



ISO/IEC JTC1/SC7
Software Engineering
Secretariat: CANADA (SCC)

ISO/IEC JTC1/SC7 N2393

2000-11-14

Document Type	FDIS Ballot
Title	FDIS 15437: Information technology - Enhancements to LOTOS (E-LOTOS)
Source	JTC1/SC7 Secretariat
Project	07.20.02.03
Status	Sent to ITTF for FDIS ballot
References	N2392
Action ID	FYI or ACT
Due Date	
Mailing Date	2000-11-14
Distribution	SC7_AG, ITTF Secr., JTC1 Sec.
Medium	Encoded Acrobat
No. Of Pages	197
Note	This is NOT a ballot. It is circulated to SC7 member bodies FOR INFORMATION PURPOSE ONLY. The document has been sent to ITTF for FDIS balloting.



EXPLANATORY REPORT	ISO/IEC FDIS 15437
ISO/IEC JTC 1/SC 7 N2393	
Will supersede: SC7 N1994	Secretariat: Canada (SCC)

This form should be sent to ITTF, together with the committee draft, by the secretariat of the joint technical committee or sub-committee concerned.

The accompanying document is submitted for circulation to member body vote as a FDIS, following consensus of the P-members of the committee obtained on:		1998-10-03
<input checked="" type="checkbox"/>	by postal ballot initiated on:	1998-06-03
P-members in favour:		6
P-members voting against:		0
P-members abstaining:		2
P-members who did not vote:		18
O-members in favour		0

Remarks:	
This document meets all the criteria for FDIS balloting.	
Project: 07.20.02.03	
I hereby confirm that this draft meets the requirements of part 3 of the IEC/ISO Directives.	
Date:	2000-11-14
Name and signature of the secretary:	Jean-Normand Drouin

ISO/IEC JTC 1/SC 7

Date: 2000-10-24

ISO/IEC DIS 15437

ISO/IEC JTC 1/SC 7/WG

Secretariat: Standards Council of Canada

Information Technology – E-LOTOS

Document type: International Standard

Document subtype: Not applicable

Document stage: (40) Approval

Document language: E

Document1 ISOSTD ISO Template Version 3.3 1998-01-12

Contents

1	Scope	1
2	Conformance	1
3	Normative references	1
4	Terms, definitions and notation	1
5	Symbols	2
6	E-LOTOS grammar	2
6.1	Lexical Structure	2
6.1.1	Character set	2
6.1.2	Comments and separators	2
6.1.3	Identifiers	3
6.1.4	Reserved words	3
6.1.5	Identifiers classes	4
6.1.6	Non-terminals classes	4
6.2	Syntax of the language	5
6.2.1	Specification	5
6.2.2	Top-level declaration	6
6.2.3	Module body	6
6.2.4	Module expression	6
6.2.5	Module formal parameters	6
6.2.6	Interface expressions	6
6.2.7	Record module expression	6
6.2.8	Interface body	7
6.2.9	Formal parameter list	7
6.2.10	Renaming/Instantiation	7
6.2.11	Equations declaration	7
6.2.12	Simple equations declaration	8
6.2.13	Declarations	8
6.2.14	Expressions	8
6.2.15	Expression Atoms	8
6.2.16	Behaviour expressions	9
6.2.17	Disabling behaviour expression	10

6.2.18	Synchronization behaviour expression	10
6.2.19	Concurrency behaviour expression	10
6.2.20	Selection behaviour expression	10
6.2.21	Suspend/Resume behaviour expression	10
6.2.22	Interleaving behaviour expression	10
6.2.23	Behaviour term	10
6.2.24	Behaviour atom	11
6.2.25	Type expressions	12
6.2.26	Record type expressions	12
6.2.27	Value expressions	12
6.2.28	Record value expressions	13
6.2.29	Patterns	13
6.2.30	Gate parameter list	13
6.2.31	Actual parameter list	13
6.2.32	Exception parameter list	13
6.2.33	Record patterns	14
6.2.34	Behaviour pattern matching	14
6.2.35	Expression pattern matching	14
6.2.36	In parameter list	14
6.2.37	In parameter	14
7	E-LOTOS abstract syntax	14
7.1	Overview	14
7.1.1	Syntactic sugar	15
7.1.2	Abstract syntax	15
7.2	Concrete to abstract syntactic translation	16
7.2.1	Interface body	16
7.2.2	Formal parameter list	18
7.2.3	Declarations	18
7.2.4	Expressions	21
7.2.5	Expression atoms	21
7.2.6	Behaviour expressions	24
7.2.7	Interleaving behaviour expression	24
7.2.8	Behaviour atom	24
7.2.9	Type expressions	30
7.2.10	Record type expressions	31
7.2.11	Record value expressions	31
7.2.12	Gate parameter list	32
7.2.13	Actual parameter list	32
7.2.14	Exception parameter list	32
7.2.15	Record patterns	32
7.2.16	In parameter list	33
7.2.17	In parameter	33
8	E-LOTOS semantics	34
8.1	Overview	34
8.2	Static Semantics	35
8.2.1	Static semantic objects for Base	35
8.2.2	Judgements on static semantics for Base	36
8.2.3	Extended identifiers	37
8.2.4	Static semantic objects for Modules	38
8.2.5	Judgements on static semantics for Modules	38
8.2.6	Cycle freedom	39
8.2.7	Context morphism	39

8.2.8	Realization	39
8.2.9	Interface Instantiation	39
8.2.10	Interface Matching	40
8.2.11	Renaming/Instantiation	40
8.3	Untimed dynamic semantics	43
8.3.1	Untimed dynamic semantic objects for Base	43
8.3.2	Judgements on untimed dynamic semantics for Base	44
8.3.3	Dynamic semantic objects for Modules	45
8.3.4	Judgements on untimed dynamic semantics for Modules	46
8.3.5	Environment morphism	47
8.3.6	Signature Instantiation	47
8.3.7	Renaming/Instantiation	47
8.4	Timed dynamic semantics	48
8.4.1	Judgements on timed dynamic semantics	48
8.5	Write-many variables: the value substitution operator	48
9	The E-LOTOS modules	51
9.1	Specification	51
9.1.1	Specification	51
9.2	Top-level declaration	52
9.2.1	Module not constrained by an interface	52
9.2.2	Module constrained by an interface	53
9.2.3	Generic module not constrained by an interface	53
9.2.4	Generic module constrained by an interface	54
9.2.5	Interface declaration	54
9.2.6	Sequential top declaration	55
9.3	Module body	55
9.3.1	Block declaration	56
9.3.2	Module Expression	56
9.4	Module expression	56
9.4.1	Module aliasing not constrained by an interface	56
9.4.2	Module aliasing constrained by an interface	57
9.4.3	Generic module actualization not constrained by an interface	57
9.4.4	Generic module actualization constrained by an interface	58
9.4.5	Generic module renaming/instantiation	58
9.5	Module formal parameters	59
9.5.1	Single	59
9.5.2	Disjoint union	59
9.6	Interface expressions	60
9.6.1	Interface identifier	60
9.6.2	Simple renaming	60
9.6.3	Explicit body	61
9.7	Interface body	61
9.7.1	Type hiding the implementation	61
9.7.2	Type synonym	61
9.7.3	Constructed type	62
9.7.4	Named record type	62
9.7.5	Process declaration	62
9.7.6	Equations	63
9.7.7	Sequential declaration	63
9.8	Record module expression	63
9.8.1	Single	63
9.8.2	Disjoint union	64

9.8.3	Renaming tuple	64
9.9	Equation declarations	64
9.9.1	Equations declaration	64
9.9.2	Sequential	65
9.10	Simple equation declaration	65
9.11	Declarations	65
9.11.1	Type synonym	65
9.11.2	Type declaration	66
9.11.3	Named record type	66
9.11.4	Process declaration	66
9.11.5	Sequential declarations	67
10	The E-LOTOS base language	68
10.1	Introduction	68
10.2	Behaviours	68
10.2.1	Disabling behaviour expression	68
10.2.2	Synchronization behaviour expression	69
10.2.3	Concurrency behaviour expression	70
10.2.4	Selection behaviour expression	72
10.2.5	Suspend/Resume behaviour expression	72
10.2.6	Sequential composition	73
10.2.7	Action	74
10.2.8	Internal action	75
10.2.9	Successful termination without values	75
10.2.10	Successful termination	75
10.2.11	Inaction	76
10.2.12	Time block	76
10.2.13	Delay	76
10.2.14	Assignment	77
10.2.15	Nondeterministic Assignment	77
10.2.16	Choice over values	78
10.2.17	Trap	79
10.2.18	General parallel	80
10.2.19	Parallel over values	82
10.2.20	Variable declaration	83
10.2.21	Gate hiding	84
10.2.22	Renaming	85
10.2.23	Process instantiation	86
10.2.24	loop iteration	87
10.2.25	Case	87
10.2.26	Case with tuples	88
10.2.27	Signalling	89
10.3	Type expressions	89
10.3.1	Type identifier	89
10.3.2	Empty type	90
10.3.3	Universal type	90
10.3.4	Record type	90
10.4	Record type expressions	90
10.4.1	Singleton record	90
10.4.2	Universal record	91
10.4.3	Record disjoint union	91
10.4.4	Empty record	91
10.5	Value expressions	92

10.5.1	Primitive constants	92
10.5.2	Variables	92
10.5.3	Record values	92
10.5.4	Constructor application	92
10.6	Record value expressions	92
10.6.1	Singleton record	92
10.6.2	Record disjoint union	93
10.6.3	Empty record	93
10.7	Patterns	93
10.7.1	Record pattern	93
10.7.2	Wildcard	94
10.7.3	Variable binding	94
10.7.4	Expression pattern	94
10.7.5	Constructor application	95
10.7.6	Explicit typing	95
10.8	Record patterns	96
10.8.1	Singleton record pattern	96
10.8.2	Record wildcard	96
10.8.3	Record match	97
10.8.4	Record disjoint union	97
10.8.5	Empty record pattern	97
10.9	Record of variables	98
10.9.1	Singleton record variable	98
10.9.2	Record disjoint union	98
10.10	Behaviour pattern-matching	98
10.10.1	Single match	98
10.10.2	Multiple match	99
11	Predefined library	100
11.1	Booleans	100
11.2	Natural Numbers	102
11.3	Integral Numbers	106
11.4	Rational Numbers	108
11.5	Floating Point Numbers	111
11.6	Characters	113
11.7	Strings	113
11.8	Enumerated Type Scheme	114
11.9	Record Type Scheme	116
11.10	Set Type Scheme	116
11.11	List Type Scheme	119
A	Tutorial	121
A.1	The base language	121
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	157
A.3.3	E-LOTOS Specification of the trader	158
A.3.4	The complete specification	161
B	Guidelines for LOTOS to E-LOTOS translation	179
B.1	Introduction	179
B.1.1	Specification and process definition	180

B.1.2	Basic LOTOS	180
B.1.3	Data Types	181
B.1.4	Full LOTOS	184

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.

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 (\gg) pair. For example $a;B$ or $B1;B2$. In LOTOS the \gg 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

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.

Information Technology – E-LOTOS

1 Scope

This document defines the syntax and semantics of the enhanced LOTOS language (IS8807), named E-LOTOS. E-LOTOS is in general 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 standard if and only if it is derivable according to the syntactic rules defined in clause 6 and the semantics is unambiguously derivable from the semantic definition in clause 8.

3 Normative references

ISO 8807 Information Processing Systems - Open Systems Interconnection
 LOTOS - A formal description technique based on the temporal ordering of observational behaviour

4 Terms, definitions and notation

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

<i>Symbol</i>	<i>meaning</i>
	alternative definition
<i>D, K, V, S, RT...</i>	terminal and non-terminal symbols
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 to note that the metalinguistic '()' and '[]' must not be confused with the E-LOTOS symbols '[', ']', '(', ')', and '|'. Anyway, in the syntax, these E-LOTOS symbols appear between singles quotes. Other

symbols (: , -> , etc.) will appear without quotes to keep the International Standard readable.

Syntax for predefined types are described in clause 11.

5 Symbols

6 E-LOTOS grammar

6.1 Lexical Structure

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

6.1.1 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.

6.1.2 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 change their meaning.

6.1.3 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 finished by 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, as “+”, “/”, etc. They are built from special characters, characters and digits.

```
<special-identifier> ::=
    <special-character> { <special-character> }
    | <digit> { [ "_" ] <normal-character> }
```

6.1.4 Reserved words

The reserved words for the language are:

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

The reserved lexical tokens for the language are:

```

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

```

6.1.5 Identifiers classes

The terminals of the language syntax are:

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

6.1.6 Non-terminals classes

The non-terminals for the language are:

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

6.2 Syntax of the language

6.2.1 Specification

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

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

6.2.2 Top-level declaration

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

6.2.3 Module body

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

6.2.4 Module expression

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

6.2.5 Module formal parameters

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

6.2.6 Interface expressions

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

6.2.7 Record module expression

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

6.2.8 Interface body

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

6.2.9 Formal parameter list

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

6.2.10 Renaming/Instantiation

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

6.2.11 Equations declaration

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

6.2.12 Simple equations declaration

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

6.2.13 Declarations

$D ::=$

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

6.2.14 Expressions

$E ::=$

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

6.2.15 Expression Atoms

$EA ::=$

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

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

6.2.16 Behaviour expressions

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

6.2.17 Disabling behaviour expression

$\begin{aligned} \text{DisB} ::= & \text{BT } [> \text{BT} \\ & \text{BT } [> \text{DisB} \end{aligned}$	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding-right: 10px;"><i>singleton</i></td> <td>(DisB1)</td> </tr> <tr> <td><i>disjoint union</i></td> <td>(DisB2)</td> </tr> </table>	<i>singleton</i>	(DisB1)	<i>disjoint union</i>	(DisB2)
<i>singleton</i>	(DisB1)				
<i>disjoint union</i>	(DisB2)				

6.2.18 Synchronization behaviour expression

$\begin{aligned} \text{SyncB} ::= & \text{BT } \text{BT} \\ & \text{BT } \text{SyncB} \end{aligned}$	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding-right: 10px;"><i>synchronization</i></td> <td>(SyncB1)</td> </tr> <tr> <td><i>synchronization</i></td> <td>(SyncB2)</td> </tr> </table>	<i>synchronization</i>	(SyncB1)	<i>synchronization</i>	(SyncB2)
<i>synchronization</i>	(SyncB1)				
<i>synchronization</i>	(SyncB2)				

6.2.19 Concurrency behaviour expression

$\begin{aligned} \text{ConcB} ::= & \text{BT } [[G(,G)^*] \text{BT} \\ & \text{BT } [[G(,G)^*] \text{ConcB} \end{aligned}$	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding-right: 10px;"><i>concurrency</i></td> <td>(ConcB1)</td> </tr> <tr> <td><i>concurrency</i></td> <td>(ConcB2)</td> </tr> </table>	<i>concurrency</i>	(ConcB1)	<i>concurrency</i>	(ConcB2)
<i>concurrency</i>	(ConcB1)				
<i>concurrency</i>	(ConcB2)				

6.2.20 Selection behaviour expression

$\begin{aligned} \text{SelB} ::= & \text{BT } \square \text{BT} \\ & \text{BT } \square \text{SelB} \end{aligned}$	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding-right: 10px;"><i>choice</i></td> <td>(SelB1)</td> </tr> <tr> <td><i>choice</i></td> <td>(SelB2)</td> </tr> </table>	<i>choice</i>	(SelB1)	<i>choice</i>	(SelB2)
<i>choice</i>	(SelB1)				
<i>choice</i>	(SelB2)				

6.2.21 Suspend/Resume behaviour expression

$\begin{aligned} \text{SuspendB} ::= & \text{BT } [X > \text{BT} \\ & \text{BT } [X > \text{SuspendB} \end{aligned}$	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding-right: 10px;"><i>suspend/resume</i></td> <td>(SuspendB1)</td> </tr> <tr> <td><i>suspend/resume</i></td> <td>(SuspendB2)</td> </tr> </table>	<i>suspend/resume</i>	(SuspendB1)	<i>suspend/resume</i>	(SuspendB2)
<i>suspend/resume</i>	(SuspendB1)				
<i>suspend/resume</i>	(SuspendB2)				

6.2.22 Interleaving behaviour expression

$\begin{aligned} \text{InterB} ::= & \text{BT } \text{BT} \\ & \text{BT } \text{InterB} \end{aligned}$	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding-right: 10px;"><i>interleaving</i></td> <td>(InterB1)</td> </tr> <tr> <td><i>interleaving</i></td> <td>(InterB2)</td> </tr> </table>	<i>interleaving</i>	(InterB1)	<i>interleaving</i>	(InterB2)
<i>interleaving</i>	(InterB1)				
<i>interleaving</i>	(InterB2)				

6.2.23 Behaviour term

$\begin{aligned} \text{BT} ::= & \text{BA} \\ & \text{BA} ; \text{BT} \end{aligned}$	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding-right: 10px;"><i>behaviour atom</i></td> <td>(BT1)</td> </tr> <tr> <td><i>sequential composition</i></td> <td>(BT2)</td> </tr> </table>	<i>behaviour atom</i>	(BT1)	<i>sequential composition</i>	(BT2)
<i>behaviour atom</i>	(BT1)				
<i>sequential composition</i>	(BT2)				

6.2.24 Behaviour atom

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

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

6.2.25 Type expressions

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

6.2.26 Record type expressions

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

6.2.27 Value expressions

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

6.2.28 Record value expressions

$RN ::= V \Rightarrow N(, V \Rightarrow N)^*$	<i>named</i> (RN1)
$N(, N)^*$	<i>positional</i> (RN2)

6.2.29 Patterns

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

6.2.30 Gate parameter list

$GPL ::= G \Rightarrow G (, G \Rightarrow G)^* [, \dots]$	<i>explicit instantiation</i> (GPL1)
\dots	<i>explicit instantiation</i> (GPL2)
$G(, G)^*$	<i>positional instantiation</i> (GPL3)

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

6.2.31 Actual parameter list

$APL ::= V \Rightarrow (E P) (, V \Rightarrow (E P))^* [, \dots]$	<i>explicit instantiation</i> (APL1)
\dots	<i>explicit instantiation</i> (APL2)
$(E P)(, (E P))^*$	<i>positional instantiation</i> (APL3)

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

6.2.32 Exception parameter list

$XPL ::= X \Rightarrow X (, X \Rightarrow X)^* [, \dots]$	<i>explicit instantiation</i> (XPL1)
\dots	<i>explicit instantiation</i> (XPL2)
$X(, X)^*$	<i>positional instantiation</i> (XPL3)

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

6.2.33 Record patterns

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

6.2.34 Behaviour pattern matching

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

6.2.35 Expression pattern matching

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

6.2.36 In parameter list

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

6.2.37 In parameter

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

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

7 E-LOTOS abstract syntax

7.1 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.

7.1.1 Syntactic sugar

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

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

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

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

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

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

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

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

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

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

7.1.2 Abstract syntax

The following constructions are defined:

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

Abstract syntax is a superset of the syntax defined by the grammar in Clause 6. 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 7.2.10, 7.2.11, and 7.2.15 respectively.

RT is considered a *T* in abstract syntax, see Subclause 7.2.9, although it is not allowed to express anonymous record types in E-LOTOS text, this rule ease the semantics definition.

7.2 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 removed clauses, ‘+’ 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.

7.2.1 Interface body

7.2.1.1 Overview

Concrete syntax

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

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

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

7.2.1.2 Process declaration

Concrete syntax

```

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

```

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

Context

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

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

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

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

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

Syntax sugar

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

where (ordered with ascendant indexes):

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

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

7.2.1.3 Function declaration

Concrete syntax

```

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

```

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

Syntax sugar

$$\left(\begin{array}{l} \text{function } F \quad [' (' FPL ') '] [: T] \\ \quad \text{raises } ' [' X_1[: T_1], \dots, X_n[: T_n] '] ' \end{array} \right) \stackrel{\text{def}}{=} \left(\begin{array}{l} \text{process } F \quad [' (' FPL ') '] [: T] \\ \quad \text{raises } ' [' X_1[: T_1], \dots, X_n[: T_n] '] ' \end{array} \right)$$

7.2.1.4 Infix function declaration

Concrete syntax

$$\mathbf{function} \ F \ \mathbf{infix} \ [\ ' \ (\ [(\mathbf{in}|\mathbf{out})]V:T, [(\mathbf{in}|\mathbf{out})]V:T \ ' \) \ ' \] \ [: T] \ [\ \mathbf{raises} \ ' \ [\ ' \ X[: T] \ (, X[: T])^* \ ' \] \ ' \] \]$$

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

Syntax sugar

$$\left(\mathbf{function} \ F \ \mathbf{infix} \ [\ ' \ (\ FP_1, FP_2 \ ' \) \ ' \] \ [: T] \ [\ \mathbf{raises} \ ' \ [\ ' \ X_1[: T_1], \dots, X_n[: T_n] \ ' \] \ ' \] \] \right) \stackrel{\text{def}}{=} \left(\mathbf{process} \ F \ [\ ' \ (\ FP_1, FP_2 \ ' \) \ ' \] \ [: T] \ [\ \mathbf{raises} \ ' \ [\ ' \ X_1[: T_1], \dots, X_n[: T_n] \ ' \] \ ' \] \] \right)$$

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

7.2.1.5 Constant value

Concrete syntax

$$\mathbf{value} \ V : S$$

Syntax sugar

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

7.2.2 Formal parameter list

7.2.2.1 Overview

Concrete syntax

$FPL ::= V:T$	<i>singleton</i> (FPL1)
* $\mathbf{in} \ V:T$	<i>input singleton</i> (FPL2)
* $\mathbf{out} \ V:T$	<i>output singleton</i> (FPL3)
FPL, FPL	<i>disjoint union</i> (FPL4)

This category has disappeared, see Subclause 7.2.1.2, 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**.

7.2.3 Declarations

7.2.3.1 Overview

Concrete syntax

$D ::= \mathbf{type} \ S \ \mathbf{renames} \ T \ \mathbf{endtype}$	<i>type synonym</i> (D1)
$\mathbf{type} \ S \ \mathbf{is} \ C[\ ' \ (\ RT \ ' \) \ ' \] \ (\ ' \ \ ' \ C[\ ' \ (\ RT \ ' \) \ ' \] \)^* \ \mathbf{endtype}$	<i>type declaration</i> (D2)
$\mathbf{type} \ S \ \mathbf{is} \ ' \ (\ RT \ ' \) \ ' \ \mathbf{endtype}$	<i>named records type declaration</i> (D3)
* $\mathbf{process} \ \Pi \ [\ ' \ [\ ' \ [G[: T] \ (, G[: T])^* \ ' \] \ ' \] \ [\ ' \ (\ FPL \ ' \) \ ' \] \ [\ \mathbf{raises} \ [X[: T] \ (, X[: T])^* \] \]$	<i>process with in/out parameters</i> (D4)
$\mathbf{is} \ B$	
$\mathbf{endproc}$	

- ★ | **function** F [' (' FPL ') '] [: T] *function with in/out parameters* (D5)
 [**raises** ' [' X [: T] (, X [: T]) * '] ']
 is E
 endfunc
- ★ | **function** F **infix** ' (' [(**in|out**) V : T , [(**in|out**) V : T '] '] *infix function* (D6)
 [: T]
 [**raises** ' [' X [: T] (, X [: T]) * '] ']
 is E
 endfunc
- ★ | **value** V : S **is** E **endval** *constant value declaration* (D7)
 | D D *sequential declaration* (D8)
- + | **process** Π [' [' [G [: T] (, G [: T]) *] '] '] *process with return value* (D9)
 [' (' V : T (, V : T) * ') ']
 : T
 [**raises** ' [' X [: T] (, X [: T]) * '] ']

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

7.2.3.2 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 (**etc**), the default in parameter is (), the default result type is **none**, the default exception list is [] and the default exception type is ().

