INTERNATIONAL STANDARD

ISO/IEC 15437

First edition
2001-08-15

Information technology — Enhancements to LOTOS (E-LOTOS)

Technologies de l'information — Mises en valeur de LOTOS (E-LOTOS)

Reference number
ISO/IEC 15437:2001(E)

© ISO/IEC 2001

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2001

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.ch
Web www.iso.ch

Printed in Switzerland

--`,,```,,,,````-`-`,,`,,`,`,,`---

::

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

© ISO/IEC 2001 – All rights reserved

Not for Resale

Contents

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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 Succesful termination without values . . . 76
9.2.10 Succesful 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 **loop** 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

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 15437 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 7, *Software engineering*.

Annexes A and B of this International Standard are for information only.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Introduction

This International Standard contains the definition of the revised version of the LOTOS standard (ISO8807) known as E-LOTOS. The definition of E-LOTOS was based on the initial LOTOS goals enriched with feedback coming from the application of LOTOS to system design in industrial environments, as well as with the goal of supporting the ODP framework (Open Distributed Processing).

The semantics of the language is formed of a behavioural process algebra part which generalizes various LOTOS operators, and of a functional data definition part which is executable and more user friendly. The number of enhancements is high and all of them are very intertwined. A list is given now which tries to highlight from the user point of view the new features from which an E-LOTOS user should benefit when using the language. The most notable differences are:

- Modularity: E-LOTOS includes the following modularization facilities: module and interface definitions; export, import and visibility control; and generic modules. The modules can contain definitions of types, functions and/or processes.

- Data typing. E-LOTOS includes the following facilities for data definitions: predefined types, union types, recursive types, records, extensible records and record subtyping. The predefined data types and schemes are pervasive, i.e. they do not have to be declared before use in a module. As a consequence, they are directly part of the semantics model. At the semantic level, functions have been defined as a particular kind of processes which are deterministic and do not perform any visible event except termination and exception signaling.

  A two level approach has been devised for backward compatibility with LOTOS. The first level is declarative. The second level provides executable definitions of the data types which should satisfy the declarative definition of level one. With this scheme, backwards compatibility with LOTOS is achieved by having declarative specifications at level one, either in ACT ONE or in an enhanced version of ACT ONE. Functional level two data definitions should satisfy level one specifications.

- Time: E-LOTOS models events as atomic and instantaneous occurrences. The specifier can define the time at which actions or behaviours may occur. For example, time restrictions can be added to action denotations restricting the occurrence to a given set of time instants or wait statements can be used to delay the occurrence of a behaviour in time.

- Sequential composition: a new operator " ; " is introduced which substitutes the operators existing in LOTOS for sequential composition, action prefix and exit/enabling (>>) pair. For example $a;B$ or $B1;B2$. In LOTOS the >> operator always generates an internal action when the enabling is performed. In E-LOTOS, internal actions are generated only when necessary. This produces some minor differences with the semantics in LOTOS of some operators.

- Unification of processes and functions at the semantic level. A function is a process which performs only a termination action upon termination.

- Introduction of write-many variables. Write many variables are included in E-LOTOS, with a safe use assured by static semantic means. For example: assignment (write) must be performed before usage (read); no dangerous use of shared variables is made by parallel behaviours; ....

- Introduction in E-LOTOS of output variables in processes and functions as the means to pass values in sequential composition. This substitutes exit statements with value passing coupled with accept statements. The existence of input and output variables in processes and functions provides a unified approach for value communication among sequentially composed behaviours which is much more readable and concise. It is in line with notations used in ODP, like IDL (Interface Definition Language). To

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

illustrate the new approach to how sequential composition and value passing are performed let us see an example which includes variable assignment, processes and functions.

```
?x := 2;
phase1 [...] (x, ?result) ; (* a process *)
compute (result, ?res1, ?res2) ; (* a function *)
(lphase2 [...] (x, res1) ||| rphase2 [...] (x, res2))
```

- A more general parallel operator has been introduced in E-LOTOS which has the actual LOTOS parallel operator as a particular case. This operator is n-ary and supports the synchronization of J processes among K composed processes where $J \leq K$. Therefore synchronization patterns of 2 among N can be modeled in E-LOTOS. The new operator is also more readable because it clearly identifies the synchronizing gates for each behaviour composed.
- The Suspend/Resume operator which generalizes disabling. The new operator generalizes the LOTOS disabling operator by allowing a disabled behaviour to be resumed by a specific action of the disabling behaviour.
- Introduction of exceptions and exception handling in the behavioural and data parts with a uniform approach. Exception mechanisms permit new ways of structuring and are demanded by ODP.
- Explicit renaming operator for observable actions or exceptions. The renaming operator allows not only to change the name of the events occurring but to add and remove fields from the structure of events, or to merge and to split gates.
- Typed gates and partial synchronization. Gates must be explicitly typed in E-LOTOS. The use of the record subtyping relation permits partial synchronization of gates as well as provides backward compatibility with standard LOTOS untyped gates.

The introduction of enhancements has been performed trying to minimize the growth in the complexity of the language. Rather than introducing new operators, the enhancements have been introduced by generalizing existing operators. Only new statements and/or operators have been introduced when absolutely necessary. E-LOTOS is, as LOTOS, a language which permits a rich variety of specification styles which may be used to model different aspects of the design process. The styles existing in LOTOS, constraint oriented, resource oriented, state oriented, EFSM oriented, monolithic, ... can be also used in E-LOTOS. In addition the existence of exceptions, partial synchronization, event renaming, and other new constructs open new ways of specifying.

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

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:

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Symbol	*meaning*
|	aternative definition
$D, K, V, S, RT \ldots$	terminal and non-terminal symbols
text	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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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:

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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:

identifier domain	*meaning*	*abbreviation*
Con	constructor identifier	C
Const	constant identifier	K
Exc	exception identifier	X
Fun	function identifier	F
Gat	gate identifier	G
GenId	generics identifiers	*gen-id*
IntId	interface identifiers	*int-id*
ModId	module identifiers	*mod-id*
Proc	process identifier	Π
Spec	specification identifier	Σ
Typ	type identifier	S
Var	variable identifier	V

5.1.6 Non-terminals classes

The non-terminals for the language are:

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

5.2 Syntax of the language

5.2.1 Specification

spec ::= [*top-dec*] *specification* (spec1)
specification Σ [**imports** *mod-exp*(, *mod-exp*)*] **is**
[**gates** $G[:T](,G[:T])^*$] [**exceptions** $X[:T](,X[:T])^*$]
(**behaviour** B) | (**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 `()`.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.2.2 Top-level declaration

top-dec	::=	**module** *mod-id* [: *int-exp*] [**imports** *mod-exp*(, *mod-exp*)*] **is** *m-body* **endmod**	*module*	(top-dec1)
	|	**module** *mod-id* : *int-exp* **is external endmod**	*external module*	(top-dec2)
	|	**generic** *gen-id* '(' *MP* ')' [: *int-exp*] [**imports** *mod-exp*(, *mod-exp*)*] **is** *m-body* **endgen**	*generic module*	(top-dec3)
	|	**interface** *int-id* [**imports** *int-exp*(, *int-exp*)*] **is** *int-exp* **endint**	*interface*	(top-dec4)
	|	*top-dec top-dec*	*sequential top declaration*	(top-dec5)

5.2.3 Module body

m-body	::=	*D*	*block declaration*	(m-body1)
	|	*mod-exp*	*module expression*	(m-body2)

5.2.4 Module expression

mod-exp	::=	*mod-id* [: *int-exp*] [**renaming** '(' *reninst* ')']	*module aliasing*	(mod-exp1)
	|	*gen-id* '(' [*RME*] ')' [: *int-exp*] [**renaming** '(' *reninst* ')']	*actualization*	(mod-exp2)
	|	*gen-id* '(' *reninst* ')'	*renaming/instantiation*	(mod-exp3)

5.2.5 Module formal parameters

MP	::=	*mod-id* : *int-exp*	*single*	(MP1)
	|	*MP* , *MP*	*disjoint union*	(MP2)

5.2.6 Interface expressions

int-exp	::=	*int-id*	*interface id*	(int-exp1)
	|	'[' *int-id* **renaming** '(' *reninst* ')' ']'	*simple renaming*	(int-exp2)
	|	'[' *i-body* [**renaming** '(' *reninst* ')'] ']'	*explicit body*	(int-exp3)

5.2.7 Record module expression

RME	::=	*mod-id* => *mod-exp*	*single*	(RME1)
	|	*RME* , *RME*	*disjoint union*	(RME2)

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.2.8 Interface body

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

5.2.9 Formal parameter list

FPL	::=	V : T	*singleton*	(fpl1)
	|	**in** V : T	*input singleton*	(fpl2)
	|	**out** V : T	*output singleton*	(fpl3)
	|	FPL , FPL	*disjoint union*	(fpl4)

5.2.10 Renaming/Instantiation

reninst	::=	**types** S := S(,S := S)*	*types*	(reninst1)
	|	**opns** (C := C | F := F)(,C := C | F := F)*	*constructors and functions*	(reninst2)
	|	**procs** Π := Π(,Π := Π)*	*processes*	(reninst3)
	|	**values** V := V(,V := V)*	*value*	(reninst4)
	|	*reninst* , *reninst*	*renaming/instantiation tuple*	(reninst5)

5.2.11 Equations declaration

eqn-dec-list	::=	[**forall** RT] $eqn-dec$ ($eqn-dec$)*	*equations declaration*	(eqn-dec-list1)
	|	*eqn-dec-list eqn-dec-list*	*sequential*	(eqn-dec-list2)

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.2.12 Simple equations declaration

eqn-dec ::= **ofsort** *T* [**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* ')'])* **endtype** — *type declaration* (D2)

| **type** *S* **is** '(' *RT* ')' **endtype** — *named records type declaration* (D3)

| **process** Π ['[' *G*[: *T*](,*G*[: *T*])* ']'] ['(' *FPL* ')'] [**raises** '[' *X*[: *T*](,*X*[: *T*])* ']'] **is** *B* **endproc** — *process with in/out parameters* (D4)

| **function** *F* ['(' *FPL* ')'] [: *T*] [**raises** '[' *X*[: *T*](,*X*[: *T*])* ']'] **is** *E* **endfunc** — *function with in/out parameters* (D5)

| **function** *F* **infix** '(' [(**in**|**out**)]*V* : *T* , [(**in**|**out**)]*V* : *T* ')' [: *T*] [**raises** '[' *X*[: *T*](,*X*[: *T*])* ']'] **is** *E* **endfunc** — *infix function* (D6)

| **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** (**exception** *X* ['(' *IPL* ')'] **is** *E* **endexn**)* [**exit** [*P*] **is** *E* **endexit**] **in** *E* **endtrap** — *trap* (EA2)

| **var** *V* : *T*[:=*E*](,*V* : *T*[:=*E*])* **in** *E* **endvar** — *variable declaration* (EA3)

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

	rename (**signal** *X* ['(' *IPL* ')'] **is** *X* [*E*])* **in** *E* **endren**	*renaming* (EA4)
	loop *E* **endloop**	*iteration* (EA5)
	loop *X* [:*T*] **in** *E* **endloop**	*breakable iteration* (EA6)

	while *E* **do**
	  E
	endwhile
		while iteration (EA7)

	for *E* **while** *E* **by** *E* **do**
	  E
	endfor
		for iteration (EA8)

	case *E* **is** *EM* **endcase**	*case* (EA9)
	case '(' *E*(,*E*)* ')' **is** *EM* **endcase**	*case with tuples* (EA10)

	if *E* **then** *E*
	  (**elsif** *E* **then** *E*)*
	  [**else** *E*]
	endif
		if-then-else (EA11)

	F ['(' *APL* ')'] ['[' *XPL* ']']	*function instantiation* (EA12)
	(*EA*|*P*) *F* (*EA*|*P*) ['[' *XPL* ']']	*infix function instantiation* (EA13)
	raise *X* ['(' *E* ')']	*raising exception* (EA14)
	break [*X*] ['(' *E* ')']	*breaking iteration* (EA15)
	N	*value* (EA16)
	EA **andalso** *EA*	*conjunction* (EA17)
	EA **orelse** *EA*	*disjunction* (EA18)
	EA = *EA*	*equality* (EA19)
	EA <> *EA*	*inequality* (EA20)
	EA . *V*	*select field* (EA21)
	EA : *T*	*explicit typing* (EA22)
	'(' *E* ')'	*parenthesized expression* (EA23)

5.2.16 Behaviour expressions

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.2.17 Disabling behaviour expression

DisB	::=	*BT* [> *BT*	*singleton*	(DisB1)
	|	*BT* [> *DisB*	*disjoint union*	(DisB2)

5.2.18 Synchronization behaviour expression

SyncB	::=	*BT* || *BT*	*synchronization*	(SyncB1)
	|	*BT* || *SyncB*	*synchronization*	(SyncB2)

5.2.19 Concurrency behaviour expression

ConcB	::=	*BT* |[$G(,G)^*$]| *BT*	*concurrency*	(ConcB1)
	|	*BT* |[$G(,G)^*$]| *ConcB*	*concurrency*	(ConcB2)

5.2.20 Selection behaviour expression

SelB	::=	*BT* [] *BT*	*choice*	(SelB1)
	|	*BT* [] *SelB*	*choice*	(SelB2)

5.2.21 Suspend/Resume behaviour expression

SuspendB	::=	*BT* [*X*> *BT*	*suspend/resume*	(SuspendB1)
	|	*BT* [*X*> *SuspendB*	*suspend/resume*	(SuspendB2)

5.2.22 Interleaving behaviour expression

InterB	::=	*BT* ||| *BT*	*interleaving*	(InterB1)
	|	*BT* ||| *InterB*	*interleaving*	(InterB2)

5.2.23 Behaviour term

BT	::=	*BA*	*behaviour atom*	(BT1)
	|	*BA* ; *BT*	*sequential composition*	(BT2)

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.2.24 Behaviour atom

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

	|	**while** *E* **do**
		B
		endwhile
			while iteration	(BA25)

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

5.2.25 Type expressions

T	::=	*S*	*type identifier*	(T1)
	|	**none**	*empty type*	(T2)
	|	**any**	*universal type*	(T3)

5.2.26 Record type expressions

RT	::=	*V* => *T* (,*V* =>*T*)* [,**etc**]	*named*	(RT1)
	|	**etc**	*universal record*	(RT2)
	|	*T*(,*T*)*	*positional*	(RT3)

5.2.27 Value expressions

N	::=	*K*	*primitive constant*	(N1)
	|	*V*	*variables*	(N2)
	|	'(' *RN* ')'	*record values*	(N3)
	|	*C* [*N*]	*constructor application*	(N4)

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.2.28 Record value expressions

RN	::=	*V* => *N*(,*V* => *N*)*	*named*	(RN1)
	|	*N*,*N*(,*N*)*	*positional*	(RN2)

5.2.29 Patterns

P	::=	'(' *RP* ')'	*records*	(P1)
	|	**any**:*T*	*wildcard*	(P2)
	|	?*V*	*variable*	(P3)
	|	!*EA*	*expression*	(P4)
	|	*C* [*P*]	*constructor application*	(P5)
	|	*P*:*T*	*explicit typing*	(P6)

5.2.30 Gate parameter list

GPL	::=	*G* => *G* (,*G* => *G*)* [, ...]	*explicit instantiation*	(GPL1)
	|	...	*explicit instantiation*	(GPL2)
	|	*G*(,*G*)*	*positional instantiation*	(GPL3)

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

5.2.31 Actual parameter list

APL	::=	*V* => (*E*|*P*) (,*V* => (*E*|*P*))* [, ...]	*explicit instantiation*	(APL1)
	|	...	*explicit instantiation*	(APL2)
	|	(*E*|*P*)(,(*E*|*P*))*	*positional instantiation*	(APL3)

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

5.2.32 Exception parameter list

XPL	::=	*X* => *X* (,*X* => *X*)* [, ...]	*explicit instantiation*	(XPL1)
	|	...	*explicit instantiation*	(XPL2)
	|	*X*(,*X*)*	*positional instantiation*	(XPL3)

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.2.33 Record patterns

RP	::=	*V* => *P* (,*V* => *P*)*[,**etc**]	*named*	(RP1)
	|	**etc**	*wildcard*	(RP2)
	|	*P* (,*P*)*	*positional*	(RP3)

5.2.34 Behaviour pattern matching

BM	::=	*P* ['[' *E* ']'] -> *B*	*single match*	(BM1)
	|	*BM* '|' *BM*	*multiple match*	(BM2)

5.2.35 Expression pattern matching

EM	::=	*P* ['[' *E* ']'] -> *E*	*single match*	(EM1)
	|	*EM* '|' *EM*	*multiple match*	(EM2)

5.2.36 In parameter list

IPL	::=	*IP* (,*IP*)*	*disjoint union*	(IPL1)

5.2.37 In parameter

IP	::=	*V* => [*P*:]*T*	*singleton*	(IP1)
	|	**etc**	*wildcard*	(IP2)
	|	*P* **as** *IP*	*record match*	(IP3)
	|	[*P*:]*T*	*tuple*	(IP4)
	|	'(' *IPL* ')'	*parenthesized parameters*	(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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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:

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

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 $\mathcal{S} \vdash \Pi \Rightarrow Pos$.

$$
\begin{array}{lcll}
\mathcal{S} & ::= & \Pi \Rightarrow Pos & \textit{positional information} \quad \text{(S1)} \\
& \mid & \Pi \Rightarrow \textbf{gates}([G_1, \ldots, G_n]) & \textit{formal gate list} \quad \text{(S2)} \\
& \mid & \Pi \Rightarrow \textbf{params}([V_1, \ldots, V_p]) & \textit{formal parameter list} \quad \text{(S3)} \\
& \mid & \Pi \Rightarrow \textbf{exceptions}([X_1, \ldots, X_m]) & \textit{formal exception list} \quad \text{(S4)} \\
& \mid & \mathcal{S}, \mathcal{S} & \textit{disjoint union} \quad \text{(S5)}
\end{array}
$$

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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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-body	::=	**type** S	*abstract data type*	(i-body1)
	|	**type** S **renames** T **endtype**	*type synonym*	(i-body2)
	|	**type** S **is** C['(' *RT* ')'] ('|' C['(' *RT* ')'])* **endtype**	*constructed type*	(i-body3)
	|	**type** S **is** '(' *RT* ')' **endtype**	*named records type declaration*	(i-body4)
⋆	|	**process** Π ['[' $[G[:T](,G[:T])^*]$ ']'] ['(' *FPL* ')'] [**raises** '[' $X[:T](,X[:T])^*$ ']']	*process*	(i-body5)
⋆	|	**function** F ['(' *FPL* ')'] [: T] [**raises** '[' $X[:T](,X[:T])^*$ ']']	*function*	(i-body6)
⋆	|	**function** F **infix** '(' $[(\textbf{in}\|\textbf{out})]V:T$, $[(\textbf{in}\|\textbf{out})]V:T$ ')' [: T] [**raises** '[' $X[:T](,X[:T])^*$ ']']	*infix function*	(i-body7)
⋆	|	**value** V : S	*constant value*	(i-body8)
	|	**eqns** *eqn-dec-list* **endeqns**	*equations*	(i-body9)
	|	*i-body i-body*	*sequential*	(i-body10)
+	|	**process** Π ['[' $[G[:T](,G[:T])^*]$ ']'] ['(' $V:T(,V:T)^*$ ')'] : T [**raises** '[' $X[:T](,X[:T])^*$ ']']	*process with return value*	(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 ['[' $G_1[:T_1], \ldots, G_n[:T_n]$ ']'] ['(' *FPL* ')'] [**raises** $[X_1[:T_1''], \ldots, X_m[:T_m'']]$]

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

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Context

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

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

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

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

with $FPL = [\textbf{in}|\textbf{out}]\ V_1 : T_1', \ldots, [\textbf{in}|\textbf{out}]\ V_p : T_p'$

Syntax sugar

$$\left(\begin{array}{ll} \textbf{process}\ \Pi & [\ G_1 : T_1, \ldots, G_n : T_n\ ] \\ & (\ FPL\ ) \\ & \textbf{raises}\ [X_1 : T_1'', \ldots, X_m : T_m''] \end{array}\right) \stackrel{\text{def}}{=} \left(\begin{array}{ll} \textbf{process}\ \Pi & [\ G_1 : T_1, \ldots, G_n : T_n\ ] \\ & (\ I\ ) \\ & : OT \\ & \textbf{raises}\ [X_1 : T_1'', \ldots, X_m : T_m''] \end{array}\right)$$

where (ordered with ascendant indexes):
$I = \{V_i : T_i \mid [\textbf{in}]\ V_i : T_i \in FPL\}$ and
$OT = (\{T_i \mid \textbf{out}\ V_i : T_i \in FPL\})$. If there is no **out** parameters, the return type is **none**.

6.2.1.3 Function declaration

Concrete syntax

$$\begin{array}{ll} \textbf{function}\ F & [\ \text{'('}\ FPL\ \text{')'}\ ] \\ & [\ :\ T\ ] \\ & [\textbf{raises}\ \text{'['}\ X[:T](,X[:T])^*\ \text{']'}] \end{array}$$

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

Syntax sugar

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

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

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

$$\begin{array}{ll} \textbf{function}\ F\ \textbf{infix} & \text{'('}\ [(\textbf{in}|\textbf{out})]V:T\ ,\ [(\textbf{in}|\textbf{out})]V:T\ \text{')'} \\ & [\ :\ T\ ] \\ & [\textbf{raises}\ \text{'['}\ X[:T](,X[:T])^*\ \text{']'}] \end{array}$$

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Syntax sugar

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

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

6.2.1.5 Constant value

Concrete syntax

value V : S

Syntax sugar

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

6.2.2 Formal parameter list

6.2.2.1 Overview

Concrete syntax

FPL	::=	$V : T$	*singleton*	(FPL1)
	$\star$ |	**in** $V : T$	*input singleton*	(FPL2)
	$\star$ |	**out** $V : T$	*output singleton*	(FPL3)
	|	FPL, FPL	*disjoint union*	(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	::=	**type** S **renames** T **endtype**	*type synonym*	(D1)
	|	**type** S **is** C['(' RT ')'] ('|' C['(' RT ')'])* **endtype**	*type declaration*	(D2)
	|	**type** S **is** '(' RT ')' **endtype**	*named records type declaration*	(D3)
	$\star$ |	**process** Π ['[' [G[: T](, G[: T])*] ']'] ['(' FPL ')'] [**raises** [X[: T](, X[: T])*]] **is** B **endproc**	*process with in/out parameters*	(D4)
	$\star$ |	**function** F ['(' FPL ')'] [: T] [**raises** '[' X[: T](, X[: T])* ']'] **is** E **endfunc**	*function with in/out parameters*	(D5)

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$\star$ |	**function** F **infix**	'(' [(**in**|**out**)]$V:T$, [(**in**|**out**)]$V:T$ ')' [: T] [**raises** '[' X[: T](,X[: T])* ']']	*infix function*	(D6)
	is E			
	endfunc			
$\star$ |	**value** V : S **is** E **endval**		*constant value declaration*	(D7)
|	D D		*sequential declaration*	(D8)
+ |	**process** Π	['[' [G[: T](,G[: T])*] ']'] ['(' $V:T$(,$V:T$)* ')'] : T [**raises** '[' X[: T](,X[: T])* ']']	*process with return value*	(D9)
	is B			
	endproc			

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 Π ['[' 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 pararameter list is `()`, the default exception list is `[]` and the default exception type is `()`.

Context

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

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

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

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

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Syntax sugar

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

where (every tuple is ordered with ascendant indexes):
$I = (V_{i_1} : T_{i_1}, \ldots, V_{i_k} : T_{i_k})$ with $\{V_{i_1} : T_{i_1}, \ldots, V_{i_k} : T_{i_k}\} = \{V_i : T_i \mid [\textbf{in}]\ V_i : T_i \in FPL\}$,
$O = \{V_i : T_i \mid \textbf{out}\ V_i : T_i \in FPL\}$,
$OT =$ '('$\{T_i \mid \textbf{out}\ V_i : T_i \in FPL\}$')' (if there is no **out** parameters, the return type is **none**), and
$RV = \{V_i \mid \textbf{out}\ V_i : T_i \in FPL\}$.

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

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

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

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**]$V_1 : T_1$, [**in**|**out**]$V_2 : T_2$ ')'
  [: T]
  [**raises** '[' X[: T](,X[: T])* ']']
  is E
endfunc

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

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Syntax sugar

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

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

6.2.3.5 Constant value declarations

Concrete syntax

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

Syntax sugar

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

6.2.4 Expressions

6.2.4.1 Overview

Concrete syntax

E	::= EA	*expression atom*	(E1)
	| EA ; E	*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	*assignment*	(EA1)
	| **trap** (**exception** X ['(' *IPL* ')'] **is** E **endexn**)* [**exit** [P] **is** E **endexit**] **in** E **endtrap**	*trap*	(EA2)
	| **var** $V : T$[:=E](, $V : T$[:=E])* **in** E **endvar**	*variable declaration*	(EA3)
	| **rename** (**signal** X['(' *IPL* ')'] **is** X [E])* **in** E **endren**	*renaming*	(EA4)
	$\star$| **loop** E **endloop**	*iteration*	(EA5)

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

⋆ |	**loop** X [:T] **in** E **endloop**	*breakable iteration*	(EA6)

⋆ |	**while** E **do**
	  E
	endwhile
		while iteration	(EA7)

⋆ |	**for** E **while** E **by** E **do**
	  E
	endfor
		for iteration	(EA8)

⋆ |	**case** E **is** EM **endcase**	*case*	(EA9)
⋆ |	**case** '(' E(,E)* ')' **is** EM **endcase**	*case with tuples*	(EA10)

⋆ |	**if** E **then** E
	  (**elsif** E **then** E)*
	  [**else** E]
	endif
		if-then-else	(EA11)

⋆ |	F ['(' [APL] ')'] ['[' XPL ']']	*function instantiation*	(EA12)
⋆ |	(EA|P) F (EA|P) ['[' XPL ']']	*infix function instantiation*	(EA13)
⋆ |	**raise** X ['(' E ')']	*raising exception*	(EA14)
⋆ |	**break** [X] ['(' E ')']	*breaking iteration*	(EA15)
⋆ |	N	*value*	(EA16)
⋆ |	EA **andalso** EA	*conjunction*	(EA17)
⋆ |	EA **orelse** EA	*disjunction*	(EA18)
⋆ |	EA = EA	*equality*	(EA19)
⋆ |	EA <> EA	*inequality*	(EA20)
⋆ |	EA . V	*select field*	(EA21)
⋆ |	EA : T	*explicit typing*	(EA22)
|	'(' E ')'	*parenthesized expression*	(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 (δ) or exception (X) transitions, and not internal (**i**), gate (G) or delay (ε) 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 [].

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Syntax sugar

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

where $AP_i = EA_i|P_i$.

6.2.5.3 Value

Concrete syntax

N

Syntax sugar

$(N) \stackrel{\text{def}}{=} (\textbf{exit}\texttt{(\$1=>}N\texttt{)})$

6.2.5.4 Conjunction

Concrete syntax

EA **andalso** EA

Syntax sugar

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

6.2.5.5 Disjunction

Concrete syntax

EA **orelse** EA

Syntax sugar

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

6.2.5.6 Equality

Concrete syntax

EA = EA

Syntax sugar

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

6.2.5.7 Inequality

Concrete syntax

EA <> EA

Syntax sugar

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

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.2.5.8 Field select

Concrete syntax

$EA \ . \ V$

Syntax sugar

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

6.2.5.9 Explicit typing

Concrete syntax

$EA : T$

Syntax sugar

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

6.2.6 Behaviour expressions

Concrete Syntax

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

Syntax sugar Note that this category is introduced to priorize 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

$InterB$	$\star$::=	$BT \ \|\|\| \ BT$	*interleaving*	(InterB1)
	$\star$|	$BT \ \|\|\| \ InterB$	*interleaving*	(InterB2)

Syntax sugar

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

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.2.8 Behaviour atom

6.2.8.1 Overview

Concrete syntax

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

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

6.2.8.3 Bracketed disabled expression

Concrete syntax

dis *DisB* **enddis**

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Syntax sugar

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

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

$$\textbf{choice}\ P\ []\ B\ \textbf{endch}$$

Syntax sugar

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

6.2.8.5 Variable declaration

Concrete syntax

$$\textbf{var}\ V_1 : T_1\,[:=E_1]\ ,\ \ldots\ ,V_n : T_n\,[:=E_n]\ \textbf{in}\ B\ \textbf{endvar}$$

Syntax sugar

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

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
$B' = ?V_{j_1} :=E_{j_1};\ \ldots\ ;?V_{j_m} :=E_{j_m}$ where
$\{j_1,\ldots,j_m\} = \{k | V_k : T_k :=E_k \in \vec{DV}\}$ and $j_1,\ldots,j_m$ are in ascending order.

6.2.8.6 Process instantiation with abbreviated parameter lists

Concrete syntax

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

Syntax sugar

$$\frac{\mathcal{S} \vdash \Pi \Rightarrow \textbf{gates}([G_1,\ldots,G_n]) \quad \mathcal{S} \vdash \Pi \Rightarrow \textbf{params}([V_1,\ldots,V_p]) \quad \mathcal{S} \vdash \Pi \Rightarrow \textbf{exceptions}([X_1,\ldots,X_m])}{(\Pi\ [GPL]\,(APL)\,[XPL]) \stackrel{\text{def}}{=} (\Pi\ [g_1,\ldots,g_n]\,(ep_1,\ldots,ep_p)\,[x_1,\ldots,x_m])}$$

where

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

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

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

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

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

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

Π '[' $[G(,G)^*]$ ']' '(' [APL] ')' ['[' $X(,X)^*$ ']']

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

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

where

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

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

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.

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

6.2.8.9 Breakable iteration

Concrete syntax

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

The default exception type is `()`.

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

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

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

6.2.8.10 while iteration

Concrete syntax

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

Syntax sugar

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

6.2.8.11 for iteration

Concrete syntax

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

Syntax sugar

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.2.8.12 Case

Concrete syntax

case E **is**	**case** '(' $E(,E)^*$ ')' **is**
BM	BM
endcase	**endcase**

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.

$$\left(\begin{array}{l}\textbf{case } E \textbf{ is}\\ \quad BM\\ \textbf{endcase}\end{array}\right) \stackrel{\text{def}}{=} \left(\begin{array}{l}\textit{case } E \textit{ is}\\ \quad BM\\ \quad \mid \textbf{any -> raise} \text{ Match}\\ \textit{endcase}\end{array}\right)$$

$$\left(\begin{array}{l}\textbf{case } (\vec{E}) \textbf{ is}\\ \quad BM\\ \textbf{endcase}\end{array}\right) \stackrel{\text{def}}{=} \left(\begin{array}{l}\textit{case } (\vec{E}) \textit{ is}\\ \quad BM\\ \quad \mid \textbf{any -> raise} \text{ Match}\\ \textit{endcase}\end{array}\right)$$

6.2.8.13 If-then-else

Concrete syntax

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

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

Syntax sugar

(**if** E **then** B_1 **else** B_2 **endif**) $\stackrel{\text{def}}{=}$ (**case** E:bool **is** true -> B_1 | false -> B_2 **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

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

6.2.8.15 Raising exception

Concrete syntax

raise X ['(' E ')']

The default expression is ().

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Syntax sugar

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

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

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

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

(**break** `(` E `)`) $\stackrel{\text{def}}{=}$ (**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

T	::=	S	*type identifier*	(T1)
	|	**none**	*empty type*	(T2)
	|	**any**	*universal type*	(T3)

Abstract syntax

T	::=	S	*type identifier*	(T1)
	|	**none**	*empty type*	(T2)
	|	**any**	*universal type*	(T3)
	+|	`'('`RT`')'`	*records*	(T4)

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.2.10 Record type expressions

6.2.10.1 Overview

Concrete syntax

RT	$::=$	V => T $(,V$ =>$T)^*$ $[,$**etc**$]$	*named*	(RT1)
	$\mid$	**etc**	*universal record*	(RT2)
$\star\mid$		T $(,T)^*$	*positional*	(RT3)

Abstract syntax

RT	$::=$	V => T $(,V$ =>$T)^*$ $[,$**etc**$]$	*named*	(RT1)
	$\mid$	**etc**	*universal record*	(RT2)
$+\mid$			*empty record*	(RT3)

6.2.10.2 Positional

Concrete syntax

$T(,T)^*$

Syntax sugar

$(T_1,\ldots,T_n) \stackrel{\text{def}}{=} (\$1{=>}T_1,\ldots,\$n{=>}T_n)$

6.2.11 Record value expressions

6.2.11.1 Overview

Concrete syntax

RN	$::=$	V => N $(,V$ => $N)^*$	*named*	(RN1)
$\star\mid$		$N,N(,N)^*$	*positional*	(RN2)

Abstract syntax

RN	$::=$	V => N $(,V$ => $N)^*$	*named*	(RN1)
$+\mid$			*empty record*	(RN2)

6.2.11.2 Positional

Concrete syntax

$N(,N)^*$

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Syntax sugar

$(N_1,\ldots,N_n) \stackrel{\text{def}}{=} (\$1{=>}N_1,\ldots,\$n{=>}N_n)$

6.2.12 Gate parameter list

GPL	$\star$::=	G => $G(,G$ => $G)^*$ $[,\ \ldots]$	*explicit instantiation*	(GPL1)
	$\star\mid$	$\ldots$	*implicit instantiation*	(GPL2)
	$\mid$	$G(,G)^*$	*positional instantiation*	(GPL3)

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

APL	$\star$::=	V => $(E\mid P)$ $(,V$ => $(E\mid P))^*$ $[,\ \ldots]$	*explicit instantiation*	(APL1)
	$\star\mid$	$\ldots$	*implicit instantiation*	(APL2)
	$\mid$	$(E\mid P)(,(E\mid P))^*$	*positional instantiation*	(APL3)

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

XPL	$\star$::=	X => $X(,X$ => $X)^*$ $[,\ \ldots]$	*explicit instantiation*	(XPL1)
	$\star\mid$	$\ldots$	*implicit instantiation*	(XPL2)
	$\mid$	$X(,X)^*$	*positional instantiation*	(XPL3)

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

RP	::=	V => P $(,V$ => $P)^*$ $[,$**etc**$]$	*named*	(RP1)
	$\mid$	**etc**	*wildcard*	(RP2)
	$\star\mid$	$P(,P)^*$	*positional*	(RP3)

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Abstract syntax

RP	$::=$	V => P ($,V$ => $P)^*$ [**,etc**]	*named*	(RP1)
	$\mid$	**etc**	*wildcard*	(RP2)
	$+\mid$	P **as** RP	*record match*	(RP3)
	$+\mid$		*empty record*	(RP4)

6.2.15.2 Positional

Concrete syntax

$P(,P)^*$

Syntax sugar

$(P_1,\ldots,P_n) \stackrel{\text{def}}{=} (\$1{=>}P_1,\ldots,\$n{=>}P_n)$

6.2.16 In parameter list

Concrete syntax

$IP\ (,IP)^*$

Syntax sugar

$(IP_1,\ldots,IP_n) \stackrel{\text{def}}{=} (\$1 \Rightarrow IP_1,\ldots,\$n \Rightarrow IP_n)$

6.2.17 In parameter

6.2.17.1 Overview

Concrete syntax

IP	$\star::=$	$V \Rightarrow [P:]T$	*singleton*	(IP1)
	$\star\mid$	**etc**	*wildcard*	(IP2)
	$\star\mid$	P **as** IP	*record match*	(IP3)
	$\star\mid$	$[P:]T$	*tuple*	(IP4)
	$\star\mid$	'(' IPL ')'	*parenthesized parameters*	(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 **as** RP : RT

6.2.17.2 Singleton parameter list

Concrete syntax

$V \Rightarrow [P:]T$

The default pattern is **any**.

Syntax sugar

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

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.2.17.3 Wildcard

Concrete syntax

etc

Syntax sugar

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

6.2.17.4 Record match

Concrete syntax

P **as** *IP*

Syntax sugar

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

6.2.17.6 Parenthesized parameters

Concrete syntax

'(' *IPL* ')'

Syntax sugar

((*IPL*)) $\stackrel{\text{def}}{=}$ (($1 ⇒ *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:

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- **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:

$\mu ::= a \mid \delta \qquad a ::= G \mid X \mid \mathbf{i}$

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:

$\mathcal{C}$	::=	$V \Rightarrow T$	*initialized variable*	($\mathcal{C}$1)
	|	$V \Rightarrow ?T$	*typed variable*	($\mathcal{C}$2)
	|	$S \Rightarrow \mathbf{type}$	*type*	($\mathcal{C}$3)
	|	$S \equiv T$	*type equivalence*	($\mathcal{C}$4)
	|	$T \sqsubseteq T'$	*subtype*	($\mathcal{C}$5)
	|	$C \Rightarrow (RT) \rightarrow S$	*constructor*	($\mathcal{C}$6)
	|	$\Pi \Rightarrow [(\mathbf{gate}(RT))^*](RT)[(\mathbf{exn}(RT))^*] \rightarrow \mathbf{exit}(T)$	*process identifer*	($\mathcal{C}$7)
	|	$\Pi \Rightarrow [(\mathbf{gate}(RT))^*](RT)[(\mathbf{exn}(RT))^*] \rightarrow \mathbf{guarded}(T)$	*process identifer*	($\mathcal{C}$8)
	|	$G \Rightarrow \mathbf{gate}(RT)$	*gate*	($\mathcal{C}$9)
	|	$X \Rightarrow \mathbf{exn}(RT)$	*exception*	($\mathcal{C}$10)
	|		*trivial*	($\mathcal{C}$11)
	|	$\mathcal{C}, \mathcal{C}$	*disjoint union*	($\mathcal{C}$12)

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 $\mathcal{C}_1 \odot \mathcal{C}_2$ to denote the matching union on context. It is well defined only if the common names of $\mathcal{C}_1$ and C_2 have the same bindings in both contexts. Its formal definition is:

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

We shall write $\mathcal{C}_1; \mathcal{C}_2$ for context over-riding (with all the bindings of $\mathcal{C}_2$, and any bindings from $\mathcal{C}_1$ not overridden by $\mathcal{C}_2$).

We shall write $\mathcal{C} - \{V_1, \ldots, V_n\}$ for subtracting variables from the context (if $V_1 \Rightarrow T_1, \ldots, V_n \Rightarrow T_n$ and $V_1 \Rightarrow ?T_1, \ldots, V_1 \Rightarrow ?T_n$ belongs to $\mathcal{C}$, 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 $\mathcal{C}[\sigma]$ ($\mathcal{B}[\sigma]$) for the contexts that are obtained by applying the substitution σ in $\mathcal{C}$ ($\mathcal{B}$).

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).

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

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
$\mathcal{C} \vdash B \Rightarrow \mathbf{exit}(RT)$[1]
meaning "in context $\mathcal{C}$, behaviour B has result type (RT)" or
$\mathcal{C} \vdash B \Rightarrow \mathbf{guarded}(RT)$
meaning "in context $\mathcal{C}$, behaviour B has result type (RT) and cannot exit initially".
Besides, the following rules apply:

$$\frac{\mathcal{C} \vdash B \Rightarrow \mathbf{exit}(RT) \quad \mathcal{C} \vdash RT \sqsubseteq RT'}{\mathcal{C} \vdash B \Rightarrow \mathbf{exit}(RT')}$$

$$\frac{\mathcal{C} \vdash B \Rightarrow \mathbf{guarded}(RT) \quad \mathcal{C} \vdash RT \sqsubseteq RT'}{\mathcal{C} \vdash B \Rightarrow \mathbf{guarded}(RT')}$$

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

Type expressions Subtyping is a preorder:

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

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

$$\mathcal{C} \vdash T_1 \sqcup T_2 \Rightarrow T \qquad \mathcal{C} \vdash T_1 \sqcap T_2 \Rightarrow T$$

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:

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

$$\mathcal{C} \vdash RT \sqsubseteq RT'$$

Subtyping is a preorder:

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

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

[1] **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.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

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

$$\mathcal{C} \vdash N \Rightarrow T$$

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

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

$$\mathcal{C} \vdash RN \Rightarrow RT$$

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

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

$$\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT)$$

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

Behaviour pattern-matching

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

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

General axioms

$$\mathcal{C}, J \vdash J$$

This axiom means that any binding J from a context $\mathcal{C}, J$ may be infered.

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 ProcId is a pair $< scp, id >$. 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 ProcId are extended identifiers.

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.2.4 Static semantic objects for Modules

For modules we define one context. We use $\mathcal{B}$ for contexts produced by top-level declarations. It is defined by the following grammar:

$\mathcal{B}$	::=	*mod-id* $\Rightarrow \mathcal{C}$	*module*	($\mathcal{B}$1)
	|	*gen-id* $\Rightarrow$ (*mod-id* $\Rightarrow \mathcal{C})^* \rightarrow \mathcal{C}$	*generics*	($\mathcal{B}$2)
	|	*int-id* $\Rightarrow \mathcal{C}$	*interface*	($\mathcal{B}$3)
	|		*empty*	($\mathcal{B}$4)
	|	$\mathcal{B}$, $\mathcal{B}$	*disjoint union*	($\mathcal{B}$5)

We also use a *matching union* operation on contexts.

7.2.5 Judgements on static semantics for Modules

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

Specification

$\vdash$ *spec* $\Rightarrow$ `ok`

Top-level declaration

$\mathcal{B} \vdash$ *top-dec* $\Rightarrow \mathcal{B}'$

Module body

$\mathcal{B}, \mathcal{C} \vdash$ *m-body*, *id* $\Rightarrow \mathcal{C}'$

The context $\mathcal{C}$ represents the set of imported objects which may be used by *m-body*. The source of the objects defined by *m-body* is *id*. These objects are given by $\mathcal{C}'$.

Module expression

$\mathcal{B}, \mathcal{C} \vdash$ *mod-exp*, *id* $\Rightarrow \mathcal{C}'$

A module expression need both the bindings of module language (for aliasing, and actualization) and the bindings of the base language (for instantiation).

The identifier *id* is the name of the source module (or generic module) for the objects created by the expression. A module expression creates always objects except when it is an aliasing (i.e. a module identifier).

Module formal parameters

$\mathcal{B} \vdash MP \Rightarrow \mathcal{B}'$

Interface expressions

$\mathcal{B} \vdash$ *int-exp*, *id* $\Rightarrow \mathcal{C}$

The identifier *id* is the source of the objects newly declared by *int-exp*.

Interface body

$\mathcal{C} \vdash$ *i-body*, *id* $\Rightarrow \mathcal{C}'$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Record module expression

$$\mathcal{B} \vdash (RME \Rightarrow (\vec{mod\text{-}id} \Rightarrow \vec{\mathcal{C}})), id \Rightarrow \phi$$

Its static semantics need informations about the formal parameters list to be instantiated and returns the realization resulted from the substitution of the actual parameters with the formal ones.

Declarations

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

7.2.6 Cycle freedom

In our semantic definition we assume no cycle of modules names; that is, there is no sequence

$$mod\text{-}id_0 \rightarrow \cdots \rightarrow mod\text{-}id_k = mod\text{-}id_0 \quad (k > 0)$$

of modules names, where the " $\rightarrow$ " relation is the use relation of the "**imports**" construct.

7.2.7 Context morphism

Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two contexts (for example the bindings of two modules). A *context morphism* $g : \mathcal{C}_1 \rightarrow \mathcal{C}_2$ is a 3-uple of functions

$$g \stackrel{\text{def}}{=} < g_T : \mathcal{C}_1.T \rightarrow \mathcal{C}_2.T, g_C : \mathcal{C}_1.C \rightarrow \mathcal{C}_2.C, g_\Pi : \mathcal{C}_1.\Pi \rightarrow \mathcal{C}_2.\Pi, >$$

such that:

$$\begin{aligned}
&g_C(C_1 \Rightarrow (RT) \rightarrow S) \stackrel{\text{def}}{=} C_2 \Rightarrow (g_T(RT)) \rightarrow g_T(S)\\
&g_\Pi(\Pi_1 \Rightarrow [(\mathbf{gate}(RT))^*]\,(RT)\,[(\mathbf{exn}(RT))^*] \rightarrow \mathbf{exit}(T))\\
&\qquad\qquad \stackrel{\text{def}}{=} \Pi_2 \Rightarrow [(\mathbf{gate}(g_T(RT)))^*]\,(g_T(RT))\,[(\mathbf{exn}(g_T(RT)))^*] \rightarrow \mathbf{exit}(g_T(T))\\
&g_\Pi(\Pi_1 \Rightarrow [(\mathbf{gate}(RT))^*]\,(RT)\,[(\mathbf{exn}(RT))^*] \rightarrow \mathbf{guarded}(T))\\
&\qquad\qquad \stackrel{\text{def}}{=} \Pi_2 \Rightarrow [(\mathbf{gate}(g_T(RT)))^*]\,(g_T(RT))\,[(\mathbf{exn}(g_T(RT)))^*] \rightarrow \mathbf{guarded}(g_T(T))
\end{aligned}$$

where C_i, Π_i, T_i are extended identifiers.

A context morphism maps a context to another context. The arguments and the results of bindings should be consistently mapped, i.e. the arguments/result type of the binding of an operation, should be equal to the image of the arguments/result type of that operation.

7.2.8 Realization

A *realization* is a special context morphism ϕ which is an identity on simple identifiers. For example, for the type context,

$$\phi_T(< scp_1, S > \Rightarrow \mathbf{type}) \stackrel{\text{def}}{=} < scp_2, S > \Rightarrow \mathbf{type}$$

The realization changes only the scope of identifiers, i.e. their definition module, interface, or generic module name.

7.2.9 Interface Instantiation

Intuitively, a module is *an instance* of an interface if the former provides definitions of all objects declared in the second one, and only for these objects.

Formally, a module binding $\mathcal{C}_1$ is *an instance* of an interface binding $\mathcal{C}_2$, written $\mathcal{C}_2 \leq \mathcal{C}_1$, if there exists a realization morphism ϕ such that $\phi(\mathcal{C}_2) = \mathcal{C}_1$.

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.2.10 Interface Matching

Intuitively, in matching a module to an interface, the module will be allowed to define more components than those declared by the interface. In this case, the module will be a superset of an instance of the interface.

Formally, a module binding $\mathcal{C}_1$ *matches* an interface binding $\mathcal{C}_2$, written $\mathcal{C}_1 \succ \mathcal{C}_2$ if exists a module binding $\mathcal{C}'$ such that $\mathcal{C}_2 \leq \mathcal{C}' \subseteq \mathcal{C}_1$.

NOTE: This relation is a combination of interface instantiation and module enrichment. The module enrichment is, in our case, the set inclusion.

7.2.11 Renaming/Instantiation

Overview A renaming is a list of mappings from old names to new names. These mappings may be used either as a renaming to completely new names, or like an actualization of some formal objects with new (actual) objects.

The judgments for *reninst* considered as a *renaming* have the form:

$$\mathcal{C} \vdash \mathit{reninst}, \mathit{id} \Rightarrow g$$

where $\mathcal{C}$ is the context of the renamed names, *id* is the source name of the new objects created by the renaming, and g is the morphism built from *reninst*.

The judgments for *reninst* considered as an *actualization* have the form:

$$\mathcal{C} \vdash \mathit{reninst}/\mathcal{C}' \Rightarrow g$$

where $\mathcal{C}$ is the context of the actual parameters, $\mathcal{C}'$ is the context of the formal parameters, and g is the morphism built from *reninst*.

Note that g is an identity for all identifiers which are not in the support of g.