Context

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

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

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

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

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

Syntax sugar

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

where (every tuple is ordered with ascendant indexes):

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

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

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

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

7.2.3.3 Function declaration

Concrete syntax

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

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

Syntax sugar

$$\left(\begin{array}{l} \text{function } F [\text{' (' } FPL \text{' ')}] [: T] \\ \quad \text{raises } [\text{' X}_1[: T_1], \dots, X_n[: T_n] \text{' '}] \\ \quad \text{is } E \\ \text{endfunc} \end{array} \right) \stackrel{\text{def}}{=} \left(\begin{array}{l} \text{process } F [\text{' (' } FPL \text{' ')}] [: T] \\ \quad \text{raises } [\text{' X}_1[: T_1], \dots, X_n[: T_n] \text{' '}] \\ \quad \text{is } E \\ \text{endproc} \end{array} \right)$$

7.2.3.4 Infix function declaration

Concrete syntax

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

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

Syntax sugar

$$\left(\begin{array}{l} \text{function } F \text{ infix } [\text{' (' } FP_1, FP_2 \text{' ')}] [: T] \\ \quad \text{raises } [\text{' X}_1[: T_1], \dots, X_n[: T_n] \text{' '}] \\ \quad \text{is } E \\ \text{endfunc} \end{array} \right) \stackrel{\text{def}}{=} \left(\begin{array}{l} \text{process } F [\text{' (' } FP_1, FP_2 \text{' ')}] [: T] \\ \quad \text{raises } [\text{' X}_1[: T_1], \dots, X_n[: T_n] \text{' '}] \\ \quad \text{is } E \\ \text{endproc} \end{array} \right)$$

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

7.2.3.5 Constant value declarations

Concrete syntax

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

Syntax sugar

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

7.2.4 Expressions

7.2.4.1 Overview

Concrete syntax

$E ::= EA$ *expression atom* (E1)
 | $EA ; E$ *sequential composition* (E2)

Syntax sugar See notes on syntax sugar for clause 7.2.5.

7.2.5 Expression atoms

7.2.5.1 Overview

Concrete syntax

$EA ::= P := EA$ *assignment* (EA1)
 | **trap** *trap* (EA2)
 (**exception** X [$'(IPL)'$] **is** E **endexn**)*
 [**exit** [P] **is** E **endexit**]
 in E
 endtrap
 | **var** $V : T[:=E](, V : T[:=E])^*$ **in** *variable declaration* (EA3)
 E
 endvar
 | **rename** *renaming* (EA4)
 (**signal** X [$'(IPL)'$] **is** X [E])^{*}
 in E
 endren
 * | **loop** E **endloop** *iteration* (EA5)
 * | **loop** X [$:T$] **in** *breakable iteration* (EA6)
 E
 endloop
 * | **while** E **do** *while iteration* (EA7)
 E
 endwhile
 * | **for** E **while** E **by** E **do** *for iteration* (EA8)
 E
 endfor

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

Syntax sugar Some of these clauses are syntax sugar, some of them with an explicit translation as described in following subclauses and some with the same translation that applies for behaviours. The rest are particular cases of behaviours only capable of performing termination (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.

7.2.5.2 Infix function instantiation

Concrete syntax

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

The default exception list is [].

Syntax sugar

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

where $AP_i = EA_i|P_i$.

7.2.5.3 Value

Concrete syntax

$$N$$

Syntax sugar

$$N \stackrel{\text{def}}{=} \mathbf{exit}(\$1 \Rightarrow N)$$
7.2.5.4 Conjunction**Concrete syntax**

$$EA \mathbf{andalso} EA$$
Syntax sugar

$$EA_1 \mathbf{andalso} EA_2 \stackrel{\text{def}}{=} \mathbf{if} EA_1 \mathbf{then} EA_2 \mathbf{else} \mathbf{false}$$
7.2.5.5 Disjunction**Concrete syntax**

$$EA \mathbf{orelse} EA$$
Syntax sugar

$$EA_1 \mathbf{orelse} EA_2 \stackrel{\text{def}}{=} \mathbf{if} EA_1 \mathbf{then} \mathbf{true} \mathbf{else} EA_2$$
7.2.5.6 Equality**Concrete syntax**

$$EA = EA$$
Syntax sugar

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

$$EA \langle \rangle EA$$
Syntax sugar

$$EA_1 \langle \rangle EA_2 \stackrel{\text{def}}{=} \mathbf{if} EA_1 = EA_2 \mathbf{then} \mathbf{false} \mathbf{else} \mathbf{true}$$
7.2.5.8 Field select**Concrete syntax**

$$EA . V$$
Syntax sugar

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

7.2.5.9 Explicit typing

Concrete syntax

$$EA : T$$

Syntax sugar

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

7.2.6 Behaviour expressions

Syntax

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

Syntax sugar Note that this category is introduced to prioritize operators, and it is just the union of other categories. Those categories that need syntactic translation are included in the next sections. The rest will be revisited in the semantic Clause 10.

7.2.7 Interleaving behaviour expression

Concrete syntax

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

Syntax sugar

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

7.2.8 Behaviour atom

7.2.8.1 Overview

Concrete syntax

$BA ::= \star G [P] [@P] [' [' E '] ']$	<i>action</i> (BA1)
+ $G P @P ' [' E '] ' \text{start}(N)$	<i>action</i> (BA2)
i	<i>internal action</i> (BA3)
null	<i>successful termination</i> (BA4)
stop	<i>inaction</i> (BA5)

	block	<i>time block</i>	(BA6)
	wait ' (' E ') '	<i>delay</i>	(BA7)
	$P := EA$	<i>assignment</i>	(BA8)
	$P := \text{any } T [' [' E '] ']$	<i>nondeterministic assignment</i>	(BA9)
*	dis <i>DisB</i> enddis	<i>bracketed disabled expression</i>	(BA10)
*	fullsync <i>SyncB</i> endfullsync	<i>bracketed synchronizaton expression</i>	(BA11)
*	conc <i>ConcB</i> endconc	<i>bracketed concurrency expression</i>	(BA12)
*	sel <i>SelB</i> endsel	<i>bracketed choice expression</i>	(BA13)
*	suspend <i>SuspendB</i> endsuspend	<i>bracketed suspend/resume expression</i>	(BA14)
*	inter <i>InterB</i> endinter	<i>bracketed interleaving expression</i>	(BA15)
*	choice $P \square B$ endch	<i>choice over values</i>	(BA16)
+	choice $P \text{ after}(N) \square B$	<i>choice over values</i>	(BA17)
	trap (exception $X [' (' IPL ') '] \text{ is } B \text{ endexn}$)* [exit $[P] \text{ is } B \text{ endexit}$] in B endtrap	<i>trap</i>	(BA18)
	par $[G\#n(, G\#n)^*] \text{ in}$ ' [' $[G(, G)^*] '] ' \rightarrow B(' [' [G(, G)^*] '] ' \rightarrow B)^*$ endpar	<i>general parallel</i>	(BA19)
	par $P \text{ in } N B$ endpar	<i>parallel over values</i>	(BA20)
	var $V : T [:= E] (, V : T [:= E])^* \text{ in}$ B endvar	<i>variable declaration</i>	(BA21)
	hide $G [: T] (, G [: T])^* \text{ in}$ B endhide	<i>gate hiding</i>	(BA22)
	rename ((gate $G [(IPL)] \text{ is } G [P]$) (signal $X [(IPL)] \text{ is } X [E]$)) in B endren	<i>renaming</i>	(BA23)
*	$\Pi \ ' [' [GPL] '] ' '$ ' (' [APL] ') '	<i>process instantiation</i>	(BA24)
	[' [' XPL '] ']		
*	loop B endloop	<i>iteration</i>	(BA25)
*	loop $X [: T] \text{ in}$ B endloop	<i>breakable iteration</i>	(BA26)
*	while E do B endwhile	<i>while iteration</i>	(BA27)
*	for E while E by E do B endfor	<i>for iteration</i>	(BA28)

	case <i>E</i> is <i>BM</i> endcase	<i>case</i> (BA29)
	case '(' (<i>E</i> , <i>E</i>)* ')' is <i>BM</i> endcase	<i>case with tuples</i> (BA30)
★	if <i>E</i> then <i>B</i> (elsif <i>E</i> then <i>B</i>)* [else <i>B</i>] endif	<i>if-then-else</i> (BA31)
	signal <i>X</i> [' (' <i>E</i> ') ']	<i>signalling</i> (BA32)
★	raise <i>X</i> [' (' <i>E</i> ') ']	<i>raising exception</i> (BA33)
★	break [<i>X</i>] [' (' <i>E</i> ') ']	<i>breaking iteration</i> (BA34)
★	' (' <i>B</i> ') '	<i>parenthesized behaviour</i> (BA35)
+	exit (<i>RN</i>)	<i>Successful termination with return value</i> (BA36)

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

7.2.8.2 Action

Concrete syntax

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

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

Abstract syntax

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

7.2.8.3 Bracketed disabled expression

Concrete syntax

dis *DisB* **enddis**

Abstract syntax

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

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

7.2.8.4 Choice over values

Concrete syntax

choice *P* [] *B* **endch**

Abstract syntax

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

7.2.8.5 Process instantiation with abbreviated parameter lists

Concrete syntax

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

Syntax sugar

$$\begin{array}{l} \mathcal{S} \vdash \Pi \Rightarrow \mathbf{gates}([G_1, \dots, G_n]) \\ \mathcal{S} \vdash \Pi \Rightarrow \mathbf{params}([V_1, \dots, V_p]) \\ \mathcal{S} \vdash \Pi \Rightarrow \mathbf{exceptions}([X_1, \dots, X_m]) \end{array} \frac{}{(\Pi \text{ ' [' GPL] ' } (\text{ ' APL ' }) \text{ ' [' XPL] ' }) \stackrel{\text{def}}{=} (\Pi [g_1, \dots, g_n] \text{ ' (' ep}_1, \dots, \text{ep}_p \text{ ') ' } [x_1, \dots, x_m])}$$

where

$$\begin{array}{l} GPL = [G_{k1} \Rightarrow G'_{k1}, \dots, G_{kp} \Rightarrow G'_{kp}, \dots] \quad G_{ki} \in \{G_1, \dots, G_n\} \\ g_i = \begin{cases} G'_i & \text{iff } G_i \Rightarrow G'_i \in GPL \\ G_i & \text{otherwise} \end{cases} \\ APL = [V_{j1} \Rightarrow (E|P)'_{j1}, \dots, V_{js} \Rightarrow (E|P)'_{js}, \dots] \quad V_{ji} \in \{V_1, \dots, V_p\} \\ ep_i = \begin{cases} E'_i & \text{iff } V_i \Rightarrow E'_i \in APL \\ P'_i & \text{iff } V_i \Rightarrow P'_i \in APL \\ V_i & \text{otherwise} \end{cases} \\ XPL = [X_{l1} \Rightarrow X'_{l1}, \dots, X_{lq} \Rightarrow X'_{lq}, \dots] \quad X_{li} \in \{X_1, \dots, X_p\} \\ x_i = \begin{cases} X'_i & \text{iff } X_i \Rightarrow X'_i \in XPL \\ X_i & \text{otherwise} \end{cases} \end{array}$$

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

7.2.8.6 Process instantiation with in/out parameters

Concrete syntax

$$\Pi \text{ ' [' [G(,G)*] '] ' ' (' [APL] ') ' ' [' [' X(X)* '] '] ' }$$

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 7.2.8.5).

Syntax sugar

$$\frac{\mathcal{S} \vdash \Pi \Rightarrow Pos}{(\Pi \text{ ' [' } \vec{G} \text{ '] ' } (\text{ ' AP}_1, \dots, \text{AP}_n \text{ ') ' } [\vec{X}]) \stackrel{\text{def}}{=} \left(\begin{array}{l} \mathbf{trap\ exit\ ?x\ is\ (RP)\ :=\ x} \\ \mathbf{in\ \Pi\ [\vec{G}\ (RE)\ [\vec{X}]} \end{array} \right)}$$

where

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

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

and patterns P_i should be irrefutable.

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

7.2.8.7 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.

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

7.2.8.8 Breakable iteration

Concrete syntax

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

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

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

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

7.2.8.9 while iteration

Concrete syntax

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

Syntax sugar

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

7.2.8.10 for iteration

Concrete syntax

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

Syntax sugar

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

7.2.8.11 Case

Concrete syntax

case *E* **is** *BM* **endcase** **case** '(*E*(,*E*)*)' **is** *BM* **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 infinite recursion.

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

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

7.2.8.12 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** $\stackrel{\text{def}}{=} \mathbf{case} \ E : \mathbf{bool} \ \mathbf{is} \ \mathbf{true} \ \rightarrow \ B_1 \ | \ \mathbf{false} \ \rightarrow \ B_2 \ \mathbf{endcase}$

7.2.8.13 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

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

7.2.8.14 Raising exception**Concrete syntax**

raise X [$'(E)'$]

The default expression is $()$.

Syntax sugar

raise X [$'(E)'$] $\stackrel{\text{def}}{=} \mathbf{signal} X$ [$'(E)'$]; **block**

7.2.8.15 Breaking iteration**Concrete syntax**

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

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

Syntax sugar

break $X \stackrel{\text{def}}{=} \mathbf{raise} X$

break X [$'(E)'$] $\stackrel{\text{def}}{=} \mathbf{raise} X$ [$'(E)'$]

break $\stackrel{\text{def}}{=} \mathbf{raise inner}$

break [$'(E)'$] $\stackrel{\text{def}}{=} \mathbf{raise inner}$ [$'(E)'$]

7.2.8.16 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 $()$.

7.2.9 Type expressions**Concrete syntax**

$T ::= S$
 | **none**
 | **any**

type identifier (T1)

empty type (T2)

universal type (T3)

Abstract syntax

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

7.2.10 Record type expressions**7.2.10.1 Overview****Concrete syntax**

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

Abstract syntax

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

7.2.10.2 Positional**Concrete syntax**

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

$$(T_1, \dots, T_n) \stackrel{\text{def}}{=} (\$1 \Rightarrow T_1, \dots, \$n \Rightarrow T_n)$$
7.2.11 Record value expressions**7.2.11.1 Overview****Concrete syntax**

$RN ::= V \Rightarrow N (, V \Rightarrow N)^*$	<i>named</i> (RN1)
* $N (, N)^*$	<i>positional</i> (RN2)

Abstract syntax

$RN ::= V \Rightarrow N (, V \Rightarrow N)^*$	<i>named</i> (RN1)
+	<i>empty record</i> (RN2)

7.2.11.2 Positional

Concrete syntax

$$N(,N)^*$$

Syntax sugar

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

7.2.12 Gate parameter list

$GPL \star ::= G \Rightarrow G(,G \Rightarrow G)^* [, \dots]$	<i>explicit instantiation</i> (GPL1)
$\star \dots$	<i>implicit instantiation</i> (GPL2)
$ G(,G)^*$	<i>positional instantiation</i> (GPL3)

See Subclause 7.2.8.5, as syntax sugar for explicit instantiation needs context information (formal parameter list of the process to be instantiated).

7.2.13 Actual parameter list

$APL \star ::= V \Rightarrow (E P) (,V \Rightarrow (E P))^* [, \dots]$	<i>explicit instantiation</i> (APL1)
$\star \dots$	<i>implicit instantiation</i> (APL2)
$ (E P)(,(E P))^*$	<i>positional instantiation</i> (APL3)

See Subclause 7.2.8.5, as syntax sugar for explicit instantiation needs context information (formal parameter list of the process to be instantiated).

7.2.14 Exception parameter list

$XPL \star ::= X \Rightarrow X(,X \Rightarrow X)^* [, \dots]$	<i>explicit instantiation</i> (XPL1)
$\star \dots$	<i>implicit instantiation</i> (XPL2)
$ X(,X)^*$	<i>positional instantiation</i> (XPL3)

See Subclause 7.2.8.5, as syntax sugar for explicit instantiation needs context information (formal parameter list of the process to be instantiated).

7.2.15 Record patterns

7.2.15.1 Overview

Concrete syntax

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

Abstract syntax

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

7.2.15.2 Positional**Concrete syntax**

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

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

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

$$IP_1, \dots, IP_n \stackrel{\text{def}}{=} \$1 \Rightarrow IP_1, \dots, \$n \Rightarrow IP_n$$
7.2.17 In parameter**7.2.17.1 Overview****Concrete syntax**

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

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

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

7.2.17.2 Singleton parameter list**Concrete syntax**

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

The default pattern is **any**.

Syntax sugar

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

7.2.17.3 Wildcard**Concrete syntax***etc***Syntax sugar** $\text{etc} \stackrel{\text{def}}{=} \$\text{argv as etc} : \text{etc}$ **7.2.17.4 Record match****Concrete syntax***P as IP***Syntax sugar** $P \text{ as } \$\text{argv as } RP : RT \stackrel{\text{def}}{=} \$\text{argv as } P \text{ as } RP : RT$ **7.2.17.5 Tuple parameter list****Concrete syntax** $[P:]T$ The default pattern is **any**.**Syntax sugar** $(P:T) \stackrel{\text{def}}{=} (\$1 \Rightarrow P:T)$ **7.2.17.6 Parenthesized parameters****Concrete syntax** $\text{' } (\text{' IPL '}) \text{'}$ **Syntax sugar** $\text{' } (\text{' IPL '}) \text{' } \stackrel{\text{def}}{=} (\$1 \Rightarrow IPL)$ **8 E-LOTOS semantics****8.1 Overview**

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

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

- **after**: used to define the semantics of time evolution, see Subclause 10.2.16. Used as abstract syntax.
- **start**: local clock in semantics of actions, see Subclause 10.2.7. 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 10.2.10.

Besides, the following apply:

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

We assume the following types declared:

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

8.2 Static Semantics

8.2.1 Static semantic objects for Base

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

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

where each identifier only has one binding.

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

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

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

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

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

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

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

Note that the grammar for record types overlaps with that of contexts. Whenever RT does not contain any occurrences of **etc**, we shall allow RT to range over contexts (for example in the type rule for sequential composition in Subclause 10.2.6).

8.2.2 Judgements on static semantics for Base

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

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

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

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

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

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

Besides, the following rules apply:

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

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

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

Type expressions Subtyping is a preorder:

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

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

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

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

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

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

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

Subtyping is a preorder:

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

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

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.

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

$$C \vdash N \Rightarrow T$$

$$C \vdash N \Rightarrow T$$

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

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

$$C \vdash RN \Rightarrow RT$$

$$C \vdash RN \Rightarrow RT$$

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

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

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

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

Record of variables

$$C \vdash (RV \Rightarrow RT) \Rightarrow (RT')$$

$$C \vdash (RV \Rightarrow RT) \Rightarrow (RT')$$

$$\frac{C \vdash RT' \equiv RT''}{C \vdash (RV \Rightarrow RT) \Rightarrow (RT')}$$

Behaviour pattern-matching

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

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

General axioms

$$C, J \vdash J$$

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

8.2.3 Extended identifiers

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

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

8.2.4 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:

$\mathcal{B} ::=$	$mod-id \Rightarrow C$	<i>module</i>	(B1)
	$gen-id \Rightarrow (mod-id \Rightarrow C)^* \rightarrow C$	<i>generics</i>	(B2)
	$int-id \Rightarrow C$	<i>interface</i>	(B3)
		<i>empty</i>	(B4)
	\mathcal{B}, \mathcal{B}	<i>disjoint union</i>	(B5)

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

8.2.5 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

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

The context 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 C' .

Module expression

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

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

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

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

Interface body

$$C \vdash i-body, id \Rightarrow C'$$

Record module expression

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

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

$$C \vdash D, id \Rightarrow C'$$

8.2.6 Cycle freedom

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

$$mod-id_0 \rightarrow \dots \rightarrow mod-id_k = mod-id_0 \quad (k > 0)$$

of modules names, where the “ \rightarrow ” relation is the use relation of the “**imports**” construct.

8.2.7 Context morphism

Let C_1 and C_2 be two contexts (for example the bindings of two modules). A *context morphism* $g : C_1 \rightarrow C_2$ is a 3-uple of functions

$$g \stackrel{\text{def}}{=} \langle g_T : C_1.T \rightarrow C_2.T, g_C : C_1.C \rightarrow C_2.C, g_\Pi : C_1.\Pi \rightarrow C_2.\Pi, \rangle$$

such that:

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

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.

8.2.8 Realization

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

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

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

8.2.9 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 C_1 is an *instance* of an interface binding C_2 , written $C_2 \leq C_1$, if there exists a realization morphism ϕ such that $\phi(C_2) = C_1$.

8.2.10 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 C_1 *matches* an interface binding C_2 , written $C_1 \succ C_2$ if exists a module binding C' such that $C_2 \leq C' \subseteq C_1$.

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

8.2.11 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:

$$C \vdash \text{reninst}, id \Rightarrow g$$

where 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:

$$C \vdash \text{reninst}/C' \Rightarrow g$$

where C is the context of the actual parameters, 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

$$\frac{C \vdash S_1 \Rightarrow \text{type}}{C \vdash (S_1 := S_2), id \Rightarrow (S_1 \mapsto \langle id, S_2 \rangle)}$$

The identifier S_1 may be a long or a short identifier, and it shall be defined in 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.

$$\frac{C \vdash C_1 \Rightarrow (RT) \rightarrow S}{C \vdash (C_1 := C_2), id \Rightarrow (C_1 \mapsto \langle id, C_2 \rangle)}$$

The identifier C_1 may be a long or a short identifier, and it shall be defined in 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.

$$\frac{C \vdash \Pi_1 \Rightarrow [(\text{gate}(RT))^*] (RT) [(\text{exn}(RT))^*] \rightarrow \text{exit}(T)}{C \vdash (\Pi_1 := \Pi_2), id \Rightarrow (\Pi_1 \mapsto \langle id, \Pi_2 \rangle)}$$

$$\frac{C \vdash \Pi_1 \Rightarrow [(\text{gate}(RT))^*] (RT) [(\text{exn}(RT))^*] \rightarrow \text{guarded}(T)}{C \vdash (\Pi_1 := \Pi_2), id \Rightarrow (\Pi_1 \mapsto \langle id, \Pi_2 \rangle)}$$

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

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.

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

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 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:

$$g : C \rightarrow g(C)$$

where:

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

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 $id := id'$ shall be distinct.

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

Actualization Rules

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

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 C' , and the second in the actual context C . In the mapping, the extended identifiers are used. The remainder of formal elements given by C' , shall have a realization in the actual context C (implicit instantiation). The realization ϕ may match the morphism given by the explicit instantiation.

$$\frac{C \vdash C_2 \Rightarrow (V'_1 \Rightarrow T'_1, \dots, V'_n \Rightarrow T'_n) \rightarrow S_2}{C \vdash (C_1 := C_2) / C', C_1 \Rightarrow (V_1 \Rightarrow T_1, \dots, V_n \Rightarrow T_n) \rightarrow S_1 \Rightarrow (C_1 \mapsto C_2, T_1 \mapsto T'_1, \dots, T_n \mapsto T'_n, S_1 \mapsto S_2)} [\phi(C') \subseteq C]$$

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

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

$$\Rightarrow (\Pi_1 \mapsto \Pi_2, g_1 \odot g_2 \odot g_3 \odot \phi)$$

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

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

$$\frac{\begin{array}{l} \vdash GL_1 \Rightarrow GL_2 \Rightarrow g \\ \vdash GL'_1 \Rightarrow GL'_2 \Rightarrow g' \end{array}}{\vdash GL_1, GL'_1 \Rightarrow GL_2, GL'_2 \Rightarrow g \odot g'}$$

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

$$\frac{\begin{array}{l} \vdash FPL_1 \Rightarrow FPL_2 \Rightarrow g \\ \vdash FPL'_1 \Rightarrow FPL'_2 \Rightarrow g' \end{array}}{\vdash FPL_1, FPL'_1 \Rightarrow FPL_2, FPL'_2 \Rightarrow g \odot g'}$$

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

$$\frac{\begin{array}{l} \vdash XL_1 \Rightarrow XL_2 \Rightarrow g \\ \vdash XL'_1 \Rightarrow XL'_2 \Rightarrow g' \end{array}}{\vdash XL_1, XL'_1 \Rightarrow XL_2, XL'_2 \Rightarrow g \odot g'}$$

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.

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

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.

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

$$g : C' \rightarrow C$$

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

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

8.3 Untimed dynamic semantics

8.3.1 Untimed dynamic semantic objects for Base

The untimed dynamic semantics² 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 identifiers, and is given by the grammar:

$$\begin{aligned}
\mathcal{E} ::= & S \equiv T && \text{type equivalence } (\mathcal{E}1) \\
& | C \Rightarrow (RT) \rightarrow S && \text{constructor } (\mathcal{E}2) \\
& | \Pi \Rightarrow \lambda[(G(RT))^*] (RP:RT) [(X(RT))^*] \rightarrow B && \text{process identifier } (\mathcal{E}3) \\
& | && \text{trivial } (\mathcal{E}4) \\
& | \mathcal{E}, \mathcal{E} && \text{disjoint union } (\mathcal{E}5)
\end{aligned}$$

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:

²We will use the term “dynamic semantics” for untimed dynamic semantics.

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

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 8.5 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 10.2.6).

8.3.2 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

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

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

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

$$\mathcal{E} \vdash N \Rightarrow T'$$

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

Record value expressions

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

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

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

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

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

Patterns

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

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

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 .

Record of variables

$$\mathcal{E} \vdash (RV \Rightarrow RT) \Rightarrow (RT')$$

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

Behaviour pattern-matching

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

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

8.3.3 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.

$\mathcal{D} ::=$	$mod-id \Rightarrow I$	<i>module</i> (D1)
	$gen-id \Rightarrow (mod-id \Rightarrow I)^* \rightarrow \mathcal{E}$	<i>generics</i> (D2)
	$int-id \Rightarrow I$	<i>interface</i> (D3)
	\mathcal{E}	<i>environment</i> (D4)
		<i>empty</i> (D5)
	\mathcal{D}, \mathcal{D}	<i>disjoint union</i> (D6)

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:

$I ::= S$	<i>type (ex-id)</i> (I1)
C	<i>constructor (ex-id)</i> (I2)
Π	<i>process (ex-id)</i> (I3)
I, I	<i>disjoint union</i> (I4)

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

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

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.

8.3.4 Judgements on untimed dynamic semantics for Modules

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

Specification

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

Top-level declaration

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

Module body

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

Module expression

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

Module formal parameters

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

Interface expressions

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

Interface body

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

Record module expression

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

Declarations

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

8.3.5 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* $g : \mathcal{E}_1 \rightarrow \mathcal{E}_2$ is a 3-uple of functions

$$g \stackrel{\text{def}}{=} \langle g_T : \mathcal{E}_1.S \rightarrow \mathcal{E}_2.S, g_C : \mathcal{E}_1.C \rightarrow \mathcal{E}_2.C, g_\Pi : \mathcal{E}_1.\Pi \rightarrow \mathcal{E}_2.\Pi, \rangle$$

such that:

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

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.

8.3.6 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 $\phi(I_2) = I_1$.

8.3.7 Renaming/Instantiation**Overview**

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

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

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

The definition is given by the static semantics rules.

8.4 Timed dynamic semantics

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

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

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:

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

8.4.1 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

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

8.5 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 reassigned 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 8.3.1) 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 ($x = 2, y = 2$).

$$\text{?x := 1; ?x := x + 1; ?y := x}$$

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

In the following example, there are two possibilities:

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

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

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 substituting operator should not perform a blind substitution of values, as some branch may change any of the bound variable and the new value need to be propagated. Therefore, the substituting operator introduces the assignments of bound variable before the subbehaviours, leaving the real substitution to the sequential ($;$) composition (see above). These assignments are called “dummy” assignments, and it is just a conceptual mechanism to carry bound variables⁴.

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

Let us see an example:

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

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

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

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

In general, we have:

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

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

The following example shows how substitution works with branching operators:

$$(B_1 [> B_2][RN] \stackrel{\text{def}}{=} B_1 [RN] [> B_2 [RN]$$

⁴Tools (as simulator, compilers) may avoid easily these “dummy” assignments.

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:

<pre>(trap exception $X_1(IPL_1)$ is B_1 endexn ⋮ exception $X_n(IPL_n)$ is B_n endexn exit P is B_e endexn in B endtrap)[RN]</pre>	$\stackrel{\text{def}}{=}$	<pre>trap exception $X_1(IPL_1)$ is $B_1[RN'_1]$ endexn ⋮ exception $X_n(IPL_n)$ is $B_n[RN'_n]$ endexn exit P is $B_e[RN'_e]$ endexn in $B[RN]$ endtrap</pre>
---	----------------------------	---

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 some behaviour B . If the exception X 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.

<pre>(case $expr$ is BM_1 ⋮ BM_n endcase)[RN]</pre>	$\stackrel{\text{def}}{=}$	<pre>case $expr[RN]$ is $BM_1[RN]$ ⋮ $BM_n[RN]$ endcase</pre>
--	----------------------------	---

where $(P[E] \rightarrow B)[RN] = P[RN][E[RN']] \rightarrow B[RN']$ and RN' are the bindings in RN except those related with variables bound by P .

Variable declaration

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

where $RN' = RN - RV$.

Loop

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

where $RN' = RN - RN''$ when $B \xrightarrow{\delta(RN'')} \cdot$.