Renaming Rules

$$\frac{\mathcal{C} \vdash S_1 \Rightarrow \mathbf{type}}{\mathcal{C} \vdash (S_1 \ := \ S_2), \mathit{id} \Rightarrow (S_1 \mapsto < \mathit{id}, S_2 >)}$$

The identifier S_1 may be a long or a short identifier, and it shall be defined in $\mathcal{C}$. The identifier S_2 shall be short. The resulting morphism maps the context of S_1 on a new context where S_2 is declared having the source *id*. The resulting morphism gives only the mapping for type names.

$$\frac{\mathcal{C} \vdash C_1 \Rightarrow (RT) \rightarrow S}{\mathcal{C} \vdash (C_1 \ := \ C_2), \mathit{id} \Rightarrow (C_1 \mapsto < \mathit{id}, C_2 >)}$$

The identifier C_1 may be a long or a short identifier, and it shall be defined in $\mathcal{C}$. The identifier C_2 shall be short. The resulting morphism maps the extended identifier C_1 on a new extended identifier C_2 is declared having the source *id*. The resulting morphism gives only the mapping for constructor names. The whole morphism is obtained using the type morphism.

$$\frac{\mathcal{C} \vdash \Pi_1 \Rightarrow [(\mathbf{gate}(RT))^*](RT)[(\mathbf{exn}(RT))^*] \rightarrow \mathbf{exit}(T)}{\mathcal{C} \vdash (\Pi_1 \ := \ \Pi_2), \mathit{id} \Rightarrow (\Pi_1 \mapsto < \mathit{id}, \Pi_2 >)}$$

$$\frac{\mathcal{C} \vdash \Pi_1 \Rightarrow [(\mathbf{gate}(RT))^*](RT)[(\mathbf{exn}(RT))^*] \rightarrow \mathbf{guarded}(T)}{\mathcal{C} \vdash (\Pi_1 \ := \ \Pi_2), \mathit{id} \Rightarrow (\Pi_1 \mapsto < \mathit{id}, \Pi_2 >)}$$

The identifier Π_1 may be a long or a short identifier, and it shall be defined in $\mathcal{C}$. The identifier Π_2 shall be a short identifier. The resulting morphism maps the extended identifier Π_1 on a new extended identifier

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Π_2 is declared having the source *id*. The resulting morphism gives only the mapping for process names. The whole morphism is obtained using the type morphism.

$$\frac{\mathcal{C} \vdash \mathit{reninst}_1, \mathit{id} \Rightarrow g_1 \quad \mathcal{C} \vdash \mathit{reninst}_2, \mathit{id} \Rightarrow g_2}{\mathcal{C} \vdash (\mathit{reninst}_1 \,,\, \mathit{reninst}_2), \mathit{id} \Rightarrow g_1, g_2} \; [g_1 \text{ and } g_2 \text{ are disjoint and } g_1, g_2 \text{ is injective}]$$

The morphisms generated by each renaming list shall be disjoint, i.e. the renaming list shall rename different identifiers. Also, the resulting morphism shall be injective, i.e. it shall map two different contexts in two different contexts.

Let $\mathcal{C}$ be the renamed context, *reninst* the renaming list, and *id* the identifier of the source module or interface. The resulting morphism g is defined as follows:

$$g : \mathcal{C} \rightarrow g(\mathcal{C})$$

where:

$$\begin{array}{lll}
g_T(< \mathit{src}, S_1 > \Rightarrow \mathbf{type}) & =< \mathit{id}, S_2 > \Rightarrow \mathbf{type} & \text{if } (S_1 \;\texttt{:=}\; S_2) \in \mathit{reninst} \\
& =< \mathit{src}, S_1 > \Rightarrow \mathbf{type} & \text{otherwise} \\
g_C(< \mathit{src}, C_1 > \Rightarrow (RT) \rightarrow S) & =< \mathit{id}, C_2 > \Rightarrow (g_T(RT)) \rightarrow g_T(S) & \text{if } (C_1 \;\texttt{:=}\; C_2) \in \mathit{reninst} \\
& =< \mathit{src}, C_1 > \Rightarrow (g_T(RT)) \rightarrow g_T(S) & \text{otherwise} \\
g_\Pi(< \mathit{src}, \Pi_1 > \Rightarrow [(\mathbf{gate}(RT))^*]\,(RT')\,[(\mathbf{exn}(RT''))^*] \rightarrow \mathbf{exit}(T)) & & \\
\quad =< \mathit{id}, \Pi_2 > \Rightarrow [(\mathbf{gate}(g_T(RT)))^*]\,(g_T(RT'))\,[(\mathbf{exn}(g_T(RT')))^*] & & \\
\quad\quad \rightarrow \mathbf{exit}(g_T(T)) & & \\
\quad\quad & & \text{if } (\Pi_1 \;\texttt{:=}\; \Pi_2) \in \mathit{reninst} \\
\quad =< \mathit{src}, \Pi_1 > \Rightarrow [(\mathbf{gate}(g_T(RT)))^*]\,(g_T(RT'))\,[(\mathbf{exn}(g_T(RT')))^*] & & \\
\quad\quad \rightarrow \mathbf{exit}(g_T(T)) & & \\
\quad\quad & & \text{otherwise}
\end{array}$$

All the requirements expressed by the rules shall be satisfied, i.e.:

Req.1 The morphism g shall be injective in each class of identifiers.

Req.2 The renaming list *reninst* shall rename different identifiers, i.e. all *is* of $\mathit{id} \;\texttt{:=}\; \mathit{id}'$ shall be distinct.

Req.3 The context $g(\mathcal{C})$ shall be well formed, i.e. an extended identifier shall be bound only once.

Actualization Rules

$$\frac{\mathcal{C} \vdash S_2 \Rightarrow \mathbf{type}}{\mathcal{C} \vdash (S_1 \;\texttt{=>}\; S_2)/\mathcal{C}', S_1 \Rightarrow \mathbf{type} \Rightarrow (S_1 \mapsto S_2) \odot \phi} \; [\phi(\mathcal{C}') \subseteq \mathcal{C}]$$

This rule deals with the elaboration of the actualization of types. The identifiers S_1 and S_2 may be long or short identifiers. The former shall be declared in the formal context $\mathcal{C}'$, and the second in the actual context $\mathcal{C}$. In the mapping, the extended identifiers are used. The remainder of formal elements given by $\mathcal{C}'$, shall have a realization in the actual context $\mathcal{C}$ (implicit instantiation). The realization ϕ may match the morphism given by the explicit instantiation.

$$\frac{\mathcal{C} \vdash C_2 \Rightarrow (V_1' \texttt{=>} T_1', \ldots, V_n' \texttt{=>} T_n') \rightarrow S_2}{\begin{array}{l}\mathcal{C} \vdash (C_1 \;\texttt{:=}\; C_2)/\mathcal{C}', C_1 \Rightarrow (V_1 \texttt{=>} T_1, \ldots, V_n \texttt{=>} T_n) \rightarrow S_1 \\ \quad \Rightarrow (C_1 \mapsto C_2, T_1 \mapsto T_1', \ldots, T_n \mapsto T_n', S_1 \mapsto S_2)\end{array}} \; [\phi(\mathcal{C}') \subseteq \mathcal{C}]$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This rule deals with the elaboration of the actualization of a formal constructor. The resulting morphism must maps the constructor names and result types.

$$\frac{\begin{array}{l}\mathcal{C} \vdash \Pi_2 \Rightarrow [GL_2]\,(FPL_2)\,[XL_2] \rightarrow \mathbf{exit}(T_2) \\ \quad \vdash (GL_1 \Rightarrow GL_2) \Rightarrow g_1 \\ \quad \vdash (FPL_1 \Rightarrow FPL_2) \Rightarrow g_2 \\ \quad \vdash (XL_1 \Rightarrow XL_2) \Rightarrow g_3\end{array}}{\begin{array}{l}\mathcal{C} \vdash (\Pi_1\ \mathtt{:=}\ \Pi_2)/\mathcal{C}', \Pi_1 \Rightarrow [GL_1]\,(FPL_1)\,[XL_1] \rightarrow \mathbf{exit}(T_1) \\ \qquad \Rightarrow (\Pi_1 \mapsto \Pi_2, g_1 \odot g_2 \odot g_3 \odot \phi)\end{array}}\ [\phi(\mathcal{C}') \subseteq \mathcal{C}]$$

$$\frac{}{\vdash () \Rightarrow () \Rightarrow \{\}}$$

$$\frac{}{\vdash ((\mathbf{gate}\ T_1) \Rightarrow (\mathbf{gate}\ T_2)) \Rightarrow (T_1 \mapsto T_2)}$$

$$\frac{\begin{array}{l}\vdash GL_1 \Rightarrow GL_2 \Rightarrow g \\ \vdash GL_1' \Rightarrow GL_2' \Rightarrow g'\end{array}}{\vdash GL_1, GL_1' \Rightarrow GL_2, GL_2' \Rightarrow g \odot g'}$$

$$\frac{}{\vdash ((V_1 \Rightarrow T_1) \Rightarrow (V_1 \Rightarrow T_2)) \Rightarrow (T_1 \mapsto T_2)}$$

$$\frac{\begin{array}{l}\vdash FPL_1 \Rightarrow FPL_2 \Rightarrow g \\ \vdash FPL_1' \Rightarrow FPL_2' \Rightarrow g'\end{array}}{\vdash FPL_1, FPL_1' \Rightarrow FPL_2, FPL_2' \Rightarrow g \odot g'}$$

$$\frac{}{\vdash ((\mathbf{exn}\ T_1) \Rightarrow (\mathbf{exn}\ T_2)) \Rightarrow (T_1 \mapsto T_2)}$$

$$\frac{\begin{array}{l}\vdash XL_1 \Rightarrow XL_2 \Rightarrow g \\ \vdash XL_1' \Rightarrow XL_2' \Rightarrow g'\end{array}}{\vdash XL_1, XL_1' \Rightarrow XL_2, XL_2' \Rightarrow g \odot g'}$$

where

$$GL_i \stackrel{\text{def}}{=} \mathbf{gate}\ T\,(,\mathbf{gate}\ T)^*$$

$$FPL_i \stackrel{\text{def}}{=} V \Rightarrow T\,(,V \Rightarrow T)^*$$

$$XL_i \stackrel{\text{def}}{=} \mathbf{exn}\ T\,(,\mathbf{exn}\ T)^*$$

This rule deals with the elaboration of the actualization of a process. The identifiers Π_1 and Π_2 may be long or short identifiers. The former shall be declared in the formal context $\mathcal{C}'$, and the second in the actual context $\mathcal{C}$. The matching of the parameter list gives three type morphisms, which shall match in the common elements. The resulting morphism maps the processes names and the types names. The realization ϕ correspond to the implicit actualization.

$$\frac{\begin{array}{l}\mathcal{C} \vdash \mathit{reninst}_1/\mathcal{C}_1 \Rightarrow g_1 \\ \mathcal{C} \vdash \mathit{reninst}_2/\mathcal{C}_2 \Rightarrow g_2\end{array}}{\mathcal{C} \vdash (\mathit{reninst}_1\,\mathtt{,}\,\mathit{reninst}_2)/\mathcal{C}_1, \mathcal{C}_2 \Rightarrow g_1 \odot g_2}\ [g_1 \odot g_2 \text{ is total on } \mathcal{C}_1, \mathcal{C}_2]$$

In the elaboration of the sequencing of actualizations, the morphisms generated by each renaming list shall match in the common elements. Also, the resulting morphism shall be total on the formal parameter contexts.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Let $\mathcal{C}'$ be the formal context, $\mathcal{C}$ the actualizing context, and *reninst* the actualizing list. The resulting morphism g is defined as follows:

$$g : \mathcal{C}' \to \mathcal{C}$$

It shall be total in $\mathcal{C}'$, i.e. for all $obj \in \mathcal{C}'$ the value $g(obj)$ is defined in $\mathcal{C}$. The morphism is defined by the following rules:

$$\begin{array}{lll}
g_T(< src', S_1 > \Rightarrow \mathbf{type}) & = < src, S_2 > \Rightarrow \mathbf{type} & \text{if } (S_1 \ := \ S_2) \in reninst \\
 & & \text{and } \mathcal{C} \vdash < src, S_2 > \Rightarrow \mathbf{type} \\
 & = < src, S_1 > \Rightarrow \mathbf{type} & \text{if } (S_1 \ := \ S_2) \notin reninst \\
 & & \text{and } \mathcal{C} \vdash < src, S_1 > \Rightarrow \mathbf{type} \\
 & = \text{undefined} & \text{otherwise} \\
g_C(< src', C_1 > \Rightarrow (RT) \to S) & = < src, C_2 > \Rightarrow (g_T(RT)) \to g_T(S) & \\
 & & \text{if } (C_1 \ := \ C_2) \in reninst \\
 & & \text{and } \mathcal{C} \vdash < src, C_2 > \Rightarrow (g_T(RT)) \to g_T(S) \\
 & = < src, C_1 > \Rightarrow (g_T(RT)) \to g_T(S) & \\
 & & \text{if } (C_1 \ := \ C_2) \notin reninst \\
 & & \text{and } \mathcal{C} \vdash < src, C_1 > \Rightarrow (g_T(RT)) \to g_T(S) \\
 & = \text{undefined} & \text{otherwise} \\
g_\Pi(< src', \Pi_1 > \Rightarrow [(\mathbf{gate}(RT))^*]\,(RT')\,[(\mathbf{exn}(RT''))^*] \to \mathbf{exit}(T)) & & \\
 & = < src, \Pi_2 > \Rightarrow [(\mathbf{gate}(g_T(RT)))^*]\,(g_T(RT'))\,[(\mathbf{exn}(g_T(RT'')))^*] & \\
 & \quad \to \mathbf{exit}(g_T(T)) & \\
 & & \text{if } (\Pi_1 \ := \ \Pi_2) \in reninst \\
 & & \text{and } \mathcal{C} \vdash < src, \Pi_2 > \Rightarrow \\
 & & \quad [(\mathbf{gate}(g_T(RT)))^*]\,(g_T(RT'))\,[(\mathbf{exn}(g_T(RT'')))^*] \\
 & & \quad \to \mathbf{exit}(g_T(T)) \\
 & & \text{otherwise}
\end{array}$$

7.3 Untimed dynamic semantics

7.3.1 Untimed dynamic semantic objects for Base

The untimed dynamic semantics[2] is given by a series of judgements, such as

$$\mathcal{E} \vdash B \xrightarrow{\delta(RN)} B'$$

meaning "in environment $\mathcal{E}$, behaviour B terminates with result (RN)". The environment gives the bindings for free identifers, and is given by the grammar:

$$\begin{array}{lll}
\mathcal{E} ::= & S \equiv T & \textit{type equivalence} \quad (\mathcal{E}1) \\
\mid & C \Rightarrow (RT) \to S & \textit{constructor} \quad (\mathcal{E}2) \\
\mid & \Pi \Rightarrow \lambda\,[(G(RT))^*]\,(RP : RT)\,[(X(RT))^*] \to B & \textit{process identifer} \quad (\mathcal{E}3) \\
\mid & & \textit{trivial} \quad (\mathcal{E}4) \\
\mid & \mathcal{E}, \mathcal{E} & \textit{disjoint union} \quad (\mathcal{E}5)
\end{array}$$

Note that environments have to carry type information. This is because LOTOS relies on run-time typing for much of its semantics, for example the semantics of the nondeterministic expression **any** T depends on the type rules for T.

The semantics for expressions with free variables uses *substitution* to replace free variables with values. The grammar for substitutions is given by:

[2] We will use the term "dynamic semantics" for untimed dynamic semantics.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$$\begin{array}{lll} \sigma & ::= & V \Rightarrow N \qquad \textit{singleton} \quad (\sigma 1) \\ & | & \qquad \textit{trivial} \quad (\sigma 2) \\ & | & \sigma, \sigma \qquad \textit{disjoint union} \quad (\sigma 3) \end{array}$$

where each variable is only bound once. We write $B[\sigma]$ for B with all free variables replaced by values given by σ with the usual α-conversion to avoid binding free variables. See Subclause 7.5 for the complete definition of substitution operator.

Note that the grammar for substitutions is the same as the grammar for record values RN, so we will use them interchangeably (for example in the dynamic semantics of sequential composition in Subclause 9.2.6).

7.3.2 Judgements on untimed dynamic semantics for Base

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

Behaviours The untimed dynamic semantics is given by judgements of the form:

$$\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B'$$

Type expressions

$$\mathcal{E} \vdash T \sqsubseteq T'$$

In each case, the judgements are the same as for the static semantics, so we omit them.

Record type expressions

$$\mathcal{E} \vdash RT \sqsubseteq RT'$$

In each case, the judgements are the same as for the static semantics, so we omit them.

Value expressions

$$\mathcal{E} \vdash N \Rightarrow T$$

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

In each case, the judgements are the same as for the static semantics, so we omit them.

Record value expressions

$$\mathcal{E} \vdash RN \Rightarrow RT$$

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

In each case, the judgements are the same as for the static semantics, so we omit them.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Patterns

$$\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN)$$

$$\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow \textbf{fail}$$

The first judgement means that in environment $\mathcal{E}$, it is possible to match the value of P with N, by means of the value bindings of (RN). The second one means that it is not possible such matching.

Variable binding

$$\mathcal{E} \vdash (V \Rightarrow ?T)$$

This judgment is introduced in local variable declarations. This means that the variable V is restricted to (sub)type T.

Behaviour pattern-matching

$$\mathcal{E} \vdash (BM \Rightarrow N) \Rightarrow \textbf{fail}$$

$$\mathcal{E} \vdash (BM \Rightarrow N) \xrightarrow{\mu(RN)} B$$

7.3.3 Dynamic semantic objects for Modules

The semantic objects for the Modules dynamic semantics are given by the grammars below. The dynamic basis is given by the dynamic environment of the base language, which is the result of top level declarations, and the environment of modules, generic modules, and interfaces.

$\mathcal{D}$	::=	*mod-id* $\Rightarrow I$	*module*	($\mathcal{D}$1)
	|	*gen-id* $\Rightarrow$ (*mod-id* $\Rightarrow I)^* \rightarrow \mathcal{E}$	*generics*	($\mathcal{D}$2)
	|	*int-id* $\Rightarrow I$	*interface*	($\mathcal{D}$3)
	|	$\mathcal{E}$	*environment*	($\mathcal{D}$4)
	|		*empty*	($\mathcal{D}$5)
	|	$\mathcal{D}, \mathcal{D}$	*disjoint union*	($\mathcal{D}$6)

Module and interfaces evaluated to *signatures* (or interfaces) I, which are a "view" of the module (interface). It contains all the extended identifiers of the module (interface). The binding of a generic module is a *functor* from a record of module binding to a base language environment. The objects of the environment $\mathcal{E}$ are uses the identifiers declared in the interface list I^*. Signatures are collection of type, constructor and process extended identifiers. They are defined by the grammar below:

I	::=	S	*type (ex-id)*	(I1)
	|	C	*constructor (ex-id)*	(I2)
	|	Π	*process (ex-id)*	(I3)
	|	I, I	*disjoint union*	(I4)

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

To extract a signature I from a dynamic environment $\mathcal{E}$ the operation Sig is used defined as follows:

$$\mathrm{Sig}(\mathcal{E}) \stackrel{\mathrm{def}}{=} \mathrm{Dom}(\mathcal{E})$$

Another further operation "$\downarrow$" is required to cut down an environment $\mathcal{E}$ to a given signature I, representing the effect of an explicit interface ascription. It is equivalent to the restriction of a function domain (here $\mathcal{E}$ domain).

A signature is also a projection of the *static* context $\mathcal{C}$; it is obtained by omitting variable, gate, and exception context, and replacing each type, constructor and process context by its domain. Thus in an implementation signatures would naturally be obtained from the static elaboration. We choose to give separate rules here for obtaining them since it is important to maintain separation between static and dynamic semantics, for reason of presentation.

After the top-level evaluation, the basis $\mathcal{D}$ contains the complete dynamic environment of the specification. This environment includes all complete specified (non-generic) type, constructor and process definitions, including the objects locally defined by a module.

7.3.4 Judgements on untimed dynamic semantics for Modules

In this subclause, we describe judgements that define the dynamic semantics for the modules language.

Specification

$$\vdash \mathit{spec} \xrightarrow{\mu(RN)} B$$

Top-level declaration

$$\mathcal{D} \vdash \mathit{top\text{-}dec} \Rightarrow \mathcal{D}'$$

Module body

$$\mathcal{D} \vdash \mathit{m\text{-}body}, \mathit{id} \Rightarrow \mathcal{E}'$$

Module expression

$$\mathcal{D} \vdash \mathit{mod\text{-}exp}, \mathit{id} \Rightarrow \mathcal{E}'$$

Module formal parameters

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

Interface expressions

$$\mathcal{D} \vdash \mathit{int\text{-}exp}, \mathit{id} \Rightarrow I$$

Interface body

$$\mathcal{D} \vdash \mathit{i\text{-}body}, \mathit{id} \Rightarrow I$$

Record module expression

$$\mathcal{D} \vdash (RME \Rightarrow (\vec{\mathit{mod}}\text{-}\mathit{id} \Rightarrow \vec{I})), \mathit{id} \Rightarrow \phi$$

Declarations

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.3.5 Environment morphism

The context morphism of the static semantics is adapted to the dynamic semantics environments. Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two environments. An *environment morphism* $g : \mathcal{E}_1 \to \mathcal{E}_2$ is a 3-uple of functions

$$g \stackrel{\text{def}}{=} < g_T : \mathcal{E}_1.S \to \mathcal{E}_2.S, g_C : \mathcal{E}_1.C \to \mathcal{E}_2.C, g_\Pi : \mathcal{E}_1.\Pi \to \mathcal{E}_2.\Pi, >$$

such that:

$$\begin{array}{rcl} g_C\,(C_1 \Rightarrow (RT) \to S) & \stackrel{\text{def}}{=} & C_2 \Rightarrow (g_T(RT)) \to S \\ g_\Pi\,(\Pi_1 \Rightarrow \lambda\,[(G\,(RT))^*]\,(RP : RT) & \stackrel{\text{def}}{=} & \Pi_2 \Rightarrow \lambda\,[(G\,(g_T(RT)))^*]\,(RP : g_T(RT)) \\ \quad [(X\,(RT))^*] \to B) & & \quad [(X\,(g_T(RT)))^*] \to g(B) \end{array}$$

where C_i, Π_i, T_i are extended identifiers. Note that g is in this case a substitution.

A signature *realization* is a special context morphism ϕ which is an identity on simple identifiers. The realization changes only the scope of identifiers, i.e. their definition module, interface, or generic module name.

7.3.6 Signature Instantiation

A signature I_1 is an instance of another signature I_2, noted $I_2 \leq I_1$, if exists a realization morphism ϕ such that $\phi(I_2) = I_1$.

7.3.7 Renaming/Instantiation

Overview

$\mathcal{E} \vdash \mathit{reninst}, \mathit{id} \Rightarrow g$

$I \vdash \mathit{reninst}, \mathit{id} \Rightarrow g$

$\mathcal{E} \vdash \mathit{reninst}/I \Rightarrow g$

The definition is given by the static semantics rules.

7.4 Timed dynamic semantics

We shall write $\mathcal{E} \vdash B \stackrel{\mu(RN@d)}{\longrightarrow} B'$ when either:

- $\mathcal{E} \vdash B \stackrel{\mu(RN)}{\longrightarrow} B'$ and $d = 0$, or
- $\mathcal{E} \vdash B \stackrel{\varepsilon(d)}{\longrightarrow} B''$ and $\mathcal{E} \vdash B'' \stackrel{\mu(RN)}{\longrightarrow} B'$

Requirements on the time domain:

1. The only closed normal forms of type time are the special constants ranged over by d.
2. The time domain is a commutative cancellative monoid $+$ with unit 0.
3. The order given by $d_1 \leq d_2$ iff $\exists d\,.\,d_1 + d = d_2$ is a total order.

Since time is a commutative cancellative monoid, it satisfies the properties:

$$d_1 + d_2 = d_2 + d_1 \quad \text{if } d_1 + d = d_2 + d \text{ then } d_1 = d_2 \quad d_1 + (d_2 + d_3) = (d_1 + d_2) + d_3 \quad d + 0 = d = 0 + d$$

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.4.1 Judgements on timed dynamic semantics

In this subclause, we describe judgements that define the timed dynamic semantics.

Behaviours The timed dynamic semantics is given by judgements of the form:

$$\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B'$$

Behaviour pattern-matching

$$\mathcal{E} \vdash (BM \Rightarrow N) \xrightarrow{\varepsilon(d)} B$$

7.5 Write-many variables: the value substitution operator

The static semantics assures that variables are used in a safe way: they always get a value (write) before usage (read). Besides, variables may be reasigned as long as they exist. However, it is highly desirable to avoid dangerous use of shared variables by parallel behaviours, as communication should be explicit.

When a variable receives a value, a "binding" is produced between its identifier and the value. The static semantics checks that a binding exists when a variable is used (this assures that it is a variable and that it has a value).

Dynamic semantics passes bindings from a behaviour to the resulting one when some action or exception takes place. Subsequent bindings are permitted, overriding existing ones. This is achieved via the value substitution operator $B[RN]$, which applies bindings RN to a behaviour B. This operator is defined as syntactic substitution of values (see Subclause 7.3.1) with the following exceptions:

Sequential composition

$$(B_1 \ ; \ B_2)\ [RN] \stackrel{\text{def}}{=} B_1[RN] \ ; \ B_2[RN']$$

with $RN' = RN - RN''$, being $RN'' = av(B_1)$ are the bindings from B_1[3].

The following behaviour makes the following binding $(x = 2, y = 2)$.

```
?x := 1; ?x := x + 1; ?y:= x
```

In the following example, there are two possibilities:

```
?x := 0;
  (a!x; ...
    |||
    ?x := x + 1; b!x; ...
  )
```

we may see $-\ a!0\ -\ b!1\ -\ \ldots$ or maybe $-\ b!1\ -\ a!0\ -\ \ldots$ as there is no "implicit" communication among behaviours. The static semantics requires that each branch produces disjoint bindings.

[3] It is easy and cumbersome to define it. In fact, it is part of the information produced by the static semantics checking.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Branching operator We consider as "branching" operators those composed by several subbehaviours and this composition may evolves as any of them: selection, disabling, suspend/resume, trap, and case are "branching" operators. Note that these are semantics operators, **if-then-else** is syntax sugar of **case**.

The subtituting operator should not perform a blind subtitution of values, as some branch may change any of the bound variable and the new value need to be propagated. Therefore, the subtituting operator introduces the assignments of bound variable before the subbehaviours, leaving the real subtitution to the sequential (;) composition (see above). These assignments are called "dummy" asignments, and it is just a conceptual mechanism to carry bound variables[4].

$$(B_1 \,[]\, B_2)[RN] \stackrel{\text{def}}{=} \textbf{exit}(RN)\,;B_1 \;[]\; \textbf{exit}(RN)\,;B_2$$

Let us see an example:

```
?x := 1;
(?x := x+1; a
[]
b
);
?y := x
```

Firstly, x is bound, with $RN = (x \Rightarrow 1)$, which the sequential composition transfers to the selection operator (note that it should stop the subtitution, as ?y:=x should not be affected yet. Therefore, the after subtitution is:

```
(?x := 1; ?x := x+1; a
[]
?x := 1; b
);
?y := x
```

Now, if the upper branch is selected, the resulting binding is $(x \Rightarrow 2)$, and if the lower branch is selected, the resulting binding is $(x \Rightarrow 1)$.

In general, we have:

$$(B_1 \; op \; B_2)[RN] \stackrel{\text{def}}{=} \textbf{exit}(RN)\,;B_1 \; op \; \textbf{exit}(RN)\,;B_2$$

with $op \in \{[], [>, [X>\}$.

The following example shows how substitution works with branching operators:

$$(B_1 \;[>\; B_2)[RN] \stackrel{\text{def}}{=} \textbf{exit}(RN)\,;B_1 \;[>\; \textbf{exit}(RN)\,;B_2$$

bindings in B_1 do not affect B_2, in the line of forgetting bindings when a behaviour is interrupted.

```
?x := 0; ((?x := x+1; a; c; ...) [> (b; ?x := x + 10))
```

If we see the sequence $-a-b->$ the binding is $(x = 10)$.

For **trap** and **case** it is more cumbersome:

```
(trap                                          trap
   exception X1(IPL1) is B1 endexn               exception X1(IPL1) is exit(RN'1);B1 endexn
   ⋮                                             ⋮
   exception Xn(IPLn) is Bn endexn      def      exception Xn(IPLn) is exit(RN'n);Bn endexn
   exit P is Be endexn                           exit P is exit(RN'e);Be endexn
in                                             in
   B                                             exit(RN);B
endtrap)[RN]                                   endtrap
```

[4]Tools (as simulator, compilers) may avoid easily these "dummy" asignments.

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

where $RN'_i = RN - RN''_i$ and RN''_i are the bindings produced by IPL_i (by P for the successful termination).

Intuitively, we try the behaviour B. If the exception X_i is raised, the partial bindings produced by B are forgotten. Behaviour B_i is launched with $[RN'_i]$, without implicit communication between B and the exception managers.

$$
\begin{array}{ll}
(\textbf{case } expr \textbf{ is} & \textbf{case } expr[RN] \textbf{ is} \\
\quad BM_1 & \quad BM_1[RN] \\
\quad \vdots & \stackrel{\text{def}}{=} \quad \vdots \\
\quad BM_n & \quad BM_n[RN] \\
\textbf{endcase})[RN] & \textbf{endcase}
\end{array}
$$

where $(P\,[E]\,\text{->}B)[RN] = P[RN]\,[E[RN']]\,\text{->}\textbf{exit}(RN');B$ and RN' are the bindings in RN except those related with variables bound by P.

Variable declaration

$$(\textbf{var } LV \textbf{ in } B)\ [RN] \stackrel{\text{def}}{=} \textbf{var } LV[RN] \textbf{ in } B[RN']$$

where $RN' = RN - LV$, i.e. the substitution $[RN']$ is the same as $[RN]$ but constrained to the variables which are not declared in LV.

Loop

$$(\textbf{loop } B \textbf{ endloop})\ [RN] \stackrel{\text{def}}{=} B[RN]\ ;\ \textbf{loop } B\ [RN'] \textbf{ endloop}$$

where $RN' = RN - av(B)$ being $av(B)$ the bindings from B i.e. the substitution $[RN']$ is the same as $[RN]$ but constrained to the variables which are not bound in B. Note the static semantics provides the way to determine $av(B)$.

B may produce some bindings, propagated through the ";" (see above). We could take some advantage here as "untouched" variables may be substituted directly (this is the reason of introducing $[RN']$ inside the second unfolding of the loop).

```
?x:= 1;
loop
  a ! x;
  ?x:= x + 1;
  if x= 10 then break endif
endloop
; c!x
```

This behaviour will offer $a!1 - a!2 - a!3 - ... - a!8 - a!9 - c!1 - >$.