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 $a!1 - a!2 - a!3 - \dots - a!8 - a!9 - c!1 - \dots$.
 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;
```

9 The E-LOTOS modules

9.1 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.

9.1.1 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

$$\frac{\begin{array}{l} \vdash top-dec \Rightarrow \mathcal{B} \\ \mathcal{B} \vdash mod-exp_1, sp-id \Rightarrow C_1 \dots \mathcal{B} \vdash mod-exp_p, sp-id \Rightarrow C_p \\ C_1, \dots, C_p, \\ G_1 \Rightarrow \mathbf{gate}(T_1) \dots G_m \Rightarrow \mathbf{gate}(T_m), \\ X_1 \Rightarrow \mathbf{exn}(T'_1) \dots X_n \Rightarrow \mathbf{exn}(T'_n) \quad \vdash B \Rightarrow \mathbf{exit}(RT) \end{array}}{\vdash (top-dec \quad \mathbf{specification} \quad sp-id \quad \mathbf{imports} \quad mod-exp_1, \dots, mod-exp_p \quad \mathbf{is} \quad \mathbf{gates} \quad G_1 : T_1, \dots, G_m : T_m \quad \mathbf{exceptions} \quad X_1 : T'_1, \dots, X_n : T'_n \quad \mathbf{behaviour} \quad B) \Rightarrow \text{ok}}$$

Dynamic semantic

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

9.2 Top-level declaration

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

9.2.1 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

$$\frac{\begin{array}{l} \mathcal{B} \vdash \text{mod-exp}_1, \text{mod-id} \Rightarrow \mathcal{C}_1 \ \dots \ \mathcal{B} \vdash \text{mod-exp}_m, \text{mod-id} \Rightarrow \mathcal{C}_m \\ \mathcal{B}, \mathcal{C}_1 \odot \dots \odot \mathcal{C}_m \vdash \text{m-body}, \text{mod-id} \Rightarrow \mathcal{C} \end{array}}{\mathcal{B} \vdash (\text{module } \text{mod-id} \text{ imports } \text{mod-exp}_1, \dots, \text{mod-exp}_m \text{ is } \text{m-body}) \\ \Rightarrow (\text{mod-id} \Rightarrow \mathcal{C}, \mathcal{C}_1 \odot \dots \odot \mathcal{C}_m)}$$

with side condition [\mathcal{C} and $\mathcal{C}_1 \odot \dots \odot \mathcal{C}_m$ 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

$$\frac{\begin{array}{l} \mathcal{D} \vdash \text{mod-exp}_1, \text{mod-id} \Rightarrow \mathcal{E}_1 \ \dots \ \mathcal{D} \vdash \text{mod-exp}_m, \text{mod-id} \Rightarrow \mathcal{E}_m \\ \mathcal{D} \odot \mathcal{E}_1 \odot \dots \odot \mathcal{E}_m \vdash \text{m-body}, \text{mod-id} \Rightarrow \mathcal{E} \end{array}}{\mathcal{D} \vdash (\text{module } \text{mod-id} \text{ imports } \text{mod-exp}_1, \dots, \text{mod-exp}_m \text{ is } \text{m-body}) \\ \Rightarrow (\text{mod-id} \Rightarrow \text{Sig}(\mathcal{E}, \mathcal{E}_1 \odot \dots \odot \mathcal{E}_m), \mathcal{E}, \mathcal{E}_1 \odot \dots \odot \mathcal{E}_m)}$$

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.

9.2.2 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

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

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

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*.

Dynamic semantics

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

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

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

9.2.3 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

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

with side condition $[\mathcal{C}$ and $\vec{\mathcal{C}} \odot \mathcal{C}_1 \odot \dots \odot \mathcal{C}_m$ 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

$$\frac{\begin{array}{l} \mathcal{D} \vdash MP \Rightarrow (\vec{mod-id} \Rightarrow \vec{I}) \\ \mathcal{D}, \vec{\mathcal{E}} \vdash mod-exp_1, gen-id \Rightarrow \mathcal{E}_1 \dots \mathcal{D}, \vec{\mathcal{E}} \vdash mod-exp_m, gen-id \Rightarrow \mathcal{E}_m \\ \mathcal{D} \odot \vec{\mathcal{E}} \odot \mathcal{E}_1 \odot \dots \odot \mathcal{E}_m \vdash m-body, gen-id \Rightarrow \mathcal{E} \end{array}}{\mathcal{D} \vdash (\mathbf{generic } gen-id (MP) \mathbf{imports } mod-exp_1, \dots, mod-exp_m \mathbf{is } m-body) \Rightarrow (gen-id \Rightarrow (\vec{mod-id} \Rightarrow \vec{I}) \rightarrow (\mathcal{E}, \vec{\mathcal{E}} \odot \mathcal{E}_1 \odot \dots \odot \mathcal{E}_m))}$$

with side condition $[\mathcal{E}$ and $\vec{\mathcal{E}} \odot \mathcal{E}_1 \odot \dots \odot \mathcal{E}_m$ 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.

9.2.4 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

$$\frac{\begin{array}{l} \mathcal{B} \vdash int-exp \Rightarrow \mathcal{C}' \\ \mathcal{B} \vdash MP \Rightarrow \vec{mod-id} \Rightarrow \vec{\mathcal{C}} \\ \mathcal{B}, \vec{\mathcal{C}} \vdash mod-exp_1, gen-id \Rightarrow \mathcal{C}_1 \dots \mathcal{B}, \vec{\mathcal{C}} \vdash mod-exp_m, gen-id \Rightarrow \mathcal{C}_m \\ \mathcal{B}, \vec{\mathcal{C}} \odot \mathcal{C}_1 \odot \dots \odot \mathcal{C}_m \vdash m-body \Rightarrow \mathcal{C} \end{array}}{\mathcal{B} \vdash (\mathbf{generic } gen-id (MP) : int-exp \mathbf{imports } mod-exp_1, \dots, mod-exp_m \mathbf{is } m-body) \Rightarrow (gen-id \Rightarrow (\vec{mod-id} \Rightarrow \vec{\mathcal{C}}) \rightarrow \mathcal{C}'')}$$

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

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

$$\frac{\begin{array}{l} \mathcal{D} \vdash int-exp, gen-id \Rightarrow I \\ \mathcal{D} \vdash MP \Rightarrow (\vec{mod-id} \Rightarrow \vec{I}) \\ \mathcal{D}, \vec{\mathcal{E}} \vdash mod-exp_1, gen-id \Rightarrow \mathcal{E}_1 \dots \mathcal{D}, \vec{\mathcal{E}} \vdash mod-exp_m, gen-id \Rightarrow \mathcal{E}_m \\ \mathcal{D} \odot \vec{\mathcal{E}} \odot \mathcal{E}_1 \odot \dots \odot \mathcal{E}_m \vdash m-body, gen-id \Rightarrow \mathcal{E} \end{array}}{\mathcal{D} \vdash (\mathbf{generic } gen-id (MP) : int-exp \mathbf{imports } mod-exp_1, \dots, mod-exp_m \mathbf{is } m-body) \Rightarrow (gen-id \Rightarrow (\vec{mod-id} \Rightarrow \vec{I}) \rightarrow (\mathcal{E}, \vec{\mathcal{E}} \odot \mathcal{E}_1 \odot \dots \odot \mathcal{E}_m) \downarrow I')}$$

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

9.2.5 Interface declaration**Abstract syntax**

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

The default import clause is empty.

Static semantics

$$\frac{\mathcal{B} \vdash \text{int-exp}_1, \text{int-id} \Rightarrow C_1 \ \dots \ \mathcal{B} \vdash \text{int-exp}_m, \text{int-id} \Rightarrow C_m \quad C_1 \odot \dots \odot C_m \vdash i\text{-body}, \text{int-id} \Rightarrow C}{\mathcal{B} \vdash (\mathbf{interface} \ \text{int-id} \ \mathbf{imports} \ \text{int-exp}_1, \dots, \text{int-exp}_m \ \mathbf{is} \ i\text{-body}) \Rightarrow (\text{int-id} \Rightarrow C, C_1 \odot \dots \odot C_m)}$$

with side condition [C and $C_1 \odot \dots \odot C_m$ 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.

Dynamic semantics

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

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

9.2.6 Sequential top declaration

Abstract syntax

top-dec top-dec

Static semantics

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

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

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

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.

9.3 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.

9.3.1 Block declaration

Abstract syntax

D

Static semantics

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

Dynamic semantics

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

9.3.2 Module Expression

Abstract syntax

$mod\text{-}exp$

Static semantics

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

The resulting context is given by the module expression check.

Dynamic semantics

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

9.4 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.

9.4.1 Module aliasing not constrained by an interface

Abstract syntax

$mod\text{-}id [\text{renaming } \text{'('} \text{' } \text{reninst } \text{'')}']$

The default renaming is $\text{'('} \text{'}$.

Static semantics

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

The objects declared by $mod\text{-}id$ 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

$$\frac{\mathcal{D} \vdash \text{mod-id} \Rightarrow I \quad \mathcal{D} \downarrow I \vdash \text{reninst}, id \Rightarrow g}{\mathcal{D} \vdash \text{mod-id} \text{ renaming } \langle \langle \text{reninst} \rangle \rangle, id \Rightarrow g(\mathcal{D} \downarrow I)}$$

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.

9.4.2 Module aliasing constrained by an interface

Abstract syntax

$$\text{mod-id} : \text{int-exp} [\text{renaming } \langle \langle \text{reninst} \rangle \rangle]$$

The default renaming is $\langle \langle \rangle \rangle$.

Static semantics

$$\frac{\begin{array}{l} \mathcal{B} \vdash \text{mod-id} \Rightarrow C_1 \\ \mathcal{B} \vdash \text{int-exp}, id \Rightarrow C_2 \\ C_3 \vdash \text{reninst}, id \Rightarrow g \end{array}}{\mathcal{B}, C \vdash \text{mod-id} : \text{int-exp} \text{ renaming } \langle \langle \text{reninst} \rangle \rangle, id \Rightarrow g(C_3)} [C_1 \supseteq C_3 \geq C_2]$$

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

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

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'' .

9.4.3 Generic module actualization not constrained by an interface

Abstract syntax

$$\text{gen-id} \langle \langle RME \rangle \rangle [\text{renaming } \langle \langle \text{reninst} \rangle \rangle]$$

The default renaming is $\langle \langle \rangle \rangle$.

Static semantics

$$\frac{\begin{array}{l} \mathcal{B} \vdash \text{gen-id} \Rightarrow (\overrightarrow{\text{mod-id}} \Rightarrow \overrightarrow{C}) \rightarrow \overrightarrow{C}, C' \\ \mathcal{B} \vdash (RME \Rightarrow (\overrightarrow{\text{mod-id}} \Rightarrow \overrightarrow{C})), id \Rightarrow \phi \\ \phi(\overrightarrow{C}, C'[id/gen-id]) \vdash \text{reninst}, id \Rightarrow g \end{array}}{\mathcal{B}, C \vdash \text{gen-id} \langle \langle RME \rangle \rangle \text{ renaming } \langle \langle \text{reninst} \rangle \rangle, id \Rightarrow g(\phi(\overrightarrow{C}, C'[id/gen-id]))}$$

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 ($\overrightarrow{\text{mod-id}}$) 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.

Dynamic semantics

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

9.4.4 Generic module actualization constrained by an interface

Abstract syntax

$$gen-id \text{ ' (' } RME \text{ ') ' : int-exp [renaming ' (' reninst ') ']$$

The default renaming is ' () ' .

Static semantics

$$\frac{\begin{array}{l} \mathcal{B} \vdash gen-id \Rightarrow (mod-id \Rightarrow \vec{C}) \rightarrow \vec{C}, C_1 \\ \mathcal{B} \vdash (RME \Rightarrow (mod-id \Rightarrow \vec{C})) \Rightarrow \phi \\ \mathcal{B} \vdash int-exp, id \Rightarrow C_2 \\ C_3 \vdash reninst, id \Rightarrow g \end{array}}{\mathcal{B}, C \vdash gen-id \text{ ' (' } RME \text{ ') ' : int-exp renaming ' (' reninst ') ' , id} \Rightarrow g(C_3)}$$

with side condition $[\phi(\vec{C}, C_1[id/gen-id]) \supseteq C_3 \geq C_2]$

Dynamic semantics

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

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

9.4.5 Generic module renaming/instantiation

Abstract syntax

$$gen-id \text{ ' (' } reninst \text{ ') '}$$

The default renaming is ' () ' .

Static semantics

$$\frac{\mathcal{B} \vdash \text{gen-id} \Rightarrow (\vec{\text{mod-id}} \Rightarrow \vec{C}) \rightarrow \vec{C}, \mathcal{C}' \quad \mathcal{C} \vdash \text{reninst} / \vec{C} \Rightarrow g}{\mathcal{B}, \mathcal{C} \vdash \text{gen-id}' (\text{' reninst '}), \text{id} \Rightarrow g(\vec{C}) \odot g(\mathcal{C}'[id/\text{gen-id}]')}$$

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 ($g(\vec{C})$), and to actualize the objects using these formal parameters ($g(\mathcal{C}'[id/\text{gen-id}])$), which are defined in the *gen-id* module. These objects change their source module to *id* (are new objects).

Dynamic semantics

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

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.

9.5 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.

9.5.1 Single**Abstract syntax**

$$\text{mod-id} : \text{int-exp}$$

Static semantics

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

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

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

9.5.2 Disjoint union**Abstract syntax**

$$MP, MP$$

Static semantics

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

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.

9.6 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).

9.6.1 Interface identifier**Abstract syntax**

int-id

Static semantics

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

The context generated is the context of the interface identifier.

Dynamic semantics

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

9.6.2 Simple renaming**Abstract syntax**

$\text{'} [\text{'} \textit{int-id}$ **renaming** $\text{'} (\text{'} \textit{reninst}$ $\text{'}) \text{'} \text{'} \text{'} \text{'} \text{'}$

Static semantics

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

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

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

9.6.3 Explicit body**Abstract syntax**

$$\text{' [' } i\text{-body [renaming ' (' reninst ') '] '] \text{'}$$

The default renaming is '()' .

Static semantics

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

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*.

Dynamic semantics

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

9.7 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.

9.7.1 Type hiding the implementation**Abstract syntax**

$$\text{type } S$$

Static semantics

$$\overline{C \vdash \text{type } S, id \Rightarrow \langle id, S \rangle \Rightarrow \text{type}}$$

Dynamic semantics

$$\overline{\mathcal{D} \vdash \text{type } S, id \Rightarrow \langle id, S \rangle}$$

9.7.2 Type synonym**Abstract syntax**

$$\text{type } S \text{ renames } T$$

Static semantics

$$\frac{C \vdash T \Rightarrow \mathbf{type}}{C \vdash (\mathbf{type } S \text{ renames } T), id \Rightarrow \langle id, S \rangle \Rightarrow \mathbf{type}, S \equiv T}$$

Dynamic semantics

$$\overline{\mathcal{D} \vdash (\mathbf{type } S \text{ renames } T), id \Rightarrow \langle id, S \rangle}$$

9.7.3 Constructed type**Abstract syntax**

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

The default constructor argument type is ().

Static semantics

$$\frac{C, \langle id, S \rangle \Rightarrow \mathbf{type} \vdash (RT_1) \Rightarrow \mathbf{type} \cdots C, \langle id, S \rangle \Rightarrow \mathbf{type} \vdash (RT_n) \Rightarrow \mathbf{type}}{C \vdash (\mathbf{type } S \text{ is } C_1(RT_1) \mid \cdots \mid C_n(RT_n)), id \Rightarrow \langle id, S \rangle \Rightarrow \mathbf{type}, \langle id, C_1 \rangle \Rightarrow (RT_1) \rightarrow S, \dots, \langle id, C_n \rangle \Rightarrow (RT_n) \rightarrow S}$$

Note that the $\langle id, S \rangle \Rightarrow \mathbf{type}$ is assumed in the context to allow recursive instantiation in B .

Dynamic semantics

$$\overline{\mathcal{D} \vdash (\mathbf{type } S \text{ is } C_1(RT_1) \mid \cdots \mid C_n(RT_n)), id \Rightarrow \langle id, S \rangle, \langle id, C_1 \rangle, \dots, \langle id, C_n \rangle}$$

9.7.4 Named record type**Abstract syntax**

$$\mathbf{type } S \text{ is ' (' } RT \text{ ') ' }$$

Static semantics

$$\frac{C \vdash (RT) \Rightarrow \mathbf{type}}{C \vdash (\mathbf{type } S \text{ is ' (' } RT \text{ ') ' }, id \Rightarrow \langle id, S \rangle \Rightarrow \mathbf{type}, S \equiv (RT)}$$

Dynamic semantics

$$\overline{\mathcal{D} \vdash (\mathbf{type } S \text{ is ' (' } RT \text{ ') ' }, id \Rightarrow \langle id, S \rangle}$$

9.7.5 Process declaration**Abstract syntax**

$$\mathbf{process } \Pi \quad \left[\begin{array}{l} \text{' [' } [G[: T]](, G[: T])^* \text{' } \text{' } \text{' } \\ \text{' (' } [V[: T]](, V[: T])^* \text{' } \text{' } \text{' } \\ \text{' : } T \text{' } \\ \text{[raises ' [' } X[: T](, X[: T])^* \text{' } \text{' } \text{' } \end{array} \right]$$

The default gate list is [], the default gate type is (**etc**), 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

$$\begin{array}{c}
C \vdash T_1 \Rightarrow \mathbf{type} \quad \dots \quad C \vdash T_m \Rightarrow \mathbf{type} \\
C \vdash T'_1 \Rightarrow \mathbf{type} \quad \dots \quad C \vdash T'_p \Rightarrow \mathbf{type} \\
C \vdash T''_1 \Rightarrow \mathbf{type} \quad \dots \quad C \vdash T''_n \Rightarrow \mathbf{type} \\
C \vdash T_e \Rightarrow \mathbf{type} \\
\hline
C \vdash (\mathbf{process} \Pi [G_1:T_1, \dots, G_m:T_m] (V_1:T'_1, \dots, V_p:T'_p) \\
\quad :T_e \mathbf{raises} [X_1:T''_1, \dots, X_n:T''_n], id \\
\quad \Rightarrow \langle id, \Pi \rangle \Rightarrow [\mathbf{gate}(T_1), \dots, \mathbf{gate}(T_m)] (V_1 \Rightarrow T'_1, \dots, V_p \Rightarrow T'_p) \\
\quad [\mathbf{exn}(T''_1), \dots, \mathbf{exn}(T''_n)] \rightarrow \mathbf{exit}(T_e))
\end{array}$$

Dynamic semantics

$$\frac{}{\mathcal{D} \vdash (\mathbf{process} \Pi [G_1:T_1, \dots, G_m:T_m] (V_1 \Rightarrow T'_1, \dots, V_p \Rightarrow T'_p) \\
\quad :T_e \mathbf{raises} [X_1:T''_1, \dots, X_n:T''_n], id) \Rightarrow \langle id, \Pi \rangle}$$

9.7.6 Equations**Abstract syntax**

eqns eqn-dec-list

Static semantics

$$\frac{C \vdash \mathit{eqn-dec-list} \Rightarrow \mathbf{ok}}{C \vdash \mathit{eqns eqn-dec-list} \Rightarrow ()}$$

9.7.7 Sequential declaration**Abstract syntax**

i-body i-body

Static Semantics

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

Dynamic Semantics

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

9.8 Record module expression

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

9.8.1 Single**Abstract syntax**

mod-id => mod-exp

Static semantics

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

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 *C*. The realization of this instance relation is given by ϕ .

Dynamic semantics

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

9.8.2 Disjoint union**Abstract syntax**

RME , *RME*

Static semantics

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

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

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

9.8.3 Renaming tuple**Abstract syntax**

reninst , *reninst*

Static semantics

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

9.9 Equation declarations**9.9.1 Equations declaration****Abstract syntax**

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

The default value of *RT* is ().

Static semantics

$$\frac{\begin{array}{l} C \vdash (RT) \Rightarrow \mathbf{type} \\ C, RT \vdash \mathit{eqn-dec}_1 \Rightarrow \mathit{ok} \\ \dots \\ C, RT \vdash \mathit{eqn-dec}_n \Rightarrow \mathit{ok} \end{array}}{C \vdash \mathbf{forall} RT \rightarrow \mathit{eqn-dec}_1 \dots \mathit{eqn-dec}_n \Rightarrow \mathit{ok}}$$

9.9.2 Sequential**Abstract syntax**

eqn-decs eqn-decs

Static semantics

$$\frac{\begin{array}{l} C \vdash \mathit{eqn-decs}_1 \Rightarrow \mathit{ok} \\ C \vdash \mathit{eqn-decs}_2 \Rightarrow \mathit{ok} \end{array}}{C \vdash \mathit{eqn-decs}_1 \mathit{eqn-decs}_2 \Rightarrow \mathit{ok}}$$

9.10 Simple equation declaration**Abstract syntax**

ofsort *T* [**forall** *RT*] *E*

The default value of *RT* is ().

Static semantics

$$\frac{\begin{array}{l} C \vdash T \Rightarrow \mathbf{type} \\ C \vdash (RT) \Rightarrow \mathbf{type} \\ C, RT \vdash E \Rightarrow \mathbf{exit}(\mathbf{bool}) \end{array}}{C \vdash \mathbf{ofsort} T \mathbf{forall} RT \rightarrow E \Rightarrow \mathit{ok}}$$

9.11 Declarations**9.11.1 Type synonym****Abstract syntax**

type *S* **renames** *T*

Static semantics

$$\frac{C \vdash T \Rightarrow \mathbf{type}}{C \vdash (\mathbf{type} S \mathbf{renames} T), id \Rightarrow (\langle id, S \rangle \Rightarrow \mathbf{type}, S \equiv T)}$$

Dynamic semantics

$$\overline{\mathcal{E} \vdash (\mathbf{type} S \mathbf{renames} T) \Rightarrow (S \equiv T)}$$

9.11.2 Type declaration

Abstract syntax

type S **is** $C[(' RT ')] (' | ' C[(' RT ')])^*$

The default constructor argument type is ().

Static semantics

$$\frac{C, \langle id, S \rangle \Rightarrow \mathbf{type} \vdash (RT_1) \Rightarrow \mathbf{type} \cdots C, \langle id, S \rangle \Rightarrow \mathbf{type} \vdash (RT_n) \Rightarrow \mathbf{type}}{C \vdash (\mathbf{type} S \mathbf{is} C_1(RT_1) \mid \cdots \mid C_n(RT_n)), id \Rightarrow (\langle id, S \rangle \Rightarrow \mathbf{type}, \langle id, C_1 \rangle \Rightarrow (RT_1) \rightarrow S, \dots, \langle id, C_n \rangle \Rightarrow (RT_n) \rightarrow S)}$$

Note that the $\langle id, S \rangle \Rightarrow \mathbf{type}$ is assumed in the context to allow recursive instantiation in B .

Dynamic semantics

$$\overline{\mathcal{E} \vdash (\mathbf{type} S \mathbf{is} C_1(RT_1) \mid \cdots \mid C_n(RT_n)) \Rightarrow (C_1 \Rightarrow (RT_1) \rightarrow S, \dots, C_n \Rightarrow (RT_n) \rightarrow S)}$$

9.11.3 Named record type

Abstract syntax

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

Static semantics