We could get the value via trapping the result:

```
?x:=1;
trap
  exception result (?y:int) is ?x:= y;endexn
in
  loop
    a!x;
    ?x:= x + 1;
    if x=10 then break result (x) endif
  endloop
endtrap;
c!x;
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8 The E-LOTOS modules

8.1 Specification

An E-LOTOS specification is a sequence of top level declarations followed by a block which declares a behaviour or an expression as *entry point*. The result of a such specification is the evaluation of this entry point, after the elaboration of the top-level declarations.

8.1.1 Specification

[*top-dec*]
specification Σ [**imports** *mod-exp*(, *mod-exp*)*] **is**
  [**gates** $G:T(,G:T)^*$] [**exceptions** $X:T(,X:T)^*$]
  (**behaviour** B) | (**value** E)

The import, gate and exceptions clauses are empty by default.

Static semantics

$$\frac{\begin{array}{l}\vdash \textit{top-dec} \Rightarrow \mathcal{B} \\ \mathcal{B} \vdash \textit{mod-exp}_1, \textit{sp-id} \Rightarrow \mathcal{C}_1 \ \cdots \ \mathcal{B} \vdash \textit{mod-exp}_p, \textit{sp-id} \Rightarrow \mathcal{C}_p \\ \mathcal{C}_1, \ldots, \mathcal{C}_p, \\ G_1 \Rightarrow \mathbf{gate}(T_1) \ \cdots \ G_m \Rightarrow \mathbf{gate}(T_m), \\ X_1 \Rightarrow \mathbf{exn}(T'_1) \ \cdots \ X_n \Rightarrow \mathbf{exn}(T'_n) \quad \vdash B \Rightarrow \mathbf{exit}(RT)\end{array}}{\begin{array}{l}\vdash (\textit{top-dec} \\ \quad \textbf{specification } \textit{sp-id} \\ \quad \textbf{imports } \textit{mod-exp}_1, \ldots, \textit{mod-exp}_p \textbf{ is} \\ \quad \textbf{gates } G_1:T_1, \ldots, G_m:T_m \\ \quad \textbf{exceptions } X_1:T'_1, \ldots, X_n:T'_n \\ \quad \textbf{behaviour } B\ ) \Rightarrow \mathtt{ok}\end{array}}$$

Dynamic semantic

$$\frac{\begin{array}{l}\vdash \textit{top-dec} \Rightarrow \mathcal{D} \\ \mathcal{D} \vdash \textit{mod-exp}_1, \textit{sp-id} \Rightarrow \mathcal{E}_1 \ \cdots \ \mathcal{D} \vdash \textit{mod-exp}_p, \textit{sp-id} \Rightarrow \mathcal{E}_p \\ \mathcal{E}_1, \ldots, \mathcal{E}_p \vdash B \xrightarrow{\mu(RN)} B'\end{array}}{\begin{array}{l}\vdash (\textit{top-dec} \\ \quad \textbf{specification } \textit{sp-id} \\ \quad \textbf{imports } \textit{mod-exp}_1, \ldots, \textit{mod-exp}_p \textbf{ is} \\ \quad \textbf{gates } G_1:T_1, \ldots, G_m:T_m \\ \quad \textbf{exceptions } X_1:T'_1, \ldots, X_n:T'_n \\ \quad \textbf{behaviour } B) \xrightarrow{\mu(RN)} B'\end{array}}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.2 Top-level declaration

A top-level declaration is a sequence (maybe empty) of module declarations, generic module declarations and interface declarations.

8.2.1 Module not constrained by an interface

Abstract syntax

module *mod-id* [**imports** *mod-exp*(, *mod-exp*)*] **is** *m-body*

The default import clause is empty.

Static semantics

$$\frac{\begin{array}{c}\mathcal{B} \vdash \textit{mod-exp}_1, \textit{mod-id} \Rightarrow \mathcal{C}_1 \ \cdots \ \mathcal{B} \vdash \textit{mod-exp}_m, \textit{mod-id} \Rightarrow \mathcal{C}_m \\ \mathcal{B}, \mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m \vdash \textit{m-body}, \textit{mod-id} \Rightarrow \mathcal{C}\end{array}}{\begin{array}{l}\mathcal{B} \vdash (\textbf{module } \textit{mod-id } \textbf{imports } \textit{mod-exp}_1 \textbf{ , } \ldots \textbf{ , } \textit{mod-exp}_m \textbf{ is } \textit{m-body}) \\ \qquad \Rightarrow (\textit{mod-id} \Rightarrow \mathcal{C}, \mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m)\end{array}}$$

with side condition [$\mathcal{C}$ and $\mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m$ have disjoint fields]

The module body is checked in the context of objects imported by the importation clause. The module name is carried on to assign the source part of the extended identifier of the objects declared by *m-body*. The imported contexts are composed using the matching operator "$\odot$" in order to allow the diamond importation scheme (to import several time the same object). However, the objects declared by the module body shall be disjoint from the imported ones. For this reason is used the disjunct composition operator "," between contexts. The interface of the module is given by the imported objects and the (new) declared objects.

Dynamic semantics

$$\frac{\begin{array}{c}\mathcal{D} \vdash \textit{mod-exp}_1, \textit{mod-id} \Rightarrow \mathcal{E}_1 \ \cdots \ \mathcal{D} \vdash \textit{mod-exp}_m, \textit{mod-id} \Rightarrow \mathcal{E}_m \\ \mathcal{D} \odot \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m \vdash \textit{m-body}, \textit{mod-id} \Rightarrow \mathcal{E}\end{array}}{\begin{array}{l}\mathcal{D} \vdash (\textbf{module } \textit{mod-id } \textbf{imports } \textit{mod-exp}_1, \ldots, \textit{mod-exp}_m \textbf{ is } \textit{m-body}) \\ \qquad \Rightarrow (\textit{mod-id} \Rightarrow \mathrm{Sig}(\mathcal{E}, \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m)), \mathcal{E}, \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m\end{array}}$$

The evaluation of a module declaration returns a basis composed from the environment of all imported and locally declared objects, and the binding of the module identifier to the signature of the environment.

8.2.2 Module constrained by an interface

Abstract syntax

module *mod-id* : *int-exp* [**imports** *mod-exp*(, *mod-exp*)*] **is** *m-body*

The default import clause is empty.

Static semantics

$$\frac{\begin{array}{c}\mathcal{B} \vdash \textit{int-exp}, \textit{mod-id} \Rightarrow \mathcal{C}' \\ \mathcal{B} \vdash \textit{mod-exp}_1, \textit{mod-id} \Rightarrow \mathcal{C}_1 \ \cdots \ \mathcal{B} \vdash \textit{mod-exp}_m, \textit{mod-id} \Rightarrow \mathcal{C}_m \\ \mathcal{B}, \mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m \vdash \textit{m-body}, \textit{mod-id} \Rightarrow \mathcal{C}\end{array}}{\begin{array}{l}\mathcal{B} \vdash (\textbf{module } \textit{mod-id} : \textit{ int-exp} \\ \qquad \textbf{imports } \textit{mod-exp}_1 \textbf{ , } \ldots \textbf{ , } \textit{mod-exp}_m \textbf{ is } \textit{m-body}) \Rightarrow (\textit{mod-id} \Rightarrow \mathcal{C}'')\end{array}}$$

with side condition [$\mathcal{C}, \mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m \supseteq \mathcal{C}'' \geq \mathcal{C}'$]

The final interface of the module is those matched by the context generated by *int-exp*. The scope of the new objects declared by *int-exp* is (by default) the module name *mod-id*.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Dynamic semantics

$$\frac{\begin{array}{c}\mathcal{D} \vdash \textit{int-exp}, \textit{mod-id} \Rightarrow I \\ \mathcal{D} \vdash \textit{mod-exp}_1, \textit{mod-id} \Rightarrow \mathcal{E}_1 \ \cdots \ \mathcal{D} \vdash \textit{mod-exp}_m, \textit{mod-id} \Rightarrow \mathcal{E}_m \\ \mathcal{D} \odot \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m \vdash \textit{m-body}, \textit{mod-id} \Rightarrow \mathcal{E}\end{array}}{\begin{array}{l}\mathcal{D} \vdash (\textbf{module}\ \textit{mod-id} : \textit{int-exp}\ \textbf{imports}\ \textit{mod-exp}_1, \ldots, \textit{mod-exp}_m\ \textbf{is}\ \textit{m-body}) \\ \quad \Rightarrow (\textit{mod-id} \Rightarrow I'), \mathcal{E}, \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m\end{array}}$$

with side condition $[\mathrm{Sig}(\mathcal{E}, \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m) \supseteq I' \geq I]$

The final signature of the module is those matched by the signature generated by *int-exp*.

8.2.3 Generic module not constrained by an interface

Abstract syntax

generic *gen-id* '(' *MP* ')' [**imports** *mod-exp*(, *mod-exp*)*] **is** *m-body*

The default import clause is empty.

Static semantics

$$\frac{\begin{array}{c}\mathcal{B} \vdash MP \Rightarrow \vec{\textit{mod-id}} \Rightarrow \vec{\mathcal{C}} \\ \mathcal{B}, \vec{\mathcal{C}} \vdash \textit{mod-exp}_1, \textit{gen-id} \Rightarrow \mathcal{C}_1 \ \cdots \ \mathcal{B}, \vec{\mathcal{C}} \vdash \textit{mod-exp}_m, \textit{gen-id} \Rightarrow \mathcal{C}_m \\ \mathcal{B}, \vec{\mathcal{C}} \odot \mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m \vdash \textit{m-body}, \textit{gen-id} \Rightarrow \mathcal{C}\end{array}}{\begin{array}{l}\mathcal{B} \vdash (\textbf{generic}\ \textit{gen-id}\ (MP) \\ \qquad \textbf{imports}\ \textit{mod-exp}_1, \ldots, \textit{mod-exp}_m \\ \qquad \textbf{is}\ \textit{m-body}) \Rightarrow (\textit{gen-id} \Rightarrow (\vec{\textit{mod-id}} \Rightarrow \vec{\mathcal{C}}) \rightarrow (\mathcal{C}, \vec{\mathcal{C}} \odot \mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m))\end{array}}$$

with side condition [$\mathcal{C}$ and $\vec{\mathcal{C}} \odot \mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m$ have disjoint fields]

The generic module parameters gives a list of bindings (formal) module identifiers, declared objects. Informally, the objects of $\vec{\mathcal{C}}$ are the generic (or formal) objects of the module. These generic objects may be used in the imported module expressions in order to allow partial instantiation of generic modules and/or generic modules parameterized by generic modules . The module body is checked in the context of generic and imported objects. The context declared by the module body must be disjoint from those of parameters and of importation objects. The target context (also called the complete specification) is formed by the objects defined by the module body, the parameters, and the imported objects.

Dynamic semantics

$$\frac{\begin{array}{c}\mathcal{D} \vdash MP \Rightarrow (\vec{\textit{mod-id}} \Rightarrow \vec{I}) \\ \mathcal{D}, \vec{\mathcal{E}} \vdash \textit{mod-exp}_1, \textit{gen-id} \Rightarrow \mathcal{E}_1 \ \cdots \ \mathcal{D}, \vec{\mathcal{E}} \vdash \textit{mod-exp}_m, \textit{gen-id} \Rightarrow \mathcal{E}_m \\ \mathcal{D} \odot \vec{\mathcal{E}} \odot \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m \vdash \textit{m-body}, \textit{gen-id} \Rightarrow \mathcal{E}\end{array}}{\begin{array}{l}\mathcal{D} \vdash (\textbf{generic}\ \textit{gen-id}\ (MP)\ \textbf{imports}\ \textit{mod-exp}_1, \ldots, \textit{mod-exp}_m\ \textbf{is}\ \textit{m-body}) \\ \quad \Rightarrow (\textit{gen-id} \Rightarrow (\vec{\textit{mod-id}} \Rightarrow \vec{I}) \rightarrow (\mathcal{E}, \vec{\mathcal{E}} \odot \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m))\end{array}}$$

with side condition [$\mathcal{E}$ and $\vec{\mathcal{E}} \odot \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m$ are disjoint]

No environment is elaborated from a generic module declaration. That means that only complete instantiated objects will be included by the basis $\mathcal{D}$ at the end of evaluation. The differences with the static rule is the use of the "$\odot$" operator instead of the "," operator. The reason is that now, the basis $\mathcal{D}$ contains environments bindings which may be the same with the imported environments.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.2.4 Generic module constrained by an interface

Abstract syntax

generic *gen-id* '(' *MP* ')' : *int-exp* [**imports** *mod-exp*(, *mod-exp*)*] **is** *m-body*

The default import clause is empty.

Static semantics

$$\frac{\begin{array}{c}\mathcal{B} \vdash \textit{int-exp} \Rightarrow \mathcal{C}' \\ \mathcal{B} \vdash \textit{MP} \Rightarrow \vec{\textit{mod-id}} \Rightarrow \vec{\mathcal{C}} \\ \mathcal{B}, \vec{\mathcal{C}} \vdash \textit{mod-exp}_1, \textit{gen-id} \Rightarrow \mathcal{C}_1 \ \cdots \ \mathcal{B}, \vec{\mathcal{C}} \vdash \textit{mod-exp}_m, \textit{gen-id} \Rightarrow \mathcal{C}_m \\ \mathcal{B}, \vec{\mathcal{C}} \odot \mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m \vdash \textit{m-body} \Rightarrow \mathcal{C}\end{array}}{\begin{array}{l}\mathcal{B} \vdash (\textbf{generic } \textit{gen-id } (\textit{MP}) : \textit{int-exp} \\ \quad \textbf{imports } \textit{mod-exp}_1, \ldots, \textit{mod-exp}_m \\ \quad \textbf{is } \textit{m-body}) \Rightarrow (\textit{gen-id} \Rightarrow (\vec{\textit{mod-id}} \Rightarrow \vec{\mathcal{C}}) \rightarrow \mathcal{C}'')\end{array}}$$

with side conditions [$\mathcal{C}$ and $\vec{\mathcal{C}} \odot \mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m$ are disjoint] and [$\mathcal{C}, \vec{\mathcal{C}} \odot \mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m \supseteq \mathcal{C}'' \geq \mathcal{C}'$]

The final interface of the module (including the formal parameters) is $\mathcal{C}''$ which instantiate the interface given by *int-exp*, and shall match the complete specification of the generic module.

Dynamic semantics

$$\frac{\begin{array}{c}\mathcal{D} \vdash \textit{int-exp}, \textit{gen-id} \Rightarrow I \\ \mathcal{D} \vdash \textit{MP} \Rightarrow (\vec{\textit{mod-id}} \Rightarrow \vec{I}) \\ \mathcal{D}, \vec{\mathcal{E}} \vdash \textit{mod-exp}_1, \textit{gen-id} \Rightarrow \mathcal{E}_1 \ \cdots \ \mathcal{D}, \vec{\mathcal{E}} \vdash \textit{mod-exp}_m, \textit{gen-id} \Rightarrow \mathcal{E}_m \\ \mathcal{D} \odot \vec{\mathcal{E}} \odot \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m \vdash \textit{m-body}, \textit{gen-id} \Rightarrow \mathcal{E}\end{array}}{\begin{array}{l}\mathcal{D} \vdash (\textbf{generic } \textit{gen-id } (\textit{MP}) : \textit{int-exp } \textbf{imports } \textit{mod-exp}_1, \ldots, \textit{mod-exp}_m \textbf{ is } \textit{m-body}) \\ \quad \Rightarrow (\textit{gen-id} \Rightarrow (\vec{\textit{mod-id}} \Rightarrow \vec{I}) \rightarrow (\mathcal{E}, \vec{\mathcal{E}} \odot \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m) \downarrow I')\end{array}}$$

with side conditions [$\mathcal{E}$ and $\vec{\mathcal{E}} \odot \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m$ are disjoint] and [$\mathrm{Sig}(\mathcal{E}, \vec{\mathcal{E}} \odot \mathcal{E}_1 \odot \ldots \odot \mathcal{E}_m) \supseteq I' \geq I$]

8.2.5 Interface declaration

Abstract syntax

interface *int-id* [**imports** *int-exp*(, *int-exp*)*] **is** *int-exp*

The default import clause is empty.

Static semantics

$$\frac{\begin{array}{c}\mathcal{B} \vdash \textit{int-exp}_1, \textit{int-id} \Rightarrow \mathcal{C}_1 \ \cdots \ \mathcal{B} \vdash \textit{int-exp}_m, \textit{int-id} \Rightarrow \mathcal{C}_m \\ \mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m \vdash \textit{i-body}, \textit{int-id} \Rightarrow \mathcal{C}\end{array}}{\mathcal{B} \vdash (\textbf{interface } \textit{int-id } \textbf{imports } \textit{int-exp}_1, \ldots, \textit{int-exp}_m \textbf{ is } \textit{i-body}) \Rightarrow (\textit{int-id} \Rightarrow \mathcal{C}, \mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m)}$$

with side condition [$\mathcal{C}$ and $\mathcal{C}_1 \odot \ldots \odot \mathcal{C}_m$ have disjoint fields]

The interface body is checked in the context of objects imported by the importation clause. The objects imported may satisfy the "diamond" rule, but the objects declared by the interface body shall be disjoint from the imported ones.

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Dynamic semantics

$$\frac{\mathcal{D} \vdash \textit{int-exp}_1, \textit{int-id} \Rightarrow I_1 \ \cdots \ \mathcal{D} \vdash \textit{int-exp}_m, \textit{int-id} \Rightarrow I_m \quad I_1 \odot \ldots \odot I_m \vdash \textit{i-body}, \textit{int-id} \Rightarrow I}{\mathcal{D} \vdash (\textbf{interface}\ \textit{int-id}\ \textbf{imports}\ \textit{int-exp}_1, \ldots, \textit{int-exp}_m\ \textbf{is}\ \textit{i-body}) \Rightarrow (\textit{int-id} \Rightarrow I, I_1 \odot \ldots \odot I_m)}$$

with side condition [I and $I_1 \odot \ldots \odot I_m$ are disjoint]

8.2.6 Sequential top declaration

Abstract syntax

top-dec *top-dec*

Static semantics

$$\frac{\mathcal{B} \vdash \textit{top-dec}_1 \Rightarrow \mathcal{B}' \quad \mathcal{B}, \mathcal{B}' \vdash \textit{top-dec}_2 \Rightarrow \mathcal{B}''}{\mathcal{B} \vdash \textit{top-dec}_1\ \textit{top-dec}_2 \Rightarrow \mathcal{B}', \mathcal{B}''}$$

The elaboration of the sequencing phrase works on a description ordered by the "dependency" relation . This ordered specification may be obtained in a first step of static analysis. The order is well founded (non-cyclic) according to cycle-freedom hypothesis. The module bindings generated by each top level declaration shall be disjoint, i.e. each module, generic module, or interface name shall be bound once.

Dynamic semantics

$$\frac{\mathcal{D} \vdash \textit{top-dec}_1 \Rightarrow \mathcal{D}_1 \quad \mathcal{D} \odot \mathcal{D}_1 \vdash \textit{top-dec}_2 \Rightarrow \mathcal{D}_2}{\mathcal{D} \vdash \textit{top-dec}_1\, \textit{top-dec}_2 \Rightarrow \mathcal{D}_1 \odot \mathcal{D}_2}$$

The evaluation of the sequencing phrase takes advantage from the static semantics check: the application of the "$\odot$" operator is safe because in the static semantics is checks the double declaration of a module, interface, or generic module identifier. We need to apply it in order to consider multiple importations of dynamic objects.

8.3 Module body

The module body defines the content of the module. It can be an explicit (generative) declaration, or an instantiation of a module expression.

8.3.1 Block declaration

Abstract syntax

D

Static semantics

$$\frac{\mathcal{C} \vdash D, id \Rightarrow \mathcal{C}'}{\mathcal{B}, \mathcal{C} \vdash D, id \Rightarrow \mathcal{C}'}$$

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Dynamic semantics

$$\frac{\mathcal{E} \vdash D, id \Rightarrow \mathcal{E}'}{\mathcal{D}, \mathcal{E} \vdash D, id \Rightarrow \mathcal{E}'}$$

8.3.2 Module Expression

Abstract syntax

mod-exp

Static semantics

$$\frac{\mathcal{B}, \mathcal{C} \vdash \textit{mod-exp}, id \Rightarrow \mathcal{C}'}{\mathcal{B}, \mathcal{C} \vdash \textit{mod-exp}, id \Rightarrow \mathcal{C}'}$$

The resulting context is given by the module expression check.

Dynamic semantics

$$\frac{\mathcal{D} \vdash \textit{mod-exp}, id \Rightarrow \mathcal{E}}{\mathcal{D} \vdash \textit{mod-exp}, id \Rightarrow \mathcal{E}}$$

8.4 Module expression

Module expressions can be an aliasing to an (already declared) module or an instantiation of a given generic module, probably renaming some identifiers and constraining to a given interface.

8.4.1 Module aliasing not constrained by an interface

Abstract syntax

mod-id [**renaming** '(' *reninst* ')']

The default renaming is '()' .

Static semantics

$$\frac{\mathcal{B} \vdash \textit{mod-id} \Rightarrow \mathcal{C}' \quad \mathcal{C}' \vdash \textit{reninst}, id \Rightarrow g}{\mathcal{B}, \mathcal{C} \vdash \textit{mod-id}\ \textbf{renaming}\ \text{'('}\ \textit{reninst}\ \text{')'}, id \Rightarrow g(\mathcal{C}')}$$

The objects declared by *mod-id* give the binding of the module expression. The aliasing does not change the source of objects declared in the module.

The morphism generated by the renaming is formed (see *reninst* rules for renaming). The resulting binding is renamed according to the renaming morphism g. Note that the objects which are modified by the renaming, change their source to *id*.

Dynamic semantics

$$\frac{\mathcal{D} \vdash \textit{mod-id} \Rightarrow I \qquad \mathcal{D} \downarrow I \vdash \textit{reninst}, id \Rightarrow g}{\mathcal{D} \vdash \textit{mod-id}\ \textbf{renaming}\ \text{'('}\ \textit{reninst}\ \text{')'}, id \Rightarrow g(\mathcal{D} \downarrow I)}$$

The module shall be already declared in the basis. By consequence, all its objects are in $\mathcal{D}$. The environment is obtained by the projection of the basis on the module signature.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.4.2 Module aliasing constrained by an interface

Abstract syntax

mod-id : *int-exp* [**renaming** '(' *reninst* ')']

The default renaming is '()' .

Static semantics

$$\frac{\begin{array}{l}\mathcal{B} \vdash \textit{mod-id} \Rightarrow \mathcal{C}_1 \\ \mathcal{B} \vdash \textit{int-exp}, \textit{id} \Rightarrow \mathcal{C}_2 \\ \mathcal{C}_3 \vdash \textit{reninst}, \textit{id} \Rightarrow g\end{array}}{\mathcal{B}, \mathcal{C} \vdash \textit{mod-id} : \textit{int-exp}\ \textbf{renaming}\ \text{'('}\ \textit{reninst}\ \text{')'}, \textit{id} \Rightarrow g(\mathcal{C}_3)}\ [\mathcal{C}_1 \supseteq \mathcal{C}_3 \geq \mathcal{C}_2]$$

The context elaborated by the module expression shall match the interface expression context. The new objects declared by the module expression or by the interface expression have as source name the identifier *id*.

Dynamic semantics

$$\frac{\begin{array}{l}\mathcal{D} \vdash \textit{mod-id} \Rightarrow I \\ \mathcal{D} \vdash \textit{int-exp}, \textit{id} \Rightarrow I' \quad (\mathcal{D} \downarrow I) \downarrow I'' \vdash \textit{reninst}, \textit{id} \Rightarrow g\end{array}}{\mathcal{D} \vdash \textit{mod-id} : \textit{int-exp}\ \textbf{renaming}\ \text{'('}\ \textit{reninst}\ \text{')'}, \textit{id} \Rightarrow g((\mathcal{D} \downarrow I) \downarrow I'')}\ [Sig(\mathcal{D} \downarrow I) \supseteq I'' \geq I']$$

The module expression *mod-id* is evaluated to an environment which signature shall contain a signature I'' matching the signature resulting from the evaluation of interface expression *int-exp*. This constraint overlaps the static semantics constraint. The resulting environment is obtained by projection of the *mod-id* environment on the signature I''.

8.4.3 Generic module actualization not constrained by an interface

Abstract syntax

gen-id '(' *RME* ')' [**renaming** '(' *reninst* ')']

The default renaming is '()' .

Static semantics

$$\frac{\begin{array}{c}\mathcal{B} \vdash \textit{gen-id} \Rightarrow (\vec{\textit{mod-id}} \Rightarrow \vec{\mathcal{C}}) \rightarrow \vec{\mathcal{C}}, \mathcal{C}' \\ \mathcal{B} \vdash (\textit{RME} \Rightarrow (\vec{\textit{mod-id}} \Rightarrow \vec{\mathcal{C}})), \textit{id} \Rightarrow \phi \\ \phi(\vec{\mathcal{C}}, \mathcal{C}'[\textit{id}/\textit{gen-id}]) \vdash \textit{reninst}, \textit{id} \Rightarrow g\end{array}}{\mathcal{B}, \mathcal{C} \vdash \textit{gen-id}\ \text{'('}\ \textit{RME}\ \text{')'}\ \textbf{renaming}\ \text{'('}\ \textit{reninst}\ \text{')'}, \textit{id} \Rightarrow g(\phi(\vec{\mathcal{C}}, \mathcal{C}'[\textit{id}/\textit{gen-id}]))}$$

The second premise checks that the actual module parameters are compatible with the formal ones, and returns the context of the actual parameters. We need the formal parameters names ($\vec{\textit{mod-id}}$) in order to handle (statically) the tuples of actual parameters. The realization ϕ gives the correspondence between formal module names and actual module names. This morphism is used to actualize all the formal parameters used with the actual ones, and also to modify the occurrences of the formal parameters used by the objects defined in the generic module. The morphism is only a source names substitution because the objects declared by the actual modules must have the same name as the objects declared by the formal modules (see *RME* rules). At the realization ϕ is added the substitution *id*/*gen-id* in order to create new objects (the actualized ones) from the objects defined by the generic module.

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Dynamic semantics

$$\frac{\begin{array}{l}\mathcal{D} \vdash \textit{gen-id} \Rightarrow (\vec{\textit{mod}}\textit{-id} \Rightarrow \vec{I}) \rightarrow \mathcal{E} \\ \mathcal{D} \vdash (\textit{RME} \Rightarrow (\vec{\textit{mod}}\textit{-id} \Rightarrow \vec{I})) \Rightarrow \phi \\ \phi(\mathcal{E}[\textit{id}/\textit{gen-id}]) \vdash \textit{reninst}, \textit{id} \Rightarrow g\end{array}}{\mathcal{D} \vdash \textit{gen-id}\ \texttt{'('}\ \textit{RME}\ \texttt{')'}\ \textbf{renaming}\ \texttt{'('}\ \textit{reninst}\ \texttt{')'}, \textit{id} \Rightarrow g(\phi(\mathcal{E}[\textit{id}/\textit{gen-id}]))}$$

8.4.4 Generic module actualization constrained by an interface

Abstract syntax

gen-id `'('` *RME* `')'` `:` *int-exp* [**renaming** `'('` *reninst* `')'`]

The default renaming is `'()'` .

Static semantics

$$\frac{\begin{array}{l}\mathcal{B} \vdash \textit{gen-id} \Rightarrow (\vec{\textit{mod}}\textit{-id} \Rightarrow \vec{\mathcal{C}}) \rightarrow \vec{\mathcal{C}}, \mathcal{C}_1 \\ \mathcal{B} \vdash (\textit{RME} \Rightarrow (\vec{\textit{mod}}\textit{-id} \Rightarrow \vec{\mathcal{C}})) \Rightarrow \phi \\ \mathcal{B} \vdash \textit{int-exp}, \textit{id} \Rightarrow \mathcal{C}_2 \\ \mathcal{C}_3 \vdash \textit{reninst}, \textit{id} \Rightarrow g\end{array}}{\mathcal{B}, \mathcal{C} \vdash \textit{gen-id}\ \texttt{'('}\ \textit{RME}\ \texttt{')'}\ \texttt{:}\ \textit{int-exp}\ \textbf{renaming}\ \texttt{'('}\ \textit{reninst}\ \texttt{')'}, \textit{id} \Rightarrow g(\mathcal{C}_3)}$$

with side condition $[\phi(\vec{\mathcal{C}}, \mathcal{C}_1[\textit{id}/\textit{gen-id}]) \supseteq \mathcal{C}_3 \geq \mathcal{C}_2]$

Dynamic semantics

$$\frac{\begin{array}{l}\mathcal{D} \vdash \textit{gen-id} \Rightarrow (\vec{\textit{mod}}\textit{-id} \Rightarrow \vec{I}) \rightarrow \mathcal{E} \\ \mathcal{D} \vdash (\textit{RME} \Rightarrow (\vec{\textit{mod}}\textit{-id} \Rightarrow \vec{I})) \Rightarrow \phi \\ \mathcal{D} \vdash \textit{int-exp}, \textit{id} \Rightarrow I \\ \phi(\mathcal{E}[\textit{id}/\textit{gen-id}] \downarrow I') \vdash \textit{reninst}, \textit{id} \Rightarrow g\end{array}}{\mathcal{D} \vdash \textit{gen-id}\ \texttt{'('}\ \textit{RME}\ \texttt{')'}\ \texttt{:}\ \textit{int-exp}\ \textbf{renaming}\ \texttt{'('}\ \textit{reninst}\ \texttt{')'}, \textit{id} \Rightarrow g(\phi(\mathcal{E}[\textit{id}/\textit{gen-id}] \downarrow I'))}$$

with side condition $[\textit{Sig}(\mathcal{E}[\textit{id}/\textit{gen-id}]) \supseteq I' \geq I]$

8.4.5 Generic module renaming/instantiation

Abstract syntax

gen-id `'('` *reninst* `')'`

The default renaming is `'()'` .

Static semantics

$$\frac{\begin{array}{l}\mathcal{B} \vdash \textit{gen-id} \Rightarrow (\vec{\textit{mod}}\textit{-id} \Rightarrow \vec{\mathcal{C}}) \rightarrow \vec{\mathcal{C}}, \mathcal{C}' \\ \mathcal{C} \vdash \textit{reninst}/\vec{\mathcal{C}} \Rightarrow g\end{array}}{\mathcal{B}, \mathcal{C} \vdash \textit{gen-id}\ \texttt{'('}\ \textit{reninst}\ \texttt{')'}, \textit{id} \Rightarrow g(\vec{\mathcal{C}}) \odot g(\mathcal{C}'[\textit{id}/\textit{gen-id}])}$$

This rule deals with the elaboration of explicit actualization of a generic module. The context $\mathcal{C}$ shall contain the declaration of all actual parameters. The correspondence between formal and actual parameters is given by the list *reninst*, which plays here the role of an actualization. For this reason the form of the inference rule is changed. The morphism g must be total (see *reninst* rules for actualization), and is used to actualize the formal parameters ($g(\vec{\mathcal{C}})$), and to actualize the objects using these formal parameters ($g(\mathcal{C}'[\textit{id}/\textit{gen-id}])$), which are defined in the *gen-id* module. These objects change their source module to *id* (are new objects).

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Dynamic semantics

$$\frac{\mathcal{D} \vdash \mathit{gen\text{-}id} \Rightarrow (\vec{\mathit{mod}}\text{-}\mathit{id} \Rightarrow \vec{I}) \rightarrow \mathcal{E}_0, \mathcal{E}_1 \quad \mathcal{E} \vdash \mathit{ren\text{-}list}/\vec{I} \Rightarrow g}{\mathcal{D}, \mathcal{E} \vdash (\mathit{gen\text{-}id}\,(\mathit{ren\text{-}list})), \mathit{id} \Rightarrow (\mathcal{E} \downarrow g(\vec{I})) \odot g(\mathcal{E}_1[\mathit{id}/\mathit{gen\text{-}id}])} \left[\mathrm{Sig}(\mathcal{E}_0) = \vec{I}\right]$$

It is similar to the static semantics rule, where instead of formal parameter context is used the signature of the parameters, and for the environment of these formal parameters is used the projection of the generic module environment on the interface of parameters.

8.5 Module formal parameters

The formal parameters of a generic module is a list of (formal) module identifier, module interface. It bound for each module identifier the context declared by its interface.

8.5.1 Single

Abstract syntax

mod-id : *int-exp*

Static semantics

$$\frac{\mathcal{B} \vdash \mathit{int\text{-}exp}, \mathit{mod\text{-}id} \Rightarrow \mathcal{C}}{\mathcal{B} \vdash \mathit{mod\text{-}id}\ :\ \mathit{int\text{-}exp} \Rightarrow (\mathit{mod\text{-}id} \Rightarrow \mathcal{C})}$$

The elaboration of the interface expressions typing the parameters is made using the formal module name as scope name for the generative declarations.

Dynamic semantics

$$\frac{\mathcal{D} \vdash \mathit{int\text{-}exp}, \mathit{mod\text{-}id} \Rightarrow I}{\mathcal{D} \vdash \mathit{mod\text{-}id}\ :\ \mathit{int\text{-}exp} \Rightarrow (\mathit{mod\text{-}id} \Rightarrow I)}$$

8.5.2 Disjoint union

Abstract syntax

MP , MP

Static semantics

$$\frac{\mathcal{B} \vdash MP_1 \Rightarrow \mathcal{C} \quad \mathcal{B} \vdash MP_2 \Rightarrow \mathcal{C}'}{\mathcal{B} \vdash MP_1\ ,\ MP_2 \Rightarrow \mathcal{C}, \mathcal{C}'}\ [\mathcal{C} \odot \mathcal{C}' \text{ is well defined}]$$

The side condition asserts that the bindings of formal modules shall matches in the shared objects (i.e., objects having the same names should be bound to the same profile). In this way, the objects imported by two different interface expression may be shared like in a "diamond" importation.

Dynamic semantics

$$\frac{\mathcal{D} \vdash MP_1 \Rightarrow \mathcal{D}_1 \quad \mathcal{D} \vdash MP_2 \Rightarrow \mathcal{D}_2}{\mathcal{D} \vdash MP_1\ ,\ MP_2 \Rightarrow \mathcal{D}_1, \mathcal{D}_2}$$

The side condition of the static rule is not necessary because signatures does not stock identifier' bindings.

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

8.6 Interface expressions

An interface expression is a list of object interfaces. It can be obtained either by alising to an (already declared) interface, or by renaming of an interface, or by an explicit (generative) declaration (possibly renamed).

8.6.1 Interface identifier

Abstract syntax

int-id

Static semantics

$$\frac{\mathcal{B} \vdash \textit{int-id} \Rightarrow \mathcal{C}}{\mathcal{B} \vdash \textit{int-id}, \textit{id} \Rightarrow \mathcal{C}}$$

The context generated is the context of the interface identifier.

Dynamic semantics

$$\frac{\mathcal{D} \vdash \textit{int-id} \Rightarrow I}{\mathcal{D} \vdash \textit{int-id}, \textit{id} \Rightarrow I}$$

8.6.2 Simple renaming

Abstract syntax

'[' *int-id* **renaming** '(' *reninst* ')' ']'

Static semantics

$$\frac{\mathcal{B} \vdash \textit{int-id} \Rightarrow \mathcal{C} \quad \mathcal{C} \vdash \textit{reninst}, \textit{id} \Rightarrow g}{\mathcal{B} \vdash \text{'['}\ \textit{int-id}\ \textbf{renaming}\ \text{'('}\ \textit{reninst}\ \text{')'}\ \text{']'}, \textit{id} \Rightarrow g(\mathcal{C})}$$

The context of the interface *int-id* is renaming using the renaming morphism *g* generated from *reninst* (see *reninst* rules for renaming). The source of the renamed objects becomes *id*.

Dynamic semantics

$$\frac{\mathcal{D} \vdash \textit{int-id} \Rightarrow I \quad I \vdash \textit{reninst}, \textit{id} \Rightarrow g}{\mathcal{D} \vdash \text{'['}\ \textit{int-id}\ \textbf{renaming}\ \text{'('}\ \textit{reninst}\ \text{')'}\ \text{']'}, \textit{id} \Rightarrow g(I)}$$

8.6.3 Explicit body

Abstract syntax

'[' *i-body* [**renaming** '(' *reninst* ')'] ']'

The default renaming is '()' .

Static semantics

$$\frac{\vdash \textit{i-body}, \textit{id} \Rightarrow \mathcal{C} \quad \mathcal{C} \vdash \textit{reninst}, \textit{id} \Rightarrow g}{\mathcal{B} \vdash \text{'['}\ \textit{i-body}\ \textbf{renaming}\ \text{'('}\ \textit{reninst}\ \text{')'}\ \text{']'}, \textit{id} \Rightarrow g(\mathcal{C})}$$

The declarations of *i-body* must be self contained (in the initial basis objects). The source of these objects is *id*. These object are renamed according to the renaming morphism *g* generated from *reninst*.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Dynamic semantics

$$\frac{\vdash \textit{i-body}, id \Rightarrow I \quad I \vdash \textit{reninst}, id \Rightarrow g}{\mathcal{D} \vdash \text{'['}\ \textit{i-body}\ \textbf{renaming}\ \text{'('}\ \textit{reninst}\ \text{')'}\ \text{']'}, id \Rightarrow g(I)}$$

8.7 Interface body

An interface body is a list of object interfaces. The interfaces of processes are always opaque. The interfaces for types are either opaque or completely defined.

8.7.1 Type hiding the implementation

Abstract syntax

type S

Static semantics

$$\overline{\mathcal{C} \vdash \textbf{type}\ S, id \Rightarrow (< id, S > \Rightarrow \textbf{type})}$$

Dynamic semantics

$$\overline{\mathcal{D} \vdash \textbf{type}\ S, id \Rightarrow < id, S >}$$

8.7.2 Type synonym

Abstract syntax

type S **renames** T

Static semantics

$$\frac{\mathcal{C} \vdash T \Rightarrow \textbf{type}}{\mathcal{C} \vdash (\textbf{type}\ S\ \textbf{renames}\ T), id \Rightarrow (< id, S > \Rightarrow \textbf{type}, S \equiv T)}$$

Dynamic semantics

$$\overline{\mathcal{D} \vdash (\textbf{type}\ S\ \textbf{renames}\ T), id \Rightarrow < id, S >}$$

8.7.3 Constructed type

Abstract syntax

type S **is** $C[\ \text{'('}\ RT\ \text{')'}\ ]\ (\text{'|'}\ C[\ \text{'('}\ RT\ \text{')'}\ ])^*$

The default constructor argument type is `()`.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Static semantics

$$\frac{\mathcal{C},<id,S>\Rightarrow\textbf{type}\vdash(RT_1)\Rightarrow\textbf{type}\ \cdots\ \mathcal{C},<id,S>\Rightarrow\textbf{type}\vdash(RT_n)\Rightarrow\textbf{type}}{\begin{array}{l}\mathcal{C}\vdash(\textbf{type}\ S\ \textbf{is}\ C_1(RT_1)\ \mid\ \cdots\ \mid\ C_n(RT_n)),id\Rightarrow\\ \qquad(<id,S>\Rightarrow\textbf{type},<id,C_1>\Rightarrow(RT_1)\rightarrow S,\ldots,<id,C_n>\Rightarrow(RT_n)\rightarrow S)\end{array}}$$

Note that the $<id,S>\Rightarrow\textbf{type}$ is assumed in the context to allow recursive instantiation in B.

Dynamic semantics

$$\frac{}{\mathcal{D}\vdash(\textbf{type}\ S\ \textbf{is}\ C_1(RT_1)\ \mid\ \cdots\ \mid\ C_n(RT_n)),id\Rightarrow<id,S>,<id,C_1>,\ldots,<id,C_n>}$$

8.7.4 Named record type

Abstract syntax

type S **is** '(' RT ')'

Static semantics

$$\frac{\mathcal{C}\vdash(RT)\Rightarrow\textbf{type}}{\mathcal{C}\vdash(\textbf{type}\ S\ \textbf{is}\ \text{'('}\ RT\ \text{')'}),id\Rightarrow(<id,S>\Rightarrow\textbf{type},S\equiv(RT))}$$

Dynamic semantics

$$\frac{}{\mathcal{D}\vdash(\textbf{type}\ S\ \textbf{is}\ \text{'('}\ RT\ \text{')'}),id\Rightarrow<id,S>}$$

8.7.5 Process declaration

Abstract syntax

process Π ['[' $[G[:T](,G[:T])^*]$ ']']
['(' $[V[:T](,V[:T])^*]$ ')']
[: T]
[**raises** '[' $X[:T](,X[:T])^*$ ']']

The default gate list is `[]`, the default gate type is **any**, the default in parameter list is `()`, the default result type is **none**, the default exception list is `[]` and the default exception type is `()`.

Static semantics

$$\frac{\begin{array}{l}\mathcal{C}\vdash T_1\Rightarrow\textbf{type}\ \cdots\ \mathcal{C}\vdash T_m\Rightarrow\textbf{type}\\ \mathcal{C}\vdash T_1'\Rightarrow\textbf{type}\ \cdots\ \mathcal{C}\vdash T_p'\Rightarrow\textbf{type}\\ \mathcal{C}\vdash T_1''\Rightarrow\textbf{type}\ \cdots\ \mathcal{C}\vdash T_n''\Rightarrow\textbf{type}\\ \mathcal{C}\vdash T_e\Rightarrow\textbf{type}\end{array}}{\begin{array}{l}\mathcal{C}\vdash(\textbf{process}\ \Pi\ [G_1:T_1,\ldots,G_m:T_m]\ (V_1:T_1',\ldots,V_p:T_p')\\ \qquad:T_e\ \textbf{raises}\ [X_1:T_1'',\ldots,X_n:T_n'']),id\\ \qquad\Rightarrow(<id,\Pi>\Rightarrow[\textbf{gate}(T_1),\ldots,\textbf{gate}(T_m)]\,(V_1\Rightarrow T_1',\ldots,V_p\Rightarrow T_p')\\ \qquad\quad[\textbf{exn}(T_1''),\ldots,\textbf{exn}(T_n'')]\rightarrow\textbf{exit}(T_e))\end{array}}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Dynamic semantics

$$\frac{}{\mathcal{D} \vdash (\textbf{process}\ \Pi\ [G_1 : T_1, \ldots, G_m : T_m]\ (V_1 \Rightarrow T_1', \ldots, V_p \Rightarrow T_p') : T_e\ \textbf{raises}\ [X_1 : T_1'', \ldots, X_n : T_n'']), id \Rightarrow < id, \Pi >}$$

8.7.6 Equations

Abstract syntax

eqns *eqn-dec-list*

Static semantics

$$\frac{\mathcal{C} \vdash \textit{eqn-dec-list} \Rightarrow \texttt{ok}}{\mathcal{C} \vdash \textbf{eqns}\ \textit{eqn-dec-list} \Rightarrow ()}$$

8.7.7 Sequential declaration

Abstract syntax

i-body i-body

Static Semantics

$$\frac{\mathcal{C}, \mathcal{C}'' \vdash \textit{i-body}_1, id \Rightarrow \mathcal{C}' \quad \mathcal{C}, \mathcal{C}' \vdash \textit{i-body}_2, id \Rightarrow \mathcal{C}''}{\mathcal{C} \vdash (\textit{i-body}_1\ \textit{i-body}_2), id \Rightarrow \mathcal{C}', \mathcal{C}''}$$

Dynamic Semantics

$$\frac{\mathcal{D} \vdash \textit{i-body}_1, id \Rightarrow I \quad \mathcal{D}, I \vdash \textit{i-body}_2, id \Rightarrow I'}{\mathcal{D} \vdash (\textit{i-body}_1\ \textit{i-body}_2), id \Rightarrow I, I'}$$

8.8 Record module expression

The record module expression is the actual parameters of the generic module instantiation.

8.8.1 Single

Abstract syntax

mod-id => mod-exp

Static semantics

$$\frac{\mathcal{B} \vdash \textit{mod-exp}, id \Rightarrow \mathcal{C}'}{\mathcal{B} \vdash ((\textit{mod-id}\ \texttt{=>}\ \textit{mod-exp}) \Rightarrow (\textit{mod-id} \Rightarrow \mathcal{C})), id \Rightarrow \phi}\ [\mathcal{C}' = \phi(\mathcal{C})]$$

It shall match the same module identifier *mod-id*. The new objects declared by the module expression *mod-exp* has as source the identifier *id*. The context given by the elaboration of the module expression shall be an instance of the context $\mathcal{C}$. The realization of this instance relation is given by ϕ.

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Dynamic semantics

$$\frac{\mathcal{D} \vdash \textit{mod-exp}, \textit{id} \Rightarrow \mathcal{E}}{\mathcal{B} \vdash ((\textit{mod-id} \texttt{ => } \textit{mod-exp}) \Rightarrow (\textit{mod-id} \Rightarrow I)), \textit{id} \Rightarrow \phi} [Sig(\mathcal{E}) = \phi(I)]$$

8.8.2 Disjoint union

Abstract syntax

RME , *RME*

Static semantics

$$\frac{\mathcal{B} \vdash (RME_1 \Rightarrow \vec{\textit{mod-id}}_1 \Rightarrow \vec{C}), \textit{id} \Rightarrow \phi \quad \mathcal{B} \vdash (RME_2 \Rightarrow \vec{\textit{mod-id}}_2 \Rightarrow \vec{C}'), \textit{id} \Rightarrow \phi'}{\mathcal{B} \vdash (RME_1, RME_2 \Rightarrow (\vec{\textit{mod-id}}_1 \Rightarrow \vec{C}, \vec{\textit{mod-id}}_2 \Rightarrow \vec{C}')), \textit{id} \Rightarrow \phi, \phi'}$$

Each part of the union shall match a different list of formal parameters. The morphisms resulting from each elaboration are disjoint because the module names of the formal parameters are disjoint. The resulting morphism is given by their disjoint composition.

Dynamic semantics

$$\frac{\mathcal{D} \vdash (RME_1 \Rightarrow \vec{\textit{mod-id}}_1 \Rightarrow \vec{I}), \textit{id} \Rightarrow \phi \quad \mathcal{D} \vdash (RME_2 \Rightarrow \vec{\textit{mod-id}}_2 \Rightarrow \vec{I}'), \textit{id} \Rightarrow \phi'}{\mathcal{D} \vdash (RME_1, RME_2 \Rightarrow (\vec{\textit{mod-id}}_1 \Rightarrow \vec{I}, \vec{\textit{mod-id}}_2 \Rightarrow \vec{I}')), \textit{id} \Rightarrow \phi, \phi'}$$

8.8.3 Renaming tuple

Abstract syntax

reninst , *reninst*

Static semantics

$$\frac{\mathcal{C} \vdash \textit{reninst}_1 \Rightarrow \mathcal{C}' \quad \mathcal{C}' \vdash \textit{reninst}_2 \Rightarrow \mathcal{C}''}{\mathcal{C} \vdash \textit{reninst}_1 \texttt{ , } \textit{reninst}_2 \Rightarrow \mathcal{C}''}$$

8.9 Equation declarations

8.9.1 Equations declaration

Abstract syntax

[**forall** *RT*] *eqn* − *dec* (*eqn* − *dec*)*

The default value of *RT* is ().

Static semantics

$$\frac{\begin{array}{c}\mathcal{C} \vdash (RT) \Rightarrow \textbf{type} \\ \mathcal{C}, RT \vdash \textit{eqn-dec}_1 \Rightarrow \texttt{ok} \\ \cdots \\ \mathcal{C}, RT \vdash \textit{eqn-dec}_n \Rightarrow \texttt{ok}\end{array}}{\mathcal{C} \vdash \textbf{forall } RT \texttt{ -> } \textit{eqn-dec}_1 \ \ldots \ \textit{eqn-dec}_n \Rightarrow \texttt{ok}}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.9.2 Sequential

Abstract syntax

eqn-decs eqn-decs

Static semantics

$$\frac{\mathcal{C} \vdash \textit{eqn-decs}_1 \Rightarrow \texttt{ok} \quad \mathcal{C} \vdash \textit{eqn-decs}_2 \Rightarrow \texttt{ok}}{\mathcal{C} \vdash \textit{eqn-decs}_1\ \textit{eqn-decs}_2 \Rightarrow \texttt{ok}}$$

8.10 Simple equation declaration

Abstract syntax

ofsort T [**forall** RT] E

The default value of RT is ().

Static semantics

$$\frac{\begin{array}{c}\mathcal{C} \vdash T \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (RT) \Rightarrow \textbf{type} \\ \mathcal{C}, RT \vdash E \Rightarrow \textbf{exit}(\textbf{bool})\end{array}}{\mathcal{C} \vdash \textbf{ofsort}\ T\ \textbf{forall}\ RT\ \texttt{->}\ E \Rightarrow \texttt{ok}}$$

8.11 Declarations

8.11.1 Type synonym

Abstract syntax

type S **renames** T

Static semantics

$$\frac{\mathcal{C} \vdash T \Rightarrow \textbf{type}}{\mathcal{C} \vdash (\textbf{type}\ S\ \textbf{renames}\ T), id \Rightarrow (< id, S > \Rightarrow \textbf{type}, S \equiv T)}$$

Dynamic semantics

$$\frac{}{\mathcal{E} \vdash (\textbf{type}\ S\ \textbf{renames}\ T) \Rightarrow (S \equiv T)}$$

8.11.2 Type declaration

Abstract syntax

type S **is** $C[\texttt{'('}\ RT\ \texttt{')'}]\ (\texttt{'|'}\ C[\texttt{'('}\ RT\ \texttt{')'}])^*$

The default constructor argument type is `()`.

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Static semantics

$$\frac{\mathcal{C},<id,S>\Rightarrow\textbf{type}\vdash(RT_1)\Rightarrow\textbf{type}\ \cdots\ \mathcal{C},<id,S>\Rightarrow\textbf{type}\vdash(RT_n)\Rightarrow\textbf{type}}{\begin{array}{l}\mathcal{C}\vdash(\textbf{type}\ S\ \textbf{is}\ C_1(RT_1)\ \mid\ \cdots\ \mid\ C_n(RT_n)),id\Rightarrow\\ \qquad(<id,S>\Rightarrow\textbf{type},<id,C_1>\Rightarrow(RT_1)\rightarrow S,\ldots,<id,C_n>\Rightarrow(RT_n)\rightarrow S)\end{array}}$$

Note that the $<id,S>\Rightarrow\textbf{type}$ is assumed in the context to allow recursive types.

Dynamic semantics

$$\frac{}{\mathcal{E}\vdash(\textbf{type}\ S\ \textbf{is}\ C_1(RT_1)\ \mid\ \cdots\ \mid\ C_n(RT_n))\Rightarrow(C_1\Rightarrow(RT_1)\rightarrow S,\ldots,C_n\Rightarrow(RT_n)\rightarrow S)}$$

8.11.3 Named record type

Abstract syntax

type S **is** '(' RT ')'

Static semantics

$$\frac{\mathcal{C}\vdash(RT)\Rightarrow\textbf{type}}{\mathcal{C}\vdash(\textbf{type}\ S\ \textbf{is}\ \text{'('}\ RT\ \text{')'}),id\Rightarrow(<id,S>\Rightarrow\textbf{type},S\equiv(RT))}$$

Dynamic semantics

$$\frac{}{\mathcal{E}\vdash(\textbf{type}\ S\ \textbf{is}\ \text{'('}\ RT\ \text{')'})\Rightarrow(S\equiv(RT))}$$

8.11.4 Process declaration

Abstract syntax

process Π ['[' $[G[:T](,G[:T])^*]$ ']']
['(' $[V[:T](,V[:T])^*]$ ')']
[: T]
[**raises** $[X[:T](,X[:T])^*]$]
is B

The default gate list is `[]`, the default gate type is **any**, the default in parameter list is `()`, the default result type is **none**, the default exception list is `[]` and the default exception type is `()`.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Static semantics

$$\frac{\begin{array}{r}\mathcal{C} \vdash (T_1) \Rightarrow \textbf{type} \ \cdots \ \mathcal{C} \vdash (T_m) \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (T'_1) \Rightarrow \textbf{type} \ \cdots \ \mathcal{C} \vdash (T'_p) \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (T''_1) \Rightarrow \textbf{type} \ \cdots \ \mathcal{C} \vdash (T''_n) \Rightarrow \textbf{type} \\ \mathcal{C}, G_1 \Rightarrow \textbf{gate}(T_1), \ldots, G_m \Rightarrow \textbf{gate}(T_m), \\ V_1 \Rightarrow T'_1, \ldots, V_p \Rightarrow T'_p, \\ X_1 \Rightarrow \textbf{exn}(T''_1), \ldots, X_n \Rightarrow \textbf{exn}(T''_n), \\ \mathit{Proc} \vdash B \Rightarrow \textbf{exit}(T)\end{array}}{\begin{array}{l}\mathcal{C} \vdash (\textbf{process}\ \Pi\ [G_1 : T_1, \ldots, G_m : T_m]\ (V_1 : T'_1, \ldots, V_p : T'_p) \\ \quad : T\ \textbf{raises}\ [X_1 : T''_1, \ldots, X_n : T''_n]\ \textbf{is}\ B), \mathit{id} \\ \quad \Rightarrow (< \mathit{id}, \Pi > \Rightarrow [\textbf{gate}(T_1), \ldots, \textbf{gate}(T_m)] \\ \qquad (V_1 \Rightarrow T'_1, \ldots, V_p \Rightarrow T'_p) \\ \qquad [\textbf{exn}(T''_1), \ldots, \textbf{exn}(T''_n)] \rightarrow \textbf{exit}(T))\end{array}}$$

$$\frac{\begin{array}{r}\mathcal{C} \vdash (T_1) \Rightarrow \textbf{type} \ \cdots \ \mathcal{C} \vdash (T_m) \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (T'_1) \Rightarrow \textbf{type} \ \cdots \ \mathcal{C} \vdash (T'_p) \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (T''_1) \Rightarrow \textbf{type} \ \cdots \ \mathcal{C} \vdash (T''_n) \Rightarrow \textbf{type} \\ \mathcal{C}, G_1 \Rightarrow \textbf{gate}(T_1), \ldots, G_m \Rightarrow \textbf{gate}(T_m), \\ V_1 \Rightarrow T'_1, \ldots, V_p \Rightarrow T'_p, \\ X_1 \Rightarrow \textbf{exn}(T''_1), \ldots, X_n \Rightarrow \textbf{exn}(T''_n), \\ \mathit{Proc} \vdash B \Rightarrow \textbf{guarded}(T)\end{array}}{\begin{array}{l}\mathcal{C} \vdash (\textbf{process}\ \Pi\ [G_1 : T_1, \ldots, G_m : T_m]\ (V_1 : T'_1, \ldots, V_p : T'_p) \\ \quad : T\ \textbf{raises}\ [X_1 : T''_1, \ldots, X_n : T''_n]\ \textbf{is}\ B), \mathit{id} \\ \quad \Rightarrow (< \mathit{id}, \Pi > \Rightarrow [\textbf{gate}(T_1), \ldots, \textbf{gate}(T_m)] \\ \qquad (V_1 \Rightarrow T'_1, \ldots, V_p \Rightarrow T'_p) \\ \qquad [\textbf{exn}(T''_1), \ldots, \textbf{exn}(T''_n)] \rightarrow \textbf{guarded}(T))\end{array}}$$

where

$$\begin{array}{rl}\mathit{Proc} = (< \mathit{id}, \Pi > \Rightarrow & [\textbf{gate}(T_1), \ldots, \textbf{gate}(T_m)] \\ & (V_1 \Rightarrow T'_1, \ldots, V_p \Rightarrow T'_p) \\ & [\textbf{exn}(T''_1), \ldots, \textbf{exn}(T''_n)] \rightarrow \textbf{exit}(T))\end{array}$$

Note that *Proc* (the process declaration) is assumed in the context to allow recursive instantiation in B.

Dynamic semantics

$$\frac{}{\begin{array}{l}\mathcal{E} \vdash (\textbf{process}\ \Pi\ [G_1 : T_1, \ldots, G_m : T_m]\ (V_1 : T'_1, \ldots, V_p : T'_p) \\ \quad : T\ \textbf{raises}\ [X_1 : T''_1, \ldots, X_n : T''_n]\ \textbf{is}\ B), \mathit{id} \\ \quad \Rightarrow (< \mathit{id}, \Pi > \Rightarrow \lambda\ [G_1 : T_1, \ldots, G_m : T_m]\ (V_1 : T'_1, \ldots, V_p : T'_p)\ [X_1 : T''_1, \ldots, X_n : T''_n] \rightarrow B)\end{array}}$$

8.11.5 Sequential declarations

Abstract syntax

$D\ D$

Static semantics

$$\frac{\mathcal{C}, \mathcal{C}'' \vdash D_1, \mathit{id} \Rightarrow \mathcal{C}' \quad \mathcal{C}, \mathcal{C}' \vdash D_2, \mathit{id} \Rightarrow \mathcal{C}''}{\mathcal{C} \vdash (D_1\ D_2), \mathit{id} \Rightarrow \mathcal{C}', \mathcal{C}''}$$

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Dynamic semantics

$$\frac{\mathcal{E} \vdash D_1, id \Rightarrow \mathcal{E}' \quad \mathcal{E} \vdash D_2, id \Rightarrow \mathcal{E}''}{\mathcal{E} \vdash (D_1\ D_2), id \Rightarrow \mathcal{E}', \mathcal{E}''}$$

9 The E-LOTOS base language

9.1 Introduction

In this chapter, we give semantics for terms defined by abstract syntax. Each subclause is devoted to a category, mostly of them divided in four parts:

- Abstract syntax: just a reminder.
- Static semantics: rules defining static semantics.
- Untimed dynamic semantics: rules defining untimed dynamic semantics.
- Timed dynamic semantics: rules for timed dynamic semantics.

If the category is just the join of simpler categories, you will find a number of subclauses for every clause.

9.2 Behaviours

9.2.1 Disabling behaviour expression

Abstract syntax

DisB	::=	*BT* [> *BT*	*singleton*	(DisB1)
	|	*BT* [> *DisB*	*disjoint union*	(DisB2)

Static semantics

$$\frac{\mathcal{C}, RT_1, RT_2 \vdash B_1 \Rightarrow \mathbf{guarded}(RT_1, RT) \qquad \mathcal{C}, RT_1, RT_2 \vdash B_2 \Rightarrow \mathbf{guarded}(RT_2, RT)}{\mathcal{C}, RT_1, RT_2 \vdash B_1\ [> B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2, RT)}$$

$$\frac{\mathcal{C}, RT_1, RT_2 \vdash B_1 \Rightarrow \mathbf{exit}(RT_1, RT) \qquad \mathcal{C}, RT_1, RT_2 \vdash B_2 \Rightarrow \mathbf{guarded}(RT_2, RT)}{\mathcal{C}, RT_1, RT_2 \vdash B_1\ [> B_2 \Rightarrow \mathbf{exit}(RT_1, RT_2, RT)}$$

RT_1 and RT_2 are bindings from the context, and RT are new bindings in the disabling behaviour expressions: they should be produced by both B_1 and B_2. With that, we allow expressions as:

```
?x:= 1;
(  a1; ?x:=2; ?z:= 2; a2
   [>  b1; ?z:= 3; b2
);
?y:= x;
```

As the variable x is bound beforehand, there is no need to ensure that both B_1 and B_2 bind it again. However, the static semantics checks that z is bound in both sub-behaviours. The same is done in all branching expressions (selection, case, trap, ...) and it is very cumbersome for the **trap** operator.

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G(RN)} B_1'}{\mathcal{E} \vdash B_1 \ [> B_2 \xrightarrow{G(RN)} B_1' \ [> B_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}()} B_1'}{\mathcal{E} \vdash B_1 \ [> B_2 \xrightarrow{\mathbf{i}()} B_1' \ [> B_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN)} B_1'}{\mathcal{E} \vdash B_1 \ [> B_2 \xrightarrow{\mathbf{i}()} \mathbf{exit}\ (RN)}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{X(RN)} B_1'}{\mathcal{E} \vdash B_1 \ [> B_2 \xrightarrow{\mathbf{i}()} \mathbf{signal}\ X\ (RN)\,;(B_1' \ [> B_2)}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{G(RN)} B_2'}{\mathcal{E} \vdash B_1 \ [> B_2 \xrightarrow{G(RN)} B_2'}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}()} B_2'}{\mathcal{E} \vdash B_1 \ [> B_2 \xrightarrow{\mathbf{i}()} B_2'}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{X(RN)} B_2'}{\mathcal{E} \vdash B_1 \ [> B_2 \xrightarrow{\mathbf{i}()} \mathbf{signal}\ X\ (RN)\,;B_2'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B_2'}{\mathcal{E} \vdash B_1 \ [> B_2 \xrightarrow{\varepsilon(d)} B_1' \ [> B_2'}$$

9.2.2 Synchronization behaviour expression

Abstract syntax

SyncB	::=	*BT* || *BT*	*synchronization*	(SyncB1)
	|	*BT* || *SyncB*	*synchronization*	(SyncB2)

Static semantics

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \qquad \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}(RT_2)}{\mathcal{C} \vdash B_1 \ || \ B_2 \Rightarrow \mathbf{exit}(RT_1, RT_2)}\ [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \qquad \mathcal{C} \vdash B_2 \Rightarrow \mathbf{guarded}(RT_2)}{\mathcal{C} \vdash B_1 \ || \ B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2)}\ [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{guarded}(RT_1) \qquad \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}(RT_2)}{\mathcal{C} \vdash B_1 \ || \ B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2)}\ [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}()} B_1'}{\mathcal{E} \vdash B_1 \;||\; B_2 \xrightarrow{\mathbf{i}()} B_1' \;||\; B_2}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}()} B_2'}{\mathcal{E} \vdash B_1 \;||\; B_2 \xrightarrow{\mathbf{i}()} B_1 \;||\; B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G(RN)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{G(RN)} B_2'}{\mathcal{E} \vdash B_1 \;||\; B_2 \xrightarrow{G(RN)} B_1' \;||\; B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{X(RN)} B_1'}{\mathcal{E} \vdash B_1 \;||\; B_2 \xrightarrow{\mathbf{i}()} \mathbf{signal}\ X\ (RN)\,;(B_1' \;||\; B_2)}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{X(RN)} B_2'}{\mathcal{E} \vdash B_1 \;||\; B_2 \xrightarrow{\mathbf{i}()} \mathbf{signal}\ X\ (RN)\,;(B_1 \;||\; B_2')}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\delta(RN_2)} B_2'}{\mathcal{E} \vdash B_1 \;||\; B_2 \xrightarrow{\delta(RN_1,RN_2)} B_1' \;||\; B_2'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B_2'}{\mathcal{E} \vdash B_1 \;||\; B_2 \xrightarrow{\varepsilon(d)} B_1' \;||\; B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN@d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d+d')} B_2'}{\mathcal{E} \vdash B_1 \;||\; B_2 \xrightarrow{\varepsilon(d+d')} \mathbf{exit}(RN) \;||\; B_2'}\ [0 < d']$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d+d')} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\delta(RN@d)} B_2'}{\mathcal{E} \vdash B_1 \;||\; B_2 \xrightarrow{\varepsilon(d+d')} B_1' \;||\; \mathbf{exit}(RN)}\ [0 < d']$$

9.2.3 Concurrency behaviour expression

Abstract syntax

ConcB	::=	*BT* |[$G(,G)^*$]| *BT*	*concurrency*	(ConcB1)
	|	*BT* |[$G(,G)^*$]| *ConcB*	*concurrency*	(ConcB2)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Static semantics

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \qquad \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}(RT_2) \quad \mathcal{C} \vdash G_1 \Rightarrow \mathbf{gate}(RT_1') \ \cdots \ \mathcal{C} \vdash G_n \Rightarrow \mathbf{gate}(RT_n')}{\mathcal{C} \vdash B_1 \ |[G_1, \ldots, G_n]| \ B_2 \Rightarrow \mathbf{exit}(RT_1, RT_2)} \ [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \qquad \mathcal{C} \vdash B_2 \Rightarrow \mathbf{guarded}(RT_2) \quad \mathcal{C} \vdash G_1 \Rightarrow \mathbf{gate}(RT_1') \ \cdots \ \mathcal{C} \vdash G_n \Rightarrow \mathbf{gate}(RT_n')}{\mathcal{C} \vdash B_1 \ |[G_1, \ldots, G_n]| \ B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2)} \ [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{guarded}(RT_1) \qquad \mathcal{C} \vdash B_2 \Rightarrow \mathbf{exit}(RT_2) \quad \mathcal{C} \vdash G_1 \Rightarrow \mathbf{gate}(RT_1') \ \cdots \ \mathcal{C} \vdash G_n \Rightarrow \mathbf{gate}(RT_n')}{\mathcal{C} \vdash B_1 \ |[G_1, \ldots, G_n]| \ B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2)} \ [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}()} B_1'}{\mathcal{E} \vdash B_1 \ |[\vec{G}]| \ B_2 \xrightarrow{\mathbf{i}()} B_1' \ |[\vec{G}]| \ B_2}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}()} B_2'}{\mathcal{E} \vdash B_1 \ |[\vec{G}]| \ B_2 \xrightarrow{\mathbf{i}()} B_1 \ |[\vec{G}]| \ B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G(RN)} B_1'}{\mathcal{E} \vdash B_1 \ |[\vec{G}]| \ B_2 \xrightarrow{G(RN)} B_1' \ |[\vec{G}]| \ B_2} \ \left[G \notin \vec{G}\right]$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{G(RN)} B_2'}{\mathcal{E} \vdash B_1 \ |[\vec{G}]| \ B_2 \xrightarrow{G(RN)} B_1 \ |[\vec{G}]| \ B_2'} \ \left[G \notin \vec{G}\right]$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G_i(RN)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{G_i(RN)} B_2'}{\mathcal{E} \vdash B_1 \ |[\vec{G}]| \ B_2 \xrightarrow{G_i(RN)} B_1' \ |[\vec{G}]| \ B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{X(RN)} B_1'}{\mathcal{E} \vdash B_1 \ |[\vec{G}]| \ B_2 \xrightarrow{\mathbf{i}()} \mathbf{signal}\ X\ (RN)\,;(B_1' \ |[\vec{G}]| \ B_2)}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{X(RN)} B_2'}{\mathcal{E} \vdash B_1 \ |[\vec{G}]| \ B_2 \xrightarrow{\mathbf{i}()} \mathbf{signal}\ X\ (RN)\,;(B_1 \ |[\vec{G}]| \ B_2')}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\delta(RN_2)} B_2'}{\mathcal{E} \vdash B_1 \ |[\vec{G}]| \ B_2 \xrightarrow{\delta(RN_1, RN_2)} B_1' \ |[\vec{G}]| \ B_2'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B_2'}{\mathcal{E} \vdash B_1 \ |[\vec{G}]| \ B_2 \xrightarrow{\varepsilon(d)} B_1' \ |[\vec{G}]| \ B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN@d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d+d')} B_2'}{\mathcal{E} \vdash B_1 \ |[\vec{G}]| \ B_2 \xrightarrow{\varepsilon(d+d')} \mathbf{exit}(RN) \ |[\vec{G}]| \ B_2'} \ [0 < d']$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d+d')} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\delta(RN@d)} B_2'}{\mathcal{E} \vdash B_1 \ |[\vec{G}]| \ B_2 \xrightarrow{\varepsilon(d+d')} B_1' \ |[\vec{G}]| \ \mathbf{exit}(RN)} \ [0 < d']$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2.4 Selection behaviour expression

Abstract syntax

$$\begin{array}{lcll} SelB & ::= & BT \ [] \ BT & \textit{choice} \quad \text{(SelB1)} \\ & | & BT \ [] \ SelB & \textit{choice} \quad \text{(SelB2)} \end{array}$$

Static semantics

$$\frac{\mathcal{C}, RT_1, RT_2 \vdash B_1 \Rightarrow \mathbf{guarded}(RT_1, RT) \qquad \mathcal{C}, RT_1, RT_2 \vdash B_2 \Rightarrow \mathbf{guarded}(RT_2, RT)}{\mathcal{C}, RT_1, RT_2 \vdash B_1 \ [] \ B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2, RT)}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G(RN)} B_1'}{\mathcal{E} \vdash B_1 \ [] \ B_2 \xrightarrow{G(RN)} B_1'}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{G(RN)} B_2'}{\mathcal{E} \vdash B_1 \ [] \ B_2 \xrightarrow{G(RN)} B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}()} B_1'}{\mathcal{E} \vdash B_1 \ [] \ B_2 \xrightarrow{\mathbf{i}()} B_1'}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}()} B_2'}{\mathcal{E} \vdash B_1 \ [] \ B_2 \xrightarrow{\mathbf{i}()} B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{X(RN)} B_1'}{\mathcal{E} \vdash B_1 \ [] \ B_2 \xrightarrow{\mathbf{i}()} \mathbf{signal}\ X\ (RN);B_1'}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{X(RN)} B_2'}{\mathcal{E} \vdash B_1 \ [] \ B_2 \xrightarrow{\mathbf{i}()} \mathbf{signal}\ X\ (RN);B_2'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B_2'}{\mathcal{E} \vdash B_1 \ [] \ B_2 \xrightarrow{\varepsilon(d)} B_1' \ [] \ B_2'}$$

9.2.5 Suspend/Resume behaviour expression

Abstract syntax

$$B \ [X> \ B$$

Static semantics

$$\frac{\mathcal{C}, RT_1, RT_2 \vdash B_1 \Rightarrow \mathbf{exit}(RT_1, RT) \qquad \mathcal{C}, RT_1, RT_2; X \Rightarrow \mathbf{exn}() \vdash B_2 \Rightarrow \mathbf{guarded}(RT_2, RT)}{\mathcal{C}, RT_1, RT_2 \vdash B_1 \ [X> \ B_2 \Rightarrow \mathbf{exit}(RT_1, RT_2, RT)}$$

$$\frac{\mathcal{C}, RT_1, RT_2 \vdash B_1 \Rightarrow \mathbf{guarded}(RT_1, RT) \qquad \mathcal{C}, RT_1, RT_2; X \Rightarrow \mathbf{exn}() \vdash B_2 \Rightarrow \mathbf{guarded}(RT_2, RT)}{\mathcal{C}, RT_1, RT_2 \vdash B_1 \ [X> \ B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2, RT)}$$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G(RN)} B_1'}{\mathcal{E} \vdash B_1 \ [X{>}\ B_2 \xrightarrow{G(RN)} B_1' \ [X{>}\ B_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}()} B_1'}{\mathcal{E} \vdash B_1 \ [X{>}\ B_2 \xrightarrow{\mathbf{i}()} B_1' \ [X{>}\ B_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN)} B_1'}{\mathcal{E} \vdash B_1 \ [X{>}\ B_2 \xrightarrow{\mathbf{i}()} \mathbf{exit}\ (RN)}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{X'(RN)} B_1'}{\mathcal{E} \vdash B_1 \ [X{>}\ B_2 \xrightarrow{\mathbf{i}()} \mathbf{signal}\ X'\ (RN)\,;(B_1' \ [X{>}\ B_2)}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{G(RN)} B_2'}{\mathcal{E} \vdash B_1 \ [X{>}\ B_2 \xrightarrow{G(RN)} \mathbf{trap\ exception}\ X\,()\ \mathbf{is}\ B_1 \ [X{>}\ B_2\ \mathbf{in}\ B_2'}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}()} B_2'}{\mathcal{E} \vdash B_1 \ [X{>}\ B_2 \xrightarrow{\mathbf{i}()} \mathbf{trap\ exception}\ X\,()\ \mathbf{is}\ B_1 \ [X{>}\ B_2\ \mathbf{in}\ B_2'}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{X'(RN)} B_2'}{\mathcal{E} \vdash B_1 \ [X{>}\ B_2 \xrightarrow{\mathbf{i}()} \mathbf{signal}\ X'\ (RN)\,;\mathbf{trap\ exception}\ X\,()\ \mathbf{is}\ B_1 \ [X{>}\ B_2\ \mathbf{in}\ B_2'}\ [X' \neq X]$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1' \qquad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B_2'}{\mathcal{E} \vdash B_1 \ [X{>}\ B_2 \xrightarrow{\varepsilon(d)} B_1' \ [X{>}\ B_2'}$$

9.2.6 Sequential composition

Abstract syntax

$$BA\ ;\ BT$$

Static semantics

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \qquad \mathcal{C};RT_1 \vdash B_2 \Rightarrow \mathbf{exit}(RT_2)}{\mathcal{C} \vdash B_1\ ;\ B_2 \Rightarrow \mathbf{exit}(RT_1;RT_2)}$$

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \qquad \mathcal{C};RT_1 \vdash B_2 \Rightarrow \mathbf{guarded}(RT_2)}{\mathcal{C} \vdash B_1\ ;\ B_2 \Rightarrow \mathbf{guarded}(RT_1;RT_2)}$$

$$\frac{\mathcal{C} \vdash B_1 \Rightarrow \mathbf{guarded}(RT_1) \qquad \mathcal{C};RT_1 \vdash B_2 \Rightarrow \mathbf{exit}(RT_2)}{\mathcal{C} \vdash B_1\ ;\ B_2 \Rightarrow \mathbf{guarded}(RT_1;RT_2)}\ [none \neq RT_1]$$

The side condition of third rule bans constructions such as $G!1;\mathbf{stop};G!2;\ldots$.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{a(RN)} B_1'}{\mathcal{E} \vdash B_1 \ ; \ B_2 \xrightarrow{a(RN)} B_1' \ ; \ B_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B_1' \quad \mathcal{E} \vdash B_2[RN_1] \xrightarrow{a(RN_2)} B_2'}{\mathcal{E} \vdash B_1 \ ; \ B_2 \xrightarrow{a(RN_2)} \mathbf{exit}(RN_1) \ ; \ B_2'}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B_1' \quad \mathcal{E} \vdash B_2[RN_1] \xrightarrow{\delta(RN_2)} B_2'}{\mathcal{E} \vdash B_1 \ ; \ B_2 \xrightarrow{\delta(RN_1;RN_2)} \mathbf{block}}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1'}{\mathcal{E} \vdash B_1 \ ; \ B_2 \xrightarrow{\varepsilon(d)} B_1' \ ; \ B_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1@d_1)} B_1' \quad \mathcal{E} \vdash B_2[RN_1] \xrightarrow{\varepsilon(d_2)} B_2'}{\mathcal{E} \vdash B_1 \ ; \ B_2 \xrightarrow{\varepsilon(d_1+d_2)} \mathbf{exit}(RN_1) \ ; \ B_2'}$$

9.2.7 Action

Abstract syntax

$G \ P \ @P \ E \ \mathbf{start}(N)$

Static semantics

$$\frac{\begin{array}{c}\mathcal{C} \vdash G \Rightarrow \mathbf{gate}\ (RT) \\ \mathcal{C} \vdash (P_1 \Rightarrow (RT)) \Rightarrow (RT_1) \\ \mathcal{C} \vdash (P_2 \Rightarrow \text{time}) \Rightarrow (RT_2) \\ \mathcal{C}; RT_1, RT_2 \vdash E \Rightarrow \mathbf{exit}(\text{bool})\end{array}}{\mathcal{C} \vdash G \ P_1 \ @P_2 \ E \ \mathbf{start}(0) \Rightarrow \mathbf{guarded}(RT_1, RT_2)} \ [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

Untimed dynamic semantics

$$\frac{\begin{array}{c}\mathcal{E} \vdash (P_1 \Rightarrow (RN)) \Rightarrow (RN_1) \\ \mathcal{E} \vdash (P_2 \Rightarrow d) \Rightarrow (RN_2) \\ \mathcal{E} \vdash E[RN_1, RN_2] \xrightarrow{\delta(\text{true})} E'\end{array}}{\mathcal{E} \vdash G \ P_1 \ @P_2 \ E \ \mathbf{start}(d) \xrightarrow{G(RN)} \mathbf{exit}(RN_1, RN_2)}$$

Timed dynamic semantics

$$\frac{}{\mathcal{E} \vdash G \ P_1 \ @P_2 \ E \ \mathbf{start}(d) \xrightarrow{\varepsilon(d')} G \ P_1 \ @P_2 \ E \ \mathbf{start}(d+d')} \ [0 < d']$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2.8 Internal action

Abstract syntax

i

Static semantics

$$\overline{\mathcal{C} \vdash \mathbf{i} \Rightarrow \mathbf{guarded}()}$$

Untimed dynamic semantics

$$\overline{\mathcal{E} \vdash \mathbf{i} \xrightarrow{\mathbf{i}()} \mathbf{exit}()}$$

Timed dynamic semantics None.

9.2.9 Succesful termination without values

Abstract syntax

null

Static semantics

$$\overline{\mathcal{C} \vdash \mathbf{null} \Rightarrow \mathbf{exit}()}$$

Untimed dynamic semantics

$$\overline{\mathcal{E} \vdash \mathbf{null} \xrightarrow{\delta()} \mathbf{block}}$$

Timed dynamic semantics None.

9.2.10 Succesful termination

Abstract syntax

exit $[(RN)]$

The default termination value is ().

Note that **exit** solely appears in abstract syntax, as a result of syntactic translation or as used by the semantics.

Static semantics

$$\frac{\mathcal{C} \vdash RN \Rightarrow RT}{\mathcal{C} \vdash \mathbf{exit}(RN) \Rightarrow \mathbf{exit}(RT)}$$

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Untimed dynamic semantics

$$\frac{}{\mathcal{E} \vdash \mathbf{exit}(RN) \xrightarrow{\delta(RN)} \mathbf{block}}$$

Timed dynamic semantics None.

9.2.11 Inaction

Abstract syntax

stop

Static semantics

$$\frac{}{\mathcal{C} \vdash \mathbf{stop} \Rightarrow \mathbf{guarded}(\mathbf{none})}$$

Untimed dynamic semantics None.

Timed dynamic semantics

$$\frac{}{\mathcal{E} \vdash \mathbf{stop} \xrightarrow{\varepsilon(d)} \mathbf{stop}}\ [0 < d]$$

9.2.12 Time block

Abstract syntax

block

Static semantics

$$\frac{}{\mathcal{C} \vdash \mathbf{block} \Rightarrow \mathbf{guarded}\ (\mathbf{none})}$$

Untimed dynamic semantics None.

Timed dynamic semantics None.

9.2.13 Delay

Abstract syntax

wait '(' *E* ')'

Static semantics

$$\frac{\mathcal{C} \vdash E \Rightarrow \mathbf{exit}(\text{time})}{\mathcal{C} \vdash \mathbf{wait}\ (E) \Rightarrow \mathbf{exit}()}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(0)} E'}{\mathcal{E} \vdash \textbf{wait}\ (E) \xrightarrow{\delta()} \textbf{block}}$$

$$\frac{\mathcal{E} \vdash E \xrightarrow{X(RN)} E'}{\mathcal{E} \vdash \textbf{wait}\ (E) \xrightarrow{X(RN)} \textbf{wait}\ (E')}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(d+d')} E'}{\mathcal{E} \vdash \textbf{wait}\ (E) \xrightarrow{\varepsilon(d)} \textbf{wait}\ (d')}\ [0 < d]$$

9.2.14 Assignment

Abstract syntax

P := EA

The pattern must be irrefutable.

Static semantics

$$\frac{\mathcal{C} \vdash EA \Rightarrow \textbf{exit}(T) \quad \mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT)}{\mathcal{C} \vdash P\ \texttt{:=}\ EA \Rightarrow \textbf{exit}(RT)}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash EA \xrightarrow{X(RN)} EA'}{\mathcal{E} \vdash P\ \texttt{:=}\ EA \xrightarrow{X(RN)} P\ \texttt{:=}\ EA'}$$

$$\frac{\mathcal{E} \vdash EA \xrightarrow{\delta(N)} EA' \quad \mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN)}{\mathcal{E} \vdash P\ \texttt{:=}\ EA \xrightarrow{\delta(RN)} \textbf{block}}$$

Timed dynamic semantics None.

9.2.15 Nondeterministic Assignment

Abstract syntax

P := **any** T ['[' E ']']

The pattern must be irrefutable. The default expression is [true].

Static semantics

$$\frac{\mathcal{C} \vdash T \Rightarrow \textbf{type} \quad \mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT) \quad \mathcal{C}; RT \vdash E \Rightarrow \textbf{exit}(\text{bool})}{\mathcal{C} \vdash P\ \texttt{:=}\ \textbf{any}\ T\ \texttt{'['}\ E\ \texttt{']'} \Rightarrow \textbf{guarded}(RT)}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash N \Rightarrow T \quad \mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \quad \mathcal{E} \vdash E[RN] \xrightarrow{\delta(\text{true})} E'}{\mathcal{E} \vdash P \texttt{ := } \textbf{any } T \texttt{ '[' } E \texttt{ ']'} \xrightarrow{\mathbf{i}()} \textbf{exit}(RN)}$$

Timed dynamic semantics None.

9.2.16 Choice over values

Abstract syntax

$$\textbf{choice } P \textbf{ after}(N) \texttt{ [] } B$$

Static semantics

$$\frac{\mathcal{C} \vdash (P \Rightarrow \textbf{any}) \Rightarrow (RT) \quad \mathcal{C} \vdash N \Rightarrow \text{time} \quad \mathcal{C}; RT \vdash B \Rightarrow \textbf{guarded}(RT')}{\mathcal{C} \vdash \textbf{choice } P \textbf{ after}(N) \texttt{ [] } B \Rightarrow \textbf{guarded}(RT')}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash (N \Rightarrow \textbf{any}) \quad \mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \quad \mathcal{E} \vdash B[RN] \xrightarrow{G(RN'@d)} B'}{\mathcal{E} \vdash \textbf{choice } P \textbf{ after}(d) \texttt{ [] } B \xrightarrow{G(RN')} B'}$$

$$\frac{\mathcal{E} \vdash (N \Rightarrow \textbf{any}) \quad \mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \quad \mathcal{E} \vdash B[RN] \xrightarrow{\mathbf{i}(@d)} B'}{\mathcal{E} \vdash \textbf{choice } P \textbf{ after}(d) \texttt{ [] } B \xrightarrow{\mathbf{i}()} B'}$$

$$\frac{\mathcal{E} \vdash (N \Rightarrow \textbf{any}) \quad \mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \quad \mathcal{E} \vdash B[RN] \xrightarrow{X(RN'@d)} B'}{\mathcal{E} \vdash \textbf{choice } P \textbf{ after}(d) \texttt{ [] } B \xrightarrow{\mathbf{i}()} \textbf{signal } X \texttt{ (} RN' \texttt{)} ; B'}$$

Note: This operator is the only place where the timed semantics is used in the untimed semantics, thus breaking the "classical" stratification [5, 28] used to prove the consistency of the semantics. However, as shown in [17] for the ET-LOTOS language (that has the same choice over values), another proof technique can ensure the consistency of the semantics and also that strong bisimulation is a congruence.

Timed dynamic semantics

$$\frac{\forall N \,.\, ((\mathcal{E} \vdash N \Rightarrow \textbf{any} \text{ and } \mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN)) \text{ implies } B[RN] \xrightarrow{\varepsilon(d+d')})}{\mathcal{E} \vdash \textbf{choice } P \textbf{ after}(d) \texttt{ [] } B \xrightarrow{\varepsilon(d')} \textbf{choice } P \textbf{ after}(d+d') \texttt{ [] } B} [0 < d']$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2.17 Trap

Abstract syntax

trap (**exception** X $[(IPL)]$ **is** B **endexn**)* [**exit** $[P]$ **is** B] **in** B

The default input parameter list is () and the default exit pattern is ().

Static semantics To abbreviate notation, in these static semantic rules $\mathcal{C}$ stands for $\mathcal{C}', RT_1''' \odot \ldots \odot RT_n''' \odot RT_e''' \odot RT'''$. RT_i''' are bindings available in the context, i.e., those variables that have a value, and may be actualized by some of the B_i in the **trap** operator. RT are new bindings produced in the **trap** operator, so all B_i must produces them. The resulting behaviour binds $RT_1''' \odot \ldots \odot RT_n''' \odot RT_e''' \odot RT''', RT$. With that we ensure that if new bindings are produced in the **trap** operator, any of its branches produces the same bindings, but bound variables may be reasigned freely in any of the subbehaviours.

$$\frac{\begin{array}{r}\mathcal{C}' \vdash (RT_1) \Rightarrow \textbf{type} \;\cdots\; \mathcal{C}' \vdash (RT_n) \Rightarrow \textbf{type} \\ \mathcal{C}' \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \;\cdots\; \mathcal{C}' \vdash (RP_n \Rightarrow RT_n) \Rightarrow (RT_n') \\ \mathcal{C}; RT_1' \vdash B_1 \Rightarrow \textbf{exit}(RT_1''', RT) \;\cdots\; \mathcal{C}; RT_n' \vdash B_n \Rightarrow \textbf{exit}(RT_n''', RT) \\ \mathcal{C}; X_1 \Rightarrow \textbf{exn}(RT_1), \ldots, X_n \Rightarrow \textbf{exn}(RT_n) \vdash B \Rightarrow \textbf{exit}(RT''', RT)\end{array}}{\begin{array}{l}\mathcal{C} \vdash (\textbf{trap exception } X_1(RP_1 : RT_1) \textbf{ is } B_1 \\ \quad \ldots \textbf{ exception } X_n(RP_n : RT_n) \textbf{ is } B_n \textbf{ in } B) \\ \quad \Rightarrow \textbf{exit}(RT_1''' \odot \ldots \odot RT_n''' \odot RT_e''' \odot RT''', RT)\end{array}}$$

$$\frac{\begin{array}{r}\mathcal{C}' \vdash (RT_1) \Rightarrow \textbf{type} \;\cdots\; \mathcal{C}' \vdash (RT_n) \Rightarrow \textbf{type} \\ \mathcal{C}' \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \;\cdots\; \mathcal{C}' \vdash (RP_n \Rightarrow RT_n) \Rightarrow (RT_n') \\ \mathcal{C}; RT_1' \vdash B_1 \Rightarrow \textbf{guarded}(RT_1''', RT) \;\cdots\; \mathcal{C}; RT_n' \vdash B_n \Rightarrow \textbf{guarded}(RT_n''', RT) \\ \mathcal{C}; X_1 \Rightarrow \textbf{exn}(RT_1), \ldots, X_n \Rightarrow \textbf{exn}(RT_n) \vdash B \Rightarrow \textbf{guarded}(RT''', RT)\end{array}}{\begin{array}{l}\mathcal{C} \vdash (\textbf{trap exception } X_1(RP_1 : RT_1) \textbf{ is } B_1 \\ \quad \ldots \textbf{ exception } X_n(RP_n : RT_n) \textbf{ is } B_n \textbf{ in } B) \\ \quad \Rightarrow \textbf{guarded}(RT_1''' \odot \ldots \odot RT_n''' \odot RT_e''' \odot RT''', RT)\end{array}}$$

$$\frac{\begin{array}{r}\mathcal{C}' \vdash (RT_1) \Rightarrow \textbf{type} \;\cdots\; \mathcal{C}' \vdash (RT_n) \Rightarrow \textbf{type} \\ \mathcal{C}' \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \;\cdots\; \mathcal{C}' \vdash (RP_n \Rightarrow RT_n) \Rightarrow (RT_n') \\ \mathcal{C}; RT_1' \vdash B_1 \Rightarrow \textbf{exit}(RT_1''', RT) \;\cdots\; \mathcal{C}; RT_n' \vdash B_n \Rightarrow \textbf{exit}(RT_n''', RT) \\ \mathcal{C}; X_1 \Rightarrow \textbf{exn}(RT_1), \ldots, X_n \Rightarrow \textbf{exn}(RT_n) \vdash B \Rightarrow \textbf{exit}(RT''', RT') \\ \mathcal{C} \vdash (P \Rightarrow (RT')) \Rightarrow (RT'') \\ \mathcal{C}; RT'' \vdash B' \Rightarrow \textbf{exit}(RT_e''', RT)\end{array}}{\begin{array}{l}\mathcal{C} \vdash (\textbf{trap exception } X_1(RP_1 : RT_1) \textbf{ is } B_1 \\ \quad \ldots \textbf{ exception } X_n(RP_n : RT_n) \textbf{ is } B_n \\ \quad\quad \textbf{exit } P \textbf{ is } B' \textbf{ in } B) \\ \quad \Rightarrow \textbf{exit}(RT_1''' \odot \ldots \odot RT_n''' \odot RT_e''' \odot RT''', RT)\end{array}}$$

$$\frac{\begin{array}{r}\mathcal{C}' \vdash (RT_1) \Rightarrow \textbf{type} \;\cdots\; \mathcal{C}' \vdash (RT_n) \Rightarrow \textbf{type} \\ \mathcal{C}' \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \;\cdots\; \mathcal{C}' \vdash (RP_n \Rightarrow RT_n) \Rightarrow (RT_n') \\ \mathcal{C}; RT_1' \vdash B_1 \Rightarrow \textbf{guarded}(RT_1''', RT) \;\cdots\; \mathcal{C}; RT_n' \vdash B_n \Rightarrow \textbf{guarded}(RT_n''', RT) \\ \mathcal{C}; X_1 \Rightarrow \textbf{exn}(RT_1), \ldots, X_n \Rightarrow \textbf{exn}(RT_n) \vdash B \Rightarrow \textbf{exit}(RT''', RT') \\ \mathcal{C} \vdash (P \Rightarrow (RT')) \Rightarrow (RT'') \\ \mathcal{C}; RT'' \vdash B' \Rightarrow \textbf{guarded}(RT_e''', RT)\end{array}}{\begin{array}{l}\mathcal{C} \vdash (\textbf{trap exception } X_1(RP_1 : RT_1) \textbf{ is } B_1 \\ \quad \ldots \textbf{ exception } X_n(RP_n : RT_n) \textbf{ is } B_n \\ \quad\quad \textbf{exit } P \textbf{ is } B' \textbf{ in } B) \\ \quad \Rightarrow \textbf{guarded}(RT_1''' \odot \ldots \odot RT_n''' \odot RT_e''' \odot RT''', RT)\end{array}}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$$\frac{\begin{array}{r}\mathcal{C}' \vdash (RT_1) \Rightarrow \textbf{type} \;\cdots\; \mathcal{C}' \vdash (RT_n) \Rightarrow \textbf{type} \\ \mathcal{C}' \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \;\cdots\; \mathcal{C}' \vdash (RP_n \Rightarrow RT_n) \Rightarrow (RT_n') \\ \mathcal{C}; RT_1' \vdash B_1 \Rightarrow \textbf{guarded}(RT_1''', RT) \;\cdots\; \mathcal{C}; RT_n' \vdash B_n \Rightarrow \textbf{guarded}(RT_n''', RT) \\ \mathcal{C}; X_1 \Rightarrow \textbf{exn}(RT_1), \ldots, X_n \Rightarrow \textbf{exn}(RT_n) \vdash B \Rightarrow \textbf{guarded}(RT''', RT') \\ \mathcal{C} \vdash (P \Rightarrow (RT')) \Rightarrow (RT'') \\ \mathcal{C}; RT'' \vdash B' \Rightarrow \textbf{exit}(RT_e''', RT)\end{array}}{\begin{array}{r}\mathcal{C} \vdash (\textbf{trap exception } X_1(RP_1 : RT_1) \textbf{ is } B_1 \\ \ldots \textbf{ exception } X_n(RP_n : RT_n) \textbf{ is } B_n \\ \textbf{exit } P \textbf{ is } B' \textbf{ in } B) \\ \Rightarrow \textbf{guarded}(RT_1''' \odot \ldots \odot RT_n''' \odot RT_e''' \odot RT''', RT)\end{array}}$$

Untimed dynamic semantics Here $\vec{X}$ ranges over $\vec{X}$ and **exit** (which we consider to be equal to δ).

$$\frac{\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B'}{\mathcal{E} \vdash (\textbf{trap } \vec{X} \ (\vec{RP} : \vec{RT}) \textbf{ is } \vec{B} \textbf{ in } B) \xrightarrow{\mu(RN)} (\textbf{trap } \vec{X} \ (\vec{RP} : \vec{RT}) \textbf{ is } \vec{B} \textbf{ in } B')} \left[\mu \notin \vec{X}\right]$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash B \xrightarrow{X_i(RN)} B' \\ \mathcal{E} \vdash ((RP_i) \Rightarrow (RN)) \Rightarrow (RN') \\ \mathcal{E} \vdash B_i[RN'] \xrightarrow{\mu(RN'')} B_i'\end{array}}{\mathcal{E} \vdash (\textbf{trap } \vec{X} \ (\vec{RP} : \vec{RT}) \textbf{ is } \vec{B} \textbf{ in } B) \xrightarrow{\mu(RN'')} B_i'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B'}{\mathcal{E} \vdash (\textbf{trap } \vec{X} \ (\vec{RP} : \vec{RT}) \textbf{ is } \vec{B} \textbf{ in } B) \xrightarrow{\varepsilon(d)} (\textbf{trap } \vec{X} \ (\vec{RP} : \vec{RT}) \textbf{ is } \vec{B} \textbf{ in } B')}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash B \xrightarrow{X_i(RN@d)} B' \\ \mathcal{E} \vdash ((RP_i) \Rightarrow (RN)) \Rightarrow (RN') \\ \mathcal{E} \vdash B_i[RN'] \xrightarrow{\varepsilon(d')} B_i'\end{array}}{\mathcal{E} \vdash (\textbf{trap } \vec{X} \ (\vec{RP} : \vec{RT}) \textbf{ is } \vec{B} \textbf{ in } B) \xrightarrow{\varepsilon(d+d')} B_i'}$$

9.2.18 General parallel

Abstract syntax

par [G#n(, G#n)*] **in**
  '[' [G(, G)*] ']' -> B(|| '[' [G(, G)*] ']' -> B)*

Static semantics

$$\frac{\begin{array}{l}\mathcal{C} \vdash G_1 \Rightarrow \textbf{gate}(RT_1) \;\cdots\; \mathcal{C} \vdash G_p \Rightarrow \textbf{gate}(RT_p) \\ \mathcal{C} \vdash \vec{G_1} \Rightarrow \textbf{gate}(\vec{RT_1}) \;\cdots\; \mathcal{C} \vdash \vec{G_m} \Rightarrow \textbf{gate}(\vec{RT_m}) \\ \mathcal{C} \vdash B_1 \Rightarrow \textbf{exit}(RT_1') \;\cdots\; \mathcal{C} \vdash B_m \Rightarrow \textbf{exit}(RT_m')\end{array}}{\begin{array}{l}\mathcal{C} \vdash (\textbf{par } G_1\#N_1, \ldots, G_p\#N_p \\ \quad [\vec{G_1}] \rightarrow B_1 \\ \quad \cdots \\ \quad || \, [\vec{G_m}] \rightarrow B_m) \Rightarrow \textbf{exit}(RT_1', \ldots, RT_m')\end{array}} \left[\begin{array}{c} RT_1' \ldots RT_m' \text{ have disjoint fields} \\ N_1 \leq |\{\vec{G_j} \mid G_1 \in \vec{G_j}, 1 \leq j \leq m\}| \\ \vdots \\ N_p \leq |\{\vec{G_j} \mid G_p \in \vec{G_j}, 1 \leq j \leq m\}| \end{array}\right]$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$$\frac{\begin{array}{l}\mathcal{C} \vdash G_1 \Rightarrow \mathbf{gate}(RT_1) \ \cdots \ \mathcal{C} \vdash G_p \Rightarrow \mathbf{gate}(RT_p) \\ \mathcal{C} \vdash \vec{G_1} \Rightarrow \mathbf{gate}(\vec{RT_1}) \ \cdots \ \mathcal{C} \vdash \vec{G_m} \Rightarrow \mathbf{gate}(\vec{RT_m}) \\ \mathcal{C} \vdash B_1 \Rightarrow \mathbf{exit}(RT'_1) \\ \quad \cdots \\ \mathcal{C} \vdash B_j \Rightarrow \mathbf{guarded}(RT'_j) \\ \quad \cdots \\ \mathcal{C} \vdash B_m \Rightarrow \mathbf{exit}(RT'_m)\end{array}}{\begin{array}{l}\mathcal{C} \vdash (\mathbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \quad [\vec{G_1}] \rightarrow B_1 \\ \quad \cdots \\ \quad [\,]\, [\vec{G_m}] \rightarrow B_m) \Rightarrow \mathbf{guarded}(RT'_1, \ldots, RT'_m)\end{array}} \left[\begin{array}{c} RT'_1 \ldots RT'_m \text{ have disjoint fields} \\ N_1 \leq |\{\vec{G_j} \mid G_1 \in \vec{G_j}, 1 \leq j \leq m\}| \\ \vdots \\ N_p \leq |\{\vec{G_j} \mid G_p \in \vec{G_j}, 1 \leq j \leq m\}| \end{array}\right]$$

where $\vec{G_i}$ is an abbreviation of $G_{i1}, \ldots, G_{in_i}$ and $\mathcal{C} \vdash \vec{G_i} \Rightarrow \mathbf{gate}(\vec{RT_i})$ is an abbreviation for $\mathcal{C} \vdash \vec{G_{i1}} \Rightarrow \mathbf{gate}(\vec{RT_{i1}})$, …, $\mathcal{C} \vdash \vec{G_{in_i}} \Rightarrow \mathbf{gate}(\vec{RT_{in_i}})$.

Here, we assume N ranges over natural numbers greater than zero.

Untimed dynamic semantics

$$\frac{\forall i \in \Sigma. \mathcal{E} \vdash B_i \xrightarrow{G_j(RN)} B'_i}{\begin{array}{l}\mathcal{E} \vdash (\mathbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \quad [\vec{G_1}] \rightarrow B_1 \ \cdots \ [\,]\, [\vec{G_m}] \rightarrow B_m) \\ \quad \xrightarrow{G_j(RN)} \quad (\mathbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \quad\quad [\vec{G_1}] \rightarrow B'_1 \ \cdots \ [\,]\, [\vec{G_m}] \rightarrow B'_m)\end{array}} \left[\begin{array}{c} \Sigma \subseteq \{1, \ldots, m\} \\ |\Sigma| = N_j \\ G_j \in \bigcap_{i \in \Sigma} \vec{G_i} \\ \forall i \notin \Sigma. B_i = B'_i \end{array}\right]$$

$$\frac{\forall i \in \Sigma. \mathcal{E} \vdash B_i \xrightarrow{G(RN)} B'_i}{\begin{array}{l}\mathcal{E} \vdash (\mathbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \quad [\vec{G_1}] \rightarrow B_1 \ \cdots \ [\,]\, [\vec{G_m}] \rightarrow B_m) \\ \quad \xrightarrow{G(RN)} \quad (\mathbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \quad\quad [\vec{G_1}] \rightarrow B'_1 \ \cdots \ [\,]\, [\vec{G_m}] \rightarrow B'_m)\end{array}} \left[\begin{array}{c} G \notin \{G_1, \ldots, G_p\} \\ \Sigma = \{i \mid G \in \vec{G_i}, 1 \leq i \leq m\} \\ \Sigma \neq \emptyset \\ \forall i \notin \Sigma. B_i = B'_i \end{array}\right]$$

$$\frac{\mathcal{E} \vdash B_j \xrightarrow{G(RN)} B'_j}{\begin{array}{l}\mathcal{E} \vdash (\mathbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \quad [\,]\, [\vec{G_1}] \rightarrow B_1 \ \cdots \ [\,]\, [\vec{G_j}] \rightarrow B_j \ \cdots \ [\,]\, [\vec{G_m}] \rightarrow B_m) \\ \quad \xrightarrow{\mathbf{i}()} \quad (\mathbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \quad\quad [\,]\, [\vec{G_1}] \rightarrow B_1 \ \cdots \ [\,]\, [\vec{G_j}] \rightarrow B'_j \ \cdots \ [\,]\, [\vec{G_m}] \rightarrow B_m)\end{array}} \left[G \notin \vec{G_j}\right]$$

$$\frac{\mathcal{E} \vdash B_j \xrightarrow{\mathbf{i}()} B'_j}{\begin{array}{l}\mathcal{E} \vdash (\mathbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \quad [\,]\, [\vec{G_1}] \rightarrow B_1 \ \cdots \ [\,]\, [\vec{G_j}] \rightarrow B_j \ \cdots \ [\,]\, [\vec{G_m}] \rightarrow B_m) \\ \quad \xrightarrow{\mathbf{i}()} \quad (\mathbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \quad\quad [\,]\, [\vec{G_1}] \rightarrow B_1 \ \cdots \ [\,]\, [\vec{G_j}] \rightarrow B'_j \ \cdots \ [\,]\, [\vec{G_m}] \rightarrow B_m)\end{array}}$$

$$\frac{\mathcal{E} \vdash B_j \xrightarrow{X(RN)} B'_j}{\begin{array}{l}\mathcal{E} \vdash (\mathbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \quad [\,]\, [\vec{G_1}] \rightarrow B_1 \ \cdots \ [\,]\, [\vec{G_j}] \rightarrow B_j \ \cdots \ [\,]\, [\vec{G_m}] \rightarrow B_m) \\ \quad \xrightarrow{\mathbf{i}()} \quad \mathbf{signal}\ X\ (RN); (\mathbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \quad\quad [\,]\, [\vec{G_1}] \rightarrow B_1 \ \cdots \ [\,]\, [\vec{G_j}] \rightarrow B'_j \ \cdots \ [\,]\, [\vec{G_m}] \rightarrow B_m)\end{array}}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN)} B_1' \ \cdots \ \mathcal{E} \vdash B_m \xrightarrow{\delta(RN)} B_m'}{\mathcal{E} \vdash (\textbf{par}\ G_1\#N_1, \ldots, G_p\#N_p\ [\vec{G_1}] \rightarrow B_1 \ \cdots\ ||\ [\vec{G_m}] \rightarrow B_m) \xrightarrow{\delta(RN)} (\textbf{par}\ G_1\#N_1, \ldots, G_p\#N_p\ [\vec{G_1}] \rightarrow B_1' \ \cdots\ ||\ [\vec{G_m}] \rightarrow B_m')}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B_1' \ \cdots \ \mathcal{E} \vdash B_m \xrightarrow{\varepsilon(d)} B_m'}{\mathcal{E} \vdash (\textbf{par}\ G_1\#N_1, \ldots, G_p\#N_p\ [\vec{G_1}] \rightarrow B_1 \ \cdots\ ||\ [\vec{G_m}] \rightarrow B_m) \xrightarrow{\varepsilon(d)} (\textbf{par}\ G_1\#N_1, \ldots, G_p\#N_p\ [\vec{G_1}] \rightarrow B_1' \ \cdots\ ||\ [\vec{G_m}] \rightarrow B_m')}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d+d')} B_1' \ \cdots \ \mathcal{E} \vdash B_j \xrightarrow{\delta(RN@d)} B_j' \ \cdots \ \mathcal{E} \vdash B_m \xrightarrow{\varepsilon(d+d')} B_m'}{\begin{array}{l}\mathcal{E} \vdash (\textbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \quad ||\ [\vec{G_1}] \rightarrow B_1 \ \cdots\ ||\ [\vec{G_j}] \rightarrow B_j \ \cdots\ ||\ [\vec{G_m}] \rightarrow B_m) \\ \quad \xrightarrow{\varepsilon(d+d')} (\textbf{par}\ G_1\#N_1, \ldots, G_p\#N_p \\ \qquad ||\ [\vec{G_1}] \rightarrow B_1 \ \cdots\ ||\ [\vec{G_j}] \rightarrow \textbf{exit}(RN) \ \cdots\ ||\ [\vec{G_m}] \rightarrow B_m)\end{array}} [0 < d']$$

9.2.19 Parallel over values

Abstract syntax

$$\textbf{par}\ P\ \textbf{in}\ N\ |||\ B$$

Static Semantics

$$\frac{\mathcal{C} \vdash N \Rightarrow \text{List} \quad \mathcal{C} \vdash (P \Rightarrow \textbf{any}) \Rightarrow (RT) \quad \mathcal{C}, RT \vdash B \Rightarrow \textbf{guarded}(\textbf{none})}{\mathcal{C} \vdash \textbf{par}\ P\ \textbf{in}\ N\ |||\ B \Rightarrow \textbf{guarded}(\textbf{none})}$$

$$\frac{\mathcal{C} \vdash N \Rightarrow \text{List} \quad \mathcal{C} \vdash (P \Rightarrow \textbf{any}) \Rightarrow (RT) \quad \mathcal{C}, RT \vdash B \Rightarrow \textbf{exit}(\textbf{none})}{\mathcal{C} \vdash \textbf{par}\ P\ \textbf{in}\ N\ |||\ B \Rightarrow \textbf{exit}(\textbf{none})}$$

$$\frac{\mathcal{C} \vdash N \Rightarrow \text{List} \quad \mathcal{C} \vdash (P \Rightarrow \textbf{any}) \Rightarrow (RT) \quad \mathcal{C}, RT \vdash B \Rightarrow \textbf{guarded}()}{\mathcal{C} \vdash \textbf{par}\ P\ \textbf{in}\ N\ |||\ B \Rightarrow \textbf{guarded}()}$$

$$\frac{\mathcal{C} \vdash N \Rightarrow \text{List} \quad \mathcal{C} \vdash (P \Rightarrow \textbf{any}) \Rightarrow (RT) \quad \mathcal{C}, RT \vdash B \Rightarrow \textbf{exit}()}{\mathcal{C} \vdash \textbf{par}\ P\ \textbf{in}\ N\ |||\ B \Rightarrow \textbf{exit}()}$$

Untimed dynamic semantics

$$\frac{}{\mathcal{E} \vdash \textbf{par}\ P\ \textbf{in}\ \text{nil}\ |||\ B \xrightarrow{\delta()} \textbf{block}}$$

$$\frac{\mathcal{E} \vdash (P \Rightarrow N_1) \Rightarrow (RN) \quad \mathcal{E} \vdash B[RN]\ |||\ (\textbf{par}\ P\ \textbf{in}\ N_2\ |||\ B) \xrightarrow{\mu(RN')} B'}{\mathcal{E} \vdash \textbf{par}\ P\ \textbf{in}\ \text{cons}(N_1, N_2)\ |||\ B \xrightarrow{\mu(RN')} B'} [\ \text{here}\ \mu ::= G \mid \mathbf{i} \mid \delta\ ]$$

$$\frac{\mathcal{E} \vdash (P \Rightarrow N_1) \Rightarrow (RN) \quad \mathcal{E} \vdash B[RN]\ |||\ (\textbf{par}\ P\ \textbf{in}\ N_2\ |||\ B) \xrightarrow{X(RN')} B'}{\mathcal{E} \vdash \textbf{par}\ P\ \textbf{in}\ \text{cons}(N_1, N_2)\ |||\ B \xrightarrow{\mathbf{i}()} \textbf{signal}\ X\ (RN')\,; B'}$$

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Timed dynamic semantics

$$\frac{}{\mathcal{E} \vdash \textbf{par}\ P\ \textbf{in}\ \text{nil}\ |||\ B \xrightarrow{\varepsilon(0)} \textbf{block}}$$

$$\frac{\mathcal{E} \vdash (P \Rightarrow N_1) \Rightarrow (RN) \quad \mathcal{E} \vdash B[RN]\ |||\ (\textbf{par}\ P\ \textbf{in}\ N_2\ |||\ B) \xrightarrow{\varepsilon(d)} B'}{\mathcal{E} \vdash \textbf{par}\ P\ \textbf{in}\ \text{cons}(N_1, N_2)\ |||\ B \xrightarrow{\varepsilon(d)} B'}$$

9.2.20 Variable declaration

Abstract syntax

$\textbf{var}\ V_1 : T_1, \ldots, V_n : T_n\ \textbf{in}\ B$

Static semantics

$$\frac{\mathcal{C}; (V_1 \Rightarrow ?T_1); \ldots; (V_n \Rightarrow ?T_n) \vdash B \Rightarrow \textbf{exit}(RT, RT')}{\mathcal{C} \vdash \textbf{var}\ \overrightarrow{V:T}\ \textbf{in}\ B \Rightarrow \textbf{exit}(RT)} \left[\begin{array}{l} \text{every field name in } RT \text{ is not in } \vec{V} \text{ and} \\ \text{every field name in } RT' \text{ is in } \vec{V} \end{array} \right]$$

$$\frac{\mathcal{C}; (V_1 \Rightarrow ?T_1); \ldots; (V_n \Rightarrow ?T_n) \vdash B \Rightarrow \textbf{guarded}(RT, RT')}{\mathcal{C} \vdash \textbf{var}\ \overrightarrow{V:T}\ \textbf{in}\ B \Rightarrow \textbf{guarded}(RT)} \left[\begin{array}{l} \text{every field name in } RT \text{ is not in } \vec{V} \text{ and} \\ \text{every field name in } RT' \text{ is in } \vec{V} \end{array} \right]$$

Note that $V_i \Rightarrow ?T_i$ informs about the type of V_i, but does not implicate a value.

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{a(RN)} B'}{\mathcal{E} \vdash \textbf{var}\ \overrightarrow{V:T}\ \textbf{in}\ B \xrightarrow{a(RN)} \textbf{var}\ \overrightarrow{V:T}\ \textbf{in}\ B'}$$

$$\frac{\mathcal{E} \vdash B \xrightarrow{\delta(RN, RN')} B'}{\mathcal{E} \vdash \textbf{var}\ \overrightarrow{V:T}\ \textbf{in}\ B \xrightarrow{\delta(RN)} \textbf{block}} \left[\begin{array}{l} \text{every field name in } RN \text{ is not in } \vec{V} \text{ and} \\ \text{every field name in } RN' \text{ is in } \vec{V} \end{array} \right]$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B'}{\mathcal{E} \vdash \textbf{var}\ \overrightarrow{V:T}\ \textbf{in}\ B \xrightarrow{\varepsilon(d)} \textbf{var}\ \overrightarrow{V:T}\ \textbf{in}\ B'}$$

9.2.21 Gate hiding

Abstract syntax

$\textbf{hide}\ G[:T](,G[:T])^*\ \textbf{in}\ B$

The default gate type is **any**.

Static semantics

$$\frac{\begin{array}{c}\mathcal{C} \vdash T_1 \Rightarrow \textbf{type}\ \cdots\ \mathcal{C} \vdash T_n \Rightarrow \textbf{type} \\ \mathcal{C}; G_1 \Rightarrow \textbf{gate}(T_1), \ldots, G_n \Rightarrow \textbf{gate}(T_n) \vdash B \Rightarrow \textbf{exit}(RT)\end{array}}{\mathcal{C} \vdash \textbf{hide}\ G_1 : T_1, \ldots, G_n : T_n\ \textbf{in}\ B \Rightarrow \textbf{exit}(RT)}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash T_1 \Rightarrow \textbf{type}\ \cdots\ \mathcal{C} \vdash T_n \Rightarrow \textbf{type} \\ \mathcal{C}; G_1 \Rightarrow \textbf{gate}(T_1), \ldots, G_n \Rightarrow \textbf{gate}(T_n) \vdash B \Rightarrow \textbf{guarded}(RT)\end{array}}{\mathcal{C} \vdash \textbf{hide}\ G_1 : T_1, \cdots, G_n : T_n\ \textbf{in}\ B \Rightarrow \textbf{guarded}(RT)}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B'}{\mathcal{E} \vdash \textbf{hide } \overrightarrow{G:T} \textbf{ in } B' \xrightarrow{\mu(RN)} \textbf{hide } \overrightarrow{G:T} \textbf{ in } B'} \left[\mu \notin \vec{G}\right]$$

$$\frac{\mathcal{E} \vdash B \xrightarrow{G_i(RN)} B' \quad \mathcal{E} \vdash (RN) \Rightarrow (RT_i)}{\mathcal{E} \vdash \textbf{hide } \overrightarrow{G:T} \textbf{ in } B' \xrightarrow{\mathbf{i}()} \textbf{hide } \overrightarrow{G:T} \textbf{ in } B'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B' \quad \mathcal{E} \vdash B \textbf{ refusing } (\vec{G:T}, d)}{\mathcal{E} \vdash \textbf{hide } \vec{G}(\vec{RT}) \textbf{ in } B' \xrightarrow{\varepsilon(d)} \textbf{hide } \vec{G:T} \textbf{ in } B'}$$

where $\mathcal{E} \vdash B$ **refusing** $(\vec{G:T}, d)$ if there is no $\mathcal{E} \vdash (RN) \Rightarrow (RT_i)$ and $d' < d$ such that $\mathcal{E} \vdash B \xrightarrow{G_i(RN@d')} B''$

Note that we use negative premise to define the timed dynamic semantics.

9.2.22 Renaming

Abstract syntax

rename ((**gate** $G[(IPL)]$ **is** G $[P]$) | (**signal** $X[(IPL)]$ **is** X $[E]$))* **in** B

The default gate input parameter list is **any**, the default gate pattern is !$argv, the default exception input parameter list is (), and the default exception value is $argv.

Patterns should be irrefutable.

Static semantics

$$\frac{\begin{array}{c}\mathcal{C} \vdash (RT_1) \Rightarrow \textbf{type} \ \cdots \ \mathcal{C} \vdash (RT_m) \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1'') \ \cdots \ \mathcal{C} \vdash (RP_m \Rightarrow RT_m) \Rightarrow (RT_m'') \\ \mathcal{C}; RT_1'' \vdash G_1' \ P_1 \Rightarrow \textbf{exit} \ () \ \cdots \ \mathcal{C}; RT_m'' \vdash G_m' \ P_m \Rightarrow \textbf{exit} \ () \\ \mathcal{C} \vdash (RT_1') \Rightarrow \textbf{type} \ \cdots \ \mathcal{C} \vdash (RT_m') \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (RP_1' \Rightarrow RT_1') \Rightarrow (RT_1''') \ \cdots \ \mathcal{C} \vdash (RP_n' \Rightarrow RT_n') \Rightarrow (RT_n''') \\ \mathcal{C}; RT_1''' \vdash \textbf{signal } X_1' \ E_1 \Rightarrow \textbf{exit} \ () \ \cdots \ \mathcal{C}; RT_n''' \vdash \textbf{signal } X_n' \ E_n \Rightarrow \textbf{exit} \ () \\ \mathcal{C}; G_1 \Rightarrow \textbf{gate}(RT_1) \odot \\ \ldots \odot G_m \Rightarrow \textbf{gate}(RT_m) \\ \odot X_1 \Rightarrow \textbf{exn}(RT_1') \odot \\ \ldots \odot X_m \Rightarrow \textbf{exn}(RT_n') \vdash B \Rightarrow \textbf{exit}(RT)\end{array}}{\begin{array}{l}\mathcal{C} \vdash (\textbf{rename} \\ \quad \textbf{gate } G_1(RP_1 : RT_1) \textbf{ is } G_1' \ P_1 \ \cdots \ \textbf{gate } G_m(RP_m : RT_m) \textbf{ is } G_m' \ P_m \\ \quad \textbf{signal } X_1(RP_1' : RT_1') \textbf{ is } X_1' \ E_1 \ \cdots \ \textbf{signal } X_m(RP_n' : RT_n') \textbf{ is } X_n' \ E_n \\ \textbf{in } B) \Rightarrow \textbf{exit}(RT)\end{array}}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$$\frac{\begin{array}{c}\mathcal{C} \vdash (RT_1) \Rightarrow \textbf{type} \;\cdots\; \mathcal{C} \vdash (RT_m) \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1'') \;\cdots\; \mathcal{C} \vdash (RP_m \Rightarrow RT_m) \Rightarrow (RT_m'') \\ \mathcal{C}; RT_1'' \vdash G_1'\ P_1 \Rightarrow \textbf{exit}\ () \;\cdots\; \mathcal{C}; RT_m'' \vdash G_m'\ P_m \Rightarrow \textbf{exit}\ () \\ \mathcal{C} \vdash (RT_1') \Rightarrow \textbf{type} \;\cdots\; \mathcal{C} \vdash (RT_m') \Rightarrow \textbf{type} \\ \mathcal{C} \vdash (RP_1' \Rightarrow RT_1') \Rightarrow (RT_1''') \;\cdots\; \mathcal{C} \vdash (RP_n' \Rightarrow RT_n') \Rightarrow (RT_n''') \\ \mathcal{C}; RT_1''' \vdash \textbf{signal}\ X_1'\ E_1 \Rightarrow \textbf{exit}\ () \;\cdots\; \mathcal{C}; RT_n''' \vdash \textbf{signal}\ X_n'\ E_n \Rightarrow \textbf{exit}\ () \\ \mathcal{C}; G_1 \Rightarrow \textbf{gate}(RT_1) \odot \\ \ldots \odot G_m \Rightarrow \textbf{gate}(RT_m) \\ \quad \odot X_1 \Rightarrow \textbf{exn}(RT_1') \odot \\ \ldots \odot X_m \Rightarrow \textbf{exn}(RT_n') \vdash B \Rightarrow \textbf{guarded}(RT)\end{array}}{\begin{array}{l}\mathcal{C} \vdash (\textbf{rename} \\ \qquad \textbf{gate}\ G_1(RP_1 : RT_1)\ \textbf{is}\ G_1'\ P_1 \;\cdots\; \textbf{gate}\ G_m(RP_m : RT_m)\ \textbf{is}\ G_m'\ P_m \\ \qquad \textbf{signal}\ X_1(RP_1' : RT_1')\ \textbf{is}\ X_1'\ E_1 \;\cdots\; \textbf{signal}\ X_m(RP_n' : RT_n')\ \textbf{is}\ X_n'\ E_n \\ \quad \textbf{in}\ B) \Rightarrow \textbf{guarded}(RT)\end{array}}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B'}{\mathcal{E} \vdash (\textbf{rename} \xrightarrow{\mu(RN)} (\textbf{rename}} \quad [\mu \notin \{G_1, \ldots, G_m, X_1, \ldots, X_m\}]$$

$$\begin{array}{ll}\textbf{gate}\ G_1(RP_1 : RT_1)\ \textbf{is}\ G_1'\ P_1 & \textbf{gate}\ G_1(RP_1 : RT_1)\ \textbf{is}\ G_1'\ P_1 \\ \cdots & \cdots \\ \textbf{gate}\ G_m(RP_m : RT_m)\ \textbf{is}\ G_m'\ P_m & \textbf{gate}\ G_m(RP_m : RT_m)\ \textbf{is}\ G_m'\ P_m \\ \textbf{signal}\ X_1(RP_1' : RT_1')\ \textbf{is}\ X_1'\ E_1 & \textbf{signal}\ X_1(RP_1' : RT_1')\ \textbf{is}\ X_1'\ E_1 \\ \cdots & \cdots \\ \textbf{signal}\ X_m(RP_n' : RT_n')\ \textbf{is}\ X_n'\ E_n & \textbf{signal}\ X_m(RP_n' : RT_n')\ \textbf{is}\ X_n'\ E_n \\ \textbf{in}\ B) & \textbf{in}\ B')\end{array}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash B \xrightarrow{a_i(RN)} B' \\ \mathcal{E} \vdash (((RP_i) : (RT_i)) \Rightarrow (RN)) \Rightarrow (RN') \\ \mathcal{E} \vdash B_i[RN'] \xrightarrow{a(RN'')} B_i'\end{array}}{\mathcal{E} \vdash (\textbf{rename} \xrightarrow{a(RN'')} (\textbf{rename}} \quad [a_i \in \{G_1, \ldots, G_m, X_1, \ldots, X_m\}]$$

$$\begin{array}{ll}\textbf{gate}\ G_1(RP_1 : RT_1)\ \textbf{is}\ G_1'\ P_1 & \textbf{gate}\ G_1(RP_1 : RT_1)\ \textbf{is}\ G_1'\ P_1 \\ \cdots & \cdots \\ \textbf{gate}\ G_m(RP_m : RT_m)\ \textbf{is}\ G_m'\ P_m & \textbf{gate}\ G_m(RP_m : RT_m)\ \textbf{is}\ G_m'\ P_m \\ \textbf{signal}\ X_1(RP_1' : RT_1')\ \textbf{is}\ X_1'\ E_1 & \textbf{signal}\ X_1(RP_1' : RT_1')\ \textbf{is}\ X_1'\ E_1 \\ \cdots & \cdots \\ \textbf{signal}\ X_m(RP_n' : RT_n')\ \textbf{is}\ X_n'\ E_n & \textbf{signal}\ X_m(RP_n' : RT_n')\ \textbf{is}\ X_n'\ E_n \\ \textbf{in}\ B) & \textbf{in}\ B')\end{array}$$

where $B_i = G_i'\ P_i$ in the case of gates and $B_i = \textbf{signal}\ X_i'\ E_i$ in the case of exceptions.

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B'}{\mathcal{E} \vdash (\textbf{rename} \xrightarrow{\varepsilon(d)} (\textbf{rename}}$$

$$\begin{array}{ll}\textbf{gate}\ G_1(RP_1 : RT_1)\ \textbf{is}\ G_1'\ P_1 \;\cdots & \textbf{gate}\ G_1(RP_1 : RT_1)\ \textbf{is}\ G_1'\ P_1 \;\cdots \\ \textbf{gate}\ G_m(RP_m : RT_m)\ \textbf{is}\ G_m'\ P_m & \textbf{gate}\ G_m(RP_m : RT_m)\ \textbf{is}\ G_m'\ P_m \\ \textbf{signal}\ X_1(RP_1' : RT_1')\ \textbf{is}\ X_1'\ E_1 \;\cdots & \textbf{signal}\ X_1(RP_1' : RT_1')\ \textbf{is}\ X_1'\ E_1 \;\cdots \\ \textbf{signal}\ X_m(RP_n' : RT_n')\ \textbf{is}\ X_n'\ E_n & \textbf{signal}\ X_m(RP_n' : RT_n')\ \textbf{is}\ X_n'\ E_n \\ \textbf{in}\ B) & \textbf{in}\ B)\end{array}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2.23 Process instantiation

Abstract syntax

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

The default gate and exception lists are the empty list `[]`, and the default argument is `()`.

Static semantics

$$\frac{\begin{array}{l}\mathcal{C}\vdash\Pi\Rightarrow[\mathbf{gate}(RT_1),\ldots,\mathbf{gate}(RT_m)]\,(RT')\,[\mathbf{exn}(RT'_1),\ldots,\mathbf{exn}(RT'_n)]\rightarrow\mathbf{exit}(RT)\\ \mathcal{C}\vdash G_1\Rightarrow\mathbf{gate}(RT_1)\ \cdots\ \mathcal{C}\vdash G_m\Rightarrow\mathbf{gate}(RT_m)\\ \mathcal{C}\vdash E_1\Rightarrow\mathbf{exit}(T_1)\ \ldots\ \mathcal{C}\vdash E_p\Rightarrow\mathbf{exit}(T_p)\\ \mathcal{C}\vdash X_1\Rightarrow\mathbf{exn}(RT'_1)\ \cdots\ \mathcal{C}\vdash X_n\Rightarrow\mathbf{exn}(RT'_n)\end{array}}{\mathcal{C}\vdash\Pi\ [G_1,\ldots,G_m]\ (E_1,\ldots,E_p)\ [X_1,\ldots,X_n]\Rightarrow\mathbf{exit}(RT)}\ [RT'=T_1,\ldots,T_p]$$

$$\frac{\begin{array}{l}\mathcal{C}\vdash\Pi\Rightarrow[\mathbf{gate}(RT_1),\ldots,\mathbf{gate}(RT_m)]\,(RT')\,[\mathbf{exn}(RT'_1),\ldots,\mathbf{exn}(RT'_n)]\rightarrow\mathbf{guarded}(RT)\\ \mathcal{C}\vdash G_1\Rightarrow\mathbf{gate}(RT_1)\ \cdots\ \mathcal{C}\vdash G_m\Rightarrow\mathbf{gate}(RT_m)\\ \mathcal{C}\vdash(E_1,\ldots,E_p)\Rightarrow\mathbf{exit}((RT'))\\ \mathcal{C}\vdash X_1\Rightarrow\mathbf{exn}(RT'_1)\ \cdots\ \mathcal{C}\vdash X_n\Rightarrow\mathbf{exn}(RT'_n)\end{array}}{\mathcal{C}\vdash\Pi\ [G_1,\ldots,G_m]\ (E_1,\ldots,E_p)\ [X_1,\ldots,X_n]\Rightarrow\mathbf{guarded}(RT)}$$

Untimed dynamic semantics

$$\frac{\begin{array}{l}\mathcal{E}\vdash\Pi\Rightarrow\lambda\,[\vec{G}'(\vec{RT})]\,(RP:RT)\,[\vec{X}'(\vec{RT}')]\rightarrow B\\ \mathcal{E}\vdash(\mathbf{rename}\\ \qquad\mathbf{gate}\ G'_1(RP_1:RT_1)\ \mathbf{is}\ G'_1\ P_1\ \cdots\mathbf{gate}\ G'_m(RP_m:RT_m)\ \mathbf{is}\ G'_m\ P_m\\ \qquad\mathbf{signal}\ X'_1(RP'_1:RT'_1)\ \mathbf{is}\ X'_1\ E_1\ \cdots\mathbf{signal}\ X'_n(RP'_n:RT'_n)\ \mathbf{is}\ X'_n\ E_n\\ \quad\mathbf{in\ case}\ \vec{E}:(RT)\ \mathbf{is}\ (RP)\rightarrow B\xrightarrow{\mu(RN)}B'\end{array}}{\mathcal{E}\vdash\Pi\ [\vec{G}]\ \vec{E}\ [\vec{X}]\xrightarrow{\mu(RN)}B'}$$

Timed dynamic semantics

$$\frac{\begin{array}{l}\mathcal{E}\vdash\Pi\Rightarrow\lambda\,[\vec{G}'(\vec{RT})]\,(RP:RT)\,[\vec{X}'(\vec{RT}')]\rightarrow B\\ \mathcal{E}\vdash((\mathbf{rename}\\ \qquad\mathbf{gate}\ G'_1(RP_1:RT_1)\ \mathbf{is}\ G'_1\ P_1\ \cdots\mathbf{gate}\ G'_m(RP_m:RT_m)\ \mathbf{is}\ G'_m\ P_m\\ \qquad\mathbf{signal}\ X'_1(RP'_1:RT'_1)\ \mathbf{is}\ X'_1\ E_1\ \cdots\mathbf{signal}\ X'_n(RP'_n:RT'_n)\ \mathbf{is}\ X'_n\ E_n\\ \quad\mathbf{in\ case}\ \vec{E}:(RT)\ \mathbf{is}\ (RP)\rightarrow B)\xrightarrow{\varepsilon(d)}B'\end{array}}{\mathcal{E}\vdash\Pi\ [\vec{G}]\ \vec{E}\ [\vec{X}]\xrightarrow{\varepsilon(d)}B'}$$

9.2.24 loop iteration

Abstract syntax

loop B

Static semantics

$$\frac{\mathcal{C}\vdash B\Rightarrow\mathbf{exit}(RT)}{\mathcal{C}\vdash\mathbf{loop}\ B\Rightarrow\mathbf{guarded}(\mathbf{none})}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B;\ \textbf{loop}\ B \xrightarrow{\mu(RN)} B'}{\mathcal{E} \vdash \textbf{loop}\ B \xrightarrow{\mu(RN)} B'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B;\ \textbf{loop}\ B \xrightarrow{\varepsilon(d)} B'}{\mathcal{E} \vdash \textbf{loop}\ B \xrightarrow{\varepsilon(d)} B'}$$

9.2.25 Case

Abstract syntax

case $E[:T]$ **is** BM **endcase**

The default type is the principal type of E (*note* this requires static information).

Static semantics

$$\frac{\mathcal{C} \vdash E \Rightarrow \textbf{exit}(T) \quad \mathcal{C} \vdash (BM \Rightarrow T) \Rightarrow \textbf{exit}(RT)}{\mathcal{C} \vdash \textbf{case}\ E:T\ \textbf{is}\ BM \Rightarrow \textbf{exit}(RT)}$$

$$\frac{\mathcal{C} \vdash E \Rightarrow \textbf{exit}(T) \quad \mathcal{C} \vdash (BM \Rightarrow T) \Rightarrow \textbf{guarded}(RT)}{\mathcal{C} \vdash \textbf{case}\ E:T\ \textbf{is}\ BM \Rightarrow \textbf{guarded}(RT)}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(N)} E' \quad \mathcal{E} \vdash (BM \Rightarrow N) \xrightarrow{\mu(RN)} B}{\mathcal{E} \vdash \textbf{case}\ E:T\ \textbf{is}\ BM \xrightarrow{\mu(RN)} B}$$

$$\frac{\mathcal{E} \vdash E \xrightarrow{X(RN)} E'}{\mathcal{E} \vdash \textbf{case}\ E:T\ \textbf{is}\ BM \xrightarrow{X(RN)} \textbf{case}\ E':T\ \textbf{is}\ BM}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(N)} E' \quad \mathcal{E} \vdash (BM \Rightarrow N) \xrightarrow{\varepsilon(d)} B}{\mathcal{E} \vdash \textbf{case}\ E:T\ \textbf{is}\ BM \xrightarrow{\varepsilon(d)} B}$$

9.2.26 Case with tuples

Abstract syntax

case '(' $E(,E)^*$ ')' **is**
  BM
endcase

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Static semantics

$$\frac{\mathcal{C} \vdash (E_1,\ldots,E_n) \Rightarrow \mathbf{exit}(RT) \quad \mathcal{C} \vdash (BM \Rightarrow (RT)) \Rightarrow \mathbf{exit}(RT')}{\mathcal{C} \vdash \mathbf{case}\ (E_1,\ldots,E_n)\ \mathbf{is}\ BM \Rightarrow \mathbf{exit}(RT')}$$

$$\frac{\mathcal{C} \vdash (E_1,\ldots,E_n) \Rightarrow \mathbf{exit}(RT) \quad \mathcal{C} \vdash (BM \Rightarrow (RT)) \Rightarrow \mathbf{guarded}(RT')}{\mathcal{C} \vdash \mathbf{case}\ (E_1,\ldots,E_n)\ \mathbf{is}\ BM \Rightarrow \mathbf{guarded}(RT')}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash E_1,\ldots,E_n \xrightarrow{\delta(RN)} E'_1,\ldots,E'_n \quad \mathcal{E} \vdash (BM \Rightarrow (RN)) \xrightarrow{\mu(RN')} B}{\mathcal{E} \vdash \mathbf{case}\ (E_1,\ldots,E_n)\ \mathbf{is}\ BM \xrightarrow{\mu(RN')} B}$$

$$\frac{\mathcal{E} \vdash E_j \xrightarrow{X(N)} E'_j}{\mathcal{E} \vdash \mathbf{case}\ (E_1,\ldots,E_n)\ \mathbf{is}\ BM \xrightarrow{X(N)} \mathbf{case}\ (E'_1,\ldots,E'_n)\ \mathbf{is}\ BM}\ [\text{with } E'_i = E_i \text{ if } i \neq j]$$

Note that $E_1,\ldots,E_n \xrightarrow{\delta(RN)} E'_1,\ldots,E'_n$ corresponds intuitively to parallel evaluation. Anycase, evaluation of E_i cannot produce bindings, so they may be evaluate in any arbitrary order.

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash E_1,\ldots,E_n \xrightarrow{\delta(RN)} E'_1,\ldots,E'_n \quad \mathcal{E} \vdash (BM \Rightarrow (RN)) \xrightarrow{\varepsilon(d)} B}{\mathcal{E} \vdash \mathbf{case}\ (E_1,\ldots,E_n)\ \mathbf{is}\ BM \xrightarrow{\varepsilon(d)} B}$$

9.2.27 Signalling

Abstract syntax

signal X ['(' E ')']

The default expression is **exit**().

Static semantics

$$\frac{\mathcal{C} \vdash E \Rightarrow \mathbf{exit}((RT)) \quad \mathcal{C} \vdash X \Rightarrow \mathbf{exn}(RT)}{\mathcal{C} \vdash \mathbf{signal}\ X\ E \Rightarrow \mathbf{guarded}()}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta((RN))} E'}{\mathcal{E} \vdash \mathbf{signal}\ X\ E \xrightarrow{X(RN)} \mathbf{exit}()}$$

$$\frac{\mathcal{E} \vdash E \xrightarrow{X'(RN)} E'}{\mathcal{E} \vdash \mathbf{signal}\ X\ E \xrightarrow{X'(RN)} \mathbf{signal}\ X\ E'}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Timed dynamic semantics None.

9.3 Type expressions

9.3.1 Type identifier

Abstract syntax

S

Static semantics

$$\overline{\mathcal{C}, S \Rightarrow \textbf{type} \vdash S \Rightarrow \textbf{type}}$$

$$\overline{\mathcal{C}, S \equiv T \vdash S \equiv T}$$

9.3.2 Empty type

Abstract syntax

none

Static semantics

$$\overline{\mathcal{C} \vdash \textbf{none} \Rightarrow \textbf{type}}$$

$$\overline{\mathcal{C} \vdash \textbf{none} \sqsubseteq T}$$

$$\overline{\mathcal{C} \vdash \textbf{none} \equiv (V \Rightarrow \textbf{none}, RT)}$$

9.3.3 Universal type

Abstract syntax

any

Static semantics

$$\overline{\mathcal{C} \vdash \textbf{any} \Rightarrow \textbf{type}}$$

$$\overline{\mathcal{C} \vdash T \sqsubseteq \textbf{any}}$$

9.3.4 Record type

Abstract syntax

(RT)

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Static semantics

$$\frac{\mathcal{C} \vdash RT \Rightarrow \textbf{record}}{\mathcal{C} \vdash (RT) \Rightarrow \textbf{type}}$$

$$\frac{\mathcal{C} \vdash RT \sqsubseteq RT'}{\mathcal{C} \vdash (RT) \sqsubseteq (RT')}$$

9.4 Record type expressions

Rules for record type expressions will be given in a compositional way, given first the semantics for items **etc** and $V => T$ and then the semantic for disjoint union (RT, RT).

9.4.1 Singleton record

Abstract syntax

$$V \Rightarrow T$$

Static semantics

$$\frac{\mathcal{C} \vdash T \Rightarrow \textbf{type}}{\mathcal{C} \vdash (V \Rightarrow T) \Rightarrow \textbf{record}}$$

$$\frac{\mathcal{C} \vdash T \sqsubseteq T'}{\mathcal{C} \vdash (V \Rightarrow T) \sqsubseteq (V \Rightarrow T')}$$

9.4.2 Universal record

Abstract syntax

etc

Static semantics

$$\frac{}{\mathcal{C} \vdash \textbf{etc} \Rightarrow \textbf{record}}$$

$$\frac{}{\mathcal{C} \vdash RT \sqsubseteq \textbf{etc}}$$

9.4.3 Record disjoint union

Abstract syntax

$$RT, RT$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Static semantics

$$\frac{\mathcal{C} \vdash RT_1 \Rightarrow \textbf{record} \qquad \mathcal{C} \vdash RT_2 \Rightarrow \textbf{record}}{\mathcal{C} \vdash RT_1, RT_2 \Rightarrow \textbf{record}} [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

$$\frac{\mathcal{C} \vdash RT_1 \sqsubseteq RT_1' \qquad \mathcal{C} \vdash RT_2 \sqsubseteq RT_2'}{\mathcal{C} \vdash RT_1, RT_2 \sqsubseteq RT_1', RT_2'}$$

$$\overline{\mathcal{C} \vdash RT_1, RT_2 \equiv RT_2, RT_1}$$

$$\overline{\mathcal{C} \vdash (RT_1, RT_2), RT_3 \equiv RT_1, (RT_2, RT_3)}$$

$$\overline{\mathcal{C} \vdash (), RT \equiv RT}$$

9.4.4 Empty record

Abstract syntax

$()$

Static semantics

$$\overline{\mathcal{C} \vdash () \Rightarrow \textbf{record}}$$

9.5 Value expressions

9.5.1 Primitive constants

Abstract syntax

K

Static semantics In this chapter we will not discuss the static semantics of primitives—this is left to the design of the standard libraries.

9.5.2 Variables

Abstract syntax

V

Static semantics

$$\overline{\mathcal{C}, V \Rightarrow T \vdash V \Rightarrow T}$$

9.5.3 Record values

Abstract syntax

'(' *RN* ')'

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Static semantics

$$\frac{\mathcal{C} \vdash RN \Rightarrow RT}{\mathcal{C} \vdash (RN) \Rightarrow (RT)}$$

9.5.4 Constructor application

Abstract syntax

$C\ [N]$

The default argument is `()`.

Static semantics

$$\frac{\mathcal{C} \vdash C \Rightarrow ((RT) \rightarrow S) \quad \mathcal{C} \vdash N \Rightarrow (RT)}{\mathcal{C} \vdash C\ N \Rightarrow S}$$

9.6 Record value expressions

9.6.1 Singleton record

Abstract syntax

$V \Rightarrow N$

Static semantics

$$\frac{\mathcal{C} \vdash N \Rightarrow T}{\mathcal{C} \vdash (V \Rightarrow N) \Rightarrow (V \Rightarrow T)}$$

9.6.2 Record disjoint union

Abstract syntax

RN, RN

Static semantics

$$\frac{\mathcal{C} \vdash RN_1 \Rightarrow RT_1 \qquad \mathcal{C} \vdash RN_2 \Rightarrow RT_2}{\mathcal{C} \vdash RN_1, RN_2 \Rightarrow RT_1, RT_2}\ [RN_1 \text{ and } RN_2 \text{ have disjoint fields}]$$

9.6.3 Empty record

Abstract syntax

$()$

Static semantics

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

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.7 Patterns

In this section the semantics of pattern matching (the sole use of patterns) is defined.

9.7.1 Record pattern

Abstract syntax

'(' *RP* ')'

Static semantics

$$\frac{\mathcal{C} \vdash (RP \Rightarrow RT') \Rightarrow (RT)}{\mathcal{C} \vdash ((RP) \Rightarrow (RT')) \Rightarrow (RT)}$$

$$\frac{\mathcal{C} \vdash ((V_1 \Rightarrow P_1, \ldots, V_n \Rightarrow P_n) \Rightarrow (V_1 \Rightarrow \mathbf{any}, \ldots, V_n \Rightarrow \mathbf{any})) \Rightarrow (RT)}{\mathcal{C} \vdash ((V_1 \Rightarrow P_1, \ldots, V_n \Rightarrow P_n) \Rightarrow \mathbf{any}) \Rightarrow (RT)}$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash (RP \Rightarrow RN') \Rightarrow (RN)}{\mathcal{E} \vdash ((RP) \Rightarrow N) \Rightarrow (RN)} [N = (RN')]$$

$$\frac{\mathcal{E} \vdash (RP \Rightarrow RN') \Rightarrow \mathbf{fail}}{\mathcal{E} \vdash ((RP) \Rightarrow N) \Rightarrow \mathbf{fail}} [N = (RN')]$$

$$\frac{}{\mathcal{E} \vdash ((RP) \Rightarrow N) \Rightarrow \mathbf{fail}} [\nexists\ RN' \mid N = (RN')]$$

The last condition means that N is not a bracketer record of values.

9.7.2 Wildcard

Abstract syntax

$\mathbf{any} : T$

Static semantics

$$\frac{\mathcal{C} \vdash T \Rightarrow \mathbf{type}}{\mathcal{C} \vdash (\mathbf{any} : T \Rightarrow T) \Rightarrow ()}$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash N \Rightarrow T}{\mathcal{E} \vdash (\mathbf{any} : T \Rightarrow N) \Rightarrow ()}$$

9.7.3 Variable binding

Abstract syntax

?V

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Static semantics

$$\frac{\mathcal{C} \vdash T \sqsubseteq T'}{\mathcal{C} \vdash (?V \Rightarrow T) \Rightarrow (V \Rightarrow T)} [V \Rightarrow ?T' \in \mathcal{C}]$$

$$\frac{}{\mathcal{C} \vdash (?V \Rightarrow T) \Rightarrow (V \Rightarrow T)} [\nexists V \Rightarrow ?T' \mid V \Rightarrow ?T' \in \mathcal{C}]$$

If there exists a restriction via a local variable declaration, V may be a value of a subtype of T' (the declared type). If there is no variable declaration, any type would match T.

Dynamic semantics

$$\frac{\mathcal{E} \vdash N \Rightarrow T \quad \mathcal{E} \vdash T \sqsubseteq T'}{\mathcal{E} \vdash (?V \Rightarrow N) \Rightarrow (V \Rightarrow N)} [V \Rightarrow ?T' \in \mathcal{E}]$$

$$\frac{}{\mathcal{E} \vdash (?V \Rightarrow N) \Rightarrow (V \Rightarrow N)} [\nexists V \Rightarrow ?T' \mid V \Rightarrow ?T' \in \mathcal{E}]$$

9.7.4 Expression pattern

Abstract syntax

$$!EA$$

Static semantics

$$\frac{\mathcal{C} \vdash EA \Rightarrow \mathbf{exit}(T)}{\mathcal{C} \vdash (!EA \Rightarrow T) \Rightarrow ()}$$

Note that no bindings can be produced in EA.

Dynamic semantics

$$\frac{\mathcal{E} \vdash EA \xrightarrow{\delta(N')} EA'}{\mathcal{E} \vdash (!EA \Rightarrow N) \Rightarrow ()} [N = N']$$

$$\frac{\mathcal{E} \vdash EA \xrightarrow{\delta(N')} EA'}{\mathcal{E} \vdash (!EA \Rightarrow N) \Rightarrow \mathbf{fail}} [N \neq N']$$

$$\frac{\mathcal{E} \vdash EA \xrightarrow{X(RN)} EA'}{\mathcal{E} \vdash (!EA \Rightarrow N) \Rightarrow \mathbf{fail}}$$

9.7.5 Constructor application

Abstract syntax

$$C\ [P]$$

The default pattern is `()`.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Static semantics

$$\mathcal{C} \vdash C \Rightarrow (RT) \rightarrow S$$
$$\mathcal{C} \vdash S \sqsubseteq T$$
$$\frac{\mathcal{C} \vdash (P \Rightarrow (RT)) \Rightarrow (RT')}{\mathcal{C} \vdash (C\ P \Rightarrow T) \Rightarrow (RT')}$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash (P \Rightarrow (RN)) \Rightarrow (RN')}{\mathcal{E} \vdash (C\ P \Rightarrow N) \Rightarrow (RN')}\ [N = C(RN)]$$

$$\frac{\mathcal{E} \vdash (P \Rightarrow (RN)) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash (C\ P \Rightarrow N) \Rightarrow \textbf{fail}}\ [N = C(RN)]$$

$$\frac{}{\mathcal{E} \vdash (C\ P \Rightarrow N) \Rightarrow \textbf{fail}}\ [\nexists\ C\ N' \mid N = C\ N']$$

9.7.6 Explicit typing

Abstract syntax

$$P : T$$

Static semantics

$$\mathcal{C} \vdash T \Rightarrow \textbf{type}$$
$$\mathcal{C} \vdash T \sqsubseteq T'$$
$$\frac{\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT)}{\mathcal{C} \vdash (P : T \Rightarrow T') \Rightarrow (RT)}$$

Dynamic semantics

$$\mathcal{E} \vdash N \Rightarrow T$$
$$\frac{\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN)}{\mathcal{E} \vdash (P : T \Rightarrow N) \Rightarrow (RN)}$$

$$\mathcal{E} \vdash N \Rightarrow T$$
$$\frac{\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash (P : T \Rightarrow N) \Rightarrow \textbf{fail}}$$

$$\mathcal{E} \vdash N \Rightarrow T'$$
$$\frac{\mathcal{E} \vdash T \sqcap T' \Rightarrow \textbf{none}}{\mathcal{E} \vdash (P : T \Rightarrow N) \Rightarrow \textbf{fail}}$$

9.8 Record patterns

Static semantics

$$\mathcal{C} \vdash (RP \Rightarrow RT) \Rightarrow (RT')$$

Dynamic semantics

$$\mathcal{E} \vdash (RP \Rightarrow RN) \Rightarrow (RN')$$

$$\mathcal{E} \vdash (RP \Rightarrow RN) \Rightarrow \textbf{fail}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.8.1 Singleton record pattern

Abstract syntax

$V \Rightarrow P$

Static semantics

$$\frac{\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT)}{\mathcal{C} \vdash ((V \Rightarrow P) \Rightarrow (V \Rightarrow T)) \Rightarrow (RT)}$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN)}{\mathcal{E} \vdash ((V \Rightarrow P) \Rightarrow (V \Rightarrow N)) \Rightarrow (RN)}$$

$$\frac{\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash ((V \Rightarrow P) \Rightarrow (V \Rightarrow N)) \Rightarrow \textbf{fail}}$$

9.8.2 Record wildcard

Abstract syntax

etc

Static semantics

$$\frac{}{\mathcal{C} \vdash (\textbf{etc} \Rightarrow RT) \Rightarrow ()}$$

Dynamic semantics

$$\frac{}{\mathcal{E} \vdash (\textbf{etc} \Rightarrow RN) \Rightarrow ()}$$

9.8.3 Record match

Abstract syntax

P **as** RP

Static semantics

$$\frac{\mathcal{C} \vdash (P \Rightarrow (RT)) \Rightarrow (RT_1) \quad \mathcal{C} \vdash (RP \Rightarrow RT) \Rightarrow (RT_2)}{\mathcal{C} \vdash (P \textbf{ as } RP \Rightarrow RT) \Rightarrow (RT_1, RT_2)} \; [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash (P \Rightarrow (RN)) \Rightarrow (RN_1) \quad \mathcal{E} \vdash (RP \Rightarrow RN) \Rightarrow (RN_2)}{\mathcal{E} \vdash (P \textbf{ as } RP \Rightarrow RN) \Rightarrow (RN_1, RN_2)}$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.8.4 Record disjoint union

Abstract syntax

RP, RP

Static semantics

$$\frac{\mathcal{C} \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \qquad \mathcal{C} \vdash (RP_2 \Rightarrow RT_2) \Rightarrow (RT_2')}{\mathcal{C} \vdash (RP_1, RP_2 \Rightarrow RT_1, RT_2) \Rightarrow (RT_1', RT_2')} \; [RT_1' \text{ and } RT_2' \text{ have disjoint fields}]$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash (RP_1 \Rightarrow RN_1) \Rightarrow (RN_1') \qquad \mathcal{E} \vdash (RP_2 \Rightarrow RN_2) \Rightarrow (RN_2')}{\mathcal{E} \vdash (RP_1, RP_2 \Rightarrow RN_1, RN_2) \Rightarrow (RN_1', RN_2')}$$

$$\frac{\mathcal{E} \vdash (RP_1 \Rightarrow RN_1) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash (RP_1, RP_2 \Rightarrow RN_1, RN_2) \Rightarrow \textbf{fail}}$$

$$\frac{\mathcal{E} \vdash (RP_2 \Rightarrow RN_2) \Rightarrow \textbf{fail}}{\mathcal{E} \vdash (RP_1, RP_2 \Rightarrow RN_1, RN_2) \Rightarrow \textbf{fail}}$$

9.8.5 Empty record pattern

Abstract syntax

$()$

Static semantics

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

Dynamic semantics

$$\frac{}{\mathcal{E} \vdash (() \Rightarrow ()) \Rightarrow ()}$$

9.9 Record of variables

9.9.1 Singleton record variable

Abstract syntax

$V \Rightarrow V$

Static semantics

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

9.9.2 Record disjoint union

Abstract syntax

RV, RV

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Static semantics

$$\frac{\mathcal{C} \vdash (RV_1 \Rightarrow RT_1) \Rightarrow (RT_1') \qquad \mathcal{C} \vdash (RV_2 \Rightarrow RT_2) \Rightarrow (RT_2')}{\mathcal{C} \vdash (RV_1, RV_2 \Rightarrow RT_1, RT_2) \Rightarrow (RT_1', RT_2')} \; [RT_1' \text{ and } RT_2' \text{ have disjoint fields}]$$

9.10 Behaviour pattern-matching

9.10.1 Single match

Abstract syntax

P [' [' E '] '] -> B

The default selection predicate is [true].

Static semantics

$$\frac{\begin{array}{l}\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT) \\ \mathcal{C}; RT \vdash E \Rightarrow \mathbf{exit}(\text{bool}) \\ \mathcal{C}; RT \vdash B \Rightarrow \mathbf{exit}(RT')\end{array}}{\mathcal{C} \vdash ((P \text{ [} E \text{] -> } B) \Rightarrow T) \Rightarrow \mathbf{exit}(RT')}$$

$$\frac{\begin{array}{l}\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT) \\ \mathcal{C}; RT \vdash E \Rightarrow \mathbf{exit}(\text{bool}) \\ \mathcal{C}; RT \vdash B \Rightarrow \mathbf{guarded}(RT')\end{array}}{\mathcal{C} \vdash ((P \text{ [} E \text{] -> } B) \Rightarrow T) \Rightarrow \mathbf{guarded}(RT')}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow \mathbf{fail}}{\mathcal{E} \vdash ((P \text{ [} E \text{] -> } B) \Rightarrow N) \Rightarrow \mathbf{fail}}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\ \mathcal{E} \vdash E[RN] \xrightarrow{\delta(\text{false})} E'\end{array}}{\mathcal{E} \vdash ((P \text{ [} E \text{] -> } B) \Rightarrow N) \Rightarrow \mathbf{fail}}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\ \mathcal{E} \vdash E[RN] \xrightarrow{X(RN)} E'\end{array}}{\mathcal{E} \vdash ((P \text{ [} E \text{] -> } B) \Rightarrow N) \Rightarrow \mathbf{fail}}$$

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\ \mathcal{E} \vdash E[RN] \xrightarrow{\delta(\text{true})} E' \\ \mathcal{E} \vdash B[RN] \xrightarrow{\mu(RN')} B'\end{array}}{\mathcal{E} \vdash ((P \text{ [} E \text{] -> } B) \Rightarrow N) \xrightarrow{\mu(RN')} B'}$$

Timed dynamic semantics

$$\frac{\begin{array}{l}\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\ \mathcal{E} \vdash E[RN] \xrightarrow{\delta(\text{true})} E' \\ \mathcal{E} \vdash B[RN] \xrightarrow{\varepsilon(d)} B'\end{array}}{\mathcal{E} \vdash ((P \text{ [} E \text{] -> } B) \Rightarrow N) \xrightarrow{\varepsilon(d)} B'}$$

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.10.2 Multiple match

Abstract syntax

BM '|' *BM*

Static semantics

$$\frac{\mathcal{C}, RT_1, RT_2 \vdash (BM_1 \Rightarrow T) \Rightarrow \mathbf{exit}(RT_1, RT) \qquad \mathcal{C}, RT_1, RT_2 \vdash (BM_2 \Rightarrow T) \Rightarrow \mathbf{exit}(RT_2, RT)}{\mathcal{C}, RT_1, RT_2 \vdash ((BM_1 \mid BM_2) \Rightarrow T) \Rightarrow \mathbf{exit}(RT_1, RT_2, RT)}$$

$$\frac{\mathcal{C}, RT_1, RT_2 \vdash (BM_1 \Rightarrow T) \Rightarrow \mathbf{guarded}(RT_1, RT) \qquad \mathcal{C}, RT_1, RT_2 \vdash (BM_2 \Rightarrow T) \Rightarrow \mathbf{guarded}(RT_2, RT)}{\mathcal{C}, RT_1, RT_2 \vdash ((BM_1 \mid BM_2) \Rightarrow T) \Rightarrow \mathbf{guarded}(RT_1, RT_2, RT)}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash (BM_1 \Rightarrow N) \xrightarrow{\mu(RN)} B}{\mathcal{E} \vdash ((BM_1 \mid BM_2) \Rightarrow N) \xrightarrow{\mu(RN)} B}$$

$$\frac{\mathcal{E} \vdash (BM_1 \Rightarrow N) \Rightarrow \mathbf{fail} \quad \mathcal{E} \vdash (BM_2 \Rightarrow N) \xrightarrow{\mu(RN)} B}{\mathcal{E} \vdash ((BM_1 \mid BM_2) \Rightarrow N) \xrightarrow{\mu(RN)} B}$$

$$\frac{\mathcal{E} \vdash (BM_1 \Rightarrow N) \Rightarrow \mathbf{fail} \quad \mathcal{E} \vdash (BM_2 \Rightarrow N) \Rightarrow \mathbf{fail}}{\mathcal{E} \vdash ((BM_1 \mid BM_2) \Rightarrow N) \Rightarrow \mathbf{fail}}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash (BM_1 \Rightarrow N) \xrightarrow{\varepsilon(d)} B}{\mathcal{E} \vdash ((BM_1 \mid BM_2) \Rightarrow N) \xrightarrow{\varepsilon(d)} B}$$

$$\frac{\mathcal{E} \vdash (BM_1 \Rightarrow N) \Rightarrow \mathbf{fail} \quad \mathcal{E} \vdash (BM_2 \Rightarrow N) \xrightarrow{\varepsilon(d)} B}{\mathcal{E} \vdash ((BM_1 \mid BM_2) \Rightarrow N) \xrightarrow{\varepsilon(d)} B}$$

10 Predefined library

This clause presents the predefined library and the predefined type scheme (e.g. strings, sets, and lists) of E-LOTOS. The types and functions of the predefined library are immediately available in each E-LOTOS specification. The type scheme declarations are translated into a set of type and functions declarations (as suggested by the "rich term syntax" of [24]).

These libraries try to be upward compatible with the existing LOTOS library. Whenever possible, the names used for types, functions, and sorts are preserved.

For each predefined type is presented the interface containing the declaration of the type and of the operations allowed for this type, and the implementation module. The static and dynamic semantics of the types and operations defined are those induced by the implementation.

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

For each type scheme is presented the interface and the implementation of the type and of the operations allowed for this type.

The interfaces contain the axiomatic description functions using ACT ONE equations.

10.1 Booleans

This subclause contains the interface and the implementation module for boolean values. The constructors "**true**" and "**false**" are defined as syntactic items and cannot be redefined as operators. Thus, they cannot be overloaded at all.