$$\frac{C \vdash (RT) \Rightarrow \mathbf{type}}{C \vdash (\mathbf{type} S \mathbf{is} ' (' RT ')'), id \Rightarrow (\langle id, S \rangle \Rightarrow \mathbf{type}, S \equiv (RT))}$$

Dynamic semantics

$$\overline{\mathcal{E} \vdash (\mathbf{type} S \mathbf{is} ' (' RT ')') \Rightarrow (S \equiv (RT))}$$

9.11.4 Process declaration

Abstract syntax

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

The default gate list is [], the default gate type is (**etc**), 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

$$\begin{array}{c}
C \vdash (T_1) \Rightarrow \mathbf{type} \quad \dots \quad C \vdash (T_m) \Rightarrow \mathbf{type} \\
C \vdash (T'_1) \Rightarrow \mathbf{type} \quad \dots \quad C \vdash (T'_p) \Rightarrow \mathbf{type} \\
C \vdash (T''_1) \Rightarrow \mathbf{type} \quad \dots \quad C \vdash (T''_n) \Rightarrow \mathbf{type} \\
C, G_1 \Rightarrow \mathbf{gate}(T_1), \dots, G_m \Rightarrow \mathbf{gate}(T_m), \\
V_1 \Rightarrow T'_1, \dots, V_p \Rightarrow T'_p, \\
X_1 \Rightarrow \mathbf{exn}(T''_1), \dots, X_n \Rightarrow \mathbf{exn}(T''_n), \\
Proc \vdash B \Rightarrow \mathbf{exit}(T) \\
\hline
C \vdash (\mathbf{process} \Pi [G_1 : T_1, \dots, G_m : T_m] (V_1 : T'_1, \dots, V_p : T'_p) \\
: T \mathbf{raises} [X_1 : T''_1, \dots, X_n : T''_n] \mathbf{is} B), id \\
\Rightarrow (\langle id, \Pi \rangle \Rightarrow [\mathbf{gate}(T_1), \dots, \mathbf{gate}(T_m)] \\
(V_1 \Rightarrow T'_1, \dots, V_p \Rightarrow T'_p) \\
[\mathbf{exn}(T''_1), \dots, \mathbf{exn}(T''_n)] \rightarrow \mathbf{exit}(T)) \\
\\
C \vdash (T_1) \Rightarrow \mathbf{type} \quad \dots \quad C \vdash (T_m) \Rightarrow \mathbf{type} \\
C \vdash (T'_1) \Rightarrow \mathbf{type} \quad \dots \quad C \vdash (T'_p) \Rightarrow \mathbf{type} \\
C \vdash (T''_1) \Rightarrow \mathbf{type} \quad \dots \quad C \vdash (T''_n) \Rightarrow \mathbf{type} \\
C, G_1 \Rightarrow \mathbf{gate}(T_1), \dots, G_m \Rightarrow \mathbf{gate}(T_m), \\
V_1 \Rightarrow T'_1, \dots, V_p \Rightarrow T'_p, \\
X_1 \Rightarrow \mathbf{exn}(T''_1), \dots, X_n \Rightarrow \mathbf{exn}(T''_n), \\
Proc \vdash B \Rightarrow \mathbf{guarded}(T) \\
\hline
C \vdash (\mathbf{process} \Pi [G_1 : T_1, \dots, G_m : T_m] (V_1 : T'_1, \dots, V_p : T'_p) \\
: T \mathbf{raises} [X_1 : T''_1, \dots, X_n : T''_n] \mathbf{is} B), id \\
\Rightarrow (\langle id, \Pi \rangle \Rightarrow [\mathbf{gate}(T_1), \dots, \mathbf{gate}(T_m)] \\
(V_1 \Rightarrow T'_1, \dots, V_p \Rightarrow T'_p) \\
[\mathbf{exn}(T''_1), \dots, \mathbf{exn}(T''_n)] \rightarrow \mathbf{guarded}(T))
\end{array}$$

where

$$Proc = (\langle id, \Pi \rangle \Rightarrow [\mathbf{gate}(T_1), \dots, \mathbf{gate}(T_m)] \\
(V_1 \Rightarrow T'_1, \dots, V_p \Rightarrow T'_p) \\
[\mathbf{exn}(T''_1), \dots, \mathbf{exn}(T''_n)] \rightarrow \mathbf{exit}(T))$$

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

Dynamic semantics

$$\frac{E \vdash (\mathbf{process} \Pi [G_1 : T_1, \dots, G_m : T_m] (V_1 : T'_1, \dots, V_p : T'_p) \\
: T \mathbf{raises} [X_1 : T''_1, \dots, X_n : T''_n] \mathbf{is} B), id \\
\Rightarrow (\langle id, \Pi \rangle \Rightarrow \lambda [G_1 : T_1, \dots, G_m : T_m] (V_1 : T'_1, \dots, V_p : T'_p) [X_1 : T''_1, \dots, X_n : T''_n] \rightarrow B)}{}$$

9.11.5 Sequential declarations

Abstract syntax

D D

Static semantics

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

Dynamic semantics

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

10 The E-LOTOS base language**10.1 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.

10.2 Behaviours**10.2.1 Disabling behaviour expression****Abstract syntax**

$$\begin{array}{l} \text{Dis}B ::= BT \ [> \ BT \qquad \qquad \qquad \text{singleton} \quad (\text{DisB1}) \\ \quad \quad | \ BT \ [> \ \text{Dis}B \qquad \qquad \qquad \text{disjoint union} \quad (\text{DisB2}) \end{array}$$

Static semantics

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

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

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.

Untimed dynamic semantics

$$\begin{array}{c}
\frac{\mathcal{E} \vdash B_1 \xrightarrow{G(RN)} B'_1}{\mathcal{E} \vdash B_1 \text{ [> } B_2 \xrightarrow{G(RN)} B'_1 \text{ [> } B_2}} \\
\frac{\mathcal{E} \vdash B_1 \xrightarrow{i(O)} B'_1}{\mathcal{E} \vdash B_1 \text{ [> } B_2 \xrightarrow{i(O)} B'_1 \text{ [> } B_2}} \\
\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN)} B'_1}{\mathcal{E} \vdash B_1 \text{ [> } B_2 \xrightarrow{i(O)} \mathbf{exit} (RN)} \\
\frac{\mathcal{E} \vdash B_1 \xrightarrow{X(RN)} B'_1}{\mathcal{E} \vdash B_1 \text{ [> } B_2 \xrightarrow{i(O)} \mathbf{signal} X (RN) ; (B'_1 \text{ [> } B_2)} \\
\frac{\mathcal{E} \vdash B_2 \xrightarrow{G(RN)} B'_2}{\mathcal{E} \vdash B_1 \text{ [> } B_2 \xrightarrow{G(RN)} B'_2}} \\
\frac{\mathcal{E} \vdash B_2 \xrightarrow{i(O)} B'_2}{\mathcal{E} \vdash B_1 \text{ [> } B_2 \xrightarrow{i(O)} B'_2}} \\
\frac{\mathcal{E} \vdash B_2 \xrightarrow{X(RN)} B'_2}{\mathcal{E} \vdash B_1 \text{ [> } B_2 \xrightarrow{i(O)} \mathbf{signal} X (RN) ; B'_2}
\end{array}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B'_2}{\mathcal{E} \vdash B_1 \text{ [> } B_2 \xrightarrow{\varepsilon(d)} B'_1 \text{ [> } B'_2}$$

10.2.2 Synchronization behaviour expression**Abstract syntax**

$$\begin{array}{ll}
\mathit{Sync}B ::= BT \parallel BT & \textit{synchronization} \quad (\text{SyncB1}) \\
\quad \quad \quad | BT \parallel \mathit{Sync}B & \textit{synchronization} \quad (\text{SyncB2})
\end{array}$$

Static semantics

$$\begin{array}{c}
\frac{C \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \quad C \vdash B_2 \Rightarrow \mathbf{exit}(RT_2)}{C \vdash B_1 \parallel B_2 \Rightarrow \mathbf{exit}(RT_1, RT_2)} \text{ [} RT_1 \text{ and } RT_2 \text{ have disjoint fields]} \\
\frac{C \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \quad C \vdash B_2 \Rightarrow \mathbf{guarded}(RT_2)}{C \vdash B_1 \parallel B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2)} \text{ [} RT_1 \text{ and } RT_2 \text{ have disjoint fields]} \\
\frac{C \vdash B_1 \Rightarrow \mathbf{guarded}(RT_1) \quad C \vdash B_2 \Rightarrow \mathbf{exit}(RT_2)}{C \vdash B_1 \parallel B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2)} \text{ [} RT_1 \text{ and } RT_2 \text{ have disjoint fields]}
\end{array}$$

Untimed dynamic semantics

$$\begin{array}{c}
\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}(O)} B'_1}{\mathcal{E} \vdash B_1 \parallel B_2 \xrightarrow{\mathbf{i}(O)} B'_1 \parallel B_2} \\
\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}(O)} B'_2}{\mathcal{E} \vdash B_1 \parallel B_2 \xrightarrow{\mathbf{i}(O)} B_1 \parallel B'_2} \\
\frac{\mathcal{E} \vdash B_1 \xrightarrow{G(RN)} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{G(RN)} B'_2}{\mathcal{E} \vdash B_1 \parallel B_2 \xrightarrow{G(RN)} B'_1 \parallel B'_2} \\
\frac{\mathcal{E} \vdash B_1 \xrightarrow{X(RN)} B'_1}{\mathcal{E} \vdash B_1 \parallel B_2 \xrightarrow{\mathbf{i}(O)} \mathbf{signal} X (RN); (B'_1 \parallel B_2)} \\
\frac{\mathcal{E} \vdash B_2 \xrightarrow{X(RN)} B'_2}{\mathcal{E} \vdash B_1 \parallel B_2 \xrightarrow{\mathbf{i}(O)} \mathbf{signal} X (RN); (B_1 \parallel B'_2)} \\
\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{\delta(RN_2)} B'_2}{\mathcal{E} \vdash B_1 \parallel B_2 \xrightarrow{\delta(RN_1, RN_2)} B'_1 \parallel B'_2}
\end{array}$$

Timed dynamic semantics

$$\begin{array}{c}
\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B'_2}{\mathcal{E} \vdash B_1 \parallel B_2 \xrightarrow{\varepsilon(d)} B'_1 \parallel B'_2} \\
\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN@d)} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d+d')} B'_2}{\mathcal{E} \vdash B_1 \parallel B_2 \xrightarrow{\varepsilon(d+d')} \mathbf{exit}(RN) \parallel B'_2} [0 < d'] \\
\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d+d')} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{\delta(RN@d)} B'_2}{\mathcal{E} \vdash B_1 \parallel B_2 \xrightarrow{\varepsilon(d+d')} B'_1 \parallel \mathbf{exit}(RN)} [0 < d']
\end{array}$$

10.2.3 Concurrency behaviour expression

Abstract syntax

$$\begin{array}{ll}
\mathit{Conc}B ::= BT \parallel [[G(,G)^*]] \parallel BT & \textit{concurrency} \quad (\text{ConcB1}) \\
\quad \mid BT \parallel [[G(,G)^*]] \parallel \mathit{Conc}B & \textit{concurrency} \quad (\text{ConcB2})
\end{array}$$

Static semantics

$$\frac{C \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \quad C \vdash B_2 \Rightarrow \mathbf{exit}(RT_2)}{C \vdash G_1 \Rightarrow \mathbf{gate}(RT'_1) \cdots C \vdash G_n \Rightarrow \mathbf{gate}(RT'_n)} [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

$$\frac{C \vdash B_1 \mid [G_1, \dots, G_n] \mid B_2 \Rightarrow \mathbf{exit}(RT_1, RT_2)}{C \vdash B_1 \mid [G_1, \dots, G_n] \mid B_2 \Rightarrow \mathbf{exit}(RT_1, RT_2)}$$

$$\frac{C \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \quad C \vdash B_2 \Rightarrow \mathbf{guarded}(RT_2)}{C \vdash G_1 \Rightarrow \mathbf{gate}(RT'_1) \cdots C \vdash G_n \Rightarrow \mathbf{gate}(RT'_n)} [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

$$\frac{C \vdash B_1 \mid [G_1, \dots, G_n] \mid B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2)}{C \vdash B_1 \mid [G_1, \dots, G_n] \mid B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2)}$$

$$\frac{C \vdash B_1 \Rightarrow \mathbf{guarded}(RT_1) \quad C \vdash B_2 \Rightarrow \mathbf{exit}(RT_2)}{C \vdash G_1 \Rightarrow \mathbf{gate}(RT'_1) \cdots C \vdash G_n \Rightarrow \mathbf{gate}(RT'_n)} [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

$$\frac{C \vdash B_1 \mid [G_1, \dots, G_n] \mid B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2)}{C \vdash B_1 \mid [G_1, \dots, G_n] \mid B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2)}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\mathbf{i}(\circ)} B'_1}{\mathcal{E} \vdash B_1 \mid [\vec{G}] \mid B_2 \xrightarrow{\mathbf{i}(\circ)} B'_1 \mid [\vec{G}] \mid B_2}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\mathbf{i}(\circ)} B'_2}{\mathcal{E} \vdash B_1 \mid [\vec{G}] \mid B_2 \xrightarrow{\mathbf{i}(\circ)} B_1 \mid [\vec{G}] \mid B'_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G(RN)} B'_1}{\mathcal{E} \vdash B_1 \mid [\vec{G}] \mid B_2 \xrightarrow{G(RN)} B'_1 \mid [\vec{G}] \mid B_2} [G \notin \vec{G}]$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{G(RN)} B'_2}{\mathcal{E} \vdash B_1 \mid [\vec{G}] \mid B_2 \xrightarrow{G(RN)} B_1 \mid [\vec{G}] \mid B'_2} [G \notin \vec{G}]$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{G_i(RN)} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{G_i(RN)} B'_2}{\mathcal{E} \vdash B_1 \mid [\vec{G}] \mid B_2 \xrightarrow{G_i(RN)} B'_1 \mid [\vec{G}] \mid B'_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{X(RN)} B'_1}{\mathcal{E} \vdash B_1 \mid [\vec{G}] \mid B_2 \xrightarrow{\mathbf{i}(\circ)} \mathbf{signal} X (RN); (B'_1 \mid [\vec{G}] \mid B_2)}$$

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{X(RN)} B'_2}{\mathcal{E} \vdash B_1 \mid [\vec{G}] \mid B_2 \xrightarrow{\mathbf{i}(\circ)} \mathbf{signal} X (RN); (B_1 \mid [\vec{G}] \mid B'_2)}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1)} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{\delta(RN_2)} B'_2}{\mathcal{E} \vdash B_1 \mid [\vec{G}] \mid B_2 \xrightarrow{\delta(RN_1, RN_2)} B'_1 \mid [\vec{G}] \mid B'_2}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B'_2}{\mathcal{E} \vdash B_1 \mid [\vec{G}] \mid B_2 \xrightarrow{\varepsilon(d)} B'_1 \mid [\vec{G}] \mid B'_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN@d)} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d+d')} B'_2}{\mathcal{E} \vdash B_1 \mid [\vec{G}] \mid B_2 \xrightarrow{\varepsilon(d+d')} \mathbf{exit}(RN) \mid [\vec{G}] \mid B'_2} [0 < d']$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d+d')} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{\delta(RN@d)} B'_2}{\mathcal{E} \vdash B_1 \mid [\vec{G}] \mid B_2 \xrightarrow{\varepsilon(d+d')} B'_1 \mid [\vec{G}] \mid \mathbf{exit}(RN)} [0 < d']$$

10.2.4 Selection behaviour expression

Abstract syntax

$$\begin{aligned} SelB & ::= BT \sqcap BT && \text{choice (SelB1)} \\ & | BT \sqcap SelB && \text{choice (SelB2)} \end{aligned}$$

Static semantics

$$\frac{C, RT_1, RT_2 \vdash B_1 \Rightarrow \mathbf{guarded}(RT_1, RT) \quad C, RT_1, RT_2 \vdash B_2 \Rightarrow \mathbf{guarded}(RT_2, RT)}{C, RT_1, RT_2 \vdash B_1 \sqcap B_2 \Rightarrow \mathbf{guarded}(RT_1, RT_2, RT)}$$

Untimed dynamic semantics

$$\begin{aligned} & \frac{\mathcal{E} \vdash B_1 \xrightarrow{G(RN)} B'_1}{\mathcal{E} \vdash B_1 \sqcap B_2 \xrightarrow{G(RN)} B'_1} \\ & \frac{\mathcal{E} \vdash B_2 \xrightarrow{G(RN)} B'_2}{\mathcal{E} \vdash B_1 \sqcap B_2 \xrightarrow{G(RN)} B'_2} \\ & \frac{\mathcal{E} \vdash B_1 \xrightarrow{iO} B'_1}{\mathcal{E} \vdash B_1 \sqcap B_2 \xrightarrow{iO} B'_1} \\ & \frac{\mathcal{E} \vdash B_2 \xrightarrow{iO} B'_2}{\mathcal{E} \vdash B_1 \sqcap B_2 \xrightarrow{iO} B'_2} \\ & \frac{\mathcal{E} \vdash B_1 \xrightarrow{X(RN)} B'_1}{\mathcal{E} \vdash B_1 \sqcap B_2 \xrightarrow{iO} \mathbf{signal} X (RN); B'_1} \\ & \frac{\mathcal{E} \vdash B_2 \xrightarrow{X(RN)} B'_2}{\mathcal{E} \vdash B_1 \sqcap B_2 \xrightarrow{iO} \mathbf{signal} X (RN); B'_2} \end{aligned}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B'_2}{\mathcal{E} \vdash B_1 \sqcap B_2 \xrightarrow{\varepsilon(d)} B'_1 \sqcap B'_2}$$

10.2.5 Suspend/Resume behaviour expression

Abstract syntax

$$B [X > B$$

Static semantics

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

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

Untimed dynamic semantics

$$\begin{array}{c}
\frac{\mathcal{E} \vdash B_1 \xrightarrow{G(RN)} B'_1}{\mathcal{E} \vdash B_1 [X > B_2 \xrightarrow{G(RN)} B'_1 [X > B_2]} \\
\frac{\mathcal{E} \vdash B_1 \xrightarrow{i(O)} B'_1}{\mathcal{E} \vdash B_1 [X > B_2 \xrightarrow{i(O)} B'_1 [X > B_2]} \\
\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN)} B'_1}{\mathcal{E} \vdash B_1 [X > B_2 \xrightarrow{i(O)} \mathbf{exit} (RN)} \\
\frac{\mathcal{E} \vdash B_1 \xrightarrow{X'(RN)} B'_1}{\mathcal{E} \vdash B_1 [X > B_2 \xrightarrow{i(O)} \mathbf{signal} X' (RN) ; (B'_1 [X > B_2]} \\
\frac{\mathcal{E} \vdash B_2 \xrightarrow{G(RN)} B'_2}{\mathcal{E} \vdash B_1 [X > B_2 \xrightarrow{G(RN)} \mathbf{trap exception} X () \mathbf{is} B_1 [X > B_2 \mathbf{in} B'_2]} \\
\frac{\mathcal{E} \vdash B_2 \xrightarrow{i(O)} B'_2}{\mathcal{E} \vdash B_1 [X > B_2 \xrightarrow{i(O)} \mathbf{trap exception} X () \mathbf{is} B_1 [X > B_2 \mathbf{in} B'_2]} \\
\frac{\mathcal{E} \vdash B_2 \xrightarrow{X'(RN)} B'_2}{\mathcal{E} \vdash B_1 [X > B_2 \xrightarrow{i(O)} \mathbf{signal} X' (RN) ; \mathbf{trap exception} X () \mathbf{is} B_1 [X > B_2 \mathbf{in} B'_2} [X' \neq X]
\end{array}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B'_1 \quad \mathcal{E} \vdash B_2 \xrightarrow{\varepsilon(d)} B'_2}{\mathcal{E} \vdash B_1 [X > B_2 \xrightarrow{\varepsilon(d)} B'_1 [X > B'_2]}$$

10.2.6 Sequential composition**Abstract syntax**

$$BA ; BT$$

Static semantics

$$\begin{array}{c}
\frac{C \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \quad C; RT_1 \vdash B_2 \Rightarrow \mathbf{exit}(RT_2)}{C \vdash B_1 ; B_2 \Rightarrow \mathbf{exit}(RT_1; RT_2)} \\
\frac{C \vdash B_1 \Rightarrow \mathbf{exit}(RT_1) \quad C; RT_1 \vdash B_2 \Rightarrow \mathbf{guarded}(RT_2)}{C \vdash B_1 ; B_2 \Rightarrow \mathbf{guarded}(RT_1; RT_2)} \\
\frac{C \vdash B_1 \Rightarrow \mathbf{guarded}(RT_1) \quad C; RT_1 \vdash B_2 \Rightarrow \mathbf{exit}(RT_2)}{C \vdash B_1 ; B_2 \Rightarrow \mathbf{guarded}(RT_1; RT_2)} [none \neq RT_1]
\end{array}$$

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

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

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B'_1}{\mathcal{E} \vdash B_1 ; B_2 \xrightarrow{\varepsilon(d)} B'_1 ; B_2}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN_1@d_1)} B'_1 \quad \mathcal{E} \vdash B_2[RN_1] \xrightarrow{\varepsilon(d_2)} B'_2}{\mathcal{E} \vdash B_1 ; B_2 \xrightarrow{\varepsilon(d_1+d_2)} \mathbf{exit}(RN_1) ; B'_2}$$

10.2.7 Action**Abstract syntax**

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

Static semantics

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

Untimed dynamic semantics

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

Timed dynamic semantics

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

10.2.8 Internal action

Abstract syntax

i

Static semantics

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

Untimed dynamic semantics

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

Timed dynamic semantics None.

10.2.9 Successful termination without values

Abstract syntax

null

Static semantics

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

Untimed dynamic semantics

$$\frac{}{\mathcal{E} \vdash \mathbf{null} \xrightarrow{\delta\emptyset} \mathbf{block}}$$

Timed dynamic semantics None.

10.2.10 Successful 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

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

Untimed dynamic semantics

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

Timed dynamic semantics None.

10.2.11 Inaction**Abstract syntax**

stop

Static semantics

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

Untimed dynamic semantics None.

Timed dynamic semantics

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

10.2.12 Time block**Abstract syntax**

block

Static semantics

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

Untimed dynamic semantics None.

Timed dynamic semantics None.

10.2.13 Delay**Abstract syntax**

wait '(' E ')'

Static semantics

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

Untimed dynamic semantics

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

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

Timed dynamic semantics

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

10.2.14 Assignment**Abstract syntax**

$$P := EA$$

The pattern must be irrefutable.

Static semantics

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

Untimed dynamic semantics

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

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

Timed dynamic semantics None.

10.2.15 Nondeterministic Assignment**Abstract syntax**

$$P := \mathbf{any} T [\text{' [' E '] ' }]$$

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

Static semantics

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

Untimed dynamic semantics

$$\begin{array}{l}
\mathcal{E} \vdash N \Rightarrow T \\
\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\
\mathcal{E} \vdash E[RN] \xrightarrow{\delta(\text{true})} E' \\
\hline
\mathcal{E} \vdash P := \mathbf{any} T \text{ , } [\text{ , } E \text{ ' }] \text{ , } \xrightarrow{\mathbf{i}0} \mathbf{exit}(RN)
\end{array}$$

Timed dynamic semantics None.

10.2.16 Choice over values**Abstract syntax**

choice P **after** (N) \square B

Static semantics

$$\begin{array}{l}
C \vdash (P \Rightarrow \mathbf{any}) \Rightarrow (RT) \\
C \vdash N \Rightarrow \text{time} \\
C; RT \vdash B \Rightarrow \mathbf{guarded}(RT') \\
\hline
C \vdash \mathbf{choice} P \mathbf{after} (N) \square B \Rightarrow \mathbf{guarded}(RT')
\end{array}$$

Untimed dynamic semantics

$$\begin{array}{l}
\mathcal{E} \vdash (N \Rightarrow \mathbf{any}) \\
\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\
\mathcal{E} \vdash B[RN] \xrightarrow{G(RN'@d)} B' \\
\hline
\mathcal{E} \vdash \mathbf{choice} P \mathbf{after} (d) \square B \xrightarrow{G(RN')} B' \\
\\
\mathcal{E} \vdash (N \Rightarrow \mathbf{any}) \\
\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\
\mathcal{E} \vdash B[RN] \xrightarrow{\mathbf{i}(@d)} B' \\
\hline
\mathcal{E} \vdash \mathbf{choice} P \mathbf{after} (d) \square B \xrightarrow{\mathbf{i}0} B' \\
\\
\mathcal{E} \vdash (N \Rightarrow \mathbf{any}) \\
\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\
\mathcal{E} \vdash B[RN] \xrightarrow{X(RN'@d)} B' \\
\hline
\mathcal{E} \vdash \mathbf{choice} P \mathbf{after} (d) \square B \xrightarrow{\mathbf{i}0} \mathbf{signal} X (RN') ; B'
\end{array}$$

Note: This operator is the only place where the timed semantics is used in the untimed semantics, thus breaking the “classical” stratification [5, 25] used to prove the consistency of the semantics. However, as shown in [16] 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

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

10.2.17 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 C stands for $C', RT_1''' \odot \dots \odot RT_n''' \odot RT_e''' \odot RT''''$. 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 produce them. The resulting behaviour binds $RT_1''' \odot \dots \odot RT_n''' \odot RT_e''' \odot RT''''$. 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 reassigned freely in any of the subbehaviours.

$$\begin{array}{c}
 C' \vdash (RT_1) \Rightarrow \mathbf{type} \quad \dots \quad C' \vdash (RT_n) \Rightarrow \mathbf{type} \\
 C' \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \quad \dots \quad C' \vdash (RP_n \Rightarrow RT_n) \Rightarrow (RT_n') \\