```
interface Boolean is

    type bool is
        true | false
    endtype

    function not (x: bool) : bool

    function infix or (x: bool, y: bool) : bool

    function infix and (x: bool, y: bool) : bool

    function infix implies (x: bool, y: bool) : bool

    function infix iff (x: bool, y: bool) : bool

    function infix xor (x: bool, y: bool) : bool

    function infix == (x: bool, y: bool) : bool

    function infix != (x: bool, y: bool) : bool

    eqns forall x, y: bool
        ofsort bool

         not (true) = false ;
         not (false) = true ;

         (x or true) = true ;
         (x or false) = x ;

         (x and true) = x ;
         (x and false) = false ;

         (x implies y) = (y or not x) ;

         (x iff y) = ((x implies y) and (y implies x)) ;

         (x xor y) = ((x and not y) or (y and not x)) ;

         (x == y) = (x iff y) ;
         (x != y) = (x xor y)
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
   endeqns

endint
```

The implementation of the boolean type is given by the `Boolean` module below.

```
module Boolean is

    type bool is
        true | false
    endtype

    function not (x: bool) : bool is
        case x in
            true -> false
        |   false -> true
        endcase
    endfun

    function infix or (x: bool, y: bool) : bool is
        case (x, y) in
            (false, false) -> false
        |   (any: bool, any: bool) -> true
        endcase
    endfun

    function infix and (x: bool, y: bool) : bool is
        case (x, y) in
            (true, true) -> true
        |   (any: bool, any: bool) -> false
        endcase
    endfun

    function infix implies (x: bool, y: bool) : bool is
        y or not(x)
    endfun

    function infix iff (x: bool, y: bool) : bool is
        x implies y and (y implies x)
    endfun

    function infix xor (x: bool, y: bool) : bool is
        x and not(y) or (y and not(x))
    endfun

    function infix = (x: bool, y: bool) : bool is
        x iff y
    endfun

    function infix <> (x: bool, y: bool) : bool is
        x xor y
    endfun
```


© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
endmod
```

10.2 Natural Numbers

This subclause contains the interface and the implementation module for natural values. The constants of type natural are recognized by the parser (token <nat>), and may be used like in traditional programming languages. The constructors of type `nat` are `0` and `Succ` .

The compatibility with the Standard LOTOS library is kept at the level of `NaturalNumber` module. However, the `BasicNaturalNumber` module and interface are not given.

```
interface NaturalNumbers imports Boolean
is
    type nat is
        0, Succ (n: nat)    (* All the constants 0, 1, ... are available *)
    endtype

    function infix + (m: nat, n: nat) : nat raises RANGE_ERROR

    function infix * (m: nat, n: nat) : nat raises RANGE_ERROR

    function infix ** (m: nat, n: nat) : nat raises RANGE_ERROR

    function infix - (m: nat, n: nat) : nat raises RANGE_ERROR

    function infix div (m: nat, n: nat) : nat raises ZERO_DIVISION

    function infix mod (m: nat, n: nat) : nat raises ZERO_DIVISION

    function infix pred (m: nat) : nat raises RANGE_ERROR

    function infix == (m: nat, n: nat) : bool

    function infix != (m: nat, n: nat) : bool

    function infix < (m: nat, n: nat) : bool

    function infix <= (m: nat, n: nat) : bool

    function infix >= (m: nat, n: nat) : bool

    function infix > (m: nat, n: nat) : bool

    eqns forall m, n : nat
        ofsort nat
          m + 0 = m;
          m + Succ(n) = Succ(m) + n;

          m * 0 = 0;
          m * Succ(n) = m + (m * n);

          m ** 0 = Succ(0);
          m ** Succ(n) = m * (m ** n)
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
          0 - 0 = 0;
          m - 0 = m;
          Succ (m) - Succ (n) = m - n;

          (m < n) =>
            m div n = 0;
          (m >= n) and (n > 0) =>
            m div n = ((m - n) div n) + 1;

          (m < n) =>
            m mod n = m;
          (m >= n) =>
            m mod n = ((m - n) mod n);

          pred (Succ (n)) = n;

        ofsort bool
          0 == 0 = true;
          0 == Succ(m) = false;
          Succ(m) == 0 = false;
          Succ(m) == Succ(n) = m = n;

          m != n = not(m == n);

          0 < 0 = false;
          0 < Succ(n) = true;
          Succ(n) < 0 = false;
          Succ(m) < Succ(n) = m < n;

          m <= n = m < n or (m = n);

          m >= n = not (m < n);

          m > n = not (m <= n);
    endeqns

endint

module NaturalNumbers imports Boolean
is
    type nat is
        0, Succ (n: nat) (* all the constants 0, 1, ... available *)
    endtype

    function infix + (m: nat, n: nat) : nat raises RANGE_ERROR is
        (* if m > MAX_NAT then raise RANGE_ERROR endif *)
        case n in
          0 -> m
        | Succ (n1: nat) -> Succ (m) + n1
        endcase
    endfunc
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
function infix * (m: nat, n: nat) : nat raises RANGE_ERROR is
    (* if m > MAX_NAT then raise RANGE_ERROR endif *)
    case n in
      0 -> 0
    | Succ (n1: nat) -> m + (m * n1)
    endcase
endfunc

function infix ** (m: nat, n: nat) : nat raises RANGE_ERROR is
    (* if m > MAX_NAT then raise RANGE_ERROR endif *)
    case n in
      0 -> 0
    | Succ (n1: nat) -> m * (m ** n1)
    endcase
endfunc

function infix - (m: nat, n: nat) : nat raises RANGE_ERROR is
    case (m, n) is
        (0, 0) -> 0
    |   (0, Succ (any: nat)) -> raise RANGE_ERROR
    |   (any: nat, 0) -> m
    |   (Succ (m1: nat), Succ (n1: nat)) -> m1 - n1
    endcase
endfun

function infix div (m: nat, n: nat) : nat raises ZERO_DIVISION is
    if (n = 0) then
        raise ZERO_DIVISION
    elsif (m < n) then
        0
    else
        ((m-n) div n) + 1
    endif
endfun

function infix mod (m: nat, n: nat) : nat raises ZERO_DIVISION is
    if (n = 0) then
        raise ZERO_DIVISION
    elsif (m < n) then
        m
    else
        ((m-n) mod n)
    endif
endfun

function infix pred (n: nat) : nat raises RANGE_ERROR is
    case n is
        0 -> raise RANGE_ERROR
    |   Succ (n1: nat) -> n1
    endcase
endfun
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    function infix == (m: nat, n: nat) : bool is
        case (m, n) is
           (0, 0) -> true
        |  (0, Succ (any: nat)) -> false
        |  (Succ (any: nat), 0) -> false
        |  (Succ (m1: nat), Succ (n1: nat)) -> m1 == n1
        endcase
    endfunc

    function infix <> (m: nat, n: nat) : bool is
        not (m == n)
    endfunc

    function infix < (m: nat, n: nat) : bool is
        case (m, n) is
            (any: nat, 0) -> false
        |   (0, Succ (any: nat)) -> true
        |   (Succ (m1: nat), Succ (n1: nat)) -> m1 < n1
        endcase
    endfunc

    function infix <= (m: nat, n: nat) : bool is
        m < n or (m = n)
    endfunc

    function infix >= (m: nat, n: nat) : bool is
        not (m < n)
    endfunc

    function infix > (m: nat, n: nat) : bool is
        not (m <= n)
    endfunc

endmod
```

10.3 Integral Numbers

This subclause contains the interface and the implementation module for integer values. The constants of type integral are signed natural values. Since natural constants are recognized by the parser (token <nat>), the integral values may be used like in traditional programming languages (with an unary operator "+" or "-" in front of the unsigned natural value). The constructors of type `int` are `Pos` (for positive integers) and `Neg` (for negative integers).

```
interface IntegerNumbers imports NaturalNumbers
is
    type int is
        Pos (n: nat)
    |   Neg (n: nat)
    endtype
    (* Pos (X) == X ; Neg (X) == - X - 1 *)
    (* Pos (Pos (X)) == Neg (Neg (X)) == X *)
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
function succ (n: int) : int raises RANGE_ERROR

function pred (n: int) : int raises RANGE_ERROR

function sign (n: int) : int

function abs (n: int) : int

function - (n: int) : int raises RANGE_ERROR

function infix + (m: int, n: int) : int raises RANGE_ERROR

function infix * (m: int, n: int) : int raises RANGE_ERROR

function infix ** (m: int, n: nat) : int raises RANGE_ERROR

function infix - (m: int, n: int) : int raises RANGE_ERROR

function infix div (m: int, n: int) : int raises ZERO_DIVISION

function infix mod (m: int, n: int) : int raises ZERO_DIVISION

function infix pred (m: int, n: int) : int raises RANGE_ERROR

function infix == (m: int, n: int) : bool

function infix != (m: int, n: int) : bool

function infix < (m: int, n: int) : bool

function infix <= (m: int, n: int) : bool

function infix >= (m: int, n: int) : bool

function infix > (m: int, n: int) : bool

function nat (k: int) : nat raises RANGE_ERROR

function int (k: nat) : int raises RANGE_ERROR

eqns forall M, N: nat, X, Y: int
    ofsort int
        succ (Pos (N)) = Pos (Succ (N));
        succ (Neg (0)) = Pos (0);
        succ (Neg (Succ (N))) = Neg (N);

        pred (Pos (0)) = Neg (0);
        pred (Pos (Succ (N))) = Pos (N);
        pred (Neg (N)) = Neg (Succ (N));

        sign (Pos (0)) = 0;
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
sign (Pos (Succ (N))) = 1;
sign (Neg (N)) = Neg (Succ (0));

abs (Pos (N)) = Pos (N);
abs (Neg (N)) = Pos (Succ (N));

- (Pos (0)) = Pos (0);
- (Pos (Succ (N))) = Neg (N);
- (Neg (N)) = Pos (Succ (N));

Pos (0) + X = X;
Pos (Succ (N)) + X = Pos (N) + succ (X);
Neg (0) + X = pred (X);
Neg (Succ (N)) + X = Neg (N) + pred (X);

X - Y = X + - (Y);

Pos (M) * Pos (N) = Pos (M * N);
Pos (M) * Neg (N) = succ (Neg (M * Succ (N)));
Neg (M) * Pos (N) = succ (Neg (Succ (M) * N));
Neg (M) * Neg (N) = Pos (Succ (M) * Succ (N));

X ** 0 = Succ (0);
X ** Succ (N) = X * (X ** N);

(m < n) =>
  m div n = 0;
(m >= n) =>
  m div n = ((m - n) div n) + 1;

(m < n) =>
  m mod n = m;
(m >= n) =>
  m mod n = ((m - n) mod n);

Pos (M) == Pos (N) = M == N;
Pos (M) == Neg (N) = false;
Neg (M) == Pos (N) = false;
Neg (M) == Neg (N) = M == N;

X != Y = not (X == Y);

Pos (M) < Pos (N) = M < N;
Pos (M) < Neg (N) = false;
Neg (M) < Pos (N) = true;
Neg (M) < Neg (N) = M > N;

X <= Y = (X < Y) or (X eq Y);

X > Y = not (X <= Y);

X >= Y = not (X < Y);
```

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
            nat (Pos (N)) = N;

            int (N) = Pos (N);

    endeqns

endint (* IntegralNumbers *)
```

The module `IntegralNumbers` may be either an external module, or may implement the equations above raising exceptions for the unspecified cases.

10.4 Rational Numbers

This subclause contains the interface and implementation module for rational values. The constants of type rational are recognized by the parser (token <rational>), and may be used like in traditional programming languages. The constructor of type `rational` is `frac` . This constructor receives two irreducible integral numbers, the second one (the denominator) should be greater than 0.

Note that the LOTOSPHERE project proposes the name "`Frac` " for the rational numbers.