 C; RT_1' \vdash B_1 \Rightarrow \mathbf{exit}(RT_1''', RT) \quad \dots \quad C; RT_n' \vdash B_n \Rightarrow \mathbf{exit}(RT_n''', RT) \\
 C; X_1 \Rightarrow \mathbf{exn}(RT_1), \dots, X_n \Rightarrow \mathbf{exn}(RT_n) \vdash B \Rightarrow \mathbf{exit}(RT''', RT) \\
 \hline
 C \vdash (\mathbf{trap} \text{ exception } X_1 (RP_1 : RT_1) \text{ is } B_1 \\
 \dots \text{ exception } X_n (RP_n : RT_n) \text{ is } B_n \text{ in } B) \\
 \Rightarrow \mathbf{exit}(RT_1''' \odot \dots \odot RT_n''' \odot RT_e''' \odot RT''', RT) \\
 \\
 C' \vdash (RT_1) \Rightarrow \mathbf{type} \quad \dots \quad C' \vdash (RT_n) \Rightarrow \mathbf{type} \\
 C' \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \quad \dots \quad C' \vdash (RP_n \Rightarrow RT_n) \Rightarrow (RT_n') \\
 C; RT_1' \vdash B_1 \Rightarrow \mathbf{guarded}(RT_1''', RT) \quad \dots \quad C; RT_n' \vdash B_n \Rightarrow \mathbf{guarded}(RT_n''', RT) \\
 C; X_1 \Rightarrow \mathbf{exn}(RT_1), \dots, X_n \Rightarrow \mathbf{exn}(RT_n) \vdash B \Rightarrow \mathbf{guarded}(RT''', RT) \\
 \hline
 C \vdash (\mathbf{trap} \text{ exception } X_1 (RP_1 : RT_1) \text{ is } B_1 \\
 \dots \text{ exception } X_n (RP_n : RT_n) \text{ is } B_n \text{ in } B) \\
 \Rightarrow \mathbf{guarded}(RT_1''' \odot \dots \odot RT_n''' \odot RT_e''' \odot RT''', RT) \\
 \\
 C' \vdash (RT_1) \Rightarrow \mathbf{type} \quad \dots \quad C' \vdash (RT_n) \Rightarrow \mathbf{type} \\
 C' \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \quad \dots \quad C' \vdash (RP_n \Rightarrow RT_n) \Rightarrow (RT_n') \\
 C; RT_1' \vdash B_1 \Rightarrow \mathbf{exit}(RT_1''', RT) \quad \dots \quad C; RT_n' \vdash B_n \Rightarrow \mathbf{exit}(RT_n''', RT) \\
 C; X_1 \Rightarrow \mathbf{exn}(RT_1), \dots, X_n \Rightarrow \mathbf{exn}(RT_n) \vdash B \Rightarrow \mathbf{exit}(RT''', RT') \\
 C \vdash (P \Rightarrow (RT')) \Rightarrow (RT'') \\
 C; RT'' \vdash B' \Rightarrow \mathbf{exit}(RT_e''', RT) \\
 \hline
 C \vdash (\mathbf{trap} \text{ exception } X_1 (RP_1 : RT_1) \text{ is } B_1 \\
 \dots \text{ exception } X_n (RP_n : RT_n) \text{ is } B_n \\
 \mathbf{exit } P \text{ is } B' \text{ in } B) \\
 \Rightarrow \mathbf{exit}(RT_1''' \odot \dots \odot RT_n''' \odot RT_e''' \odot RT''', RT) \\
 \\
 C' \vdash (RT_1) \Rightarrow \mathbf{type} \quad \dots \quad C' \vdash (RT_n) \Rightarrow \mathbf{type} \\
 C' \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1') \quad \dots \quad C' \vdash (RP_n \Rightarrow RT_n) \Rightarrow (RT_n') \\
 C; RT_1' \vdash B_1 \Rightarrow \mathbf{guarded}(RT_1''', RT) \quad \dots \quad C; RT_n' \vdash B_n \Rightarrow \mathbf{guarded}(RT_n''', RT) \\
 C; X_1 \Rightarrow \mathbf{exn}(RT_1), \dots, X_n \Rightarrow \mathbf{exn}(RT_n) \vdash B \Rightarrow \mathbf{exit}(RT''', RT') \\
 C \vdash (P \Rightarrow (RT')) \Rightarrow (RT'') \\
 C; RT'' \vdash B' \Rightarrow \mathbf{guarded}(RT_e''', RT) \\
 \hline
 C \vdash (\mathbf{trap} \text{ exception } X_1 (RP_1 : RT_1) \text{ is } B_1 \\
 \dots \text{ exception } X_n (RP_n : RT_n) \text{ is } B_n \\
 \mathbf{exit } P \text{ is } B' \text{ in } B) \\
 \Rightarrow \mathbf{guarded}(RT_1''' \odot \dots \odot RT_n''' \odot RT_e''' \odot RT''', RT)
 \end{array}$$

$$\begin{array}{c}
C' \vdash (RT_1) \Rightarrow \mathbf{type} \quad \dots \quad C' \vdash (RT_n) \Rightarrow \mathbf{type} \\
C' \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT'_1) \quad \dots \quad C' \vdash (RP_n \Rightarrow RT_n) \Rightarrow (RT'_n) \\
C; RT'_1 \vdash B_1 \Rightarrow \mathbf{guarded}(RT''_1, RT) \quad \dots \quad C; RT'_n \vdash B_n \Rightarrow \mathbf{guarded}(RT''_n, RT) \\
C; X_1 \Rightarrow \mathbf{exn}(RT_1), \dots, X_n \Rightarrow \mathbf{exn}(RT_n) \vdash B \Rightarrow \mathbf{guarded}(RT''_e, RT') \\
C \vdash (P \Rightarrow (RT')) \Rightarrow (RT'') \\
C; RT'' \vdash B' \Rightarrow \mathbf{exit}(RT''_e, RT) \\
\hline
C \vdash (\mathbf{trap\ exception} \ X_1 (RP_1 : RT_1) \ \mathbf{is} \ B_1 \\
\quad \dots \ \mathbf{exception} \ X_n (RP_n : RT_n) \ \mathbf{is} \ B_n \\
\quad \mathbf{exit} \ P \ \mathbf{is} \ B' \ \mathbf{in} \ B) \\
\Rightarrow \mathbf{guarded}(RT''_1 \odot \dots \odot RT''_n \odot RT''_e \odot RT'', RT)
\end{array}$$

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

$$\begin{array}{c}
\frac{\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B'}{\mathcal{E} \vdash (\mathbf{trap} \ \vec{X} \ (\vec{RP} : \vec{RT}) \ \mathbf{is} \ \vec{B} \ \mathbf{in} \ B) \xrightarrow{\mu(RN)} (\mathbf{trap} \ \vec{X} \ (\vec{RP} : \vec{RT}) \ \mathbf{is} \ \vec{B} \ \mathbf{in} \ B)} \left[\mu \notin \vec{X} \right]} \\
\\
\frac{\mathcal{E} \vdash B \xrightarrow{\mu_i(RN)} B' \quad \mathcal{E} \vdash ((RP_i) \Rightarrow (RN)) \Rightarrow (RN') \quad \mathcal{E} \vdash B_i[RN'] \xrightarrow{\mu(RN'')} B'_i}{\mathcal{E} \vdash (\mathbf{trap} \ \vec{X} \ (\vec{RP} : \vec{RT}) \ \mathbf{is} \ \vec{B} \ \mathbf{in} \ B) \xrightarrow{\mu(RN'')} B'_i}
\end{array}$$

Timed dynamic semantics

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

10.2.18 General parallel

Abstract syntax

$$\mathbf{par} \ [G\#n(, G\#n)^*] \ \mathbf{in} \ \text{' [' [G(, G)^*] '] ' -> B(|| \text{' [' [G(, G)^*] '] ' -> B)^*}$$

Static semantics

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

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

where \vec{G}_i is an abbreviation of G_{i1}, \dots, G_{in_i} and $C \vdash \vec{G}_i \Rightarrow \mathbf{gate}(\vec{RT}_i)$ is an abbreviation for $C \vdash \vec{G}_{i1} \Rightarrow \mathbf{gate}(RT_{i1}), \dots, C \vdash \vec{G}_{in_i} \Rightarrow \mathbf{gate}(RT_{in_i})$.

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

Untimed dynamic semantics

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

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

$$\frac{\mathcal{E} \vdash B_j \xrightarrow{G(RN)} B'_j}{\mathcal{E} \vdash (\mathbf{par} \ G_1 \# N_1, \dots, G_p \# N_p} \\
\quad || [\vec{G}_1] \rightarrow B_1 \quad \dots \quad || [\vec{G}_j] \rightarrow B_j \quad \dots \quad || [\vec{G}_m] \rightarrow B_m) \\
\quad \xrightarrow{i(O)} (\mathbf{par} \ G_1 \# N_1, \dots, G_p \# N_p} \\
\quad || [\vec{G}_1] \rightarrow B_1 \quad \dots \quad || [\vec{G}_j] \rightarrow B'_j \quad \dots \quad || [\vec{G}_m] \rightarrow B_m)}
\left[G \notin \vec{G}_j \right]$$

$$\frac{\mathcal{E} \vdash B_j \xrightarrow{i(O)} B'_j}{\mathcal{E} \vdash (\mathbf{par} \ G_1 \# N_1, \dots, G_p \# N_p} \\
\quad || [\vec{G}_1] \rightarrow B_1 \quad \dots \quad || [\vec{G}_j] \rightarrow B_j \quad \dots \quad || [\vec{G}_m] \rightarrow B_m) \\
\quad \xrightarrow{i(O)} (\mathbf{par} \ G_1 \# N_1, \dots, G_p \# N_p} \\
\quad || [\vec{G}_1] \rightarrow B_1 \quad \dots \quad || [\vec{G}_j] \rightarrow B'_j \quad \dots \quad || [\vec{G}_m] \rightarrow B_m)}$$

$$\frac{\mathcal{E} \vdash B_j \xrightarrow{X(RN)} B'_j}{\mathcal{E} \vdash (\mathbf{par} \ G_1 \# N_1, \dots, G_p \# N_p} \\
\quad || [\vec{G}_1] \rightarrow B_1 \quad \dots \quad || [\vec{G}_j] \rightarrow B_j \quad \dots \quad || [\vec{G}_m] \rightarrow B_m) \\
\quad \xrightarrow{i(O)} \mathbf{signal} \ X \ (RN) ; (\mathbf{par} \ G_1 \# N_1, \dots, G_p \# N_p} \\
\quad || [\vec{G}_1] \rightarrow B_1 \quad \dots \quad || [\vec{G}_j] \rightarrow B'_j \quad \dots \quad || [\vec{G}_m] \rightarrow B_m)}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\delta(RN)} B'_1 \dots \mathcal{E} \vdash B_m \xrightarrow{\delta(RN)} B'_m}{\mathcal{E} \vdash (\mathbf{par} G_1 \# N_1, \dots, G_p \# N_p [\vec{G}_1] \rightarrow B_1 \dots \parallel [\vec{G}_m] \rightarrow B_m) \xrightarrow{\delta(RN)} (\mathbf{par} G_1 \# N_1, \dots, G_p \# N_p [\vec{G}_1] \rightarrow B'_1 \dots \parallel [\vec{G}_m] \rightarrow B'_m)}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d)} B'_1 \dots \mathcal{E} \vdash B_m \xrightarrow{\varepsilon(d)} B'_m}{\mathcal{E} \vdash (\mathbf{par} G_1 \# N_1, \dots, G_p \# N_p [\vec{G}_1] \rightarrow B_1 \dots \parallel [\vec{G}_m] \rightarrow B_m) \xrightarrow{\varepsilon(d)} (\mathbf{par} G_1 \# N_1, \dots, G_p \# N_p [\vec{G}_1] \rightarrow B'_1 \dots \parallel [\vec{G}_m] \rightarrow B'_m)}$$

$$\frac{\mathcal{E} \vdash B_1 \xrightarrow{\varepsilon(d+d')} B'_1 \dots \mathcal{E} \vdash B_j \xrightarrow{\delta(RN@d)} B'_j \dots \mathcal{E} \vdash B_m \xrightarrow{\varepsilon(d+d')} B'_m}{\mathcal{E} \vdash (\mathbf{par} G_1 \# N_1, \dots, G_p \# N_p \parallel [\vec{G}_1] \rightarrow B_1 \dots \parallel [\vec{G}_j] \rightarrow B_j \dots \parallel [\vec{G}_m] \rightarrow B_m) \xrightarrow{\varepsilon(d+d')} (\mathbf{par} G_1 \# N_1, \dots, G_p \# N_p \parallel [\vec{G}_1] \rightarrow B_1 \dots \parallel [\vec{G}_j] \rightarrow \mathbf{exit}(RN) \dots \parallel [\vec{G}_m] \rightarrow B_m)} [0 < d']$$

10.2.19 Parallel over values

Abstract syntax

$\mathbf{par} P \mathbf{in} N \parallel \parallel B$

Static Semantics

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

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

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

$$\frac{C \vdash N \Rightarrow \text{List} \quad C \vdash (P \Rightarrow \mathbf{any}) \Rightarrow (RT) \quad C, RT \vdash B \Rightarrow \mathbf{exit}(\)}{C \vdash \mathbf{par} P \mathbf{in} N \parallel \parallel B \Rightarrow \mathbf{exit}(\)}$$

Untimed dynamic semantics

$$\mathcal{E} \vdash \mathbf{par} P \mathbf{in} \mathbf{nil} \parallel \parallel B \xrightarrow{\delta(\)} \mathbf{block}$$

$$\frac{\mathcal{E} \vdash (P \Rightarrow N_1) \Rightarrow (RN) \quad \mathcal{E} \vdash B[RN] \parallel \parallel (\mathbf{par} P \mathbf{in} N_2 \parallel \parallel B) \xrightarrow{\mu(RN')} B'}{\mathcal{E} \vdash \mathbf{par} P \mathbf{in} \mathbf{cons}(N_1, N_2) \parallel \parallel B \xrightarrow{\mu(RN')} B'} \quad [\text{here } \mu ::= G \mid \mathbf{i} \mid \delta]$$

$$\frac{\mathcal{E} \vdash (P \Rightarrow N_1) \Rightarrow (RN) \quad \mathcal{E} \vdash B[RN] \parallel \parallel (\mathbf{par} P \mathbf{in} N_2 \parallel \parallel B) \xrightarrow{X(RN')} B'}{\mathcal{E} \vdash \mathbf{par} P \mathbf{in} \mathbf{cons}(N_1, N_2) \parallel \parallel B \xrightarrow{\mathbf{i}(\)} \mathbf{signal} X (RN'); B'}$$

Timed dynamic semantics

$$\frac{}{\mathcal{E} \vdash \mathbf{par} P \mathbf{in} \mathbf{nil} \mid \mid \mid B \xrightarrow{\varepsilon^{(0)}} \mathbf{block}}$$

$$\frac{\mathcal{E} \vdash (P \Rightarrow N_1) \Rightarrow (RN) \quad \mathcal{E} \vdash B[RN] \mid \mid \mid (\mathbf{par} P \mathbf{in} N_2 \mid \mid \mid B) \xrightarrow{\varepsilon^{(d)}} B'}{\mathcal{E} \vdash \mathbf{par} P \mathbf{in} \mathbf{cons}(N_1, N_2) \mid \mid \mid B \xrightarrow{\varepsilon^{(d)}} B'}$$

10.2.20 Variable declaration

Abstract syntax

$$\mathbf{var} V_1 : T_1 [:= E_1], \dots, V_n : T_n [:= E_n] \mathbf{in} B$$

Static semantics

$$\frac{C - \{V_1, \dots, V_n\} \vdash (V_1, \dots, V_n \Rightarrow T_1, \dots, T_n) \Rightarrow (RT', RT'') \quad C \vdash B_2 \Rightarrow \mathbf{exit}(RT')}{C; (V_1 \Rightarrow ?T_1); \dots; (V_n \Rightarrow ?T_n); RT' \vdash B \Rightarrow \mathbf{exit}(RT, RT''')} \left[\begin{array}{l} RT \text{ is disjoint with } RT', RT'' \\ \text{and } RT''' \subseteq RT', RT'' \end{array} \right]$$

$$\frac{C - \{V_1, \dots, V_n\} \vdash (V_1, \dots, V_n \Rightarrow T_1, \dots, T_n) \Rightarrow (RT', RT'') \quad C \vdash B_2 \Rightarrow \mathbf{exit}(RT')}{C; (V_1 \Rightarrow ?T_1); \dots; (V_n \Rightarrow ?T_n); RT' \vdash B \Rightarrow \mathbf{guarded}(RT'', RT)} \left[\begin{array}{l} RT \text{ is disjoint with } RT', RT'' \\ \text{and } RT''' \subseteq RT', RT'' \end{array} \right]$$

where

$$B_2 = (?V_{j1}, \dots, ?V_{jm}) := (E_{j1}, \dots, E_{jm})$$

$$\{j1, \dots, jm\} = \{k \mid V_k : T_k := E_k\}$$

If there is no variable initialization, $B_2 = \mathbf{exit}()$. $(V_i \Rightarrow ?T_i)$ informs about the type of V_i , but does not implicate a value.

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B_2 \xrightarrow{\delta^{(RN)}} B'_2 \quad \mathcal{E} \vdash B[RN] \xrightarrow{\mu^{(RN')}} B'}{\mathcal{E} \vdash \mathbf{var} V_1 : T_1 [:= E_1], \dots, V_n : T_n [:= E_n] \mathbf{in} B \xrightarrow{\mu^{(RN')}} \mathbf{var} V_1 : T_1, \dots, V_n : T_n \mathbf{in} B'}$$

$$\frac{\mathcal{E} \vdash B \xrightarrow{a^{(RN)}} B'}{\mathcal{E} \vdash \mathbf{var} RV : RT \mathbf{in} B \xrightarrow{a^{(RN)}} \mathbf{var} V_1 : T_1, \dots, V_n : T_n \mathbf{in} B'}$$

$$\frac{\mathcal{E} \vdash B \xrightarrow{\delta^{(RN', RN'')}} B' \quad \mathcal{E} \vdash RN' \Rightarrow RT' \quad \mathcal{E} \vdash (RV \Rightarrow RT) \Rightarrow (RT')}{\mathcal{E} \vdash \mathbf{var} V_1 : T_1, \dots, V_n : T_n \mathbf{in} B \xrightarrow{\delta^{(RN')}} \mathbf{block}}$$

where

$$\begin{aligned} B_2 &= (?V_{j_1}, \dots, ?V_{j_m}) := (E_{j_1}, \dots, E_{j_m}) \\ \{j_1, \dots, j_m\} &= \{k \mid V_k : T_k := E_k\} \\ RN' &= RN - \{V_1, \dots, V_n\} \end{aligned}$$

Note that the first rule only applies when the variable declaration includes instantiation of at least one variable. In such case, the appropriate values are substituted in B , and the behaviour evolves in a form without instantiations.

The second rule only applies when no instantiation is provided in the declaration, or when these instantiations have been substituted in B .

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B'}{\mathcal{E} \vdash \mathbf{var} V_1 : T_1, \dots, V_n : T_n \mathbf{in} B \xrightarrow{\varepsilon(d)} \mathbf{var} V_1 : T_1, \dots, V_n : T_n \mathbf{in} B'}$$

10.2.21 Gate hiding

Abstract syntax

hide $G[: T](, G[: T])^* \mathbf{in} B$

The default gate type is **(etc)**.

Static semantics

$$\frac{C \vdash T_1 \Rightarrow \mathbf{type} \ \dots \ C \vdash T_n \Rightarrow \mathbf{type} \quad C; G_1 \Rightarrow \mathbf{gate}(T_1), \dots, G_n \Rightarrow \mathbf{gate}(T_n) \vdash B \Rightarrow \mathbf{exit}(RT)}{C \vdash \mathbf{hide} G_1 : T_1, \dots, G_n : T_n \mathbf{in} B \Rightarrow \mathbf{exit}(RT)}$$

$$\frac{C \vdash T_1 \Rightarrow \mathbf{type} \ \dots \ C \vdash T_n \Rightarrow \mathbf{type} \quad C; G_1 \Rightarrow \mathbf{gate}(T_1), \dots, G_n \Rightarrow \mathbf{gate}(T_n) \vdash B \Rightarrow \mathbf{guarded}(RT)}{C \vdash \mathbf{hide} G_1 : T_1, \dots, G_n : T_n \mathbf{in} B \Rightarrow \mathbf{guarded}(RT)}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B'}{\mathcal{E} \vdash \mathbf{hide} G : T \mathbf{in} B \xrightarrow{\mu(RN)} \mathbf{hide} G : T \mathbf{in} B' \left[\mu \notin \vec{G} \right]}$$

$$\frac{\mathcal{E} \vdash B \xrightarrow{G_i(RN)} B' \quad \mathcal{E} \vdash (RN) \Rightarrow (RT_i)}{\mathcal{E} \vdash \mathbf{hide} G : T \mathbf{in} B \xrightarrow{i\circ} \mathbf{hide} G : T \mathbf{in} B'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B' \quad \mathcal{E} \vdash B \mathbf{refusing} (G : T, d)}{\mathcal{E} \vdash \mathbf{hide} \vec{G}(\vec{RT}) \mathbf{in} B \xrightarrow{\varepsilon(d)} \mathbf{hide} \vec{G} : T \mathbf{in} B'}$$

where $\mathcal{E} \vdash B \mathbf{refusing} (G : T, d)$ 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.

10.2.22 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 **(etc)**, 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

$$\begin{array}{c}
C \vdash (RT_1) \Rightarrow \mathbf{type} \quad \dots \quad C \vdash (RT_m) \Rightarrow \mathbf{type} \\
C \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1'') \quad \dots \quad C \vdash (RP_m \Rightarrow RT_m) \Rightarrow (RT_m'') \\
C; RT_1'' \vdash G_1' P_1 \Rightarrow \mathbf{exit} () \quad \dots \quad C; RT_m'' \vdash G_m' P_m \Rightarrow \mathbf{exit} () \\
C \vdash (RT_1') \Rightarrow \mathbf{type} \quad \dots \quad C \vdash (RT_m') \Rightarrow \mathbf{type} \\
C \vdash (RP_1' \Rightarrow RT_1''') \Rightarrow (RT_1''') \quad \dots \quad C \vdash (RP_n' \Rightarrow RT_n') \Rightarrow (RT_n''') \\
C; RT_1''' \vdash \mathbf{signal} X_1' E_1 \Rightarrow \mathbf{exit} () \quad \dots \quad C; RT_n''' \vdash \mathbf{signal} X_n' E_n \Rightarrow \mathbf{exit} () \\
C; G_1 \Rightarrow \mathbf{gate}(RT_1) \odot \\
\dots \odot G_m \Rightarrow \mathbf{gate}(RT_m) \\
\odot X_1 \Rightarrow \mathbf{exn}(RT_1') \odot \\
\dots \odot X_m \Rightarrow \mathbf{exn}(RT_n') \vdash B \Rightarrow \mathbf{exit}(RT) \\
\hline
C \vdash (\mathbf{rename} \\
\quad \mathbf{gate} G_1(RP_1:RT_1) \mathbf{is} G_1' P_1 \quad \dots \quad \mathbf{gate} G_m(RP_m:RT_m) \mathbf{is} G_m' P_m \\
\quad \mathbf{signal} X_1(RP_1':RT_1') \mathbf{is} X_1' E_1 \quad \dots \quad \mathbf{signal} X_m(RP_n':RT_n') \mathbf{is} X_n' E_n \\
\quad \mathbf{in} B) \Rightarrow \mathbf{exit}(RT)
\end{array}$$

$$\begin{array}{c}
C \vdash (RT_1) \Rightarrow \mathbf{type} \quad \dots \quad C \vdash (RT_m) \Rightarrow \mathbf{type} \\
C \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT_1'') \quad \dots \quad C \vdash (RP_m \Rightarrow RT_m) \Rightarrow (RT_m'') \\
C; RT_1'' \vdash G_1' P_1 \Rightarrow \mathbf{exit} () \quad \dots \quad C; RT_m'' \vdash G_m' P_m \Rightarrow \mathbf{exit} () \\
C \vdash (RT_1') \Rightarrow \mathbf{type} \quad \dots \quad C \vdash (RT_m') \Rightarrow \mathbf{type} \\
C \vdash (RP_1' \Rightarrow RT_1''') \Rightarrow (RT_1''') \quad \dots \quad C \vdash (RP_n' \Rightarrow RT_n') \Rightarrow (RT_n''') \\
C; RT_1''' \vdash \mathbf{signal} X_1' E_1 \Rightarrow \mathbf{exit} () \quad \dots \quad C; RT_n''' \vdash \mathbf{signal} X_n' E_n \Rightarrow \mathbf{exit} () \\
C; G_1 \Rightarrow \mathbf{gate}(RT_1) \odot \\
\dots \odot G_m \Rightarrow \mathbf{gate}(RT_m) \\
\odot X_1 \Rightarrow \mathbf{exn}(RT_1') \odot \\
\dots \odot X_m \Rightarrow \mathbf{exn}(RT_n') \vdash B \Rightarrow \mathbf{guarded}(RT) \\
\hline
C \vdash (\mathbf{rename} \\
\quad \mathbf{gate} G_1(RP_1:RT_1) \mathbf{is} G_1' P_1 \quad \dots \quad \mathbf{gate} G_m(RP_m:RT_m) \mathbf{is} G_m' P_m \\
\quad \mathbf{signal} X_1(RP_1':RT_1') \mathbf{is} X_1' E_1 \quad \dots \quad \mathbf{signal} X_m(RP_n':RT_n') \mathbf{is} X_n' E_n \\
\quad \mathbf{in} B) \Rightarrow \mathbf{guarded}(RT)
\end{array}$$

Untimed dynamic semantics

$$\begin{array}{c}
\mathcal{E} \vdash B \xrightarrow{\mu(RN)} B' \\
\hline
\mathcal{E} \vdash (\mathbf{rename} \\
\quad \mathbf{gate} G_1(RP_1:RT_1) \mathbf{is} G_1' P_1 \\
\quad \dots \\
\quad \mathbf{gate} G_m(RP_m:RT_m) \mathbf{is} G_m' P_m \\
\quad \mathbf{signal} X_1(RP_1':RT_1') \mathbf{is} X_1' E_1 \\
\quad \dots \\
\quad \mathbf{signal} X_m(RP_n':RT_n') \mathbf{is} X_n' E_n \\
\quad \mathbf{in} B) \xrightarrow{\mu(RN)} (\mathbf{rename} \\
\quad \mathbf{gate} G_1(RP_1:RT_1) \mathbf{is} G_1' P_1 \\
\quad \dots \\
\quad \mathbf{gate} G_m(RP_m:RT_m) \mathbf{is} G_m' P_m \\
\quad \mathbf{signal} X_1(RP_1':RT_1') \mathbf{is} X_1' E_1 \\
\quad \dots \\
\quad \mathbf{signal} X_m(RP_n':RT_n') \mathbf{is} X_n' E_n \\
\quad \mathbf{in} B') \quad [\mu \notin \{G_1, \dots, G_m, X_1, \dots, X_m\}]
\end{array}$$

$$\begin{array}{c}
\mathcal{E} \vdash B \xrightarrow{a_i(RN)} B' \\
\mathcal{E} \vdash ((RP_i : RT_i) \Rightarrow (RN)) \Rightarrow (RN') \\
\mathcal{E} \vdash B_i[RN'] \xrightarrow{a(RN')} B'_i \\
\hline
\mathcal{E} \vdash (\text{rename} \quad \xrightarrow{a(RN')} \quad \text{rename}) \quad [a_i \in \{G_1, \dots, G_m, X_1, \dots, X_m\}] \\
\begin{array}{ccc}
\text{gate } G_1(RP_1 : RT_1) \text{ is } G'_1 P_1 & & \text{gate } G_1(RP_1 : RT_1) \text{ is } G'_1 P_1 \\
\dots & & \dots \\
\text{gate } G_m(RP_m : RT_m) \text{ is } G'_m P_m & & \text{gate } G_m(RP_m : RT_m) \text{ is } G'_m P_m \\
\text{signal } X_1(RP'_1 : RT'_1) \text{ is } X'_1 E_1 & & \text{signal } X_1(RP'_1 : RT'_1) \text{ is } X'_1 E_1 \\
\dots & & \dots \\
\text{signal } X_m(RP'_m : RT'_m) \text{ is } X'_m E_m & & \text{signal } X_m(RP'_m : RT'_m) \text{ is } X'_m E_m \\
\text{in } B) & & \text{in } B')
\end{array}
\end{array}$$

where $B_i = G'_i P_i$ in the case of gates and $B_i = \text{signal } X'_i E_i$ in the case of exceptions.

Timed dynamic semantics

$$\begin{array}{c}
\mathcal{E} \vdash B \xrightarrow{\varepsilon(d)} B' \\
\hline
\mathcal{E} \vdash (\text{rename} \quad \xrightarrow{\varepsilon(d)} \quad \text{rename}) \\
\begin{array}{ccc}
\text{gate } G_1(RP_1 : RT_1) \text{ is } G'_1 P_1 \dots & & \text{gate } G_1(RP_1 : RT_1) \text{ is } G'_1 P_1 \dots \\
\text{gate } G_m(RP_m : RT_m) \text{ is } G'_m P_m & & \text{gate } G_m(RP_m : RT_m) \text{ is } G'_m P_m \\
\text{signal } X_1(RP'_1 : RT'_1) \text{ is } X'_1 E_1 \dots & & \text{signal } X_1(RP'_1 : RT'_1) \text{ is } X'_1 E_1 \dots \\
\text{signal } X_m(RP'_m : RT'_m) \text{ is } X'_m E_m & & \text{signal } X_m(RP'_m : RT'_m) \text{ is } X'_m E_m \\
\text{in } B) & & \text{in } B)
\end{array}
\end{array}$$

10.2.23 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

$$\begin{array}{l}
C \vdash \Pi \Rightarrow \text{[gate}(RT_1), \dots, \text{gate}(RT_m)] (RT') \text{[exn}(RT'_1), \dots, \text{exn}(RT'_n)]} \rightarrow \text{exit}(RT) \\
C \vdash G_1 \Rightarrow \text{gate}(RT_1) \dots C \vdash G_m \Rightarrow \text{gate}(RT_m) \\
C \vdash E_1 \Rightarrow \text{exit}(T_1) \dots C \vdash E_p \Rightarrow \text{exit}(T_p) \\
C \vdash X_1 \Rightarrow \text{exn}(RT'_1) \dots C \vdash X_n \Rightarrow \text{exn}(RT'_n) \\
\hline
C \vdash \Pi \ [G_1, \dots, G_m] \ (E_1, \dots, E_p) \ [X_1, \dots, X_n] \Rightarrow \text{exit}(RT) \quad [RT' = T_1, \dots, T_p]
\end{array}$$

$$\begin{array}{l}
C \vdash \Pi \Rightarrow \text{[gate}(RT_1), \dots, \text{gate}(RT_m)] (RT') \text{[exn}(RT'_1), \dots, \text{exn}(RT'_n)]} \rightarrow \text{guarded}(RT) \\
C \vdash G_1 \Rightarrow \text{gate}(RT_1) \dots C \vdash G_m \Rightarrow \text{gate}(RT_m) \\
C \vdash (E_1, \dots, E_p) \Rightarrow \text{exit}((RT')) \\
C \vdash X_1 \Rightarrow \text{exn}(RT'_1) \dots C \vdash X_n \Rightarrow \text{exn}(RT'_n) \\
\hline
C \vdash \Pi \ [G_1, \dots, G_m] \ (E_1, \dots, E_p) \ [X_1, \dots, X_n] \Rightarrow \text{guarded}(RT)
\end{array}$$

Untimed dynamic semantics

$$\begin{array}{c}
\mathcal{E} \vdash \Pi \Rightarrow \lambda[\vec{G}'(\vec{RT})](RP : RT) [\vec{X}'(\vec{RT}')] \rightarrow B \\
\mathcal{E} \vdash \text{(rename} \\
\quad \text{gate } G'_1(RP_1 : RT_1) \text{ is } G'_1 P_1 \cdots \text{gate } G'_m(RP_m : RT_m) \text{ is } G'_m P_m \\
\quad \text{signal } X'_1(RP'_1 : RT'_1) \text{ is } X'_1 E_1 \cdots \text{signal } X'_n(RP'_n : RT'_n) \text{ is } X'_n E_n \\
\quad \text{in case } \vec{E} : (RT) \text{ is } (RP) \rightarrow B \xrightarrow{\mu(RN)} B' \\
\hline
\mathcal{E} \vdash \Pi [\vec{G}] \vec{E} [\vec{X}] \xrightarrow{\mu(RN)} B'
\end{array}$$

Timed dynamic semantics

$$\begin{array}{c}
\mathcal{E} \vdash \Pi \Rightarrow \lambda[\vec{G}'(\vec{RT})](RP : RT) [\vec{X}'(\vec{RT}')] \rightarrow B \\
\mathcal{E} \vdash \text{(rename} \\
\quad \text{gate } G'_1(RP_1 : RT_1) \text{ is } G'_1 P_1 \cdots \text{gate } G'_m(RP_m : RT_m) \text{ is } G'_m P_m \\
\quad \text{signal } X'_1(RP'_1 : RT'_1) \text{ is } X'_1 E_1 \cdots \text{signal } X'_n(RP'_n : RT'_n) \text{ is } X'_n E_n \\
\quad \text{in case } \vec{E} : (RT) \text{ is } (RP) \rightarrow B \xrightarrow{\varepsilon(d)} B' \\
\hline
\mathcal{E} \vdash \Pi [\vec{G}] \vec{E} [\vec{X}] \xrightarrow{\varepsilon(d)} B'
\end{array}$$

10.2.24 loop iteration**Abstract syntax**

loop B

Static semantics

$$\frac{C \vdash B \Rightarrow \text{exit}(RT)}{C \vdash \text{loop } B \Rightarrow \text{guarded}(\text{none})}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash B; \text{loop } B \xrightarrow{\mu(RN)} B'}{\mathcal{E} \vdash \text{loop } B \xrightarrow{\mu(RN)} B'}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash B; \text{loop } B \xrightarrow{\varepsilon(d)} B'}{\mathcal{E} \vdash \text{loop } B \xrightarrow{\varepsilon(d)} B'}$$

10.2.25 Case**Abstract syntax**

case E **is** BM **endcase**

Static semantics

$$\frac{\begin{array}{l} C \vdash E \Rightarrow \mathbf{exit}(T) \\ C \vdash (BM \Rightarrow T) \Rightarrow \mathbf{exit}(RT) \end{array}}{C \vdash \mathbf{case } E \mathbf{ is } BM \Rightarrow \mathbf{exit}(RT)}$$

$$\frac{\begin{array}{l} C \vdash E \Rightarrow \mathbf{exit}(T) \\ C \vdash (BM \Rightarrow T) \Rightarrow \mathbf{guarded}(RT) \end{array}}{C \vdash \mathbf{case } E \mathbf{ is } BM \Rightarrow \mathbf{guarded}(RT)}$$

Untimed dynamic semantics

$$\frac{\begin{array}{l} \mathcal{E} \vdash E \xrightarrow{\delta(N)} E' \\ \mathcal{E} \vdash (BM \Rightarrow N) \xrightarrow{\mu(RN)} B \end{array}}{\mathcal{E} \vdash \mathbf{case } E \mathbf{ is } BM \xrightarrow{\mu(RN)} B}$$

$$\frac{\begin{array}{l} \mathcal{E} \vdash E \xrightarrow{X(RN)} E' \\ \mathcal{E} \vdash \mathbf{case } E \mathbf{ is } BM \xrightarrow{X(RN)} \mathbf{case } E' \mathbf{ is } BM \end{array}}{\mathcal{E} \vdash \mathbf{case } E \mathbf{ is } BM \xrightarrow{X(RN)} \mathbf{case } E' \mathbf{ is } BM}$$

Timed dynamic semantics

$$\frac{\begin{array}{l} \mathcal{E} \vdash E \xrightarrow{\delta(N)} E' \\ \mathcal{E} \vdash (BM \Rightarrow N) \xrightarrow{\varepsilon(d)} B \end{array}}{\mathcal{E} \vdash \mathbf{case } E \mathbf{ is } BM \xrightarrow{\varepsilon(d)} B}$$

10.2.26 Case with tuples**Abstract syntax**

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

Static semantics

$$\frac{\begin{array}{l} C \vdash (E_1, \dots, E_n) \Rightarrow \mathbf{exit}(RT) \\ C \vdash (BM \Rightarrow (RT)) \Rightarrow \mathbf{exit}(RT') \end{array}}{C \vdash \mathbf{case } (E_1, \dots, E_n) \mathbf{ is } BM \Rightarrow \mathbf{exit}(RT')}$$

$$\frac{\begin{array}{l} C \vdash (E_1, \dots, E_n) \Rightarrow \mathbf{exit}(RT) \\ C \vdash (BM \Rightarrow (RT)) \Rightarrow \mathbf{guarded}(RT') \end{array}}{C \vdash \mathbf{case } (E_1, \dots, E_n) \mathbf{ is } BM \Rightarrow \mathbf{guarded}(RT')}$$

Untimed dynamic semantics

$$\frac{\begin{array}{l} \mathcal{E} \vdash E_1, \dots, E_n \xrightarrow{\delta(RN)} E'_1, \dots, E'_n \\ \mathcal{E} \vdash (BM \Rightarrow (RN)) \xrightarrow{\mu(RN')} B \end{array}}{\mathcal{E} \vdash \mathbf{case } (E_1, \dots, E_n) \mathbf{ is } BM \xrightarrow{\mu(RN')} B}$$

$$\frac{\begin{array}{l} \mathcal{E} \vdash E_j \xrightarrow{X(N)} E'_j \\ \mathcal{E} \vdash \mathbf{case } (E_1, \dots, E_n) \mathbf{ is } BM \xrightarrow{X(N)} \mathbf{case } (E'_1, \dots, E'_n) \mathbf{ is } BM \end{array}}{\mathcal{E} \vdash \mathbf{case } (E_1, \dots, E_n) \mathbf{ is } BM \xrightarrow{X(N)} \mathbf{case } (E'_1, \dots, E'_n) \mathbf{ is } BM} \text{ [with } E'_i = E_i \text{ if } i \neq j]$$

Note that $E_1, \dots, E_n \xrightarrow{\delta(RN)} E'_1, \dots, E'_n$ 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

$$\frac{\mathcal{E} \vdash E_1, \dots, E_n \xrightarrow{\delta(RN)} E'_1, \dots, E'_n \quad \mathcal{E} \vdash (BM \Rightarrow (RN)) \xrightarrow{\varepsilon(d)} B}{\mathcal{E} \vdash \mathbf{case} (E_1, \dots, E_n) \mathbf{is} BM \xrightarrow{\varepsilon(d)} B}$$

10.2.27 Signalling

Abstract syntax

signal X [$'(' E ')'$]

The default expression is **exit**().

Static semantics

$$\frac{C \vdash E \Rightarrow \mathbf{exit}((RT)) \quad C \vdash X \Rightarrow \mathbf{exn}(RT)}{C \vdash \mathbf{signal} X E \Rightarrow \mathbf{guarded}()}$$

Untimed dynamic semantics

$$\frac{\mathcal{E} \vdash E \xrightarrow{\delta(RN)} E'}{\mathcal{E} \vdash \mathbf{signal} X E \xrightarrow{X(RN)} \mathbf{exit}()}$$

$$\frac{\mathcal{E} \vdash E \xrightarrow{X'(RN)} E'}{\mathcal{E} \vdash \mathbf{signal} X E \xrightarrow{X'(RN)} \mathbf{signal} X E'}$$

Timed dynamic semantics None.

10.3 Type expressions

10.3.1 Type identifier

Abstract syntax

S

Static semantics

$$\overline{C, S \Rightarrow \mathbf{type} \vdash S \Rightarrow \mathbf{type}}$$

$$\overline{C, S \equiv T \vdash S \equiv T}$$

10.3.2 Empty type

Abstract syntax

none

Static semantics

$$\frac{}{\mathcal{C} \vdash \mathbf{none} \Rightarrow \mathbf{type}}$$

$$\frac{}{\mathcal{C} \vdash \mathbf{none} \sqsubseteq T}$$

$$\frac{}{\mathcal{C} \vdash \mathbf{none} \equiv (V \Rightarrow \mathbf{none}, RT)}$$

10.3.3 Universal type

Abstract syntax

any

Static semantics

$$\frac{}{\mathcal{C} \vdash \mathbf{any} \Rightarrow \mathbf{type}}$$

$$\frac{}{\mathcal{C} \vdash T \sqsubseteq \mathbf{any}}$$

10.3.4 Record type

Abstract syntax

(RT)

Static semantics

$$\frac{\mathcal{C} \vdash RT \Rightarrow \mathbf{record}}{\mathcal{C} \vdash (RT) \Rightarrow \mathbf{type}}$$

$$\frac{\mathcal{C} \vdash RT \sqsubseteq RT'}{\mathcal{C} \vdash (RT) \sqsubseteq (RT')}$$

10.4 Record type expressions

Rules for record type expressions will be given in a compositional way, given first the semantics for items **etc** and $V \Rightarrow T$ and then the semantic for disjoint union (RT, RT') .

10.4.1 Singleton record

Abstract syntax

$V \Rightarrow T$

Static semantics

$$\frac{C \vdash T \Rightarrow \mathbf{type}}{C \vdash (V \Rightarrow T) \Rightarrow \mathbf{record}}$$

$$\frac{C \vdash T \sqsubseteq T'}{C \vdash (V \Rightarrow T) \sqsubseteq (V \Rightarrow T')}$$

10.4.2 Universal record**Abstract syntax**

etc

Static semantics

$$\overline{C \vdash \mathbf{etc} \Rightarrow \mathbf{record}}$$

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

10.4.3 Record disjoint union**Abstract syntax**

RT, RT

Static semantics

$$\frac{C \vdash RT_1 \Rightarrow \mathbf{record} \quad C \vdash RT_2 \Rightarrow \mathbf{record}}{C \vdash RT_1, RT_2 \Rightarrow \mathbf{record}} [RT_1 \text{ and } RT_2 \text{ have disjoint fields}]$$

$$\frac{C \vdash RT_1 \sqsubseteq RT'_1 \quad C \vdash RT_2 \sqsubseteq RT'_2}{C \vdash RT_1, RT_2 \sqsubseteq RT'_1, RT'_2}$$

$$\overline{C \vdash RT_1, RT_2 \equiv RT_2, RT_1}$$

$$\overline{C \vdash (RT_1, RT_2), RT_3 \equiv RT_1, (RT_2, RT_3)}$$

$$\overline{C \vdash (), RT \equiv RT}$$

10.4.4 Empty record**Abstract syntax**

()

Static semantics

$$\overline{C \vdash () \Rightarrow \mathbf{record}}$$

10.5 Value expressions

10.5.1 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.

10.5.2 Variables

Abstract syntax

$$V$$

Static semantics

$$\frac{}{C, V \Rightarrow T \vdash V \Rightarrow T}$$

10.5.3 Record values

Abstract syntax

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

Static semantics

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

10.5.4 Constructor application

Abstract syntax

$$C [N]$$

The default argument is ().

Static semantics

$$\frac{C \vdash C \Rightarrow ((RT) \rightarrow S) \quad C \vdash N \Rightarrow (RT)}{C \vdash C N \Rightarrow S}$$

10.6 Record value expressions

10.6.1 Singleton record

Abstract syntax

$$V \Rightarrow N$$

Static semantics

$$\frac{C \vdash N \Rightarrow T}{C \vdash (V \Rightarrow N) \Rightarrow (V \Rightarrow T)}$$

10.6.2 Record disjoint union**Abstract syntax**

RN, RN

Static semantics

$$\frac{C \vdash RN_1 \Rightarrow RT_1 \quad C \vdash RN_2 \Rightarrow RT_2}{C \vdash RN_1, RN_2 \Rightarrow RT_1, RT_2} [RN_1 \text{ and } RN_2 \text{ have disjoint fields}]$$

10.6.3 Empty record**Abstract syntax**

()

Static semantics

$$\overline{C \vdash () \Rightarrow ()}$$

10.7 Patterns

In this section the semantics of pattern matching (the sole use of patterns) is defined.

10.7.1 Record pattern**Abstract syntax**

' (' RP ') '

Static semantics

$$\frac{C \vdash (RP \Rightarrow RT') \Rightarrow (RT)}{C \vdash ((RP) \Rightarrow (RT')) \Rightarrow (RT)}$$

$$\frac{C \vdash ((V_1 \Rightarrow P_1, \dots, V_n \Rightarrow P_n) \Rightarrow (V_1 \Rightarrow \mathbf{any}, \dots, V_n \Rightarrow \mathbf{any})) \Rightarrow (RT)}{C \vdash ((V_1 \Rightarrow P_1, \dots, V_n \Rightarrow P_n) \Rightarrow \mathbf{any}) \Rightarrow (RT)}$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash (RP \Rightarrow RN') \Rightarrow (RN)}{\mathcal{E} \vdash ((RP) \Rightarrow N) \Rightarrow (RN)} [N = (RN')]$$

$$\frac{\mathcal{E} \vdash (RP \Rightarrow RN') \Rightarrow \mathbf{fail}}{\mathcal{E} \vdash ((RP) \Rightarrow N) \Rightarrow \mathbf{fail}} [N = (RN')]$$

$$\overline{\mathcal{E} \vdash ((RP) \Rightarrow N) \Rightarrow \mathbf{fail}} [\exists RN' \mid N = (RN')]$$

The last condition means that N is not a bracketer record of values.

10.7.2 Wildcard

Abstract syntax

any : T

Static semantics

$$\frac{C \vdash T \Rightarrow \mathbf{type}}{C \vdash (\mathbf{any} : T \Rightarrow T) \Rightarrow ()}$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash N \Rightarrow T}{\mathcal{E} \vdash (\mathbf{any} : T \Rightarrow N) \Rightarrow ()}$$

10.7.3 Variable binding

Abstract syntax

? V

Static semantics

$$\frac{C \vdash T \sqsubseteq T'}{C \vdash (?V \Rightarrow T) \Rightarrow (V \Rightarrow T)} [V \Rightarrow ?T' \in C]$$

$$\frac{}{C \vdash (?V \Rightarrow T) \Rightarrow (V \Rightarrow T)} [\exists V \Rightarrow ?T' \mid V \Rightarrow ?T' \in C]$$

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

$$\frac{\mathcal{E} \vdash N \Rightarrow T \quad \mathcal{E} \vdash T \sqsubseteq T'}{\mathcal{E} \vdash (?V \Rightarrow N) \Rightarrow (V \Rightarrow N)} [V \Rightarrow ?T' \in \mathcal{E}]$$

$$\frac{}{\mathcal{E} \vdash (?V \Rightarrow N) \Rightarrow (V \Rightarrow N)} [\exists V \Rightarrow ?T' \mid V \Rightarrow ?T' \in \mathcal{E}]$$

10.7.4 Expression pattern

Abstract syntax

! EA

Static semantics

$$\frac{C \vdash EA \Rightarrow \mathbf{exit}(T)}{C \vdash (!EA \Rightarrow T) \Rightarrow ()}$$

Note that no bindings can be produced in EA .

Dynamic semantics

$$\frac{\mathcal{E} \vdash EA \xrightarrow{\delta(N')} EA'}{\mathcal{E} \vdash (!EA \Rightarrow N) \Rightarrow ()} [N = N']$$

$$\frac{\mathcal{E} \vdash EA \xrightarrow{\delta(N')} EA'}{\mathcal{E} \vdash (!EA \Rightarrow N) \Rightarrow \mathbf{fail}} [N \neq N']$$

$$\frac{\mathcal{E} \vdash EA \xrightarrow{X(RN)} EA'}{\mathcal{E} \vdash (!EA \Rightarrow N) \Rightarrow \mathbf{fail}}$$

10.7.5 Constructor application**Abstract syntax**

$$C [P]$$

The default pattern is ().

Static semantics

$$\begin{array}{l} C \vdash C \Rightarrow (RT) \rightarrow S \\ C \vdash S \sqsubseteq T \\ \frac{C \vdash (P \Rightarrow (RT)) \Rightarrow (RT')}{C \vdash (C P \Rightarrow T) \Rightarrow (RT')} \end{array}$$

Dynamic semantics

$$\frac{\mathcal{E} \vdash (P \Rightarrow (RN)) \Rightarrow (RN')}{\mathcal{E} \vdash (C P \Rightarrow N) \Rightarrow (RN')} [N = C(RN)]$$

$$\frac{\mathcal{E} \vdash (P \Rightarrow (RN)) \Rightarrow \mathbf{fail}}{\mathcal{E} \vdash (C P \Rightarrow N) \Rightarrow \mathbf{fail}} [N = C(RN)]$$

$$\frac{}{\mathcal{E} \vdash (C P \Rightarrow N) \Rightarrow \mathbf{fail}} [\exists C N' \mid N = C N']$$

10.7.6 Explicit typing**Abstract syntax**

$$P:T$$

Static semantics

$$\begin{array}{l} C \vdash T \Rightarrow \mathbf{type} \\ C \vdash T \sqsubseteq T' \\ \frac{C \vdash (P \Rightarrow T) \Rightarrow (RT)}{C \vdash (P:T \Rightarrow T') \Rightarrow (RT)} \end{array}$$

Dynamic semantics

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

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

$$\frac{\mathcal{E} \vdash N \Rightarrow T}{\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow \mathbf{fail}}$$

$$\frac{\mathcal{E} \vdash (P:T \Rightarrow N) \Rightarrow \mathbf{fail}}{\mathcal{E} \vdash (P:T \Rightarrow N) \Rightarrow \mathbf{fail}}$$

$$\frac{\mathcal{E} \vdash N \Rightarrow T'}{\mathcal{E} \vdash T \sqcap T' \Rightarrow \mathbf{none}}$$

$$\frac{\mathcal{E} \vdash T \sqcap T' \Rightarrow \mathbf{none}}{\mathcal{E} \vdash (P:T \Rightarrow N) \Rightarrow \mathbf{fail}}$$

10.8 Record patterns**Static semantics**

$$C \vdash (RP \Rightarrow RT) \Rightarrow (RT')$$

Dynamic semantics

$$\mathcal{E} \vdash (RP \Rightarrow RN) \Rightarrow (RN')$$

$$\mathcal{E} \vdash (RP \Rightarrow RN) \Rightarrow \mathbf{fail}$$

10.8.1 Singleton record pattern**Abstract syntax**

$$V \Rightarrow P$$

Static semantics

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

Dynamic semantics

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

$$\frac{\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow \mathbf{fail}}{\mathcal{E} \vdash ((V \Rightarrow P) \Rightarrow (V \Rightarrow N)) \Rightarrow \mathbf{fail}}$$

10.8.2 Record wildcard**Abstract syntax**

etc

Static semantics

$$\overline{C \vdash (\mathbf{etc} \Rightarrow RT) \Rightarrow ()}$$

Dynamic semantics

$$\overline{\mathcal{E} \vdash (\mathbf{etc} \Rightarrow RN) \Rightarrow ()}$$

10.8.3 Record match**Abstract syntax**

$P \text{ as } RP$

Static semantics

$$\begin{array}{l} C \vdash (P \Rightarrow (RT)) \Rightarrow (RT_1) \\ C \vdash (RP \Rightarrow RT) \Rightarrow (RT_2) \\ \hline C \vdash (P \text{ as } RP \Rightarrow RT) \Rightarrow (RT_1, RT_2) \quad [RT_1 \text{ and } RT_2 \text{ have disjoint fields}] \end{array}$$

Dynamic semantics

$$\begin{array}{l} \mathcal{E} \vdash (P \Rightarrow (RN)) \Rightarrow (RN_1) \\ \mathcal{E} \vdash (RP \Rightarrow RN) \Rightarrow (RN_2) \\ \hline \mathcal{E} \vdash (P \text{ as } RP \Rightarrow RN) \Rightarrow (RN_1, RN_2) \end{array}$$

10.8.4 Record disjoint union**Abstract syntax**

RP, RP

Static semantics

$$\frac{C \vdash (RP_1 \Rightarrow RT_1) \Rightarrow (RT'_1) \quad C \vdash (RP_2 \Rightarrow RT_2) \Rightarrow (RT'_2) \quad [RT'_1 \text{ and } RT'_2 \text{ have disjoint fields}]}{C \vdash (RP_1, RP_2 \Rightarrow RT_1, RT_2) \Rightarrow (RT'_1, RT'_2)}$$

Dynamic semantics

$$\begin{array}{l} \mathcal{E} \vdash (RP_1 \Rightarrow RN_1) \Rightarrow (RN'_1) \quad \mathcal{E} \vdash (RP_2 \Rightarrow RN_2) \Rightarrow (RN'_2) \\ \hline \mathcal{E} \vdash (RP_1, RP_2 \Rightarrow RN_1, RN_2) \Rightarrow (RN'_1, RN'_2) \\ \\ \mathcal{E} \vdash (RP_1 \Rightarrow RN_1) \Rightarrow \mathbf{fail} \\ \hline \mathcal{E} \vdash (RP_1, RP_2 \Rightarrow RN_1, RN_2) \Rightarrow \mathbf{fail} \\ \\ \mathcal{E} \vdash (RP_2 \Rightarrow RN_2) \Rightarrow \mathbf{fail} \\ \hline \mathcal{E} \vdash (RP_1, RP_2 \Rightarrow RN_1, RN_2) \Rightarrow \mathbf{fail} \end{array}$$

10.8.5 Empty record pattern**Abstract syntax**

$()$

Static semantics

$$\overline{C \vdash (() \Rightarrow ()) \Rightarrow ()}$$

Dynamic semantics

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

10.9 Record of variables**10.9.1 Singleton record variable****Abstract syntax**

$$V \Rightarrow V$$

Static semantics

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

10.9.2 Record disjoint union**Abstract syntax**

$$RV, RV$$

Static semantics

$$\frac{\mathcal{C} \vdash (RV_1 \Rightarrow RT_1) \Rightarrow (RT'_1) \quad \mathcal{C} \vdash (RV_2 \Rightarrow RT_2) \Rightarrow (RT'_2)}{\mathcal{C} \vdash (RV_1, RV_2 \Rightarrow RT_1, RT_2) \Rightarrow (RT'_1, RT'_2)} [RT'_1 \text{ and } RT'_2 \text{ have disjoint fields}]$$

10.10 Behaviour pattern-matching**10.10.1 Single match****Abstract syntax**

$$P [\text{' [? E ?] ' }] \rightarrow B$$

The default selection predicate is [true].

Static semantics

$$\frac{\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT) \quad \mathcal{C}; RT \vdash E \Rightarrow \mathbf{exit}(\mathbf{bool}) \quad \mathcal{C}; RT \vdash B \Rightarrow \mathbf{exit}(RT')}{\mathcal{C} \vdash ((P [E] \rightarrow B) \Rightarrow T) \Rightarrow \mathbf{exit}(RT')}$$

$$\frac{\mathcal{C} \vdash (P \Rightarrow T) \Rightarrow (RT) \quad \mathcal{C}; RT \vdash E \Rightarrow \mathbf{exit}(\mathbf{bool}) \quad \mathcal{C}; RT \vdash B \Rightarrow \mathbf{guarded}(RT')}{\mathcal{C} \vdash ((P [E] \rightarrow B) \Rightarrow T) \Rightarrow \mathbf{guarded}(RT')}$$

Untimed dynamic semantics

$$\begin{array}{c}
\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow \mathbf{fail} \\
\hline
\mathcal{E} \vdash ((P [E] \rightarrow B) \Rightarrow N) \Rightarrow \mathbf{fail} \\
\\
\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\
\mathcal{E} \vdash E[RN] \xrightarrow{\delta(\mathbf{false})} E' \\
\hline
\mathcal{E} \vdash ((P [E] \rightarrow B) \Rightarrow N) \Rightarrow \mathbf{fail} \\
\\
\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\
\mathcal{E} \vdash E[RN] \xrightarrow{X(RN)} E' \\
\hline
\mathcal{E} \vdash ((P [E] \rightarrow B) \Rightarrow N) \Rightarrow \mathbf{fail} \\
\\
\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\
\mathcal{E} \vdash E[RN] \xrightarrow{\delta(\mathbf{true})} E' \\
\mathcal{E} \vdash B[RN] \xrightarrow{\mu(RN')} B' \\
\hline
\mathcal{E} \vdash ((P [E] \rightarrow B) \Rightarrow N) \xrightarrow{\mu(RN')} B'
\end{array}$$

Timed dynamic semantics

$$\begin{array}{c}
\mathcal{E} \vdash (P \Rightarrow N) \Rightarrow (RN) \\
\mathcal{E} \vdash E[RN] \xrightarrow{\delta(\mathbf{true})} E' \\
\mathcal{E} \vdash B[RN] \xrightarrow{\varepsilon(d)} B' \\
\hline
\mathcal{E} \vdash ((P [E] \rightarrow B) \Rightarrow N) \xrightarrow{\varepsilon(d)} B'
\end{array}$$

10.10.2 Multiple match**Abstract syntax**

$$BM \text{ ' } | \text{ ' } BM$$

Static semantics

$$\begin{array}{c}
C, RT_1, RT_2 \vdash (BM_1 \Rightarrow T) \Rightarrow \mathbf{exit}(RT_1, RT) \quad C, RT_1, RT_2 \vdash (BM_2 \Rightarrow T) \Rightarrow \mathbf{exit}(RT_2, RT) \\
\hline
C, RT_1, RT_2 \vdash ((BM_1 | BM_2) \Rightarrow T) \Rightarrow \mathbf{exit}(RT_1, RT_2, RT) \\
\\
C, RT_1, RT_2 \vdash (BM_1 \Rightarrow T) \Rightarrow \mathbf{guarded}(RT_1, RT) \quad C, RT_1, RT_2 \vdash (BM_2 \Rightarrow T) \Rightarrow \mathbf{guarded}(RT_2, RT) \\
\hline
C, RT_1, RT_2 \vdash ((BM_1 | BM_2) \Rightarrow T) \Rightarrow \mathbf{guarded}(RT_1, RT_2, RT)
\end{array}$$

Untimed dynamic semantics

$$\begin{array}{c}
\mathcal{E} \vdash (BM_1 \Rightarrow N) \xrightarrow{\mu(RN)} B \\
\hline
\mathcal{E} \vdash ((BM_1 | BM_2) \Rightarrow N) \xrightarrow{\mu(RN)} B \\
\\
\mathcal{E} \vdash (BM_1 \Rightarrow N) \Rightarrow \mathbf{fail} \\
\mathcal{E} \vdash (BM_2 \Rightarrow N) \xrightarrow{\mu(RN)} B \\
\hline
\mathcal{E} \vdash ((BM_1 | BM_2) \Rightarrow N) \xrightarrow{\mu(RN)} B \\
\\
\mathcal{E} \vdash (BM_1 \Rightarrow N) \Rightarrow \mathbf{fail} \\
\mathcal{E} \vdash (BM_2 \Rightarrow N) \Rightarrow \mathbf{fail} \\
\hline
\mathcal{E} \vdash ((BM_1 | BM_2) \Rightarrow N) \Rightarrow \mathbf{fail}
\end{array}$$

Timed dynamic semantics

$$\frac{\mathcal{E} \vdash (BM_1 \Rightarrow N) \xrightarrow{\varepsilon(d)} B}{\mathcal{E} \vdash ((BM_1 | BM_2) \Rightarrow N) \xrightarrow{\varepsilon(d)} B}$$

$$\mathcal{E} \vdash (BM_1 \Rightarrow N) \Rightarrow \mathbf{fail}$$

$$\frac{\mathcal{E} \vdash (BM_2 \Rightarrow N) \xrightarrow{\varepsilon(d)} B}{\mathcal{E} \vdash ((BM_1 | BM_2) \Rightarrow N) \xrightarrow{\varepsilon(d)} B}$$

11 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 [21]).

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.

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.

11.1 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)
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

endmod