```
interface RationalNumbers imports IntegerNumbers is

    type rational is
        frac (num: int, den: int) (* den > 0 *)
    endtype

    function infix + (f1: rational, f2: rational) : rational

    function infix - (f1: rational, f2: rational) : rational

    function infix * (f1: rational, f2: rational) : rational

    function infix ** (m: rational, n: int) : rational

    function infix / (f1: rational, f2: rational) : rational raises ZERO_DIVISION

    function max (f1: rational, f2: rational) : rational
    function min (f1: rational, f2: rational) : rational

    function abs (f: rational) : rational

    function round (f: rational) : rational
    (* this function returns the nearest integral value of f, except for
       halfway cases, which are rounded to the integral value larger in
       magnitude *)

    function ceil  (f: rational) : rational
    (* this function returns the least integral value greater than or
       equal to f *)

    function floor (f: rational) : rational
    (* this function returns the greatest value less than or equal to f *)
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    function infix >  (f1: rational, f2: rational) : bool
    function infix >= (f1: rational, f2: rational) : bool
    function infix <  (f1: rational, f2: rational) : bool
    function infix <= (f1: rational, f2: rational) : bool
    function infix == (f1: rational, f2: rational) : bool
    function infix != (f1: rational, f2: rational) : bool

endint (* RationalNumbers *)
```

The implementation uses a local function "gd " (greater divisor) to reduce fractions which are not irreducible.

```
module RationalNumbers imports IntegralNumbers is

    type rational is
        frac (num: int, den: int) (* den > 0 *)
    endtype

    (* local function *)
    function gd (m: int, n: int) : int is
        case n is
            0 -> m
        |   any int -> gd (n, m mod n)
        endcase
    endfun

    function reduce (f: rational) : rational is
        var gd: int = gd (f.num, f.den)
        in
            frac (f.num div gd, f.den div gd) (* div is the integral division *)
        endvar
    endfun

     function minus (f: rational) : rational is
        frac (minus (f.num), f.den)
     endfun

    function infix + (f1: rational, f2: rational) : rational is
        reduce (frac (f1.num * f2.den + f2.num * f1.den, f1.den * f2.den))
    endfun

    function infix - (f1: rational, f2: rational) : rational is
        reduce (frac (f1.num * f2.den - f2.num * f1.den, f1.den * f2.den))
    endfun

    function infix * (f1: rational, f2: rational) : rational is
        reduce (frac (f1.num * f2.num, f1.den * f2.den))
    endfun

    function infix ** (f: rational, p: int) : rational is
        if p == 0 then
            frac (1, 1)
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
    elsif p < 0 then
        (f ** (p + 1)) / f
    else (* p > 0 *)
        (f ** (p - 1)) * f
    endif
endfun

function infix / (f1: rational, f2: rational) : rational raises ZERO_DIVISION is
    case f2 in
        frac (num => 0, den => any int) -> raise ZERO_DIVISION
    |   any rational -> reduce (frac (f1.num * f2.den, f2.num * f1.den))
    endcase
endfun

function max (f1: rational, f2: rational) : rational is
    if f1 >= f2 then f1 else f2 endif
endfun

function min (f1: rational, f2: rational) : rational is
    if f1 >= f2 then f2 else f1 endif
endfun

function abs (f: rational) : rational is
    frac (abs (f.num), f.den)
endfun

function round (f: rational) : int is
    (f.num / f.den) +
    (if (f.num rem f.den) >= (f.den / 2) then 1 else 0 endif)
endfun

function ceil (f: rational) : int is
    (f.num / f.den) +
    (if (f.num rem f.den) >= 0 then 1 else 0 endif)
endfun

function floor (f: rational) : int is
    (f.num / f.den)
endfun

function infix > (f1: rational, f2: rational) : bool is
    (f1.num * f2.den - f2.num * f1.den) > 0
endfun

function infix >= (f1: rational, f2: rational) : bool is
    (f1.num * f2.den - f2.num * f1.den) >= 0
endfun

function infix <  (f1: rational, f2: rational) : bool is
    (f1.num * f2.den - f2.num * f1.den) < 0
endfun
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    function infix <= (f1: rational, f2: rational) : bool is
        (f1.num * f2.den - f2.num * f1.den) <= 0
    endfun

    function infix == (f1: rational, f2: rational) : bool is
        (f1.num * f2.den)  == (f2.num * f1.den)
    endfun

    function infix != (f1: rational, f2: rational) : bool is not (f1 == f2) endfun

endmod (* RationalNumbers *)
```

10.5 Floating Point Numbers

This subclause contains the interface for floating point values. These values are recognized by the parser (token <float>), and may be used like in traditional programming languages. The operations provided for floating point numbers are the classical ones. They may be obtained by external implementation.

Note that this predefined type appears also in the LOTOSPHERE proposal.

```
interface FloatNumbers imports IntegralNumbers is

    type float is
        [+|-]<nat>.[<nat>] [E[+|-]<nat>]
    endtype

    function infix + (f1: float, f2: float) : float
    function infix - (f1: float, f2: float) : float
    function infix * (f1: float, f2: float) : float
    function infix ** (f1: float, f2: int) : float
    function infix / (f1: float, f2: float) : float raises ZERO_DIVISION

    function max (f1: float, f2: float) : float
    function min (f1: float, f2: float) : float

    function abs (f: float) : float

    function sqrt (f: float) : float
    function exp  (f: float) : float      (* e^x *)
    function log  (f: float) : float      (* log_10 x *)

    function sin  (f: float) : float
    function cos  (f: float) : float
    function tan  (f: float) : float
    function asin (f: float) : float
    function acos (f: float) : float
    function atan (f: float) : float

    function sinh  (f: float) : float
    function cosh  (f: float) : float
    function tanh  (f: float) : float
    function asinh (f: float) : float
    function acosh (f: float) : float
```

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
    function atanh (f: float) : float

    function pi : float
    function e  : float

    function round (f: float) : float
    (* this function returns the nearest integral value to f, except for
       halfway cases, which are rounded to the integral value larger in
       magnitude *)

    function ceil  (f: float) : float
    (* this function returns the least integral value greater than or
       equal to f *)

    function floor (f: float) : float
   (* this function returns the greatest value less than or equal to f *)

    function infix >  (f1: float, f2: float) : bool
    function infix >= (f1: float, f2: float) : bool
    function infix <  (f1: float, f2: float) : bool
    function infix <= (f1: float, f2: float) : bool
    function infix == (f1: float, f2: float) : bool
    function infix != (f1: float, f2: float) : bool

endint (* FloatNumbers *)
```

10.6 Characters

This subclause contains the interface and the implementation module for character values. The constants of type character are written between quote symbols (e.g., 'A') and are recognized by the parser (token <char>). They represent the ISO Latin-1 characters.

```
interface Characters imports NaturalNumbers is

    type char is
        (* all the  ISO Latin-1 characters between simple quotes *)
    endtype

    function pred (c: char) : char raises RANGE_ERROR
    function succ (c: char) : char raises RANGE_ERROR

    function nat  (c: char) : nat
    function char (n: nat)  : char raises RANGE_ERROR

    function tolower (c: char) : char
    function toupper (c: char) : char

    function isalpha  (c: char) : bool
    function isdigit  (c: char) : bool
    function isxdigit (c: char) : bool
    function islower  (c: char) : bool
    function isupper  (c: char) : bool
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    function isalnum  (c: char) : bool

    function infix >  (c1: char, c2: char) : bool
    function infix >= (c1: char, c2: char) : bool
    function infix <  (c1: char, c2: char) : bool
    function infix <= (c1: char, c2: char) : bool
    function infix == (c1: char, c2: char) : bool
    function infix != (c1: char, c2: char) : bool

endint (* Characters *)
```

10.7 Strings

```
interface Strings imports NaturalNumbers, Characters is

    type string is ...
    endtype

    function length (s: string)  : nat

    function concat (s1: string, s2: string) : string
    function prefix (s: string, n: nat) : string
    function suffix (s: string, n: nat) : string
    function substr (s: string, n1: nat, n2: nat) : string

    function index  (s1: string, s2: string) : nat (* search from left *)+
    function rindex (s1: string, s2: string) : nat (* search from right *)+

    function nth (s: string, n: nat) : char

    function infix >  (s1: string, s2: string) : bool
    function infix >= (s1: string, s2: string) : bool
    function infix <  (s1: string, s2: string) : bool
    function infix <= (s1: string, s2: string) : bool
    function infix == (s1: string, s2: string) : bool
    function infix != (s1: string, s2: string) : bool

    function string (c: char)  : string
    function string (n: nat)   : string
    function string (n: int)   : string
    function string (f: float) : string

    function int    (s: string, b: nat) : int   raises RANGE_ERROR
    function float  (s: string)         : float raises RANGE_ERROR

endint (* Strings *)
```

10.8 Enumerated Type Scheme

The declaration of an enumerated type `ET` with values `C1` , ..., `Cn` :

```
type ET is
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
    enum C1, ..., Cn
endtype
```

is translated into (substituted with) the following list of declarations (the interface of objects generated for an enumerated type):

```
type ET is
    C1 | ... | Cn
endtype

function min : ET
function max : ET

function pred (x: ET) : ET raises RANGE_ERROR
function succ (x: ET) : ET raises RANGE_ERROR

function infix >  (x: ET, y: ET) : bool
function infix >= (x: ET, y: ET) : bool
function infix <  (x: ET, y: ET) : bool
function infix <= (x: ET, y: ET) : bool
function infix == (x: ET, y: ET) : bool
function infix != (x: ET, y: ET) : bool

function pos    (x: ET)  : nat
function ET     (n: nat) : ET raises RANGE_ERROR
function string (x: ET)  : string
```

The implementation of these function declarations is given below:

```
function min : ET is C1 endfun

function max : ET is Cn endfun

function pred (x: ET) : ET raises RANGE_ERROR is
    case x is
        C1 -> raise RANGE_ERROR
        C2 -> C1
    |   ...
    |   Cn -> Cn-1
    endcase
endfun

function succ (x: ET) : ET raises RANGE_ERROR is
    case x is
        C1 -> C2
    |   ...
    |   Cn -> raise RANGE_ERROR
    endcase
endfun

function pos (x: ET) : Nat is
    case x is
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        C1 -> 1
    |   ...
    |   Cn -> n
    endcase
endfun

function ET (n: nat) : ET raises RANGE_ERROR is
    case n is
        1 -> C1
    |   ...
    |   n -> Cn
    |   any nat -> raise RANGE_ERROR
    endcase
endfun

function infix > (x: ET, y: ET) : Bool is pos (x) > pos (y) endfun

function infix >= (x: ET, y: ET) : Bool is pos (x) >= pos (y) endfun

function infix <  (x: ET, y: ET) : Bool is pos (x) < pos (y) endfun

function infix <= (x: ET, y: ET) : Bool is pos (x) <= pos (y) endfun

function infix == (x: ET, y: ET) : Bool is pos (x) == pos (y) endfun

function infix != (x: ET, y: ET) : Bool is pos (x) != pos (y) endfun
```

10.9 Record Type Scheme

A record type declaration:

```
type RT is
    record F1: T1, ..., Fn: Tn
endtype
```

is translated into (substituted with) the following list of declarations (the interface of objects generated for a record type):

```
type RT is
    RT (F1: T1, ..., Fn: Tn)
endtype

(* selectors of RT *)
function get_F1 (x: RT) : T1
...
function get_Fn (x: RT) : Tn

(* setup fields functions of RT *)
function set_F1 (x: RT, f: T1) : RT
...
function set_Fn (x: RT, f: Tn) : RT
```

The implementation of the function declarations above is:

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
function get_F1 (x: RT) : T1 is x.F1 endfun
...
function get_Fn (x: RT) : Tn is x.Fn endfun

function set_F1 (x: RT, f: T1) : RT is
    RT (F1 => f, F2 => x.F2, ...., Fn => x.Fn)
endfun
...
function set_Fn (x: RT, f: Tn) : RT is
    RT (F1 => x.F1, F2 => x.F2, ...., Fn => f)
endfun
```

10.10 Set Type Scheme

The declaration of a type set `ST` with elements in the scalar type `T` :

```
type ST is set of T endtype
```

is translated into (substituted with) the following list of declarations (the interface of objects generated for a set type):

```
type ST

function {}   : ST
function full : ST

function union  (s1: ST, s2: ST) : ST
function diff   (s1: ST, s2: ST) : ST
function inters (s1: ST, s2: ST) : ST

function card (s: ST) : Nat

function isin     (e: T, s: ST)      : bool
function isempty  (s: ST)            : bool
function issubset (s1: ST, s2: ST)   : bool
function isdisjoint (s1: ST, s2: ST) : bool

function infix >  (s1: ST, s2: ST) : bool
function infix >= (s1: ST, s2: ST) : bool
function infix <  (s1: ST, s2: ST) : bool
function infix <= (s1: ST, s2: ST) : bool
function infix == (s1: ST, s2: ST) : bool
function infix != (s1: ST, s2: ST) : bool
```

The type `ST` is implemented like a list of elements. The constructors are "`nil` " and "`cons` ". However, this implementation is not visible to the user, so no pattern-matching is allowed on set types.

```
type ST is
```

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    nil
|   cons (e: T, s: ST)
endtype

function {} : ST is nil () endfun
function full : ST is cons (nil () endfun

function union (s1: ST, s2: ST) : ST is
    case s1 is
       nil () -> s2
    |  cons (e: T, s: ST) ->
            if isin (e, s2) then
                union (s, s2)
            else
                cons (e, union (s, s2))
            endif
    endcase
endfun

function diff (s1: ST, s2: ST) : ST is
    case s1 is
        nil () -> nil ()
    |   cons (e: T, s: ST) ->
            if isin (e, s2) then
                diff (s, s2)
            else
                cons (e, diff (s, s2))
            endif
    endcase
enfunc

function inters (s1: ST, s2: ST) : ST is
    case s1 is
        nil () -> nil ()
    |   cons (e: T, s: ST) ->
            if isin (e, s2) then
                cons (e, Inters (s, s2))
            else
                inters (s, s2)
            endif
    endcase
endfun

function card (s: ST) : nat is
    case s is
       nil -> 0
    |  cons (any T, s1: ST) -> card (s1) + 1
    endcase
endfun

function isin (e: T, s: ST) : bool is
    case s is
```


© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        nil () -> false
    |   cons (e1: T, s1: ST) -> (e == e1) orelse isin (e, s1)
    endcase
endfun

function isempty (s: ST) : bool is
    case s is
       nil () -> true
    |  any ST -> false
    endcase
endfun

function issubset (s1: ST, s2: ST) : bool is
    case (s1, s2) is
        (nil, nil) -> true
    |   (nil, any ST) -> true
    |   (any ST, nil) -> false
    |   (cons (e1: T, s11: ST), any ST) ->
                iselem (e1, s2) andthen issubset (s11, s2)
    endcase
endfun

function isdisjoint (s1: ST, s2: ST) : bool is
    isempty (inters (s1, s2))
endfun

function infix > (s1: ST, s2: ST) : bool is
    issubset (s2, s1) and  not (isempty (diff (s1, s2)))
endfun

function infix >= (s1: ST, s2: ST) : bool is issubset (s2, s1) endfun

function infix <  (s1: ST, s2: ST) : bool is
    issubset (s1, s2) and  not (isempty (diff (s1, s2)))
endfun

function infix <= (s1: ST, s2: ST) : bool is issubset (s1, s2) endfun

function infix == (s1: ST, s2: ST) : bool is
    issubset (s1, s2) and issubset (s2, s1)
endfun

function infix != (s1: ST, s2: ST) : bool is not (s1 == s2) endfun
```

The "in extenso" (i.e. by giving the list of their elements) for a set type:

```
{ E1, ..., En }
```

is translated into:

```
union (cons (E1, nil ()), union (..., cons (En, nil ())))
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

10.11 List Type Scheme

The declaration of a type list `LT` with elements of type `T` :

```
type LT is list of T endtype
```

is translated into (substituted with) the following list of declarations (the interface of objects generated for a list type):

```
type LT
    nil ()
|   cons (e: T, l: LT)
endtype

function isempty (l: LT) : bool

function [] : LT
function tcons (l: LT, e: T) : LT

function head  (l: LT) : T  raises EMPTY_LIST
function taill (l: LT) : LT raises EMPTY_LIST

function nth    (l: LT, n: nat)  : T raises RANGE_ERROR
function concat (l1: LT, l2: LT) : LT
function length (l: LT)          : nat

function ==  (l1: LT, l2: LT) : bool
function !=  (l1: LT, l2: LT) : bool
```

The type `LT` has the constructors are "`nil` " and "`cons` ". These constructors are visible, so pattern-matching is allowed on list types. The base type of the list must have an equality function "`==` ").

```
function isempty (l: LT) : bool is
    case s in
        nil -> true
    |   any LT -> false
    endcase
endfun

function [] : LT is nil () endfun

function tcons (l: LT, e: T) : LT is
    case s in
        nil -> cons (e, nil)
    |   cons (e1: T, l1: LT) -> cons (e1, tcons (l1, e))
    endcase
endfun

function head (l: LT) : T raises EMPTY_LIST is
    case s in
        nil -> raise EMPTY_LIST
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
    |   cons (e: T, any LT) -> e
    endcase
endfun

function tail (l: LT) : LT raises EMPTY_LIST is
    case s in
        nil -> raise EMPTY_LIST
    |   cons (any T, l1: LT) -> l1
    endcase
endfun

function concat (l1: LT, l2: LT) : LT is
    case (s1, s2) is
        (nil, any LT) -> l2
    |   (any LT, nil) -> l1
    |   (any LT, cons  (e: T, l: LT)) -> concat (append (l1, e), l)
    endcase
endfun

function length (l: LT) : nat is
    case s in
        nil -> 0
    |   cons (any T, l: LT) -> Succ (length (s))
    endcase
endfun

function nth (l: LT, n: nat) : T raises RANGE_ERROR is
    case (s, n)  is
        (nil, any nat) -> if n > 0 then raise RANGE_ERROR else nil endif
    |   (cons (e: T, any LT), 0) -> e
    |   (cons (any T, l1: LT), any nat) -> nth (l1, n - 1)
    endcase
endfun

function == (l1: LT, l2: LT) : bool is
    case (s1, s2) is
        (nil, any LT) -> false
    |   (any LT, nil) -> false
    |   (cons  (e1: T, l1p: LT), cons (e2: T, l2p: LT)) ->
                (e1 == e2) andthen (l1p == l2p)
    endcase
endfun

function != (l1: LT, l2: LT) : bool is not (l1 == l2) endfun
```

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

`[ E1, ..., En ]`

where `E1` , ..., `En` are expressions of type `T` . This notation is equivalent with

`cons (E1, cons ( ... cons (En, nil ()) ... ))`

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex A
(informative)
Tutorial

A.1 The base language

The ISO formal language LOTOS [6, 4] is composed of a process algebra part (based on CCS [22] and CSP [12]) to describe behaviours, and an algebraic language (ACT ONE [8]) to describe the abstract data types. This language is mathematically well-defined and expressive: it allows the description of concurrency, nondeterminism, synchronous and asynchronous communications. It supports various levels of abstraction and provides several specification styles. Good tools exist to support specification, verification and code generation. Despite these positive features, this language is currently under revision in ISO [26] because feedback from users has indicated that the usefulness of the language is limited by certain characteristics relating both to technical capabilities and user-friendliness of the language.

Two main enhancements address datatypes and time. There is no notion of quantitative time in standard LOTOS, which precludes any precise description of real-time systems. Furthermore, the LOTOS algebraic datatypes are not user-friendly and suffer from several limitations such as the semi-decidability of equational specifications, the lack of modularity and the inability to define partial operations.

For example, a simple router of packets containing a data field and an address field might be defined in standard LOTOS:

```
process Router [inp,left,right] : noexit :=
  inp?p:packet;
  (
       [getdest(p) = L] -> left!getdata(p);Router [inp,left,right]
    [] [getdest(p) = R] -> right!getdata(p);Router [inp,left,right]
  )
endproc
```

This definition suffers from some problems of readability for non-LOTOS experts (for example the use of selection predicates and choice rather than a **case** construct) but is quite understandable compared to the

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

definition of the packet datatype:

```
type Packet is Data
  sorts
    packet, dest
  opns
    mkpacket : dest, data -> packet
    getdest : packet -> dest
    getdata : packet -> data
    L : -> dest
    R : -> dest
  eqns forall p:packet, de:dest, da:data
    ofsort packet mkpacket (getdest (p), getdata (p)) = p
    ofsort dest getdest (mkpacket (de, da)) = de
    ofsort data getdata (mkpacket (de, da)) = da
  endeqns
endtype
```

This can be compared with the equivalent process declaration in the base language presented here:

```
process Router [inp, left, right] : noexit :=
  inp?p:packet;
  (       [getdest(p) = L] -> left!getdata(p); Router [inp, left, right]
    [] [getdest(p) = R] -> right!getdata(p); Router [inp, left, right]
  )
endproc
```

with the corresponding data type declarations:

```
type dest is L | R endtype
type packet is (de=>dest, da=>data) endtype
```

Note that:

- The gates in the Router process are explicitly typed.
- We can use field projection to access the fields of the packet, rather than using hand-crafted selection functions.
- The scope of the variable p is made explicit by a **local** variable declaration.
- The **case** statement is made explicit, rather than implicit using selection predicates and choice.
- We have moved the recursive call outside the **case** statement, avoiding the need to duplicate it.
- The definition of the 'dest' type as a union, and the 'packet' type as a record is made explicit, and much shorter.

The revised LOTOS language is a two-layer language. The higher layer is the module language and it will be described in the next subclause. The lower layer is the base language which we will informally present in this subclause.

The static semantics of the base language is based on judgements such as $\mathcal{C} \vdash E \Rightarrow \mathbf{exit}\ (T)$ meaning 'in context $\mathcal{C}$ expression E has result type T' for example:

$$1 \Rightarrow \text{float}, \text{x} \Rightarrow \text{float}, / \Rightarrow (\text{float}, \text{float}) \rightarrow \mathbf{exit}\ (\text{float}) \vdash 1/\text{x} \Rightarrow \mathbf{exit}\ (\text{float})$$

means 'in a context where 1 and x are floats, and / is a function from pairs of floats to floats, then the expression $1/\text{x}$ has result type float'. The static semantics includes:

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- User-definable record, union types, and recursive types.
- Subtyping (for example we could allow integers as a subtype of floats).
- Imperative write-many variables, with a static semantics which ensures that every variable is written before read, and that shared variables cannot be used for communication between processes.
- Gates are explicitly typed (but we can use subtyping to provide the power of standard LOTOS untyped gates).

The dynamic semantics is based on judgements such as $\mathcal{E} \vdash E \xrightarrow{\alpha(N)} E'$ meaning 'in environment $\mathcal{E}$ expression E reduces (with action $\alpha(N)$) to E''. For expressions, possible values of α are an exception X or a successful termination action δ. For example the expression $1/2$ terminates with value 0.5:

$$\vdash 1/2 \xrightarrow{\delta(0.5)} \textbf{block}$$

and $1/0$ raises the exception Div:

$$\vdash 1/0 \xrightarrow{\mathrm{Div}()} \textbf{block}$$

The dynamic semantics includes:

- Behaviours communicating on gates with other behaviours.
- Behaviours or expressions raising exceptions, which may be trapped by exception handlers.
- Behaviours with real-time semantics.

In fact, the semantics of expressions is given by treating expressions as a subclass of behaviours: expressions can only perform exception or termination actions, and cannot communicate on gates, or have any real-time behaviour. Unifying expressions and behaviours in this way allows for a much simpler and uniform semantics.

The language described in this International Standard is based on previous proposals for real-timed LOTOS [18, 19, 20] and LOTOS with functional datatypes [13, 14] and exception handling [11]. Many of the language features, especially the imperative features, are based on the language proposed in [10]. A previous version of the language can be found [16].

A.1.1 Basic concepts

A.1.1.1 Declarations

A specification in the base language is given by a set of *declarations*. These declarations can be structured at the module level (see Subclause A.2).

These declarations come in three flavours: *type* declarations, *function* declarations, and *process* declarations. In the base language, all type, constructor, function and process identifiers must be unique—all treatment of overloading is left to the module language.

Type declarations A type declaration is either a *type synonym* or a *datatype* declaration. A type synonym declares a new type identifier for an existing type. For example we can declare a type code as:

```
type code renames nat endtype
```

code and nat are synonymous. We can use code and nat as the same type (for example any function expecting a code will accept a nat). More succinctly, type equality is *structural* not by *name*.

We can also declare a recursive data type of integer lists as:

```
type intlist is
      nil
    | cons(int, intlist)
endtype
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Data type declarations define new types, listing all the *constructors* for that type. Since there can be more than one constructor, we can define *union* types, for example:

```
type pdu is
  send(packet,bit) | ack(bit)
endtype
```

The base language does *not* provide a mechanism for defining parameterized types—this is left for the module system.

There are also predefined types.

Function declarations A function declaration defines a new function, which can be used in data expressions. For example:

```
function reflect (p:point) : point is
  (x=>p.y,y=>p.x)
endfunc
```

The function parameters are given as a list of typed variables – in E-LOTOS we decorate binding occurrences of variables with ?. A function can have more than one input parameter, and can return a record of results, for example (we will fill in the details later):

```
type PairOfIntLists is (intlist,intlist) endtype
function partition (x:int,xs:intlist) : PairOfIntLists is
  var
    less:intlist := all of xs less than x,
    gtr:intlist:= all of xs greater than x
  in
    (less,gtr)
  endvar
endfunc
```

This function can be called (for example):

```
function quicksort (xs:intlist) :intlist is
  case xs is
      nil ->
        nil
    | cons(?y,?ys) ->
        var
          l:intlist,g:intlist
        in
          (?l,?g) := partition (y,ys)
          append (quicksort (l),cons (y,quicksort (g)))
        endvar
  endcase
endfunc
```

This style of function is very common, so we provide some syntax sugar for it, using **in** and **out** parameters. For example, the partition function could have been written:

```
function partition (in x:int, in xs:intlist, out less:intlist, out gtr:intlist) is
  ?less := all of xs less than x;
  ?gtr := all of xs greater than x
endfunc
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

and then used in quicksort as:

```
partition (y,ys,?l,?g)
```

rather than:

```
(?l,?g) := partition (y,ys)
```

Functions may raise exceptions (described below) which have to be declared, for example:

```
function hd (xs:intlist) : int raises [Hd] is
  case xs is
      nil -> raise Hd
    | cons(?x,any:intlist) -> x
  endcase
endfunc
```

When such a function is called, the Hd exception is instantiated, for example the following will raise the exception Foo:

```
hd (nil) [Foo]
```

Most often, we use the same exception name as in the declaration:

```
hd (nil) [Hd]
```

This acts as a visual reminder that the hd function can raise the exception Hd.

Exceptions can be typed, for example:

```
function foo () raises [Foo:(string)] is
  raise Foo("Hello world")
endfunc
```

Any untyped exceptions are assumed to have type `()`.

Note that function declarations are just syntax sugar for a subclass of process declaration.

Process declarations Process declarations are very similar to function declarations: they have parameter lists, in and out parameters, and a list of typed exception parameters.

However, there are two important differences between functions and processes: processes can have real-time behaviour, and they can communicate on gates. For example, a simple counter process is defined:

```
process Counter [up:none,down:none] is
  up; (down ||| Counter [up,down])
endproc
```

By default, gates have type *any*, which allows communication of arbitrary data, for compatibility with existing LOTOS. Also, the default return type of a process is **null**.

Process behaviours are discussed further in Subclause A.1.1.4.

A.1.1.2 Typing

Type expressions We have already seen a number of type expressions, for example:

- The data type intlist, and the type synonym point are both *type identifiers*.
- The type (x=>float,y=>float) is a *record type* with *fields* x and y.
- The type (int,intlist) is a *pair type*: in fact this is syntax sugar for the record type ($1=>int,$2=>intlist).

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Record types can be *extensible*, for example the type (name=>string,**etc**) is a record type with at least one field, but which can be extended to have others.

In addition to type identifiers and record types, we have two special types:

- The empty type **none** with no values, used to give the functionality of processes such as **stop** or **Counter** which never terminate.
- The universal type **any** which is a supertype of every other type, used to give types for gates which can communicate data of any type, for compatibility with existing LOTOS.

Subtyping The base language supports *subtyping*, for example we could have integers as a subtype of floats. The built-in subtyping is on records: we allow a record type (**etc**) which is a supertype of any other record. For example, the type (name=>string,**etc**) is a record with at least one field 'name' of type string. This record type can be extended to many subtypes, for example (name=>string,age=>int,**etc**) or (name=>string,age=>int). Note the difference between these last two types: the former can be extended with further fields, where the latter cannot.

We include a special **none** type, which has no values. The type **none** is the most specialised type, and **any** is the most general type. Since a record type with a **none** field cannot have any values, we can identify it with **none**, for example the pair type (**none**,int) has no values, so is equivalent to the type **none**. This means that the one-element record type (**none**) is the most specialized record type, and (**etc**) is the most general.

For example, **stop** is a behaviour of type **exit** (**none**), meaning that it will never terminate. Since (**none**) is the least general record type, we can use **stop** wherever a process of any record type is required.

Similarly, if G is a gate of type *any* then we can communicate values of any type along G—this is the same semantics as the existing untyped gates in standard LOTOS.

A.1.1.3 Data expressions

In contrast to standard LOTOS (which has a separation between processes and functions), E-LOTOS considers functions to be restricted forms of processes (with no communication or real-time capabilities). The language of expressions is therefore very similar to the language of behaviours, and shares many features such as pattern-matching, exception raising and handling, and imperative features.

Normal forms A *normal form* is a data expression which cannot be reduced any further. For example $1+1$ is not in normal form, but 2 is. A normal form is one of the following:

- A primitive constant, such as `"Hello world"` or 2, for one of the built-in types.
- A variable, such as x or gtr.
- A record of normal forms, such as (x=>1.5,y=>$-$3.14), () or (5,nil()) (which is just syntax sugar for ($1=>5,$2=>nil())).
- A constructor applied to a normal form, such as nil() or cons(5,nil()).

We will let N range over normal forms, and (RN) range over record normal forms.

Pattern-matching The expression language includes a **case** operation, which allows branching depending on the value of an expression, for example we can find the head of a list with:

```
case xs is
    nil -> raise Hd
  | cons(?x,any:list) -> x
endcase
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This case operation consists of a value to branch on (in this case xs) together with a list of possibilities, given by *patterns*. If the list is empty, then the first pattern will match, and the Hd exception will be raised. If the list is non-empty, then the second pattern will match, x will be *bound to* the head of the list, and will then be returned as the result.

Case expressions are evaluated by evaluating the expression to normal form, and then attempting to match the resulting value against each pattern from top to bottom until a match is found. If the value does not match any pattern (which cannot occur in the above example), a special Match exception is raised.

Note that cons(?x,**any**:list) is a structured pattern. At the highest level, we find the list constructor cons, built from a record pattern that includes the elementary patterns ?x and **any**:list. For a list to match this pattern, it has to have the form cons(hd,tl).

When a list matches the pattern cons(?x,**any**:list), the variable x is bound to the head of the list, for example producing the substitution [x=>hd]. Since substitutions have the same syntax as records, we will make a pun between record normal forms and substitutions.

We also allow expressions in patterns, which are evaluated when the pattern is matched, and match any value equal to the result. This is most often used to match against constants, for example:

```
case x is
    !0 -> "zero"
  | any:int -> "nonzero"
endcase
```

Sometimes, it is useful to match against an expression, for example we can check to see if a list is a palindrome (using a function which reverses a list) with:

```
case xs is
    !reverse(xs) -> "palindrome"
  | any:list -> "nonpalindrome"
endcase
```

The main use of matching against expressions is in communication, as we shall see in Subclause A.1.1.4.

Patterns can be explicitly typed, which is useful in the presence of subtyping. For example, if nat were a subtype of int, then we could construct a **case** statement to decide whether an integer value is an natural number or not:

```
case x:int is
    any:nat -> "natural"
  | any:int -> "nonnatural"
endcase
```

Again, the main use for explicitly typed patterns is in communication.

A pattern is one of the following:

- A bound variable, for example ?x.
- A free expression for example !0 or !reverse(xs).
- The typed wildcard pattern **any**:T.
- A record pattern, for example (x=>?px,y=>?py), (), or (?x,**any**:T) (which is just syntax sugar for ($1=>?x,$2=>**any**:T)).
- An extensible record pattern, for example (x=>?px,**etc**), (**etc**), or (?x,**etc**) where **etc** is a pattern which matches any other fields. Note the difference between (?x,**any**:T) and (?x,**etc**): the former will only match tuples with two fields where the latter will match tuples with any (positive) number of fields.

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- A record pattern with an **as** clause to bind part of the record, for example (?all **as** ?x,**etc**) or (?x,?all **as etc**).
- A constructor applied to a pattern, for example nil or cons(?x,**any**:list).
- An explicitly typed pattern, for example ?y:int.
- A guarded pattern, for example ?y:int [y < 10] which matchs any integer less than 10.

It is easy to define operators such as **if**-statements as syntax sugar on top of the case operator, for example the expression:

if E **then** E_1 **else** E_2 **endif**

can be expanded to:

```
case E is
      true -> E1
    | any:bool -> E2
endcase
```

elsif-statements are also syntax sugar, for example the expression:

if E_1 **then** E_2 **elsif** E_3 **then** E_4 **else** E_5 **endif**

can be expanded to:

```
if E1 then
  E2
else
  if E3 then
    E4
  else
    E5
  endif
endif
```

Exceptions Expressions can raise exceptions, in order to signal an error of some kind, for example when we attempt to take the head of an empty list:

```
function hd (xs:intlist) : int raises [Hd] is
  case xs is
       nil -> raise Hd
    | cons(?x,any:intlist) -> x
  endcase
endfunc
```

Exceptions either propagate to the top level, or are *trapped* by an exception handler. For example we can declare a function:

```
function hd0 (xs:intlist) : int is
  trap
    exception Hd is 0 endexn
  in
    hd (xs) [Hd]
  endtrap
endfunc
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Then hd0 (cons(a,as)) returns a, and hd0 (nil) returns 0, since the Hd exception raised by hd is trapped by the exception handler.

Exceptions can be typed, for example:

```
trap
  exception Error (code:int) is
    case code is
        !0 -> "minor error"
      | !1 -> "major error"
      | any:int -> raise Unknown (code)
    endcase
  endexn
in
  ...
endtrap
```

We can declare more than one exception in a single trap operator, for example:

```
trap
  exception Foo is E1 endexn
  exception Bar is E2 endexn
in
  E
endtrap
```

Note that Foo and Bar are only trapped in E, *not* in either E_1 or E_2. So if E raises Foo or Bar, then it will be handled, but if E_1 or E_2 raises Foo or Bar then it will not.

In addition, we can write a 'handler' for the successful termination of an expression, for example:

```
trap
  exception ParseError is 0 endexn
  exit (x:string) is string2int (x) [ParseError] endexit
in
  E
endtrap
```

This is useful in the case where we want any ParseError exception raised by E to be trapped, but *not* any ParseError exception raised by the call to string2int. It is impossible to write this without the capability to handle successful termination—of the two obvious 'solutions', one does not type-check:

```
string2int (
  trap
    exception ParseError is 0 endexn
  in
    E
  endtrap
) [ParseError]
```

and the other traps the ParseError exception raised by string2int:

```
trap
  exception ParseError is 0 endexn
in
  string2int (E) [ParseError]
endtrap
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The **trap** operator both declares and traps the exception—this means it is impossible for an exception to escape outside its scope. This can be contrasted with a language such as SML where exception declaration and handling are separated, so it is possible for exceptions to escape their scope:

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

Note that the only way in which an exception can be observed by its environment is by trapping it—it is impossible for expressions to synchronize on exceptions.

Imperative features The data expression language is functional, but supports a language of record expressions which mimics an imperative language with write-many variables. For example, the imperative expression:

```
?x := 0; ?y := "hello world";
```

is equivalent to the behaviour:

```
(x=>0,y=>"hello world")
```

The simplest imperative expression is an assignment P `:=` E, where P is an irrefutable pattern and E an expression, for example:

```
?x := 4
```

As we remarked earlier, we allow the use of out parameters as syntax sugar for assignment, for example:

```
partition (y,ys,?l,?g)
```

is shorthand for:

```
(?l,?g) := partition (y,ys)
```

There is a sequential composition operator whose syntax is E_1 ; E_2. It is like the LOTOS enabling operator because it combines two expressions, but it has a slightly different semantics: it does not perform an internal **i** action.

The **var** operator is used to restrict the scope of variables, with syntax **var** *ILV* **in** E **endvar**, where *ILV* is a list of typed variables. For example:

```
var
  x:int
in
  ?x:=E; x*x
endvar
```

has the same semantics as $E * E$. Optionally, some of the local variables can be initialized, for example we could have written:

```
var
  x:int:=E
in
  x*x
endvar
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

An iteration (or loop) operator is included in the language. This operator allows recursive processes to be specified without using explicit process identifiers.

Loops with local variables can be declared—these local variables can be initialized, and should then be assigned to on each iteration of the loop. A loop can be broken with a **break** command. For example, an imperative function to sum a list of numbers can be defined:

```
function sum (xs:intlist) : int is
  var ys:intlist:=xs,
    total:int:=0
  in
    loop
      case ys is
          nil -> break (total)
        | cons(?z,?zs) -> ?total := total+z; ?ys := zs
      endcase
    endloop
  endvar
endfunc
```

The breakable loop is then defined using exception handling, for example the above loop is shorthand for:

```
trap
  exception Inner (x:int) is x endexn
in
  loop
    var ys:intlist:=xs,
      total:int:=0
    in
      case ys is
        nil -> raise Inner (total)
        | cons(?z,?zs) -> ?total := total+z; ?ys := zs
      endcase
    endvar
  endloop
endtrap
```

We also allow named loops, so that you can break a loop other than the innermost one, for example:

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

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

As an example of the imperative features, an imperative definition of quicksort partitioning is as follows:

```
function partition (x:int,xs:intlist) : (intlist,intlist) is
  loop
    var
      less:intlist:= nil,
      gtr:intlist:= nil,
      rest:intlist:= xs
    in
      case rest is
           nil -> break ((less,gtr))
        | cons(?y,?ys) [y < x] ->
             ?less := cons(y,less); ?gtr := gtr; ?rest := ys
        | cons(?y,?ys) ->
             ?less := less; ?gtr := cons(y,gtr); ?rest := ys
      endcase
    endvar
  endloop
endfunc
```

It can be compared with the functional definition:

```
function partition (x:int,xs:intlist) : (intlist,intlist) is
  case xs is
       nil ->
         (nil,nil)
    | cons(?y,?ys) ->
        var
          less:intlist,gtr:intlist
        in
          (?less,?gtr) := partition (x,ys)
          if x > y
            then (cons(y,less), gtr)
            else (less, cons(y,gtr))
          endif
        endvar
  endcase
endfunc
```

Besides, we provide usual **while** and **for**, which are syntactic sugar for **loop**.

```
function fact (n:int) : int raises [OutOfRange] is
  if n < 0 then raise OutOfRange endif
  ;
  var x:int,res:int:= 1 in
    for x:= 2 while x <= n by x:= x+1 do
      res:= res * x
    endfor
  endvar
endfunc
```

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
function NumElements (L:intlist) : int is
  var x:int,total:int:= 0 in
    while L <> NIL do
      total:= total + 1 ;
      cons (?x, ?L):= L
    endwhile
  endvar
endfunc
```

Static semantics The static semantics for expressions is given by translating them into the behaviour language described below. For expressions which do not assign to variables, the typing is given by judgements:

$$\mathcal{C} \vdash E \Rightarrow \textbf{exit}\ (T)$$

meaning 'in context $\mathcal{C}$, expression E has result type T'. The context $\mathcal{C}$ gives the type for each of the free identifiers used in E, for example we can deduce:

$$\mathrm{x} \Rightarrow \mathrm{int},\ * \Rightarrow (\mathrm{int},\mathrm{int}) \rightarrow \textbf{exit}\ (\mathrm{int}) \quad \vdash \quad \mathrm{x} * \mathrm{x} \Rightarrow \textbf{exit}\ (\mathrm{int})$$

meaning 'in a context where x is an integer and $*$ is a function from pairs of integers to integers, then $\mathrm{x} * \mathrm{x}$ returns an integer'.

Expressions which assign to variables but do not return a result have typing given by judgements:

$$\mathcal{C} \vdash E \Rightarrow \textbf{exit}\ (V_1 \Rightarrow T_1, \ldots, V_n \Rightarrow T_n)$$

meaning 'in context $\mathcal{C}$, expression E assigns to variables V_1 through to V_n the types T_1 through to T_n'. For example we can deduce:

$$2 \Rightarrow \mathrm{int} \quad \vdash \quad \mathrm{?x:=2} \Rightarrow \textbf{exit}\ (\mathrm{x} \Rightarrow \mathrm{int})$$

meaning 'in a context where 2 is an integer, then ?x:=2 assigns an integer to the variable x'.

Expressions which both assign to variables and return a result have typing given by judgements:

$$\mathcal{C} \vdash E \Rightarrow \textbf{exit}\ (T,\ V_1 \Rightarrow T_1, \ldots, V_n \Rightarrow T_n)$$

which combines the above two semantics. For example:

$$2 \Rightarrow \mathrm{int},\ * \Rightarrow (\mathrm{int},\mathrm{int}) \rightarrow \textbf{exit}\ (\mathrm{int}) \quad \vdash \quad \mathrm{?x:=2;\ x} * \mathrm{x} \Rightarrow \textbf{exit}\ (\mathrm{int},\mathrm{x} \Rightarrow \mathrm{int})$$

meaning 'in a context where 2 is an integer and $*$ is a function from pairs of integers to integers, then ?x:=2; $\mathrm{x} * \mathrm{x}$ assigns an integer to the variable x and returns an integer'.

Note that x is not free in the expression ?x:=2; $\mathrm{x} * \mathrm{x}$ since it is bound by the assignment statement. This is reflected in the type judgement above, which does not require x to be in the context.

Dynamic semantics The dynamic semantics of data expressions is defined by the translation into behaviour expressions. There are two ways in which a data expression can have observable behaviour: either it terminates successfully, or it raises an exception.

Expressions which terminate successfully with a value have dynamic semantics given by judgements:

$$\mathcal{E} \vdash E \xrightarrow{\delta(N)} E'$$

meaning 'in environment $\mathcal{E}$, the expression E returns normal form N and then behaves like E''. As it happens, E' will always be an expression with no behaviour, since an expression cannot do anything after terminating, but we use this notation for symmetry with the case of exception raising. The context gives the bindings of function identifiers, and other similar static information required at run-time. For example:

$$\vdash \quad 2 * 2 \xrightarrow{\delta(4)} \textbf{block}$$

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

maning 'the expression $2*2$ returns the value 4 and then has no observable behaviour'.

Expressions which terminate successfully having assigned values to variables have dynamic semantics given by judgements:

$$\mathcal{E} \vdash E \xrightarrow{\delta(V_1 \Rightarrow N_1, \ldots, V_n \Rightarrow N_n)} E'$$

meaning 'in context $\mathcal{E}$, the expression E assigns normal forms N_1 through to N_n to variables V_1 through to V_n'. For example:

$$\vdash \quad \texttt{?x:=2} \xrightarrow{\delta(x \Rightarrow 2)} \textbf{block}$$

meaning 'the expression ?x:=2 terminates, having assigned the value 2 to the variable x, and then has no observable behaviour'.

Expressions which both assign to variables and return a result have dynamic semantics given by judgements:

$$\mathcal{E} \vdash E \xrightarrow{\delta(N, V_1 \Rightarrow N_1, \ldots, V_n \Rightarrow N_n)} E'$$

combining the two semantics, for example:

$$\vdash \quad \texttt{?x:=2; x*x} \xrightarrow{\delta(4, x \Rightarrow 2)} \textbf{block}$$

Similarly, the semantics of exceptions is given by judgements:

$$\mathcal{E} \vdash E \xrightarrow{X(N)} E'$$

For example:

$$\textbf{raise}\ X(1) \quad \xrightarrow{X(1)} \quad \textbf{block}$$

The semantics is defined formally in Subclausess 6.2.8.15 and 9.2.27.

A.1.1.4 Behaviour expressions

Some knowledge of LOTOS is assumed in this International Standard. However, for completeness, we provide the syntax of Basic LOTOS (i.e. LOTOS without datatypes) together with some brief explanations.

$$B ::= \textbf{stop} \mid \textbf{exit} \mid \Pi[G^*] \mid G;B \mid \textbf{i};B \mid B \ [] \ B \mid B \ |[G^*]| \ B \mid \textbf{hide}\ G^*\ \textbf{in}\ B \mid B >> B \mid B \ [> B$$

The semantics is as follows:

- Deadlock: **stop** is an inactive behaviour.
- Termination: **exit** is a behaviour that terminates successfully. It performs an action on gate δ and then deadlocks.
- Process instantiation: $\Pi[\vec{G}]$ instantiates the previously delared process definition with parameters $\vec{G}$.
- Action-prefix: $G;B$ is a behaviour that first performs action G and then behaves like B.
- Internal action-prefix: $\textbf{i};B$ is a behaviour that first performs the internal action **i** and then behaves like B.
- External choice: B_1 [] B_2 is a process that can behave either like B_1 or like B_2 depending on the environment.
- Parallelism: B_1 $|[\vec{G}]|$ B_2 is the parallel composition of B_1 and B_2 with synchronisation on the gates in $\vec{G}$.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- Abstraction: **hide** $\vec{G}$ **in** B hides in behaviour B all the actions from the set $\vec{G}$, i.e. it renames them into **i**.

- Enabling: B_1 >> B_2 is the sequential composition of B_1 and B_2, i.e. B_2 can start when B_1 has terminated successfully.

- Disabling: B_1 [> B_2 allows B_2 to disable B_1 provided B_1 has not terminated successfully.

The main differences between this language and the E-LOTOS base language that we have designed are as follows:

- Actions are particular behaviours and the two forms of sequential composition (action-prefix and enabling) are unified.

- New features are added such as pattern-matching, exceptions, assignment, time and other operators (e.g. an explicit renaming operator).

The behaviour language can be seen as an extension of the data language with communication between parallel processes and real-time features.

Communication Behaviours can communicate on *gates*. The simplest communicating process is one which synchronizes on a gate G: this is just written G. Such synchronizations can then be sequentially composed, for example a behaviour which alternates between in and out actions is:

```
loop
  inp; outp
endloop
```

Behaviours can also send or receive data on gates, for example a one-place integer buffer is:

```
loop
  var x:int
  in
    inp ?x; outp !x
  endvar
endloop
```

Here the variable x is *bound* by the communication on the inp gate, and is *free* in the communication on the outp gate. The resulting behaviour copies integers from the inp gate to the outp gate.

When synchronizing on a gate, you can specify any pattern to synchronize on, for example:

```
G(age=>!28,name=>?na,address=>(number=>?no,street=>!"Acacia Ave",etc))
```

will synchronize on any person aged 28 living in Acacia Avenue, and will bind the variables na and no appropriately. This use of patterns in communications is the main reason for allowing ? and ! in patterns.

You can also specify a *selection predicate* specifying whether a synchronization should be allowed, for example to select anyone in their 20s living on Acaica Avenue, you might say:

```
G(age=>?a,name=>?na,address=>(number=>?no,street=>!"Acacia Ave",etc))
  [20 ≤ a andalso a ≤ 29]
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Gate parameters are given in process declarations, for example:

```
process Buffer [inp:int,outp:int] is
  loop
    var x:int
    in
      inp ?x; outp !x
    endvar
  endloop
endproc
```

Gates may be typed: by default each gate has type *any*, so can communicate data of any type, for example:

```
process OverloadingExample [overloaded] (out x:int, out y:bool) is
  overloaded(?x:int);
  overloaded(?y:bool)
endproc
```

The first communication on the overloaded gate has to be of type integer, and the second has to be of type boolean.