```

11.2 Natural Numbers

This subclause contains the interface and the implementation module for natural values. The constants of type `nat` 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)

    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

```

```

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

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
```

11.3 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 *)

    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;
    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);

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.

11.4 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 *)

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)
    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

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 *)

```

11.5 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
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 *)

```

11.6 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
  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 *)
```

11.7 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 *)
```

11.8 Enumerated Type Scheme

The declaration of an enumerated type ET with values C1 , ..., Cn :

```

type ET is
  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
    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

```

11.9 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:

```
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
```

11.10 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
  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
endfunc

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
        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 ())))
```

11.11 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
  | 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 ()) ...))

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 [19] and CSP [11]) 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 [23] 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 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:packet, left:data, right:data] is
  var
    p:packet
  in
    inp(?p);
    case p.de is
      L -> left(!p.da)
      | R -> right(!p.da)
    endcase;
    Router [inp, left, right]
  endvar
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 $C \vdash E \Rightarrow \mathbf{exit} (T)$ meaning 'in context C expression E has result type T ' for example:

$$1 \Rightarrow \mathbf{float}, x \Rightarrow \mathbf{float}, / \Rightarrow (\mathbf{float}, \mathbf{float}) \rightarrow \mathbf{exit} (\mathbf{float}) \vdash 1/x \Rightarrow \mathbf{exit} (\mathbf{float})$$

means 'in a context where 1 and x are floats, and $/$ is a function from pairs of floats to floats, then the expression $1/x$ has result type float'. The static semantics includes:

- 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 :

$$\vdash 1/2 \xrightarrow{\delta(0.5)} \mathbf{block}$$

and $1/0$ raises the exception \mathbf{Div} :

$$\vdash 1/0 \xrightarrow{\mathbf{Div}(\cdot)} \mathbf{block}$$

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 [17] and LOTOS with functional datatypes [12, 13]. 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 [15].

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 'point' synonymous with a record of floats as:

```
type point is
  (x=>float,y=>float)
endtype
```

and we can declare a recursive data type of integer lists as:

```
type intlist is
  nil
  | cons(int,intlist)
endtype
```

Type synonyms can be used interchangeably, for example the following declarations are the same:

```
type colpixel is
  (pt=>point,col=>colour)
endtype
type colpixel' is
  (pt=>(x=>float,y=>float),col=>colour)
endtype
```

We can use colpixel and colpixel' as the same type (for example any function expecting a colpixel will accept a colpixel'). More succinctly, type equality is *structural* not by *name*.

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
```

It is possible to define recursive data types, such as the datatype of lists above.

Finally, there is a shorthand for renaming types:

```
type code renames nat endtype
```

The base language does *not* provide a mechanism for defining parameterized types—this is left for the module system.

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):

```
function partition (x:int,xs:intlist) : (intlist,intlist) 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
```

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 (**etc**), 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)`.

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 $(\text{name} \Rightarrow \text{string}, \text{etc})$ is a record with at least one field 'name' of type string. This record type can be extended to many subtypes, for example $(\text{name} \Rightarrow \text{string}, \text{age} \Rightarrow \text{int}, \text{etc})$ or $(\text{name} \Rightarrow \text{string}, \text{age} \Rightarrow \text{int})$. 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 $(\text{none}, \text{int})$ 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 **gate(etc)** 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 $(x \Rightarrow 1.5, y \Rightarrow -3.14)$, $()$ or $(5, \text{nil}())$ (which is just syntax sugar for $(\$1 \Rightarrow 5, \$2 \Rightarrow \text{nil}())$).
- A constructor applied to a normal form, such as $\text{nil}()$ or $\text{cons}(5, \text{nil}())$.

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

```

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 $\text{cons}(?x, \text{any}:list)$ 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 $\text{cons}(\text{hd}, \text{tl})$.

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 `int` is a subtype of `float`, then we can construct a **case** statement to decide whether a value is an integer or not:

```

case x:float is
  any:int -> "integer"
  | any:float -> "noninteger"
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.
- 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 matches 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 E1 else E2 endif

```

can be expanded to:

```

case  $E$  is
  true ->  $E_1$ 
  | any:bool ->  $E_2$ 
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  $E_1$  then
   $E_2$ 
else
  if  $E_3$  then
     $E_4$ 
  else
     $E_5$ 
  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

```

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  $E_1$  endexn
  exception Bar is  $E_2$  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

```

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

```

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 ...
```

As an example of the imperative features, an imperative definition of quicksort partitioning is given in Figure A.1. It can be compared with the functional definition given in Figure A.2.


```

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

```

Figure A.1: The imperative version of partition

```

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

```

Figure A.2: The functional version of partition

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

```

```

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:

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

meaning 'in context C , expression E has result type T '. The context C gives the type for each of the free identifiers used in E , for example we can deduce:

$$x \Rightarrow \text{int}, * \Rightarrow (\text{int}, \text{int}) \rightarrow \mathbf{exit} (\text{int}) \vdash x * x \Rightarrow \mathbf{exit} (\text{int})$$

meaning 'in a context where x is an integer and $*$ is a function from pairs of integers to integers, then $x * x$ returns an integer'.

Expressions which assign to variables but do not return a result have typing given by judgements:

$$C \vdash E \Rightarrow \mathbf{exit} (V_1 \Rightarrow T_1, \dots, V_n \Rightarrow T_n)$$

meaning 'in context 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:

$$2 \Rightarrow \text{int} \vdash ?x:=2 \Rightarrow \mathbf{exit} (x \Rightarrow \text{int})$$

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:

$$C \vdash E \Rightarrow \mathbf{exit} (T, V_1 \Rightarrow T_1, \dots, V_n \Rightarrow T_n)$$

which combines the above two semantics. For example:

$$2 \Rightarrow \text{int}, * \Rightarrow (\text{int}, \text{int}) \rightarrow \mathbf{exit} (\text{int}) \vdash ?x:=2; x * x \Rightarrow \mathbf{exit} (\text{int}, x \Rightarrow \text{int})$$

meaning 'in a context where 2 is an integer and $*$ is a function from pairs of integers to integers, then $?x:=2; x * x$ assigns an integer to the variable x and returns an integer'.

Note that x is not free in the expression $?x:=2; x * x$ 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:

$$\vdash 2 * 2 \xrightarrow{\delta(4)} \mathbf{block}$$

meaning '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:

$$\mathcal{E} \vdash E \xrightarrow{\delta(V_1 \Rightarrow N_1, \dots, V_n \Rightarrow N_n)} E'$$

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:

$$\vdash ?x := 2 \xrightarrow{\delta(x \Rightarrow 2)} \mathbf{block}$$

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:

$$\mathcal{E} \vdash E \xrightarrow{\delta(N, V_1 \Rightarrow N_1, \dots, V_n \Rightarrow N_n)} E'$$

combining the two semantics, for example:

$$\vdash ?x := 2; x * x \xrightarrow{\delta(4, x \Rightarrow 2)} \mathbf{block}$$

Similarly, the semantics of exceptions is given by judgements:

$$\mathcal{E} \vdash E \xrightarrow{X(N)} E'$$

For example:

$$\mathbf{raise} X(1) \xrightarrow{X(1)} \mathbf{block}$$

The semantics is defined formally in Subclauses 7.2.8.14 and 10.2.27.

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.

$$B ::= \mathbf{stop} \mid \mathbf{exit} \mid \Pi[G^*] \mid G; B \mid \mathbf{i}; B \mid B \square B \mid B \mid B \mid [G^*] \mid B \mid \mathbf{hide} G^* \mathbf{in} B \mid B \gg B \mid B \triangleright B$$

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 declared 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: $\mathbf{i};B$ is a behaviour that first performs the internal action \mathbf{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} .
- Abstraction: **hide** \vec{G} **in** B hides in behaviour B all the actions from the set \vec{G} , i.e. it renames them into \mathbf{i} .
- Enabling: $B_1 \gg B_2$ 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]
```

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 (**etc**), 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:

```
process Router [inp: (de=>dest, etc) ,left,right] is
  var destination: dest, data: (etc)
  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=>!"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)
```

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 G1#n1, ..., Gp#np in
  [Γ1] for B1
|| ...
|| [Γn] for Bn
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* ($G_1\#n_1, \dots, G_p\#n_p$) 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

```

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
     $G$ ;
     $?x := \mathbf{any}$  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 [17] 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.

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*.

The behaviour:

```
hide  $G$  in
   $G@?t[t \geq 3]$ ;  $B$ 
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 continuously 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

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:

wait(3); B_1 [X > **wait**(1); i; **wait**(2); **raise**(X)

and not in

wait(1); B_1 [X > **wait**(1); i; **wait**(2); **raise**(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 [X > B_2 ; **raise**(X)

B_1 [X > B_2 ; **signal**(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:

$(B_1 [X_1 > G_1; G'_1; \mathbf{raise}(X_1) \mid \mid B_2 [X_2 > G_2; G'_2; \mathbf{raise}(X_2)]$
 $\mid [G_1, G_2, G'_1, G'_2] \mid$
 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

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

G (x=>int) **is** G'

in

B

endren

We can remove a field from a gate:

rename

G (x=>?i:int, y=>any:bool) **is** G' (!i)

in

B

endren

We can add a field to a gate:

```

rename
   $G(x \Rightarrow ?i : \text{int})$  is  $G'(x \Rightarrow !i, y \Rightarrow !\text{true})$ 
in
   $B$ 
endren

```

We can merge two gates G' and G'' into a single gate G :

```

rename
   $G'(x \Rightarrow ?i : \text{int})$  is  $G(x \Rightarrow !i, y \Rightarrow !\text{true})$ 
   $G''(x \Rightarrow ?i : \text{int})$  is  $G(x \Rightarrow !i, y \Rightarrow !\text{false})$ 
in
   $B$ 
endren

```

We can also split one gate G into two gates G_1 and G_2 :

```

rename
   $G(x \Rightarrow ?i : \text{int}, y \Rightarrow !1)$  is  $G_1(!i)$ 
   $G(x \Rightarrow ?i : \text{int}, y \Rightarrow !2)$  is  $G_2(!i)$ 
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 : (etc)
  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:

$$C \vdash B \Rightarrow \text{exit} (\vec{V} \Rightarrow \vec{T})$$

For example:

$$G \Rightarrow \text{gate any} \vdash (G(?x : \text{int}) \mid \mid G(?y : \text{bool})) \Rightarrow \text{exit} (x \Rightarrow \text{int}, y \Rightarrow \text{bool})$$

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:

$$C \vdash B \Rightarrow \mathbf{guarded} (\vec{V} \Rightarrow \vec{T})$$

And of course, the following rule is valid:

$$\frac{C \vdash B \Rightarrow \mathbf{guarded} (\vec{V} \Rightarrow \vec{T})}{C \vdash B \Rightarrow \mathbf{exit} (\vec{V} \Rightarrow \vec{T})}$$

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):

$$\begin{array}{l} \mathbf{i}; G(?t); \mathbf{wait}(t) \xrightarrow{\mathbf{i}()} G(?t); \mathbf{wait}(t) \\ \xrightarrow{G(3)} ?t:=3; \mathbf{wait}(3) \\ \xrightarrow{\varepsilon(2)} ?t:=3; \mathbf{wait}(1) \\ \xrightarrow{\varepsilon(1)} ?t:=3; \mathbf{wait}(0) \\ \xrightarrow{\delta(t \Rightarrow 3)} \mathbf{block} \end{array}$$

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:

$$?t:=3 \xrightarrow{\delta(t \Rightarrow 3)} \mathbf{block}$$

but:

$$?t:=3 \not\xrightarrow{\varepsilon(d)}$$

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:

$$?t:=3 \parallel \mathbf{wait}(2); ?y:=\mathbf{true}$$

We would like this to have semantics (up to strong bisimulation):

$$\begin{array}{ccc}
 ?t:=3 \parallel \mathbf{wait}(2);?y:=\mathbf{true} & \xrightarrow{\varepsilon(1)} & ?t:=3 \parallel \mathbf{wait}(1);?y:=\mathbf{true} \\
 & \xrightarrow{\varepsilon(1)} & ?t:=3 \parallel \mathbf{wait}(0);?y:=\mathbf{true} \\
 & \xrightarrow{\delta(t \Rightarrow 3, y \Rightarrow \mathbf{true})} & \mathbf{block}
 \end{array}$$

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 [20].

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

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

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 expressions.
- 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.
- 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.

The abstract syntax contains: specification declaration, module declaration, module expressions, generic module declarations, interface declarations, interface expressions, and declarations.

The static semantics of this language is formally defined. It is based on judgments such as $\mathcal{B} \vdash top\text{-}dec \Rightarrow C$ meaning ‘in the context \mathcal{B} of top-level declarations, the top level definition $top\text{-}dec$ is well formed and gives the context C ’. For example:

$$\vdash (\text{module OnePoint is type M is (x:int, y:int)}) \Rightarrow (\text{OnePoint} \Rightarrow \{M \Rightarrow \text{type, } M \equiv (x \Rightarrow \text{int}, y \Rightarrow \text{int})\})$$

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 [14] and on previous proposals for LOTOS with a module system [24].

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 [22, 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 (zero(), zero()) is
      (!x, !y) -> 0
    endcase
  endfunc
endmod
```

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: (int,int), 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: (int,int), 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 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 endgen
```

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.

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.

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) endtype
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) endtype
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, ..., *int-exp*_{*n*}” clause.
- Importation of modules into modules and generic modules using “**imports** *mod-exp*₁, ..., *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

```

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 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, arrays and sets are frequently used in specifications some rich syntax for them is suitable (see A.2.1.8).

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 `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

```

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, which need to be resolved:

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),
3. the relationship between generic modules and ACT ONE parameterized types and type renaming should be clarified.

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, and the relationship with them should be clarified.

For each module:

- We should give an interface, a module, and (where necessary) the dynamic semantics for the module.
- We should specify if the module is *pervasive* or not. A module is pervasive if it is available everywhere without explicit import reference. The identification of pervasive modules will be the subject of further discussions.
- We should specify whether the module will be defined with genericity.
- We should specify whether the types and functions contained in the module will be defined using the base language, or if they will be implemented externally (e.g. real or floating-point numbers).

In the present paper, the built-in data types and the rich term syntax are not described in detail. In the given examples, the implementation parts are often omitted and only interfaces are provided. A detailed specification should be provided maybe based on existing proposals, for example [10] [21].

We propose the following *predefined types*: the type 'int', the type 'real', 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, subrange types, record types, arrays, 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

$$\text{OrdItem} \Rightarrow \{ \text{item} \Rightarrow \text{int}, \text{leq} \Rightarrow [] \text{ (i} \Rightarrow \text{item, j} \Rightarrow \text{item)} [] \rightarrow \mathbf{exit}(\text{bool}) \}$$

In this example we suppose that integers are pervasive.

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

$$\text{Monoid} \Rightarrow \{ M \Rightarrow \mathbf{type}, 0 \Rightarrow M, + \Rightarrow [] \text{ (M, M)} [] \rightarrow \mathbf{exit}(M) \}$$

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* **endgen**

is

$$\text{gen-id} \Rightarrow (\text{mod-id} \Rightarrow C) \rightarrow C'$$

where *C* is the binding (type) of the interface identifier *int-id*, and *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 (*mod-id* \Rightarrow *C*)), and then deriving the result from *C'*, by instantiation of the formal parameters used in the generic module body.

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.

- 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) 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
```

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 [18].

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 give 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.

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.3 summarize the interactions of a trader and its clients.

Figure A.3: 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.

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:

```
type InvocationSig is
  ExportOfferInv ( serviceDescription => Servicedescription,
                  servicePropValues => PropertyValueList,
                  offerPropValues => PropertyValueList,
                  serviceInterfaceId => InterfaceId )
  | ImportOfferInv (serviceDescription => ServiceDescription,
                  matchingCriteria => Rule,
                  preferenceCriteria => Rule,
                  orderingRequirementList => OrderRequirementList,
                  servicePropertiesOfInterest => PropertiesOfInterest,
                  offerPropertiesOfInterest => PropertiesOfInterest )
  | AddLinkInv ( newLinkName => name,
                linkPropValues => PropertyValueList,
                targetInterfaceId => InterfaceId )
  | ... (* other invocation signatures *)
endtype
```

```
type TerminationSig is
  ExportOfferTer (offerId => ServiceOfferId)
  | ImportOfferTer ( detailsOfServiceOffers => ServiceOfferDetailList )
  | AddLinkTer ( linkId => 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
  (interfaceId => InterfaceIdentifier,
   originatorId => InterfaceIdentifier,
   invocation => InvocationSig)
```

```
endtype
```

```
type Termination is
  (interfaceId => InterfaceIdentifier,
   originatorId => InterfaceIdentifier,
   termination => TerminationSig)
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.

```
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 hidden 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: stateAccessCh ]
  ( 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,
- 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 is

```

```
String
endtype

type InterfaceId is
  Integer
endtype

type InterfaceIdList is
  List (* of InterfaceId *)
endtype

type GraphEdgeSpec is
  (linkId => LinkId,
   edgeName => Name,
   linkPropertyValues => PropertyValueList
   toNodeLocation => InterfaceId)
endtype

type OfferPartitionSubgraph is
  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
  (orderDirection => OrderType,
   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

type RuleList
  List (* of Rule *)
endtype

type PropertyValue is
  (name => PropertyName,
   value => LiteralPropertyValue)
endtype

type PropertyNameList is
  List (* of PropertyName *)
endtype

type PropertiesOfInterest is
  ALL
  | Interesting ( PropertyNameList )
endtype

type PropertyValueList is
  List (* of PropertyValue *)
endtype

type ServiceDescription is
  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 is
  List (* of QualificationCode *)
endtype

type ServiceOfferDetail is
  (interfaceId => InterfaceId,
```

```

    servicePropertyValues => PropertyValueList,
    serviceOfferPropertyValues => PropertyValueList,
    offerQualificationList => Qualifier)
endtype

type ServiceOfferDetailList is
    List (* of ServiceOfferDetail *)
endtype

type ServiceOfferId is
    integer
endtype

type ServiceOfferIdList is
    List (* of ServiceOfferId *)
endtype

type ServiceOfferDescription is
    (serviceOfferId => ServiceOfferId,
    serviceDescription => ServiceDescription,
    InterfaceId => InterfaceId,
    servicePropertyValues => PropertyValueList,
    serviceOfferPropertyValues => PropertyValueList)
endtype

type ServiceOfferSpace is
    List (* of ServiceOfferDescription *)
endtype

typeLinkId is
    Integer
endtype

typeLinkIdList is
    List (* of LinkId *)
endtype

type InvocationSig is
    ExportOfferInv ( serviceDescription => Servicedescription,
                    servicePropValues => PropertyValueList,
                    offerPropValues => PropertyValueList,
                    serviceInterfaceId => InterfaceId )
    | WithdrawOfferInv ( offerId => ServiceOfferId )
    | ModifyOfferInv ( offerId => ServiceOfferId,
                    servicePropValues => PropertyValueList,
                    offerPropValues => PropertyValueList )
    | ImportOfferInv (serviceDescription => ServiceDescription,
                    matchingCriteria => Rule,
                    preferenceCriteria => Rule,
                    orderingRequirementList => OrderRequirementList,
                    servicePropertiesOfInterest => PropertiesOfInterest,
                    offerPropertiesOfInterest => PropertiesOfInterest )

```



```

| QueryOfferPartitionSubgraphInv
| DescribeOfferInv ( serviceOfferIdList => ServiceOfferIdList,
                    servicePropertiesOfInterest => PropertiesOfInterest,
                    offerPropertiesOfInterest => PropertiesOfInterest )
| AddLinkInv ( newLinkName => name,
               linkPropValues => PropertyValueList,
               targetInterfaceId => InterfaceId )
| RemoveLinkInv ( linkId => LinkId )
endtype

```

```

type TerminationSig is
  ExportOfferTer (offerId => ServiceOfferId)
| WithdrawOfferTer
| ModifyOfferTer
| ImportOfferTer ( detailsOfServiceOffers => ServiceOfferDetailList )
| QueryOfferPartitionSubgraphTer
  (offerPartitionSubgraph => OfferPartitionSubgraph)
| DescribeOfferTer ( detailsOfServiceOffers => ServiceOfferDetailList )
| AddLinkTer ( linkId => LinkId )
| RemoveLinkTer
| Error (err => TraderErrorExceptions)
endtype

```

```

type Invocation is
  (interfaceId => InterfaceId,
   originatorId => InterfaceId,
   invocation => InvocationSig)
endtype

```

```

type Termination is
  (interfaceId => InterfaceId,
   originatorId => InterfaceId,
   termination => TerminationSig)
endtype

```

```

type stateAccess is (stateAcc) endtypechan

```

```

type stateAcc 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 (reason => string)
| UndefinedProperty ( pName => PropertyName )
| BadOfferIdentity ( offerId => 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]
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 ->
    raise err ( UndefinedProperty(pName) )
  Cons ( (!pName, ?v), any:PropertyValueList ) ->
    v
  Cons ( any:PropertyValue, ?tl ) ->
    propValueOf ( pName, tl ) [err]
endcase
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)
        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,
                      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
                          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
  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), ?t1) ->
      var l, l1: LiteralPropertyValue
      in
        ?l := propValueOf (pname, vs) [err];
        ?l1 := propValueOf (pname, v1s) [err];

```



```

        l < l1 orelse ( l = l1 andalso IsOrdered (t1, d, d1) )
    endvar
    Cons ((DESCENDING, ?pname), ?t1) ->
    var l, l1: LiteralPropertyValue
    in
        ?l := propValueOf (pname, vs) [err];
        ?l1 := propValueOf (pname, v1s) [err];
        l > l1 orelse ( l = l1 andalso IsOrdered (t1, 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)
    |||
    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*)

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
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
    |||
    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

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, offerIds)] ;
            ?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)
      |||
      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.

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 evolves 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 μ instruction 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]):

<pre>(*LOTOS*) process Vending [coin,candy1,candy2] : noexit := coin; (candy1; Vending [coin,candy1,candy2] []candy2; Vending [coin,candy1,candy2]) endproc</pre>	<pre>(*E – LOTOS*) process Vending [coin,candy1,candy2] is coin; (candy1; Vending [...] []candy2; Vending [...]) endproc</pre>
--	---

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  $\Rightarrow$  middle, ...]
    | [middle] |
    SingleBuffer [inp  $\Rightarrow$  middle, ...]
  endhide
endproc
```

B.1.2 Basic LOTOS

Enabling and exit In E-LOTOS, enabling operator (\gg) 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  $\gg$  c; stop  $\Rightarrow$  a;b;c; stop
a; b; (exit  $\gg$  c; stop)  $\Rightarrow$  a; b; c; stop
```


Thus **exit** becomes obsolete, and it is removed from E-LOTOS.

Operator associativity In E-LOTOS, all operators have the same precedence, but action prefix “;”, which has the higher precedence. Associativity is left-handed, as in LOTOS. However, it is not possible to mix different operators in the same expression without explicitly specifying its associativity. 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:

$$(B_1 [> B_2) ||| B_3 \quad \text{or} \quad \mathbf{dis} \ B_1 [> B_2 \ \mathbf{enddis} \ ||| \ B_3$$

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 ($([G_1, \dots, G_n])$)
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:

$$B_1 \text{ op } B_2 \text{ op } B_3 \equiv B_1 \text{ op } (B_2 \text{ op } B_3)$$

also equivalent to:

$$\textit{op-begin-keyword} \ B_1 \ \textit{op} \ B_2 \ \textit{op} \ B_3 \ \textit{op-end-keyword}$$

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*)	(*E – LOTOS*)
type Boolean is	interface Boolean is
sorts bool	type bool is
opns	true, false
true, false: \rightarrow bool	endtype
and: bool, bool \rightarrow bool	function infix and(a: bool, b: bool): bool
endtype	endint

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:

<pre> (*LOTOS*) type NaturalNumber is boolean ... endtype </pre>	<pre> (*E – LOTOS*) interface NaturalNumber imports Boolean is ... endint module NaturalNumber imports Boolean is ... endmod </pre>
---	--

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 11). 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:

```
(*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
  opnames 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
  opnames
    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.

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).

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, *Proceeding 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] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [12] 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.

- [13] 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.
- [14] 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.
- [15] Alan Jeffrey and Guy Leduc. E-LOTOS core language. Chapter 3 of [23], 1996.
- [16] L. Léonard and G. Leduc. A formal definition of time in LOTOS. *Formal Aspects of Computing*, 1998.
- [17] 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.
- [18] Giovanni 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.
- [19] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] 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.