We can use **as** patterns to match against all or some of a record. This is particularly useful when the record is extensible, for example we can write a simple router capable of handling any type of data as:

```
type S is (de=>dest,etc) endtype
type AnyRecord is (etc) endtype
process Router [inp:S,left,right] is
  var destination:dest, data:AnyRecord
  in
    inp(de=>?destination,?data as etc);
    case destination is
        L -> left(!data)
      | R -> right(!data)
    endcase;
    Router [inp,left,right]
  endvar
endproc
```

Concurrency Concurrent behaviours can synchronize on their communications. For example, two behaviours which are forced to synchronize on all communications are:

```
    G(address=>(number=>?no,street=>!"Acacia Ave",etc),etc)
|| G(age=>!28,name=>?na,address=>any:addrType)
```

Since the two behaviours are forced to synchronize on the gate *G*, this has the same semantics as:

G(age=>!28,name=>?na,address=>(number=>?no,street=>!*T"AcaciaAve*",**etc**))

Data may be communicated in both directions in a synchronization, for example:

```
    G(age=>!28,name=>?na,etc);B1
||  G(age=>?a,name=>!"Fred",etc);B2
```

has the same semantics as:

```
G(age=>!28,name=>!"Fred",etc);
(?na:="Fred"; B1) || (?a:=28; B2)
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Parallel behaviours have to synchronize on termination, for example the following will terminate immediately, after setting variables x and y:

```
?x:=1 || ?y:=2
```

Two behaviours which have no synchronizations at all (apart from synchronizing on termination) are:

```
    overloaded(?x:int)
||| overloaded(?y:bool)
```

This will communicate twice on the overloaded gate: once inputting an integer, and once inputting a boolean, but the order is unspecified. Once both inputs have happened, the process can terminate. This process has the same semantics as:

```
    overloaded(?x:int); overloaded(?y:bool)
[]  overloaded(?y:bool); overloaded(?x:int)
```

Note that the variables bound by concurrent processes are all the variables bound by the components, and that there is no possibility of communication by shared variables.

General parallel operator E-LOTOS has a parallel operator which allows explicit synchronization and "n among m" synchronization:

par $G_1\#n_1, ..., G_p\#n_p$ **in**
  $[\Gamma_1]$ **for** B_1
|| ...
|| $[\Gamma_n]$ **for** B_n
endpar

This operator says that: if some B_i can do an action with name G, and this action is specified in the Γ_i set (the *synchronization list*), then this action must be synchronized with the actions of the others behaviours as follows:

- if G is specified in the *list of degrees* $(G_1\#n_1, ..., G_p\#n_p)$ with the degree n, then B_i has to synchronize in G with n-1 other behaviours which have G in their synchronization list;
- if G does not appear in the list of degrees, then it has to synchronize with all other behaviours having G in their synchronization list.

On the other hand, if G is not in the Γ_i set, B_i can do it in interleaving with the other processes B_j.

Time Behaviours have real-time capabilities, given by three constructs:

- a time type, with addition and comparisons on times,
- a **wait** operator, to introduce delays, and
- an extended communication operator, which is sensitive to delay.

The time datatype is a total order with addition. We shall let d range over values of type time.

The delay operator is just written **wait**(d) which delays by time d and then terminates. For example a behaviour which communicates on gate G every time unit is:

loop
  G;
  wait(1)
endloop

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

We can delay by an arbitrary time expression **wait**(*E*), for example:

```
loop
  var x:time
  in
    G ?x;
    wait(x)
  endvar
endloop
```

Also, we have a simple way to write a nondeterministic delay:

```
loop
  var x:time
  in
    G;
    ?x := any time;
    wait(x)
  endvar
endloop
```

Communications can be made sensitive to time by adding a `@`*P* annotation, which matches the pattern *P* to the time at which the communication happens (measured from when the communication was enabled). For example:

G(?x:int)@?t[t < 3]

is a behaviour that agrees to accept an integer value (to be bound to variable x), provided that less than 3 time units have passed, whereas:

G(?x:int)@!3

is similar, but the action can only occur at time 3, because the pattern variable has been replaced by a pattern value !3. This behaviour has the same semantics as:

```
var
  t:time
in
  G ?x:int@?t[t = 3]
endvar
```

The time features are directly inspired by ET-LOTOS [18] but are adapted to fit with other new paradigms of the language, such as:

- action is a behaviour,
- sequential composition does not generate an **i** action,
- the presence of pattern-matching,
- the presence of exception raising and handling.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Urgency An important concept is *urgency*: a behaviour is urgent if it cannot delay—for example if there is a computation which must be performed immediately. For example, sequential composition is urgent—once the first behaviour terminates, control is immediately passed to the second without delay. For example, consider the process:

```
loop
   (loop tick endloop [> wait(1));
   (loop tock endloop [> wait(1))
endloop
```

This will perform any number of 'tick' actions during the first time interval, then at time 1 control is handed over, and any number of 'tock' actions is performed until time 2, and so on. Each of the hand-overs is urgent, so we know it is impossible for a 'tick' action to happen in an even time interval, or a 'tock' action to happen in an odd time interval.

In E-LOTOS, the urgent actions are:

- Internal (**i**) actions, whether written explicitly or caused by hiding.
- Exception raising (X) actions.
- Termination (δ) actions.

All of these actions happen immediately. However, there is one exception to the urgency of these actions: it is possible for a termination to be delayed by a parallel behaviour. For example the following behaviour will terminate at time 2:

```
wait(1) ; exit || wait(2) ; exit
```

The urgent semantics of exceptions given here is basically the same as the 'signals' model of Timed CSP [7].

Hiding The syntax for hiding is like in existing LOTOS, except that the (declared) gates are typed. For example in:

```
hide mid:int in
   Buffer [inp,mid] || Buffer [mid,outp]
endhide
```

a new mid gate is declared, which can communicate integers, and is then replaced by internal **i** actions. This operator preserves the property of urgency of all **i**, and allows the modelling of urgency on hidden synchronization. This means that one can express that a synchronization should occur as soon as made possible by all the processes involved. For example the behaviour:

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

has the same semantics as:

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

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

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The behaviour:

hide G **in**
  G@?t[t ≥ 3] ; B
endhide

has the same semantics as:

wait(3) ; ?t:=3; **i**;
hide G **in** B **endhide**

Again the earliest possible time for G to occur is after 3 time units.
The behaviour:

hide G **in**
  G@?t[t > 3] ; B
endhide

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

wait(4) ; ?t:=4; **i**;
hide G **in** B **endhide**

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

wait(3) ; **block**

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

Having to hide synchronizations to make them occur as soon as possible is sometimes criticized, because there are cases where one would like to still observe those gates. The problem here lies in the interpretation of the word 'observation'. Observing requires interaction, and interaction may lead to interference. Clearly, we would like to show the interaction to the environment without allowing it to interfere. There is a nice solution to this problem. It suffices to raise an exception (signal) immediately after the occurrence of the hidden interaction as follows. Consider two processes, Producer and Consumer, that want to synchronise on the sync event as soon as they are both ready to do so. We add a special monitoring process that synchronizes with them and sends a signal just after sync occurred:

Producer	:=	**var** t:time **in** ?t:= **any** time; **wait**(t) **endvar**; sync; Producer
Consumer	:=	sync; **var** t:time **in** ?t:= **any** time; **wait**(t) **endvar**; Consumer
Monitoring	:=	sync; **signal** yes ; Monitoring
System	:=	**hide** sync **in** (Producer || Consumer || Monitoring)

The **signal** operator is the same as **raise** except that it allows computation to carry on after the exception has been raised: **raise** X is shorthand for **signal** X ; **block**.

Suspend/Resume This operator is an extension of the LOTOS disabling which allows the resume of the interrupted behaviour. The resume is done through an exception gate which is specified inside the operator. For example, in process:

wait(4) ; B_1 [X> **wait**(1) ; i; **wait**(2) ; **raise**(X)

the left behaviour is continously suspended by the internal action after one time unit. The left behaviour is suspended during two time units and then resumes via the exception X. The left behaviour is blocked when it

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

is suspended; it does not evolve in time or in actions. It results, in our example, that this process is resumed in the following state after its first suspension:

$$\mathbf{wait}(3)\,;\ B_1\ \ [\mathrm{X}>\ \ \mathbf{wait}(1)\,;\ \mathrm{i};\ \mathbf{wait}(2)\,;\ \mathbf{raise}(\mathrm{X})$$

and not in

$$\mathbf{wait}(1)\,;\ B_1\ \ [\mathrm{X}>\ \ \mathbf{wait}(1)\,;\ \mathrm{i};\ \mathbf{wait}(2)\,;\ \mathbf{raise}(\mathrm{X}).$$

The right behaviour is always restarted after a resume, it means, for instance, that the following two expressions have the same semantics:

$$B_1\ \ [\mathrm{X}>\ \ B_2\,;\ \mathbf{raise}(\mathrm{X})$$
$$B_1\ \ [\mathrm{X}>\ \ B_2\,;\ \mathbf{signal}(\mathrm{X})\,;\ B_3$$

This quite simple operator can be used to specify more complex interruption mechanisms where, for instance, a behaviour can suspend several behaviours. For example, consider the process:

$$(B_1\ [X_1>\ G_1\,;\ G_1'\,;\mathbf{raise}(X_1)\ \ |||\ \ B_2\ [X_2>\ G_2\,;\ G_2'\,;\mathbf{raise}(X_2))$$
$$|\,[G_1,G_2,G_1',G_2']\,|$$
$$B_3$$

The behaviour B_3 control B_1 and B_2 through the gate G_i and G_i'. For instance, B_3 can suspend B_1 with the gate G_1 and resume it with G_1'. In this way, it is possible to specify more complex interruption mechanisms used, for instance, in real-time schedulers.

Let us remark that the LOTOS disabling is a special case of this Suspend/Resume operator where no exception gate has been specified.

Renaming An explicit renaming operator is introduced in the language. It allows one to rename observable actions into observable actions, or exceptions into exceptions.

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

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

rename gate
  G (x=>?i:**int**) **is** G'(x=>!i)
in
  B
endren

Note the syntactic similarity between renaming and function declaration or exception trapping. This form of renaming is so common that we provide a shorthand for it:

rename gate
  G (x=>**int**) **is** G'
in
  B
endren

We can remove a field from a gate:

rename gate
  G (x=>?i:**int**,y=>**any**:**bool**) **is** G'(!i)
in
  B
endren

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

We can add a field to a gate:

```
rename gate
  G (x=>?i:int) is G'(x=>!i,y=>!true)
in
  B
endren
```

We can merge two gates G' and G'' into a single gate G:

```
rename
  gate G' (x=>?i:int) is G(x=>!i,y=>!true)
  gate G'' (x=>?i:int) is G(x=>!i,y=>!false)
in
  B
endren
```

We can also split one gate G into two gates G_1 and G_2:

```
rename
  gate G (x=>?i:int,y=>!1) is G1(!i)
  gate G (x=>?i:int,y=>!2) is G2(!i)
in
  B
endren
```

We can rename exceptions in a similar way.

Imperative feature for behaviours As with expressions, a number of "imperative" features are supported to ease writing specifications. Some examples with **loop** has been presented. As the rest are obvious, we only present some little examples:

```
process protocol [down:packet,up] is
  var
    code:packet,
    data:AnyRecord
  in
    down(de=>?code,?data as etc) ;
    while code<>disconnect do
      up!data ;
      down(de=>?code,?data as etc)
    endwhile
  endvar
endproc
```

Static semantics The static semantics for behaviour expressions is very similar to that of data expressions, and is given by judgements:

$$\mathcal{C} \vdash B \Rightarrow \textbf{exit}\ (\vec{V} \Rightarrow \vec{T})$$

For example:

$$G \Rightarrow \textbf{gate any} \vdash (G(?x:\text{int})\ |||\ G(?y:\text{bool})) \Rightarrow \textbf{exit}\ (x \Rightarrow \text{int}, y \Rightarrow \text{bool})$$

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

However, it is useful to identify the behaviour expressions that cannot terminate initially, i.e. without having first performed an (internal or observable) action or sent a signal. These behaviours will type as:

$$\mathcal{C} \vdash B \Rightarrow \textbf{guarded}\ (\vec{V} \Rightarrow \vec{T})$$

And of course, the following rule is valid:

$$\frac{\mathcal{C} \vdash B \Rightarrow \textbf{guarded}\ (\vec{V} \Rightarrow \vec{T})}{\mathcal{C} \vdash B \Rightarrow \textbf{exit}\ (\vec{V} \Rightarrow \vec{T})}$$

By carefully requiring behaviour expressions to be so guarded in some contexts (e.g. choice, disabling and suspend/resume), we can preserve time determinism even though sequential composition and exception handling do not introduce an internal action.

Dynamic semantics The dynamic semantics of data expressions is given by two kinds of reduction:

- Successful termination $\mathcal{E} \vdash E \xrightarrow{\delta(RN)} E'$.
- Exception raising $\mathcal{E} \vdash E \xrightarrow{X(RN)} E'$.

The dynamic semantics of behaviour expressions extends this with three new kinds of judgement:

- Internal actions $\mathcal{E} \vdash B \xrightarrow{\mathbf{i}()} B'$.
- Communication $\mathcal{E} \vdash B \xrightarrow{G(RN)} B'$.
- Delay $\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B'$.

For example (up to strong bisimulation):

$$\begin{array}{lll} \mathbf{i};\ G(?\mathrm{t})\,;\mathbf{wait}(\mathrm{t}) & \xrightarrow{\mathbf{i}()} & G(?\mathrm{t})\,;\mathbf{wait}(\mathrm{t}) \\ & \xrightarrow{G(3)} & ?\mathrm{t}:=3;\ \mathbf{wait}(3) \\ & \xrightarrow{\varepsilon(2)} & ?\mathrm{t}:=3;\ \mathbf{wait}(1) \\ & \xrightarrow{\varepsilon(1)} & ?\mathrm{t}:=3;\ \mathbf{wait}(0) \\ & \xrightarrow{\delta(\mathrm{t}\Rightarrow 3)} & \mathbf{block} \end{array}$$

The urgency of internal, exception and termination actions is given by the properties:

- No behaviour B can offer both $\xrightarrow{\mathbf{i}()}$ and $\xrightarrow{\varepsilon(d)}$.
- No behaviour B can offer both $\xrightarrow{X(RN)}$ and $\xrightarrow{\varepsilon(d)}$.
- No behaviour B can offer both $\xrightarrow{\delta(RN)}$ and $\xrightarrow{\varepsilon(d)}$.

For example:

$$?\mathrm{t}:=3 \xrightarrow{\delta(\mathrm{t}\Rightarrow 3)} \mathbf{block}$$

but:

$$?\mathrm{t}:=3 \not\xrightarrow{\varepsilon(d)}$$

However, in order to get the correct synchronization semantics for termination, we have to allow terminated processes to age when placed in a parallel context. Consider the following example:

$$?\mathrm{t}:=3 \ ||\ \mathbf{wait}(2)\,;?\mathrm{y}:=\mathrm{true}$$

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

We would like this to have semantics (up to strong bisimulation):

$$
\begin{array}{lll}
\texttt{?t:=3 || } \mathbf{wait}\texttt{(2);?y:=true} & \xrightarrow{\varepsilon(1)} & \texttt{?t:=3 || } \mathbf{wait}\texttt{(1);?y:=true} \\
 & \xrightarrow{\varepsilon(1)} & \texttt{?t:=3 || } \mathbf{wait}\texttt{(0);?y:=true} \\
 & \xrightarrow{\delta(t \Rightarrow 3, y \Rightarrow true)} & \mathbf{block}
\end{array}
$$

In order to achieve this, we allow terminated processes to age in a parallel composition. The alternative would be to treat δ actions (and sequential composition) in the same way as gates (and hiding), but this would have introduced many negative premises into the semantics (for example sequential composition and exception handling), which we have tried to avoid. The semantics presented here only uses negative premises in the semantics of hiding.

A.2 The module language

LOTOS has only a limited form of modules, which encapsulate data types and operations but not processes. Moreover, this mechanism does not support abstraction: every object declared in a module is exported outside. These deficiencies make LOTOS difficult to use, and cause problems for users and tool implementors alike. A critical evaluation of LOTOS data types from the user point of view can be found, for instance, in [23].

One of the goals of E-LOTOS is to develop a modularization system, which should allow export and import, hiding, and generic modules. The modules used in the data part should be the same as those used in the behavioural part, so 'process' declarations should be allowed as well as 'type' and 'operation' declarations. For abstraction and code re-use, interfaces and generic modules are very useful.

For example, a simple router of packets containing a data field and an address field is specified in LOTOS

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

as follows:

```
specification Router[in,left,right] : noexit :=
  type Natural is
    sorts nat
    ...
  endtype
  type Data is
    formalsorts data
  endtype
  type Packet is Data
    sorts
      packet,dest
    opns
      mkpacket : dest,data -> packet
      getdest : packet -> dest
      getdata : packet -> data
      L : -> dest
      R : -> dest
    eqns forall p:packet, de:dest, da:data
      ofsort packet mkpacket (getdest (p),getdata (p)) = p
      ofsort dest getdest (mkpacket (de,da)) = de
      ofsort data getdata (mkpacket (de,da)) = da
  endtype
  type NatPacket is Packet actualizedby Natural using
    natpacket for packet
    data for nat
  endtype
  behaviour Router[in,left,right]
  where
    process Router [in,left,right] : noexit :=
    in?p:natpacket;
      (
        [getdest(p) = L] -> left!getdata(p);Router [in,left,right]
      [] [getdest(p) = R] -> right!getdata(p);Router [in,left,right]
      )
    endproc
endspec
```

Apart from readability problems of the specification language discussed in Subclause A.1, this specification suffers from some problems of re-usability. For example, it is not possible to parameterize the Router process specification by a generic data type, because the behaviour part of standard LOTOS must refer only to fully instantiated types. So, to obtain a Router dealing with boolean data, one has to re-write the process Router to accept BoolPackets instead of NatPackets.

This can be compared with the equivalent specification using the module language presented here (the

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

base language is one presented in Subclause A.1):

```
interface Data is
  type data
endint
module Destination is
  type dest is L|R endtype
endmod
generic Router(D:Data) imports Destination is
  type packet is (de => dest, da => data) endtype
  process Router[in:packet, left:data, right:data] is
    ...
  endproc
endgen
module NatRouter is
  Router (Natural renaming (types nat := data))
endmod
specification Router imports NatRouter is
  gates in:any, left:any, right:any
  behaviour
    Router [in, left, right]
endspec
```

Note that:

- Abstract data types, like data, can be declared into interfaces.
- Generic modules are parameterized by interfaces, in a functional style.
- Process and behaviours can be declared in modules and generic modules; generic data types can be used into behaviour expresions.
- Modules and generic modules can import other modules in order to use their definitions.
- Modules present a "by default" interface, which contains all the definitions. However, an interface can restrain the module through an explicit declaration.

  modules, and to avoid name clashes. However, this clumsy notation can be avoided by opening modules.

- Generic modules are instantiated by modules which match a specified interface (via a renaming). For example, the Natural module matches the interface Data if we map the type data to nat. So, the generic router module can be instantiated with Natural to obtain a router managing naturals as data.
- A specification entity can import modules already declared. The body of the specification could be a behaviour expression or an expression. The gates and exceptions used in the body have to be declared.
- The name space is flat. An extension of this proposal with a dot notation for identifiers will be investigated.

For this proposal we present a complete abstract syntax and a static semantics.

a base language as defined in [26]. To attain it, we present a complete abstract syntax, some derived forms (as syntactic sugar), and a static semantics. The flattening function should be provided.

The abstract syntax contains: specification declaration, module declaration, module expressions, generic module declarations, interface declarations, interface expressions, and declarations.

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The static semantics of this language is formally defined. It is based on judgments such as $\mathcal{B} \vdash$ *top-dec* $\Rightarrow \mathcal{C}$ meaning 'in the context $\mathcal{B}$ of top-level declarations, the top level definition *top-dec* is well formed and gives the context $\mathcal{C}$'. For example:

$$\vdash (\textbf{module}\ \text{OnePoint}\ \textbf{is}\ \textbf{type}\ \text{M}\ \textbf{is}(\text{x}:\text{int},\text{y}:\text{int})) \Rightarrow (\text{OnePoint} \Rightarrow \{\text{M} \Rightarrow \textbf{type}, \text{M} \equiv (\text{x} \Rightarrow \text{int}, \text{y} \Rightarrow \text{int})\})$$

means 'then the OnePoint module definition is bound to a context (signature) where the type M is declared, and it is equivalent to the type pair $(x : \text{int}, y : \text{int})$'. We assume in this subclause some built-in (pervasives) types like int, bool, etc. The static semantics includes:

- Abstract data types.
- Modules declaration and module expressions.
- Generic modules parameterized by interfaces, and generic module instantiation.
- Interface declarations and expressions.
- Enrichment, sub-typing, and matching relation between interfaces and modules semantic objects.
- Renaming of modules and interfaces.

The module language described in this paper is based on previous discussions for the module language [15] and on previous proposals for LOTOS with a module system [27].

A.2.1 Basic concepts

We provide here a short introduction to this proposal by giving example of how it responds to issues formulated in the questionnaire of [25, Subsection 7.2].

A.2.1.1 Naming

The domain of names is not structured. However, an extension of this proposal with a structured domain for identifiers will be investigated.

A.2.1.2 Specification structuring

A specification in modular-E-LOTOS is given as a sequence of *module* declarations, *generic modules* declarations, and *interface* declarations.

Modules Modules are sequences of *declarations* of types, constructors, processes, functions, and value constants. For types, functions, processes, and values constants the user has to provide an implementation. Modules can be declared using *module declarations*, whose simplest form is:

module *mod-id* **is** *dec* **endmod**

where *mod-id* is a module identifier and *dec* is a (base language) declaration, enriched with value constant declarations. For example, a one-point domain has the following declaration:

```
module OnePoint is
  type M is zero() endtype
  value 0:M is zero() endval
  function infix + (x:M,y:M):M is
    case(x,y) is
      (!zero(),!zero()) -> 0
    endcase
  endfunc
endmod
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Unlike the SML module system, we have not nested modules; the possibility to declare nested modules in SML is unusual, makes the module system more complex, and it is not obvious whether the extra complexity is necessary.

Processes can be declared in modules. For example, the specification of a data-flow process is:

```
module DataFlow is
  process Flow [In:PairOfInts, Out:int] is
    In(?x=>int, ?y=>int); Out!(x+y); Flow [In, Out]
  endproc
endmod
```

Note that the type **int** is a built-in type, so no importation clauses are needed.

Interfaces Intuitively, an *interface* is a module type. Whereas a module expression declares a module, an interface expression specifies a class of modules. For example, the data-flow module has the interface:

```
interface DataFlow is
  process Flow [In:pairOfInts, Out:int]
endint
```

An interface is not the type of any particular module, but rather of a whole class of modules, namely all the modules that *match* the interface. For example, the interface DataFlow can be matched by any module which has at least a a process with name Flow, gates of type (**int**, **int**) and **int**, and with functionality **exit**(**none**) (**noexit** in LOTOS).

In the language we accept equations to be specified in interfaces. For example, the Monoid interface is:

```
interface Monoid is
  type M
  value 0:M
  function infix + (M, M) : M
  eqns forall x, y, z : M ->
    (0 + x) = x;
    (x + 0) = x;
    ((x + y) + z) = (x + (y + z));
  endeqns
endint
```

Tools could treat equational specifications just as (type checked) comments, so we should ensure that (as in Extended ML) the equations can be commented out without affecting the semantics of the module.

Generic Modules Genericity is a useful tool to construct specifications and for code reuse. Here we provide a mean for genericity using generic modules. Generic modules allow standard libraries of components to be built up, and support code reuse of both components and glue. The simplest declaration of a generic module is:

generic *gen-id* (*mod-id* : *int-id*) **is** *dec* **en dgen**

where *gen-id* is a generic module identifier, *mod-id* is a identifier for a formal module matching the interface *int-id*, and *dec* are declarations of the base language.

Generic modules cannot be parameterized by generic modules.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

So, as well as specifying the exports of a module, interfaces are also used to specify the parameters of a generic module. For example, a generic list module can be implemented using monomorphic lists as follows:

```
interface List imports Monoid is
  type E
  function inj(x:E):M
endint
interface EqType is
  type E
endint
generic GenericList (Eq:EqType):List is
  type M is nil()|cons(E,M)endtype
  value 0 is nil() endval
  function infix + (s1:M,s2:M):M is
    case s1 is
        nil() → s2
      | cons(?x,?xs) → cons(x,xs+s2)
    endcase
  endfunc
  function inj(x:E):M is cons(x,nil()) endfunc
endgen
```

This module can be used as follows:

```
module ListNat:[List renaming (types E := nat)]is
  GenericList(Natural renaming (types nat := E))
endmod
```

A.2.1.3 Abstraction, Hiding

Abstraction is present by abstract data types declaration in interfaces. An abstract data type can be specified as follows:

type *S*

This abstract data type may be implemented by a lot of concrete (or manifest) data types. For example, if we declare:

```
interface Set is
  type Element
  type Set
  value empty:Set
  function insert(x:Element,s:Set):Set
  function delete(x:Element,s:Set):Set
  function member(x:Element,s:Set):bool
endint
```

several implementations for type Set may be given: using list, binary trees ...

An issue of abstract data types (ADT) is type equality. Equality is an important concept in E-LOTOS, since it is used implicitly by synchronization. So far, all types allow equality, but the module system can introduce data abstraction, so it is no longer possible to see the internal representation of a data type. Our proposal consider abstract data types as equality types. Users have to define an equality function for each abstract data type used in communication.

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

subtypes of constructed types. For example, the declaration of the following ADT: in the base language is the possibility to statically type check and an efficient implementation.

used in the in the specification to provide binding depending on context. For example, another specification of Set interface is: current environment.

Another means for abstraction is hiding. It constrains an existing module interface (or view) by another (more general) interface, provided that the module matches the second interface. The effect is to obtain different views of the module, depending on the current interface of the module. Consider, for example, the following specification:

```
interface VIEW1 is
  type S
  value x:S
  value y:S
endint
interface VIEW2 is
  type S
  type pairS is (x=>S,y=>S) endty pe
endint
module A is
  type S is ACK()|REQ() endtype
  value x : S is ACK() endval
  value y : S is REQ() endval
  type pairS is (x=>S,y=>S) endty pe
endmod
module A1:VIEW1 is A endmod
module A2:VIEW2 is A endmod
```

As a result of constraint A1 by VIEW1, only the components specified in VIEW1 are accessible for users of A1. Hence, pair, ACK and REQ are not accessible via A1.

A.2.1.4 Composition of modules and interfaces

There are several means to compose modules and interfaces:

- Importation of interfaces into interfaces using "**imports** *int-exp*$_1, \cdots,$ *int-exp*$_n$" clause.
- Importation of modules into modules and generic modules using "**imports** *mod-exp*$_1, \cdots,$ *mod-exp*$_n$".
- Renaming of interfaces and modules, whose simpler form is:

  module *mod-id*$'$ **is**
  mod-id **renaming** (**types** S := S',...**opns** C := C',...)
  endmod

- Instantiation of generic modules, whose simpler form is:

  module *mod-id* **is** *gen-id*(*mod-id*$'$) **endmod**

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Interface importation Interfaces can import other interfaces. For example, the interface for pre-orders extends that for partial orders with one equation:

```
interface PreOrder is
  type T
  function infix <=(x:T,y:T):bool
  eqns forall x,y,z:T
    x<=x;
    (x<=y andalso y<=z)=>x<=y;
  endeqns
endint
interface PartialOrder imports PreOrder is
  eqns forall x,y:T
    (x<=y andalso y<=x)=>x=y;
  endeqns
endint
```

Note that the sort bool and the operations on this sort (as => and **andalso**) are built-in.

Module importation fact, the declared interface of the module is imported. For example the monomorphic list interface is:

Modules can import other modules: the coercion of the natural numbers type (assuming an appropriate Natural module) to a Monoid structure is made by:

```
module NatMonoid:Monoid imports Natural is
  type M is nat endtype
endmod
```

In the case of multiple importation, the definitions provided by each importer must be compatible. Thus, if two or more modules provide definitions for a common name, these definitions must be the same.

Instantiation Instantiation provides code re-use. For example, generic List can be instantiated several times:

```
module ListNat:[List renaming (types E := nat)] is
  GenericList(Natural renaming (types nat := E))
endmod
module ListBool:[List renaming (types E := bool)] is
  GenericList(Boolean renaming (types bool := E))
endmod
```

The result of the first instantiation is a module having as type E the type nat, and M being a list of naturals. The interface of this module is the List interface where the E type is replaced by nat. Similarly for the second instantiation. Importing the modules ListNat and ListBool will generate a name clash for M, +, and inj. To avoid this a renaming can be applied, as described in the next paragraph.

Since lists and sets are frequently used in specifications some rich syntax for them is suitable (see A.2.1.8). an instantiation. For example, the previous example can be wrote as follows:

Renaming Renaming is used to give an unique name to objects to avoid name clashes. The solution proposed is compatible with ACT ONE renaming of types. For example, obtaining a module of integer lists with type

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ListNat is as follows:

```
module ListNat is
  GenericList(Natural renaming (types nat := E))
    renaming (types ListNat := List)
endmod
```

A.2.1.5 Equational specifications

We allow equational specifications in interfaces. For example, in the Monoid interface:

```
eqns forall x,y,z:nat
  (x+0)=x;
  x=(x+0);
  (x+(y+z))=((x+y)+z);
endeqns
```

Many tools will treat these specifications just as (type checked) comments, so we should ensure that (as with Extended ML) the equations can be commented out without effecting the semantics of the module.

An extension of the solution proposed here is presented in [9]. It allows to specify equations, relations and properties.

We have to provide a formal semantics for when equations are valid (although this is obviously not computable, so we cannot expect automatic tools for checking validity).

A.2.1.6 Relationship with the external environment

External declarations are allowed for modules to allow interfacing to other specification or implementation languages. For example, one could give an external implementation of the Monoid module by declaring:

```
module ExtMonoid:Monoid is external endmod
```

Any object declared to be **external** has no formal dynamic semantics.

A.2.1.7 Compatibility with ACT ONE

The module system of E-LOTOS will include algebraic specifications in interfaces. For example, we can compare the LOTOS specification:

```
type Monoid is
  sorts
    M
  opns
    0: -> M
    _+_:M,M -> M
  eqns forall x,y,z:M
    ofsort M
      x+0=x;
      0+x=x;
      (x+y)+z=x+(y+z)
endtype
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

with the declaration from the example data language:

```
interface Monoid is
  type M
  value 0:M
  function infix +(x:M,y:M):M
  eqns forall x,y,z:M
    (x+0)=x;
    (0+x)=x;
    ((x+y)+z)=(x+(y+z));
  endeqns
endint
module Monoid:Monoid is external endmod
```

There is a strong resemblance between such specifications and ACT ONE data type declarations. There are, however, a number of differences:

1. ACT ONE allows overloading, as long as the sort of any expression can be determined statically,
2. module Monoid is specified to be *any* structure which satisfies the axioms, not just the initial one (in particular we may wish to introduce an initial declaration similar to the current external),

A.2.1.8 Base environment

The *base environment* is a collection of signatures (i.e. interfaces) and (possibly generic) modules which are predefined, and can be used in any E-LOTOS specification. They play the same role for E-LOTOS as the standard libraries do for LOTOS.

We propose the following *predefined types*: the type 'int', the type 'nat, the type 'float, the type 'rational, the type 'bool', the type 'char', the type 'string'. All these types can be declared in a Standard module which can be pervasive.

Also we propose a set of *constructed types* with their rich term syntax: enumerated types, record types, sets, and lists.

A.2.1.9 Static semantics

Modules In static semantics, a module is a context. Modules can be declared using *module declarations*, whose simplest form is

module *mod-id* **is** *dec* **endmod**

where *mod-id* is a module identifier and *dec* is a (base language) declaration enriched with value declarations. The context associated with the module identifier *mod-id* is generated by the declarations in *dec*. For example, the declaration

```
module OrdItem is
  type item is int endtype
  function leq(i:item,j:item):bool is i<=j endfunc
endmod
```

elaborates to the following environment

$$\text{OrdItem} \Rightarrow \{\text{item} \Rightarrow \text{int}, \text{leq} \Rightarrow [\,]\ (\text{i} \Rightarrow \text{item}, \text{j} \Rightarrow \text{item})\ [\,]\ \rightarrow \textbf{exit}(\text{bool})\}$$

In this example we suppose that integers are pervasive.

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Interfaces Intuitively, an *interface* is a type for modules. Whereas a module expression declares a module, an interface expression specifies a class of modules. An interface is represented semantically by a context bound to the interface identifier. For example, the Monoid interface declaration below

```
interface Monoid is
  type M
  value 0:M
  function infix +(M,M):M
  eqns forall x,y,z:M ->
    (0+x) = x;
    (x+0) = x;
    ((x+y)+z) = (x+(y+z));
  endeqns
endint
```

elaborates to the following environment

$$\text{Monoid} \Rightarrow \{\text{M} \Rightarrow \textbf{type}, 0 \Rightarrow \text{M}, + \Rightarrow [\,](\text{M},\text{M})\,[\,] \rightarrow \textbf{exit}(\text{M})\}$$

Over interfaces bindings and modules bindings we define a *match* relation. Intuitively, a module *matches* an interface if the former provides compatible definitions for each definition given in the second one.

Generic modules The static semantics object corresponding to a generic module is an application from a record of module bindings to a module binding. For example, the result of elaborating a generic module declaration

generic *gen-id* (*mod-id* : *int-id*) **is** *dec* **en dgen**

is

$$\textit{gen-id} \Rightarrow (\textit{mod-id} \Rightarrow \mathcal{C}) \rightarrow \mathcal{C}'$$

where $\mathcal{C}$ is the binding (type) of the interface identifier *int-id*, and $\mathcal{C}'$ is the result of elaborating *dec*.

The elaboration of a generic module instantiation consists to check first that the actual module parameters match the domain of the binding (here $(\textit{mod-id} \Rightarrow \mathcal{C})$), and then deriving the result from $\mathcal{C}'$, by instantiation of the formal parameters used in the generic module body.

static semantics are: between identifiers bound for modules, generic modules, and interfaces.

may describe the same identifier twice.

Closure Restrictions The semantics presented requires no restrictions on reference to non-local identifiers. For example, it allows an interface expression to refer to external interface identifiers and to external module identifiers; it also allows a generic module and a renaming morphism to refer to external identifiers of any kinds.

However, for implementation purposes, one may want to impose the following restrictions on reference of identifiers (ignoring references to identifiers bound to the base environment, which may occur anywhere):

- In any interface binding *int-id* **imports** ... **is** *int-exp*, the only non-local references in *int-exp* are to identifiers of the imported interfaces. For example, in the following monomorphic list declaration:

  ```
  interface List imports Monoid is
    type E
    function inj(x:E):M
  endint
  ```

  the only one non-local reference is to type M imported from Monoid interface.

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

- In any generic module binding *gen-id* (*mod-id* : *int-exp*)* : *int-exp'* **imports** ... **is** *mod-exp*, the only non-local references are to interface identifiers and to identifiers of the imported modules, except that *int-exp'* may refer to *mod-id* and its identifiers. For example, in the following generic module declaration:

```
generic GenericList(Eq:EqType):List is
  type M is nil()|cons(E,M) end type
  value 0 is nil() endval
  function infix +(s1:M,s2:M):M is
    case s1 is
        nil() → s2
      | cons(?x,?xs) → cons(x,xs+s2)
    endcase
  endfunc
  function inj(x:E):M is cons(x,nil()) endfunc
endgen
```

the only non-local declarations are to interface identifier EqType and to identifiers declared into EqType, E.

A.3 An E-LOTOS specification of the ODP trader

A.3.1 Introduction

In this annex, we present an E-LOTOS specification of the ODP trader computational viewpoint. The ODP trader is an object which enables software components to find appropriate services providers within an open and dynamically changing distributed system. The trader specification is a good example of how the language can be applied to specify real problems.

Our E-LOTOS specification follows the informal computational description of the trading function given in [1]. The specification is not complete in the sense that it does not include all the functionality given in the informal description. But, it describes an important portion of this functionality and mostly features not considered here can be added directly without difficulties. A previous version of this specification is presented in [21].

In what follows, Subclause A.3.2 gives an informal overview of the trading function. Subclause A.3.3 details some relevant parts of our E-LOTOS specifications. Finally, Annex A.3.3 contains the complete E-LOTOS specification of the trader.

A.3.2 An overview of the ODP Trader

In order to use services in an open distributed systems, users need to know which services are available and who are their providers. Since sites and applications are frequently changing in large distributed systems, it seems compulsory to have a mechanism which enables software components to find appropriate services providers. This mechanism, called Trading Function, is supplied in ODP [2, 3] by the *trader object*.

Following the philosophy of ODP, the trader is specified through several viewpoints. In the next subclauses, we gives an informal description of this viewpoints.

A.3.2.1 Enterprise viewpoint

From the enterprise viewpoint, a trader is an object that enables clients to find dynamically suitable servers in an ODP system. A trader can be viewed as an advertiser where objects can announce their capabilities and become aware of capabilities of other objects.

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

An announcement in a trader is called *service offer*. It describes the characteristics or properties satisfied by the service. In addition to service properties, a service offer also contains the interface where the service is available.

Advertising a service offer is called *export*. When a trader accepts an export request, it stores the exported service offer in a centralized or distributed database. This database is often termed *service offer space*.

On the other hand, an importer can require knowledge about adequate service providers. In this case, the trader accepts a request, called *import*, containing an expression of service requirements desired by the importer. The trader matches the importer's service request with its database of service offers and selects a list (probably empty) of appropriated service offers which satisfy the requirements made by the importer.

The list of matched services offers is returned to the importer which may then interact directly with any service described in the list. The Figure A.1 summarize the interactions of a trader and its clients.

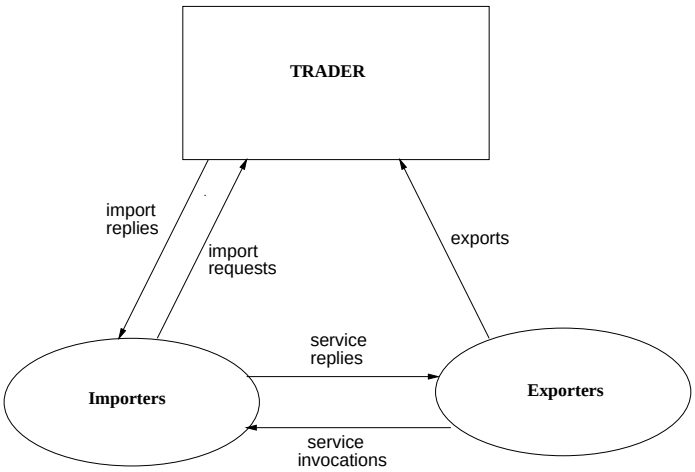

Figure A.1: Interactions of a trader and its clients

Export and import activities are governed by a *trading policy*, which comprises trader policies, importer policies and exporter policies. Where an activity involves interactions between objects, the resulting policy will be a compromise between the wishes of the interacting objects. Therefore, a trader's behaviour is limited by the policies established for these activities. In other words, trader policies determine and guide a trader's behaviour. For example, a trader policy can restrict resources used by an individual import request.

Several autonomous traders can be 'linked' in order to share their service offer spaces. Thus, a trader also can play the role of exporter or importer with respect to other trader(s). Such a group of autonomous traders is termed *interworking group*. A trader within an interworking group enlarges the service offer space for its users by including offers of other traders in the group. This enlargement of the service offer space is made indirectly when a trader propagates import requests to neighbor traders.

A.3.2.2 Information Viewpoint

The trader information viewpoint defines the information elements and the relationships between them which are manipulated by the ODP trading function. In the standard [1], this viewpoint is formally described using the formal specification notation Z. The specification includes basic concepts for information and, static invariant and dynamic schemata for the ODP trading function. In this annex we do not consider this viewpoint.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A.3.2.3 Computational Viewpoint

The trading function computational viewpoint describes an object template for a trader. This object has interfaces for service and management operations. Service operations are related to import and export activities whereas management operations are provided to add, delete or modify links to other traders.

In the standard [1], this viewpoint description comprehends:

- signature templates for the service interface and management interface (defined in CORBA IDL),
- types used in the operations parameters (defined in CORBA IDL), and
- informal descriptions of the trader's behaviour.

The trader's behaviour is given by the behaviour of every service operation and management operation, plus a set of constraints on interleaving of actions performed by these operations. In the next subclause, we give in E-LOTOS a formal description of the trader's behaviour. This formal specification could be viewed as a complement of the informal one given in [1].

A.3.3 E-LOTOS Specification of the trader

This subsection outlines our E-LOTOS specification of the ODP trader computational viewpoint. Here, we present only the most relevant parts of the specification. In particular, we pay attention in the definition of the trader's behaviour. The Subclause A.3.4 contains the complete specification including type and functions definitions which are used to define the behaviour of operations.

A trader is modeled by a process which can interact with the environment through two ports. In one port, the trader receives operation invocations from clients and management objects. In the other port, it returns the results to the respective invokers. Following the ODP terminology, we will call *termination* to the action of return a result.

The following E-LOTOS code shows a scenario where a trader communicates with other client objects. The trader receives invocations on the gate `inv` and send the respective terminations on the gate `ter`.

```
par inv#2, ter#2, bind#2
   [inv,ter] -> Trader[inv,ter](...)
 ||[inv,ter] -> importer[inv,ter]
 ||[bind] -> importerExporter[bind]
 ||[inv,ter,bind] -> bindingObject[inv,ter,bind]
endpar
```

This behavior expression represent a scenario where:

- The `Trader` and `importer` processes directly interchange data using `inv` and `ter`.
- The `Trader` and `importerExporter` processes are indirectly communicated through the `bindigObject` process which redirerects invocations and terminations of one to other process.

A.3.3.1 Type declarations

In the specification, the greatest part of the type declarations comprise types for the operation parameters. The standard [1] defines these types in CORBA IDL and translate them to E-LOTOS is very easy. Also, we translate exception declarations given in CORBA IDL to type declarations in the E-LOTOS notation. In order to made more concise the specification, we have wiped the tailing string "Type" from all type names in [1].

Ports are typed in E-LOTOS, therefore we need specify types for data exchanged (offered) by E-LOTOS processes. In particular, we need specify which are the types involved in trader interactions. In the rest of this subsection we describe these types.

The signature for operation invocations and operation terminations is defined in the following types:

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
type InvocationSig is
   ExportOfferInv ( Servicedescription, (*serviceDescription*)
                    PropertyValueList, (*servicePropValues*)
                    PropertyValueList, (*offerPropValues*)
                    InterfaceId (*serviceInterfaceId*)
                  )
 | ImportOfferInv ( ServiceDescription, (*serviceDescription*)
                    Rule, (*matchingCriteria*)
                    Rule, (*preferenceCriteria*)
                    OrderRequirementList, (*orderingRequirementList*)
                    PropertiesOfInterest, (*servicePropertiesOfInterest*)
                    PropertiesOfInterest (*offerPropertiesOfInterest*)
                  )
 | AddLinkInv ( Name, (*newLinkName*)
                PropertyValueList, (*linkPropValues*)
                InterfaceId  (*targetInterfaceId*)
              )
 | ... (* other invocation signatures *)
endtype

type TerminationSig is
   ExportOfferTer (ServiceOfferId)
 | ImportOfferTer (ServiceOfferDetailList)
 | AddLinkTer (LinkId)
 | ... (* other termination signatures *)
endtype
```

Notice that an invocation signature states the input parameters of the respective operation whereas a termination signature states the output parameters.

Every trader or client has an interface identifier. These identifiers are used in invocations and terminations to distinguish one of others. Therefore, an invocation (termination) contains the identifier of the invoked trader and the identifier of the originator client. The type of values interchanged in invocations and terminations is defined in E-LOTOS by the following declaration:

```
type Invocation is
  ( InterfaceIdentifier, (*interfaceId*)
    InterfaceIdentifier, (*originatorId*)
    InvocationSig        (*invocation*)
  )
endtype

type Termination is
  ( InterfaceIdentifier, (*interfaceId*)
    InterfaceIdentifier, (*originatorId*)
    TerminationSig       (*termination*)
  )
endtype
```

A.3.3.2 Computational Behaviour of the trader

At the most abstract level, the trader object is a parallel composition of three processes. The `ServiceInterface` and `ManagementInterface` processes provide functionality for service operations and management operations, respectively. The process `StateProc` represents the trader state.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
process Trader [inv: Invocation, ter: Termination]
               ( interfaceId: InterfaceIdentifier,
                 properties : Properties,
                 offerSpace : ServiceOfferSpace,
                 linkSpace :  linkOfferSpace  ) is
  hide sa:stateAccess in
      ( ServiceInterface [inv, ter, sa] (interfaceId)
       |||
        ManagementInterface [inv, ter, sa] (interfaceId)
      )
    |[sa]|
      StateProc [sa] ( properties, offerSpace, linkSpace )
  endhide
endproc (*Trader *)
```

Service operations only write in the trader service offer space while management operations only write in the trader link space and/or change trader properties. Therefore, beside an operations can eventually access both the service offer space and the link space of the trader, service operations and management operations can be performed in parallel without destroy the consistency of the trader state.

The process `StateProc` encapsulates three elements which conform the trader state: a set of trader properties, the service offer space and the link space. Service and management operations use the trader state through the and the link spacesa port which is hided within the trader.

In order to illustrate how the operation's behaviour are defined we show the definition of the `ServiceInterface` process. In this process, all service operations are offered to clients in parallel. However, the availability of operations is constrained in such way that accessing operations (as import offer) and modifying operations (as export offer) can not be overlapped in time. Availability of operations is defined in a constraint oriented style by composing the processes representing operations with the `OrderingConstraints` process.

```
process ServiceInterface [ inv: Invocation,
                           ter: Termination,
                           sa: stateAccess ]
                         ( interfaceId: InterfaceIdentifier ) is
    ( ImportOffer [inv, ter, sa] (interfaceId)
     |||
      exportOffer [inv, ter, sa] (interfaceId)
     |||
      .... (* other service operations *)
    )
  |[inv, ter]|
    OrderingConstraints [inv, ter]
endproc (*ServiceInterface *)
```

The most significant and complex service operation is the importation of an offer. The behaviour of the operation is given below by the `ImportOffer` process definition.

Informally, for each invocation the `ImportOffer` process performs the following activities:

- receive the invocation,
- access the state to get the offer database, trader links and trader properties,
- match the offers against arguments and propagate the import invocation to interoperate with 'neighbor' traders,

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- When some trader finds *good matches*, they are submitted in the termination.

```
process ImportOffer [ inv: Invocation, ter: Termination,
                      sa: stateAccess ]
                    (id: InterfaceId) is
    inv ( !id, ?origId,
          ImportOfferInv ( ?sd, ?matchingC, ?preferenceC,
                           ?ordering, ?spOfInt, ?opOfInt ) ) ;
    sa Read( ?traderProp, ?offers, ?links ) ;
    trap
      exception X ( offers: ServiceOfferDetailList ) is
         ter ( !id, !origId, ImportOfferTer(offers) )
      endexn
      exit is
         ter ( !id, !origId, ImportOfferTer(emptyOffers) )
      endexit
    in
       ... (* match local offers *)
     |||
       ... (* interoperate with other traders *)
    endtrap
 |||
  ImportOffer [ inv, ter, sa ] ( id )
endproc
```

The above definition is a good example to illustrate the use of the trap constructor. We use this constructor to express that, when several trader are interoperating in an import operation, the matched offers returned to the original importer are those of the 'first' trader which had successful in the matching activity.

A.3.4 The complete specification

A.3.4.1 Type Declarations

```
type List is
   Nil
 | Cons (any, List)
endtype

type Name renames
  String
endtype

type InterfaceId renames
  Integer
endtype

type InterfaceIdList renames
  List (* of InterfaceId *)
endtype

type GraphEdgeSpec is
 ( LinkId, (*linkId*)
   Name,   (*edgeName*)
```

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
   PropertyValueList, (*linkPropertyValues*)
   InterfaceId (*toNodeLocation*)
 )
endtype

type OfferPartitionSubgraph renames
  List (* of GraphEdgeSpec *)
endtype

type PropertyName is
   ServicePropertyName (Name),
 | OfferPropertyName (Name),
 | LinkPropertyName (Name),
 | TraderPropertyName (Name),
endtype

type Order is
   ASCENDING
 | DESCENDING
endtype

type OrderRequirementList is
 ( OrderType,   (*orderDirection*)
   PropertyName (*propertyName*)
 )
endtype

type LiteralPropertyValue is
 | BoolVal (Bool)
 | IntVal (Integer)
 | NameVal (Name)
 | RefVal (InterfaceId)
endtype

type LiteralOrProperty is
   Literal (LiteralPropertyValue)
 | Property (PropertyName)
endtype

type Rule is
   TRUE
 | Exist (propertyName)
 | LiteralOrProperty EQ LiteralOrProperty,
 | LiteralOrProperty LT LiteralOrProperty,
 | LiteralOrProperty LE LiteralOrProperty,
 | LiteralOrProperty GT LiteralOrProperty,
 | LiteralOrProperty GE LiteralOrProperty,
 | LiteralOrProperty NE LiteralOrProperty,
 | NOT (Rule)
 | Rule AND Rule
 | Rule OR Rule
endtype
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
type RuleList renames
  List (* of Rule *)
endtype

type PropertyValue is
 ( PropertyName,        (*name*)
   LiteralPropertyValue (*value*)
 )
endtype

type PropertyNameList renames
  List (* of PropertyName *)
endtype

type PropertiesOfInterest is
   ALL
 | Interesting ( PropertyNameList )
endtype

type PropertyValueList renames
  List (* of PropertyValue *)
endtype

type ServiceDescription renames
  integer
endtype

type QualificationCode is
   Full_succes
 | Selection_preference_not_obtained
 | Property_of_interest_not_avalilable
 | Offer_property_of_interest_not_avalilable
endtype

type Qualifier renames
  List (* of QualificationCode *)
endtype

type ServiceOfferDetail is
 ( InterfaceId,       (*interfaceId*)
   PropertyValueList, (*servicePropertyValues*)
   PropertyValueList, (*serviceOfferPropertyValues*)
   Qualifier          (*offerQualificationList*)
 )
endtype

type ServiceOfferDetailList renames
  List (* of ServiceOfferDetail *)
endtype

type ServiceOfferId renames
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
  integer
endtype

type ServiceOfferIdList renames
  List (* of ServiceOfferId *)
endtype

type ServiceOfferDescription is
 ( ServiceOfferId,     (*serviceOfferId*)
   ServiceDescription, (*serviceDescription*)
   InterfaceId,        (*InterfaceId*)
   PropertyValueList,  (*servicePropertyValues*)
   PropertyValueList   (*serviceOfferPropertyValues*)
 )
endtype

type ServiceOfferSpace renames
  List (* of ServiceOfferDescription *)
endtype

type LinkId renames
  Integer
endtype

type LinkIdList renames
  List (* of LinkId *)
endtype

type InvocationSig is
   ExportOfferInv ( ServiceDescription, (*serviceDescription*)
                    PropertyValueList,  (*servicePropValues*),
                    PropertyValueList,  (*offerPropValues*)
                    InterfaceId         (*serviceInterfaceId*)
                  )
 | WithdrawOfferInv ( ServiceOfferId )
 | ModifyOfferInv ( ServiceOfferId,    (*offerId*)
                    PropertyValueList, (*servicePropValues*)
                    PropertyValueList  (*offerPropValues*)
                  )
 | ImportOfferInv ( ServiceDescription,   (*serviceDescription*)
                    Rule,                 (*matchingCriteria*)
                    Rule,                 (*preferenceCriteria*)
                    OrderRequirementList, (*orderingRequirementList*)
                    PropertiesOfInterest, (*servicePropertiesOfInterest*)
                    PropertiesOfInterest  (*offerPropertiesOfInterest*)
                  )
 | QueryOfferPartitionSubgraphInv
 | DescribeOfferInv ( ServiceOfferIdList,   (*serviceOfferIdList*)
                      PropertiesOfInterest, (*servicePropertiesOfInterest*)
                      PropertiesOfInterest  (*offerPropertiesOfInterest*)
                    )
 | AddLinkInv ( Name, (*newLinkName*)
```

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
                PropertyValueList, (*linkPropValues*)
                InterfaceId (*targetInterfaceId*)
              )
 | RemoveLinkInv ( LinkId )
endtype

type TerminationSig is
   ExportOfferTer ( ServiceOfferId)
 | WithdrawOfferTer
 | ModifyOfferTer
 | ImportOfferTer ( ServiceOfferDetailList )
 | QueryOfferPartitionSubgraphTer ( OfferPartitionSubgraph)
 | DescribeOfferTer ( ServiceOfferDetailList )
 | AddLinkTer ( LinkId )
 | RemoveLinkTer
 | Error ( TraderErrorExceptions )
endtype


type Invocation is
  ( InterfaceId,  (*interfaceId*)
    InterfaceId,  (*originatorId*)
    InvocationSig (*invocation*)
  )
entype

type Termination is
  ( InterfaceId,   (*interfaceId*)
    InterfaceId,   (*originatorId*)
    TerminationSig (*termination*)
  )
endtype

type stateAccess is
   Read ( properties => PropertyValueList,
          offerSpace => ServiceOfferSpace,
          linkSpace =>  OfferPartitionSubgraph )
 | ReadProperties ( properties => PropertyValueList )
 | ReadOffers ( offerSpace => ServiceOfferSpace )
 | ReadLinks ( linkSpace =>  OfferPartitionSubgraph )
 | Write ( properties => PropertyValueList,
           offerSpace => ServiceOfferSpace,
           linkSpace =>  OfferPartitionSubgraph )
 | WriteProperties ( properties => PropertyValueList )
 | WriteOffers ( offerSpace => ServiceOfferSpace )
 | WriteLinks ( linkSpace =>  OfferPartitionSubgraph )
 | NewServiceOfferId ( serviceOfferId => ServiceOfferId )
 | NewLinkId ( linkId => LinkId )
endtype

type TraderErrorExceptions is
   SystemSpecificException (String)
```

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
 | UndefinedProperty (PropertyName)
 | BadOfferIdentity (ServiceOfferId)
 | InvalidArgumentSyntax
endtype
```

A.3.4.2 Function Declarations

```
(* ----------  List general functions -------------*)

function append (xs, ys: List) : List is
  case xs is
    Nil            -> ys
    Cons(?x, xxs?) -> Cons(x,append(xxs,ys))
  endcase
endfunc

function IsIn (x: any, xs: List) : Boolean
  case xs
    Nil           -> false
    Cons(?y, ys?) -> x = y orelse IsIn(x,ys)
  endcase
endfunc

function delete (x: any, xs: List) : Boolean
  case xs is
    Nil            -> Nil
    Cons (!x, ?ys) -> ys
    Cons (?y, ?ys) -> Cons (y, delete(x,ys))
  endcase
endfunc



(* ---------- LiteralOrProperty functions -------------*)

function lt (lp1, lp2: LiteralOrProperty) : Boolean
            raises [err: (TraderErrorExceptions)] is
 case (lp1,lp2)
   (IntVal(?i), Intval(?j))   -> i < j
   (NameVal(?n), Nameval(?m)) -> n < m
   otherwise                  -> raise err (InvalidArgumentSyntax)
 endcase
endfunc

function le (lp1, lp2: LiteralOrProperty) : Boolean
            raises [err: (TraderErrorExceptions)] is
  lt(lp1,lp2)[err] orelse (lp1 = lp2)
endfunc

function gt (lp1, lp2: LiteralOrProperty) : Boolean
            raises [err: (TraderErrorExceptions)] is
  not le(lp1,lp2)[err]
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
endfunc

function ge (lp1, lp2: LiteralOrProperty) : Boolean
            raises [err: (TraderErrorExceptions)] is
  not lt(lp1,lp2)[err]
endfunc

function ne (lp1, lp2: LiteralOrProperty) : Boolean is
  not (lp1 = lp2)
endfunc

function valueOf ( lp: LiteralOrProperty, props: PropertyValueList )
                 : LiteralPropertyValue
                 raises [err: (TraderErrorExceptions)]
  case lp is
    Property (?pName) -> propValueOf (pName, props) [err]
    Literal (?v) -> v
  endcase
endfunc




(* ---------- OfferPartitionSubgraph functions -------------*)

function neighbours (l: OfferPartitionSubgraph) : InterfaceIdList is
 case l
   Nil                   -> Nil
   Cons( (?lid, ?nm, ?lProps, ?id), ?tl) -> Cons(id,tl)
 endcase
endfunc


(* ----------  PropertyValueList functions -------------*)

function existProperty ( pName: PropertyName,
                         props: PropertyValueList ) : Boolean
  case props is
    Nil ->
      false
    Cons ((name => !pName, etc), any:PropertyValueList) ->
      true
    Cons (any:PropertyValue, ?tl) ->
      existProperty (nm, tl)
  endcase
endfunc

function propValueOf  ( pName: PropertyName, props: PropertyValueList )
                 : LiteralPropertyValue
                 raises [err: (TraderErrorExceptions)]
   case props is
     Nil ->
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
      raise err ( UndefinedProperty(pName) )
    Cons ((!pName, ?v), any:PropertyValueList) ->
      v
    Cons (any:PropertyValue, ?tl) ->
      propValueOf (pName, tl) [err]
  endcaspe
endfunc

function existAllProperties (pns: PropertyNameList, pVals: PropertyValueList)
                            : Boolean is
 case pns is
   Nil ->
     true
   Cons(?pn, ?xs ) ->
     existProperty (pn, pVals) andalso existAllProperties (xs, pVals)
 endcase
endfunc

function filterProps ( pns: PropertyNameList, pVals: PropertyValueList )
                     : PropertyValueList
 case pVals is
   Nil ->
     Nil
   Cons( (?pn,?v), ?pvs ) ->
     if IsIn (pn, pns)
     then
       Cons( (pn,v), FilterProps(pns,pvs) )
     else
       filterProps(pns,pvs)
     endif
 endcase
endfunc

function interestingProps ( intr: PropertiesOfInterest,
                            pVals: PropertyValueList )
                          : PropertyValueList
 case intr is
   ALL                -> pVals
   Interesting (?pns) -> filterProps(pns, pVals)
 endcase
endfunc



(* ----------  QualificationCode functions -------------*)
function insQualification (q: QualificationCode, qs: Qualifier)
                          : Qualifier is
 case qs is
   Nil -> Cons (q, Nil)
   ?rs -> if q = Full_succes
          then rs
          else Cons (q,rs)
```

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
          endif
 endcase
endfunc


(* ---------- Matching criteria function -------------*)

function conforms ( criteria: Rule,
                    sPropVal, oPropVal, traderProp: PropertyValueList )
                  : Boolean
                  raises [err: (TraderErrorExceptions)] is
 var
   props : PropertyValueList := append ( sPropVal, append(oPropVal,traderProp) )
 in
   case criteria is
     TRUE -> True
     Exist ( ?propertyName ) ->
       case propertyName is
         ?nm, linkPropertyName (any:Name) ->
           raise err (SystemSpecificException ("bad use of link property
                                               name in rule"))
         ?nm ->
            existProperty (nm, sPropVal)
       endcase
     ?lp1 EQ ?lp2 ->
       valueOf(lp1, props)[err] = valueOf (lp2, props)[err]
     ?lp1 LT ?lp2 ->
       lt (valueOf(lp1, props)[err], valueOf(lp2, props)[err]) [err]
     ?lp1 LE ?lp2 ->
       le (valueOf(lp1, props)[err], valueOf(lp2, props)[err]) [err]
     ?lp1 GT ?lp2 ->
       gt (valueOf(lp1, props)[err], valueOf(lp2, props)[err]) [err]
     ?lp1 GE ?lp2 ->
       ge (valueOf(lp1, props)[err], valueOf(lp2, props)[err]) [err]
     ?lp1 NE ?lp2 ->
       not (valueOf(lp1, props)[err] = valueOf (lp2, props)[err])
     NOT ?r ->
       not conforms (r, sPropVal, oPropVal, traderProp) [err]
     ?r1 AND ?r2 ->
       conforms (r1, sPropVal, oPropVal, traderProp) [err] andalso
       conforms (r2, sPropVal, oPropVal, traderProp) [err]
     ?r1 OR ?r2 ->
       conforms (r1, sPropVal, oPropVal, traderProp) [err] orelse
       conforms (r2, sPropVal, oPropVal, traderProp) [err]
   endcase
 endvar
endfunc


(* ----------  ServiceOfferSpace functions -------------*)

function matchOffers ( sd: ServiceDescription,
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
                      matchingC, preferenceC: Rule,
                      traderProp: PropertyValueList,
                      spOfInt, opOfInt: PropertiesOfInterest,
                      offers: ServiceOfferSpace )
                    : ServiceOfferDetailList
                    raises [err: (TraderErrorExceptions)] is
 case offers is
   Nil -> Nil
   Cons ( (?oId, !sd, ?interfId, ?sPropVal, ?oPropVal), ?tl ) ->
      var
        tlMtchs : ServiceOfferDetailList := matchOffers(sd, matchingC,
                                                        preferenceC, traderProp,
                                                        spOfInt, opOfInt, tl)[err]
      in
        if conforms ( matchingC, sPropVal, oPropVal, traderProp ) [err]
        then
          local
          var ss,os : PropertyValueList,
              sq, oq, pq : QualificationCode,
              qq: Qualifier
          in
            ?ss := InterestingProps (spOfInt, sPropVal);
            ?sq := if existAllProperties (spOfInt, sPropVal)
                   then Full_succes
                   else Property_of_interest_not_avalilable
                   endif;
            ?os := InterestingProps (opOfInt, oPropVal);
            ?oq := if existAllProperties (opOfInt, oPropVal)
                   then Full_succes
                   else Offer_property_of_interest_not_avalilable
                   endif;
            ?pq := if conforms (preferenceC, sPropVal, oPropVal, traderProp)
                               [err]
                   then Full_succes
                   else Selection_preference_not_obtained
                   endif;
            ?qq :=
              insQualification(sq,insQualification(oq,insQualification(qq,Nil)));
            Cons((interfId, ss, os, qq),tlMtchs)
          endvar
        else tlMtchs
        endif
      endvar
   Cons (any:ServiceOfferDescription, ?tl) ->
      matchOffers (sd,matchingC,preferenceC,traderProp,spOfInt,opOfInt,tl)
                  [err]
 endcase
endfunc

function getOfferDetails ( oId: ServiceOfferId,
                           offers: ServiceOfferSpace,
                           out sDes: ServiceDescription
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
                          out interfId: InterfaceId,
                          out sProps, oProps: PropertiesOfInterest )
                        : ServiceOfferDetailList
                        raises [err: (TraderErrorExceptions)] is
 case offers is
   Nil ->
     raise err ( BadOfferIdentity(oId) )
   Cons ( (!oId, ?sd, ?id, ?sPropVal, ?oPropVal) , ?tl ) ->
     ?sDes := sd ;
     ?interfId := id;
     ?sProps := sPropVal;
     ?oProps := oPropVal
   Cons (any:ServiceOfferDescription, ?tl) ->
     getOfferDetails (oId, tl, ?sDes, ?interfId, ?sProps, ?oProps) [err]
 endcase
endfunc

function existOffer (oId: ServiceOfferId, offers: ServiceOfferSpace)
                    : Boolean is
  case offers is
    Nil        ->
      false
    Cons( (?id, ?sd, ?id, ?sProps, ?oProps), ?tl ) ->
      ( id = Oid ) orelse existOffer (oId, tl)
  endcase
endfunc

function delOffer (oId: ServiceOfferId, offers: ServiceOfferSpace)
                  : ServiceOfferSpace
  case offers is
    Nil -> Nil
    Cons( (!oId, ?sd, ?id, ?sProps, ?oProps), ?tl ) ->
      tl
    Cons( ?od, ?ods ) ->
      Cons(od,delOffer(oId,tl))
  endcase
endfunc

function deleteOffer (oId: ServiceOfferId, offers: ServiceOfferSpace)
                     : ServiceOfferSpace
                     raises [err: (TraderErrorExceptions)] is
  if existOffer(oId, offers)
  then raise err ( BadOfferIdentity(oId) )
  else delOffer(oId, offers)
  endif
endfunc

function describe ( offerIds: ServiceOfferIdList,
                    spOfInt, opOfInt: PropertiesOfInterest,
                    offers: ServiceOfferSpace ) : ServiceOfferDetailList
                  raises [err: (TraderErrorExceptions)] is
 case offerIds is
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
   Nil ->
     Nil
   Cons (?oid, ?oids) ->
     var interfId: InterfaceId,
         sProps, oProps : PropertyValuleList
         ss,os : PropertyValueList,
         sq, oq : QualificationCode,
         qq: Qualifier
     in
       getOfferDetails ( oid, offers, any:ServiceDescription,
                         ?interfId, ?sProps, ?oProps ) [err];
       ?ss := InterestingProps (spOfInt, sProps);
       ?sq := if existAllProperties (spOfInt, sProps)
              then Full_succes
              else Property_of_interest_not_avalilable
              endif;
       ?os := InterestingProps (opOfInt, oProps);
       ?oq := if existAllProperties (opOfInt, oProps)
              then Full_succes
              else Offer_property_of_interest_not_avalilable
              endif;
       ?qq := insQualification(oq,insQualification(sq,Nil));
       Cons( (interfId, ss, os, qq),
             describe (offerIds, spOfInt, opOfInt, sds) [err] )
     endvar
 endcase
endfunc


(* ----------  ServiceOfferDetailList ordering functions -------------*)

function IsOrdered ( ordering: OrderRequirementList,
                     d, d1: ServiceOfferDetail ) : Boolean
                   raises [err: (TraderErrorExceptions)] is
 var
   vs: PropertyValueList := append(d.servicePropertyValues,
           d.serviceOfferPropertyValues),
   v1s: PropertyValueList := append(d1.servicePropertyValues,
            d1.serviceOfferPropertyValues)
 in
   case ordering is
     Nil -> true
     Cons ((ASCENDING, ?pName), ?tl) ->
       var l, l1: LiteralPropertyValue
       in
         ?l := propValueOf (pname, vs) [err];
         ?l1 := propValueOf (pname, v1s) [err];
         l < l1 orelse ( l = l1 andalso IsOrdered (tl, d, d1) )
       endvar
     Cons ((DESCENDING, ?pName), ?tl) ->
       var l, l1: LiteralPropertyValue
       in
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
        ?l := propValueOf (pname, vs) [err];
        ?l1 := propValueOf (pname, v1s) [err];
        l > l1 orelse ( l = l1 andalso IsOrdered (tl, d, d1) )
      endvar
   endcase
 endloc
endfunc

function partition ( in ordering: OrderRequirementList,
                     in d: ServiceOfferDetail,
                     in ds: ServiceOfferDetailList
                     out less, gtr: ServiceOfferDetailList)
                   raises [err: (TraderErrorExceptions)] is
  loop ( out less, gtr: ServiceOfferDetailList,
         in l,g, rest: ServiceOfferDetailList   )
    ?l := Nil; ?g := Nil; ?rest := ds;
    case rest is
      Nil -> break (less=>l, gtr=>g)
      Cons (?d1, ?d1s) [ IsOrdered (ordering, d, d1) [err] ] ->
        ?l := Cons(d1,l); ?g := g;  ?rest:= d1s
      Cons (?d1, ?d1s) [ not IsOrdered (ordering, d, d1) [err] ] ->
        ?l := l; ?g := Cons(d1,g); ?rest := d1s
  endloop
endfunc

function sort ( ordering: OrderRequirementList,
                offers: ServiceOfferDetailList ) : ServiceOfferDetailList
              raises [err: (TraderErrorExceptions)] is
  case offers is
    Nil ->
      Nil
    Cons(?d, ds) ->
      var less, gtr: ServiceOfferDetailList
      in
        partition (ordering, d, ds, ?less, ?gtr) [err];
        append( sort(ordering,less),  Cons(d,sort(ordering, gtr)) )
      endvar
  endcase
endfunc
```

A.3.4.3 Process Declaration

```
(* ---------- Trader Process -------------*)

process Trader [inv: Invocation, ter: Termination]
               ( interfaceId: InterfaceId,
                 properties : PropertyValueList,
                 offerSpace : ServiceOfferSpace,
                 linkSpace :  OfferPartitionSubgraph  ) is
  hide sa:stateAccess in
      ( ServiceInterface [inv, ter, sa] (interfaceId)
       |||
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
        ManagementInterface [inv, ter, sa] (interfaceId)
      )
    |[sa]|
      StateProc [sa] ( properties, offerSpace, linkSpace )
  endhide
endproc (*Trader *)

process ServiceInterface [ inv: Invocation,
                           ter: Termination,
                           sa: stateAccess ]
                         ( interfaceId: InterfaceId ) is
    ( ImportOffer [inv, ter, sa] (interfaceId)
     |||
      QueryOfferPartitionSubgraph [inv, ter, sa] (interfaceId)
     |||
      describeOffer [inv, ter, sa] (interfaceId)
     |||
      exportOffer [inv, ter, sa] (interfaceId)
     |||
      withdrawOffer [inv, ter, sa] (interfaceId)
     |||
      modifyOffer [inv, ter, sa] (interfaceId)
    )
  |[inv, ter]|
    ServiceOperationsOrderingConstraints [inv, ter]
endproc (*ServiceInterface *)

process ServiceOperationsOrderingConstraints
            [ inv: Invocation, ter: Termination ] is
 loop
    var id, origId: InterfaceId
    in
       inv ( ?id, ?origId, ExportOfferInv (etc) ) ;
       ter ( !id, !origId, ExportOfferTer (etc) )
      []
       inv ( ?id, ?origId, WithdrawOfferInv (etc) ) ;
       ter ( !id, !origId, WithdrawOfferTer (etc) )
      []
       inv ( ?id, ?origId, ModifyOfferInv (etc) ) ;
       ter ( !id, !origId, ModifyOfferTer (etc) )
    endvar
  []
    hide nr:() in
      importOfferConstraint [inv, ter, nr]
     |||
      queryOfferPartitionSubgraphConstraint [inv, ter, nr]
     |||
      describeOfferConstraint [inv, ter, nr]
    endhide
 endloop
endproc (*ServiceOperationsOrderingConstraints*)
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
process importOfferConstraint [inv: Invocation,
                               ter: Termination, nr: ()] : () is
    var id, origId: InterfaceId
    in
      inv ( ?id, ?origId, ImportOfferInv (etc) ) ;
      ter ( !id, !origId, ImportOfferTer (etc) )
    endvar
   []
    nr
 |||
   importOfferConstraint [inv, ter, nr]
endproc (*importOfferConstraint*)

process queryOfferPartitionSubgraphConstraint
               [inv: Invocation, ter: Termination, nr: ()] : () is
    var id, origId: InterfaceId
    in
      inv ( ?id, ?origId, QueryOfferPartitionSubgraphInv (etc) ) ;
      ter ( !id, !origId, QueryOfferPartitionSubgraphTer (etc) )
    endvar
   []
    nr
 |||
   queryOfferPartitionSubgraphConstraint [inv, ter, nr]
endproc (*queryOfferPartitionSubgraphConstraint*)

process describeOfferConstraint [inv: Invocation,
                                 ter: Termination, nr: ()] : () is
    var id, origId: InterfaceId
    in
      inv ( ?id, ?origId, DescribeOfferInv (etc) ) ;
      ter ( !id, !origId, DescribeOfferTer (etc) )
    endvar
   []
    nr
 |||
   describeOfferConstraint [inv, ter, nr]
endproc (*describeOfferConstraint *)

process ImportOffer [ inv: Invocation, ter: Termination, sa: stateAccess ]
                    (id: InterfaceId) is
    inv ( !id, ?origId,
          ImportOfferInv ( ?sd, ?matchingC, ?preferenceC,
                           ?ordering, ?spOfInt, ?opOfInt ) ) ;
    sa Read(?traderProp, ?offers, ?links) ;
    trap
      exception X ( offers: ServiceOfferDetailList ) is
         ter ( !id, !origId, !ImportOfferTer(offers) )
      endexn
      exception err ( e: TraderErrorExceptions ) is
         ter ( !id, !origId, !Error(e) )
      endexn
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
      exit is
         ter ( !id, !origId, !ImportOfferTer(emptyOffers) )
      endexn
    in
      var matches: ServiceOfferDetailList :=
           sort ( ordering,
                  matchOffers(sd,matchingC, preferenceC, traderProp,
                              spOfInt, opOfInt, offers)[err] )
      in
        if matches <> emptyOffers
        then raise X (matches)
        else exit
        endif
      endvar
     |||
      var neighs: InterfaceIdList := neighbours (links)
      in
        PropagateImport [ inv, ter ]
                        ( id, sd, matchingC, preferenceC, ordering,
                          spInterest, opInterest, neighs )
                        [ X ]
      endvar
    endtrap
 |||
  ImportOffer [ inv, ter, sa ] ( id )
endproc (*ImportOffer*)


process PropagateImport [ inv: Invocation, ter: Termination ]
                        ( id: InterfaceId,
                          sd: ServiceDescription,
                          matchingC, preferenceC: Rule,
                          ordering: OrderRequirementList,
                          spInterest: PropertiesOfInterest,
                          opInterest: PropertiesOfInterest,
                          neighs: InterfaceIdList )
                        raises [ X: (ServiceOfferDetailList)] : () is
 if neighs = Nil
 then exit
 else
   var x: InterfaceId
   in
     ?x := any InterfaceId [isIn(x,neighs)] ;
     ( var matches: ServiceOfferDetailList
       in
         inv ( !x, !id, !ImportOfferInv ( sd, matchingC, preferenceC,
                                          ordering, spInterest, opInterest ) )
         ter ( !x, !id, ImportOfferTer ( ?matches ) ) ;
         if matches <> emptyOffers
         then raise X (matches) endif
       endvar
      |||
```

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
       PropagateImport [ inv, ter ]
                       ( id, sd, matchingC, preferenceC, ordering,
                         spInterest, opInterest, delete (x, neighs) )
                       [ X ]
     )
   endvar
endproc (*PropagateImport*)

process QueryOfferPartitionSubgraph
           [ inv: Invocation, ter: Termination, sa: stateAccess ]
           (id: InterfaceId) is
   inv (!id, ?origId, QueryOfferPartitionSubgraphInv ) ;
   sa ReadLinks(?subg) ;
   ter (!id, !origId, !QueryOfferPartitionSubgraphTer(subg))
endproc

process describeOffer [ inv: Invocation, ter: Termination, sa: stateAccess ]
                      (id: InterfaceId) is
   inv ( !id, ?origId,
         DescribeOfferInv (?offerIds, ?spInterest, ?opInterest) ;
   sa ReadOffers(?offers) ;
   ter ( !id, !origId,
         DescribeOfferTer (!describe(offerIds, spOfInt, opOfInt, offers)) )
endproc

process exportOffer [ inv: Invocation, ter: Termination, sa: stateAccess ]
                    (id: InterfaceId) is
   inv ( !id, ?origId, ExportOfferInv (?sd, ?sProps, ?oProps, ?interfId) ) ;
   sa NewServiceOfferId(?oId) ;
   sa ReadOffers (?offers) ;
   sa !WriteOffers(Cons((oId, sd, interfId, sProps, oProps), offers)) ;
   ter ( !id, !origId, !ExportOfferTer (oId) )
endproc

process withdrawOffer [ inv: Invocation, ter: Termination, sa: stateAccess ]
                      (id: InterfaceId) is
   inv ( !id, ?origId, WithdrawOfferInv (?oId) ) ;
   sa ReadOffers(?offers) ;
   sa !WriteOffers(deleteOffer(oId,offers)) ;
   ter ( !id, !origId, WithdrawOfferTer )
endproc

process modifyOffer [ inv: Invocation, ter: Termination, sa: stateAccess ]
                      (id: InterfaceId) is
   inv ( !id, ?origId, ModifyOfferInv (?oId,?sProps,?oProps) )
   sa ReadOffers(?offers) ;
   getOfferDetails ( !oId, !offers, ?intrfId, ?sd, etc );
   sa ( WriteOffers(!Cons((oId, sd, intrfId, sProps, oProps),
                          deleteOffer(oId,offers))) );
   ter ( !id, !origId, ModifyOrderTer )
endproc
```

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
process StateProc [ s: stateAccess ]
                  ( ps: PropertyValueList, sos: ServiceOfferSpace,
                    ls: OfferPartitionSubgraph ) is
   loop
      var props: PropertyValueList := ps,
          offs: ServiceOfferSpace := sos,
          lns: OfferPartitionSubgraph := ls,
      in
           (   s Read(!props, !offs, !lns)
            [] s ReadProperties(!props)
            [] s ReadOffers(!offs)
            [] s ReadLinks(!lns)
           ); ?props := props; ?offs := offs; ?lns := lns
        []
          s Write(?props, ?offs, ?lns)
        []
          s WriteProperties(?props) ;
          ?offs := offs; ?lns := lns
        []
          s WriteOffers(?offs) ;
          ?props := props; ?lns := lns
        []
          s WriteLinks(?lns) ;
          ?props := props; ?offs := offs
      endvar
   endloop
 |||
   var offerIds: ServiceOfferIdList,
       linkIds: LinkIdList
   init
     ?offerIds := Nil;  ?linkIds := Nil
   in
      loop
          var oId: ServiceOfferId
          in
            s NewServiceOfferId(?oId) [not IsIn (oId, offersIds)] ;
            ?offerIds := Cons(oId, offerIds)
          endvar
        []
          var lnId: LinkId
          in
            s NewLinkId(?lnId) [not IsIn (lnId, linkIds)] ;
            ?linkIds := Cons(oId, linkIds)
          endvar
      endloop
   endvar
endproc

process ManagementInterface [ inv: Invocation, ter: Termination, sa: stateAccess ]
                            (id:interfaceId) is
    ( AddLink [inv, ter, sa] (interfaceId)
     |||
```

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
      RemoveLink [inv, ter, sa] (interfaceId)
    )
 |[inv, ter]|
    ManagementOperationsOrderingConstraints [inv, ter]
endproc

process AddLink  [ inv: Invocation, ter: Termination, sa: stateAccess ]
                 (id:interfaceId) is
   inv ( !id, ?origId, AddLinkInv (?nm, ?lProps, ?targetId) ) ;
   sa ReadLinks(?lns) ;
   sa NewLinkId(?lid) ;
   sa ( !WriteLinks(Cons((lid,nm,lProps,targetId), lns)) ) ;
   ter ( !id, !origId, !AddLinkTer(lid) ;
endproc

process RemoveLink [ inv: Invocation, ter: Termination, sa: stateAccess ]
                   (id:interfaceId) is
   inv ( !id, ?origId, RemoveLinkInv (?lid) )
   sa ReadLinks(?lns);
   sa !WriteLinks(deleteLink(lid,lns)) ;
   ter ( !id, !origId, RemoveLinkTer )
endproc
```

Annex B
(informative)

Guidelines for LOTOS to E-LOTOS translation

B.1 Introduction

This annex is devoted for those LOTOS users that want to update to E-LOTOS and for those interested in seeing the differences and improvements introduced in the language.

E-LOTOS has been designed with the intention of being upward compatible with LOTOS. However, this does not imply that E-LOTOS tools should be able to process LOTOS specification. It rather means that the main concepts of LOTOS (behaviour based on process algebras and data based on algebraic semantics) are the core of E-LOTOS. It is recommended to start with a (brief) looking at Annex A.

In this annex we will not describe the new features of E-LOTOS. Subclause B.1.2 describes how to translate from basic LOTOS to (basic) E-LOTOS. Subclause B.1.3 describe how data types from LOTOS are supported and can be translate to E-LOTOS data types. Subclause B.1.4 describes the translation of a LOTOS specification with data and behaviour to E-LOTOS.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

B.1.1 Specification and process definition

The first change a LOTOS user will notice is that a specification is just a small part of a E-LOTOS description. The introduction of modules in E-LOTOS caused this change. A specification in E-LOTOS is an optional sequence of module declaration and a specification. So, a LOTOS specification is just a E-LOTOS one without modules.

In E-LOTOS, the functionality of processes and specification is not fixed on the definition.

Processes cannot be nested. In fact, the **where** clause of LOTOS has been removed. So, the name space inside a module is flat, so name overriding is not allowed. The translation from LOTOS is straightforward: raise the nested processes (and everywhere in a **where** clause) to the top level definition. Maybe some name solving would be needed.

Process instantiation and recursion It is very usual to specify some system that evolutes through some states or phases, modeled as LOTOS processes, commonly all of them with the same signature (same gates and parameters). For example, a specification of some chip, with a gate per μinstruction and a number of parameters (registers, state variable, etc.). In LOTOS this should be a tedious piece of work in which all gates and variables are repeated every instantiation, without really changing but a little data. In E-LOTOS, you may abbreviate the instantiation of a process by just writing the specific changes in gates and/or parameters.

For example, the famous VendingMachine (taken from [6]):

```
(*LOTOS*)
process Vending [coin,candy1,candy2] : noexit :=
  coin;
    (candy1; Vending[coin,candy1,candy2]
    []candy2; Vending[coin,candy1,candy2]
    )
endproc
```

```
(*E-LOTOS*)
process Vending [coin,candy1,candy2] is
  coin;
    (candy1; Vending[...]
    []candy2; Vending[...]
    )
endproc
```

You may abbreviate just part of the gate list, by stating explicit instantiation, as in the following example:

```
process SingleBuffer [inp,outp] is
  inp; outp; SingleBuffer[...]
endproc

process DoubleBuffer [inp,outp] is
  hide middle in
    SingleBuffer [outp⇒middle,...]
    |[middle]|
    SingleBuffer [inp⇒middle,...]
  endhide
endproc
```

B.1.2 Basic LOTOS

Enabling and exit In E-LOTOS, enabling operator (>>) has been joined with action prefix (;) in just one operator, sequential composition (;). So, LOTOS expressions with enabling operator is easily translated into E-LOTOS, taking care of keeping precedence and removing **exit** from the left part of the expression:

```
a; b; exit >> c; stop   ⇒   a;b;c; stop
a; b; (exit >> c; stop)   ⇒   a; b; c; stop
```

Thus **exit** becomes obsolete, and it is removed from E-LOTOS.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Operator associativity In E-LOTOS, all operators have the same precedence, but action prefix "; ", which has the higher precedence. Asociativity is left-handed, as in LOTOS. However, it is not possible to mix different operators in the same expression without explicitly specifying its asociativity. So, the following expression would cause a syntactical error in E-LOTOS:

B_1 [> B_2 ||| B_3

The precedence in LOTOS is not very intuitive: in this case, this behaviour is parsed as $(B_1$ [> $B_2)$ ||| B_3. This expression can be translated into E-LOTOS as:

$(B_1$ [> $B_2)$ ||| B_3 or **dis** B_1 [> B_2 **enddis** ||| B_3

There are two ways to express precedence in E-LOTOS: via parenthesis or via specific begin-end keywords for each behaviour binary operator. So, we have:

dis	**enddis**	for disabling ([>)
fullsync	**endfullsync**	for full synchronization (||)
conc	**endconc**	for partial synchronization (|[$G_1, \ldots, G_n$]|)
sel	**endsel**	for selection ([])
suspend	**endsuspend**	for disabling ([>)
inter	**endinter**	for interleaving (|||)

The associativity with the same operator is keep, so the following expressions are equivalent both in LOTOS and in E-LOTOS:

B_1 op B_2 op B_3 $\equiv$ B_1 op (B_2 op B_3)

also equivalent to:

op-begin-keyword B_1 op B_2 op B_3 *op-end-keyword*

B.1.3 Data Types

Data types have been revised substantially in E-LOTOS, gaining advantage from practical application of LOTOS in industry. This revision includes more user friendly data types, predefined ones, partial functions, and more intuitive semantics. For a LOTOS user, the main change is in equations: E-LOTOS does not define dynamic semantics for equations. Tools can treat equations as type checked comments.

Constructors and functions E-LOTOS makes a difference between constructors and functions. Therefore, LOTOS **opns** will be split as in the following example:

```
(*LOTOS*)
type Boolean is
  sorts bool
  opns
    true,false: -> bool
    _and_: bool,bool -> bool
endtype
```

```
(*E-LOTOS*)
interface Boolean is
  type bool is
    true,false
  endtype
  function infix and(a: bool,b:bool): bool
endint
```

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Interfaces and implementations In E-LOTOS, data type implementations may come from an external module (which may be a E-LOTOS specification or any other kind of implementation). Boolean would be split in E-LOTOS in the following way:

```
interface Boolean is
  type bool is
    true,false
  endtype
  function infix and(a: bool,b:bool): bool
  eqns
    forall x, y : bool
      ofsort bool
        x and true = x;
        x and false = false;
endint

module Boolean is
  type bool is
    true,false
  endtype
  function infix and(x: bool,y:bool): bool
    case (x,y) in
      (true,true) -> true
      | (any:bool, any: bool) -> false
    endcase
  endfunc
endmod
```

Extensions and combinations of type specifications To specify data types with a large number of operations, or to enrich existing data types, there is a language constructor to combine existing specifications or to extend them. An example is:

```
(*LOTOS*)
type NaturalNumber is boolean
  ...
endtype
```

```
(*E – LOTOS*)
interface NaturalNumber imports Boolean is
  ...
endint

module NaturalNumber imports Boolean is
  ...
endmod
```


© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

A larger example The following is a possible type definition for natural numbers:

```
(*LOTOS*)
type NaturalNumbers is boolean
  sorts nat
  opns
    0: -> nat
    succ: nat -> nat
    _+_: nat,nat -> nat
    _<_: nat,nat -> bool
  eqns
    forall m, n : nat
      ofsort nat
        m + 0 = m;
        m + succ(n) = succ (m+n);
      ofsort bool
        0 < 0 = false;
        0 < succ(n) = true;
        succ(m) < 0 = false;
        succ(m) < succ(n) = m < n;
endtype
```

```
(*E – LOTOS*)
interface NaturalNumbers imports boolean is
  type nat is
    0 | succ(n:nat)
  endtype
  function infix +(m:nat,n:nat):nat
function infix <(m:nat,n:nat):bool
eqns
    forall m, n : nat
      ofsort nat
        m + 0 = m;
        m + succ(n) = succ (m+n);
      ofsort bool
        0 < 0 = false;
        0 < succ(n) = true;
        succ(m) < 0 = false;
        succ(m) < succ(n) = m < n;
endint

module NaturalNumbers imports Boolean is
  type nat is
    0 | Succ (n: nat)
  endtype

  function infix + (m: nat, n: nat): nat is
    case n in
      0 -> m
      | Succ (n1: nat) -> Succ (m) + n1
    endcase
  endfunc
endmod
```

E-LOTOS has predefined types, so you do not really need to translate basic types as Booleans, Natural, and others (see Clause 10). However, NaturalNumber is a good example for showing the differences between LOTOS and E-LOTOS data types.

The main difference in the example is that *definition* may be separated from *implementation*. This allows providing external implementation of data types.

Parameterized types A parameterized type in LOTOS can be seen as a template for building new types, or as a partial specification with general features of a data type. These specifications will be completed later, obtaining complete data type specifications. The usual applications are general containers, as "queues", "stacks", etc. In LOTOS, this is done via **formal** sorts, operations and equations.

In E-LOTOS, instead of **formal** types, we use **generic** modules. Genericity is used for constructing specifications and for code reuse. The above example would be specified in E-LOTOS as follows:

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
(*LOTOS*)
type Queue is
  formalsorts element
  formalopns e0: -> element
  sorts queue
  opns
    create: -> queue
    add: element, queue -> nat
    first: queue -> element
    eqns
    ...
endtype
```

```
(*E – LOTOS*)
generic Queue (Eq:EqType) is
  type M is
    nil | cons(E,M)
  endtype
endgen
```

and thus can be actualized by any other type, for example a queue of natural numbers:

```
(*LOTOS*)
type NatQueue is
  Queue actualizedby NaturalNumbers using
    sortnames nat for element
    opnnames 0 for e0
endtype
```

```
(*E – LOTOS*)
module ListNat is
  GenericList (Natural renaming (types nat:=E))
    renaming (types ListNat:=List)
endmod
```

However, a **list** constructor is available in E-LOTOS as predefined type, so a list of natural number could be as simple as:

```
E-LOTOS
type NatQueue is list of Nat endtype
```

Renaming Renaming of data type specifications is useful when some required semantics is already captured in an existing data type. It is also very common to apply renaming for rapid prototyping, where existing data type definition is close to the goal and further details are left for further refinement. Renaming is applied to sorts and operations. With different syntax, the same construction exists in E-LOTOS, and it applies to every component of a module (so, it is extended to rename values, processes, etc.).

```
(*LOTOS*)
type Connection is
  Queue renamedby
    sortnames
      channel for queue
      message for element
  opnnames
      send for add
      receive for first
endtype
```

```
(*E – LOTOS*)
module Connection is
  Queue renaming (
      types queue := channel
      opns element := message,
        add := send,
        first := receive
      )
  endmod
```

B.1.4 Full LOTOS

Write-many variables In E-LOTOS we have write-many variables. As in LOTOS there were just write-once variables, there is no problem, as typically in LOTOS variable definition overrides previous declared ones.

Accept **Accept** construction is not needed in E-LOTOS, as sequential composition (";") is used. Values are automatically passed from one behaviour to the next one.

 © ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Let **Let** construction translation is straightforward: in E-LOTOS it is called **var**: the only difference is that **endvar** is needed in E-LOTOS.

Successful termination with value offers In LOTOS, **exit** is used to specify the successful termination of a process, possibly with a list of values. There are two typical applications: generation of values as a result of some processing or passing values from a behaviour to another. Both cases are implicit in E-LOTOS: the first one is usually modelled via **out** parameters of a process. The second one is simply the default semantics of E-LOTOS (see **accept** above).

© ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Bibliography

[1] ODP Trading Function. Draft Rec. X9tr — ISO/IEC DIS 13235, June 1995.

[2] Open Distributed Processing – Reference Model – Part 2: Foundations. ISO/IEC 10746-2, 1995.

[3] Open Distributed Processing – Reference Model – Part 3: Architecture. ISO/IEC 10746-3, 1995.

[4] ISO 8807. *LOTOS—A formal description technique based on the temporal ordering of observational behaviour*, 1989.

[5] R.N. Bol and J.F. Groote. The meaning of negative premises in transition system specifications. *Journal of the ACM*, 43(5):863–914, September 1996. An extended abstract appeared in J. Leach Albert, B. Monien, and M. Rodriguez Artalejo, editors, Procceding of 18th ICALP, Madrid, pages 481–494, 1991.

[6] T. Bolognesi and Ed Brinksma. Introduction tho the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14, 1987.

[7] Jim Davies, Dave Jackson, and Steve Schneider. Broadcast communication for real-time processes. In J. Vytopil, editor, *Proc. Formal Techniques in Real-Time and Fault-Tolerant systems*, pages 149–170. Springer-Verlag, 1992. LNCS 571.

[8] H. Ehrig and B. Mahr. Fundamentals of algebraic specification. *Bull. Euro. Assoc. Theoret. Comp. Sci.*, 6, 1985.

[9] Hubert Garavel and Radu Mateescu. French-Romanian proposal for capture of requirements and expression of properties in E-LOTOS modules. Rapport SPECTRE 96-04, VERIMAG, Grenoble, May 1996. Input document (KC4) to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Kansas City, Missouri, USA, May, 12–21, 1996.

[10] Hubert Garavel and Mihaela Sighireanu. French-Romanian integrated proposal for the user language of E-LOTOS. Rapport SPECTRE 96-05, VERIMAG, Grenoble, May 1996. Input document (KC3) to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Kansas City, Missouri, USA, May, 12–21, 1996.

[11] Hubert Garavel and Mihaela Sighireanu. On the introduction of exceptions in LOTOS. In R. Gotzhein and J. Bredereke, editors, *Formal Description Techniques IX and Protocol Specification, Testing and Verification XVI*, pages 469–484. Chapman & Hall, London, 1996.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[13] Alan Jeffrey. Semantics for a fragment of LOTOS with functional data and abstract datatypes. In *Revised Working Draft on Enhancements to LOTOS (v3)*, ISO/IEC JTC1/SC21/WG7 N1053, chapter Annexe A. 1995.

[14] Alan Jeffrey. A core date and behaviour language for E-LOTOS. Input document (KC1) to the ISO/IEC JTC1/SC21/WG7/E-LOTOS meeting in Kansas City, May 1996.

 © ISO/IEC 2001 – All rights reserved

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

[15] Alan Jeffrey, Hubert Garavel, Guy Leduc, Charles Pecheur, and Mihaela Sighireanu. Towards a proposal for datatypes in E-LOTOS. Annex A of ISO/IEC JTC1/SC21 N10108 Second Working Draft on Enhancements to LOTOS. Output document of the edition meeting, Ottawa (Canada), July, 20–26, 1995, October 1995.

[16] Alan Jeffrey and Guy Leduc. E-LOTOS core language. Chapter 3 of [26], 1996.

[17] L. Léonard and G. Leduc. A formal definition of time in LOTOS. *Formal Aspects of Computing*, 1998.

[18] Luc Léonard and Guy Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 29(3):271–292, 1997.

[19] Luc Léonard and Guy Leduc. A formal definition of time in LOTOS. *Formal Aspects of Computing*, 10:248–266, 1998.

[20] Luc Léonard and Guy Leduc. A formal definition of time in LOTOS. *Formal Aspects of Computing - Electronic journal version*, 10:28–96, 1998. http://www.link.springer.de/link/service/journals/00165/supp/1998/801003024 8.pdf.

[21] Giovanny Lucero and Juan Quemada. An E-LOTOS specification of the ODP Trader. Input document (GR1) to the ISO/IEC JTC1/SC21/WG7/E-LOTOS meeting in Grenoble, December 1996.

[22] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[23] Harold B. Munster. LOTOS specification of the MAA standard, with an evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[24] Charles Pecheur. A proposal for data types for E-LOTOS. Technical report, University of Liège, October 1994. Annex H of ISO/IEC JTC1/SC21/WG1 N1349 Working Draft on Enhancements to LOTOS.

[25] Juan Quemada, editor. Revised working draft on enhancements to LOTOS (v2). ISO/IEC JTC1/SC21/WG7 N10108 Project 1.21.20.2.3. Output document of the Ottawa meeting, January 1996.

[26] Juan Quemada, editor. Revised working draft on enhancements to lotos (v4). ISO/IEC JTC1/SC21/WG1 N1173 Project 1.21.20.2.3. Output document of the Kansas City meeting, September 1996.

[27] Mihaela Sighireanu and Hubert Garavel. On the definition of modular E-LOTOS. Input document (GR2) to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Grenoble, France, December 9-11, 1996, December 1996.

[28] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. *Nord. J. Computation*, 2(2):274–302, 1995. Also appeared in *Proceedings CSL'93*, Swansea 1993.

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ICS 35.060; 35.080

Price based on 187 pages

© ISO/IEC 2001 – All rights reserved

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale