

INTERNATIONAL STANDARD

**ISO
8807**

First edition
1989-02-15

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

*Systèmes de traitement de l'information — Interconnexion de systèmes ouverts —
LOTOS — Technique de description formelle basée sur l'organisation temporelle de
comportement observationnel*



Reference number
ISO 8807 : 1989 (E)



Documents

Document Sharing Hub

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

Draft International Standards adopted by the technical committees are circulated to the member bodies for approval before their acceptance as International Standards by the ISO Council. They are approved in accordance with ISO procedures requiring at least 75 % approval by the member bodies voting.

International Standard ISO 8807 was prepared by Technical Committee ISO/TC 97, *Information processing systems*.

Users should note that all International Standards undergo revision from time to time and that any reference made herein to any other International Standard implies its latest edition, unless otherwise stated.

Annex A forms an integral part of this International Standard. Annexes B, C, D and E are for information only.

CONTENTS

	Page
0 Introduction	1
0.1 General	1
0.2 FDTs	1
0.3 The requirement for standard FDTs	1
0.4 The objectives to be satisfied by an FDT	1
0.5 The origin of LOTOS	2
0.6 The structure of this International Standard	2
1 Scope and field of application	3
2 References	3
3 Conformance	3
4 Basic mathematical concepts and notation	5
4.1 General	5
4.2 Sets	5
4.3 Lists	6
4.4 Strings	6
4.5 Relations and functions	6
4.6 Backus-Naur Form	7
4.7 Syntax-directed definitions	8
4.8 Derivation systems	10
5 Model	13
5.1 Introduction	13
5.2 Many-sorted algebras	13
5.3 Labelled transition systems	13
5.4 Structured labelled transition systems	14
6 Formal Syntax	15
6.1 Lexical tokens	15
6.1.1 General	15
6.1.2 Basic characters	15
6.1.3 Reserved symbols	15
6.1.3.1 Word symbols	15
6.1.3.2 Special symbols	17
6.1.4 Identifiers	17
6.1.5 Requirement	18
6.1.6 Comments	18
6.1.7 Token separators	18
6.1.8 Requirement	18
6.2 Specification text	18
6.2.1 specification	18
6.2.2 definition-block	18
6.2.3 data-type-definitions	19
6.2.4 p-expressions	19
6.2.5 sorts, operations and equations	19
6.2.6 process-definitions	20
6.2.7 behaviour-expressions	20
6.2.7.1 general structure	21
6.2.7.2 local-definition-expressions	21
6.2.7.3 sum-expressions	21

6.2.7.4	par-expressions	21
6.2.7.5	hiding-expressions	21
6.2.7.6	enable-expressions	21
6.2.7.7	disable-expressions	22
6.2.7.8	parallel-expressions	22
6.2.7.9	choice-expressions	22
6.2.7.10	guarded-expressions	22
6.2.7.11	action-prefix-expressions	22
6.2.7.12	atomic-expressions	22
6.2.8	value-expressions	23
6.2.9	declarations	23
6.2.10	special-identifiers	23
7	Semantics	25
7.1	Introduction	25
7.1.1	Structure of the static semantics definition	25
7.1.2	Structure of the dynamic semantics definition	25
7.1.3	Structure of clause 7	26
7.2	General structures and definitions	26
7.2.1	Names and related functions	26
7.2.2	Algebraic specifications, terms, and equations	27
7.2.2.1	Signature	27
7.2.2.2	Terms	27
7.2.2.3	Equations	27
7.2.2.4	Conditional equations	28
7.2.2.5	Algebraic specifications	28
7.2.3	Canonical specifications	28
7.2.3.1	Introduction	28
7.2.3.2	Behaviour-expression-structure	28
7.2.3.3	Behaviour specification	29
7.2.3.4	Canonical LOTOS specification	29
7.3	Static semantics	29
7.3.1	Introduction	29
7.3.2	General structures and definitions	30
7.3.2.1	Scope	30
7.3.2.2	Extended identifiers	30
7.3.2.3	The interpretation of extended identifiers	31
7.3.2.4	Functionality	32
7.3.2.5	Data-presentation	32
7.3.2.6	Parameterized data-presentation	32
7.3.2.7	Non-overlapping data-presentation	33
7.3.2.8	Signature morphism	33
7.3.2.9	Data-presentation morphism	33
7.3.2.10	Environments	34
7.3.2.11	Standard library	34
7.3.2.12	Complete data-presentation	35
7.3.2.13	Valid dependence order	35
7.3.3	Reconstruction of terms	35
7.3.3.1	Value-atoms	36
7.3.3.2	Value-atom, position, and argument-list of a value-expression	36
7.3.3.3	Operation-assignment	37
7.3.3.4	Consistent operation-assignment	37
7.3.3.5	Explicit sort indication	37
7.3.3.6	Sound operation-assignment	37
7.3.3.7	Generated operation-assignments	37

7.3.3.8	Scopes of an operation-assignment	38
7.3.3.9	Minimal operation-assignment	38
7.3.3.10	Reconstruction of a term	38
7.3.4	Flattening of a LOTOS specification	39
7.3.4.1	Introduction	39
7.3.4.2	Flattening of specification	39
7.3.4.3	Flattening of data-type-definitions	40
7.3.4.4	Flattening of process-definitions	48
7.3.4.5	Flattening of behaviour-expressions	49
7.3.4.6	Flattening of identifiers	55
7.3.5	Functional structure of the flattening function	56
7.4	Semantics of data-presentations	60
7.4.1	General	60
7.4.2	The derivation system of a data-presentation	60
7.4.2.1	Axioms generated by equations	60
7.4.2.2	Inference rules generated by equations	60
7.4.2.3	Generated derivation system	60
7.4.3	Congruence relation induced by a data-presentation	61
7.4.4	Quotient term algebra	61
7.5	Semantics of a canonical LOTOS specification	61
7.5.1	General	61
7.5.2	Auxiliary definitions	61
7.5.2.1	Notation	61
7.5.2.2	Extended behaviour-expressions	62
7.5.2.3	The simplification of sum- and par-expressions	62
7.5.2.4	Substitution	63
7.5.3	Transition derivation system	63
7.5.3.1	General framework	63
7.5.3.2	Axioms of transition	63
7.5.3.3	Inference rules of transition	64
7.5.4	Structured labelled transition system of a behaviour-expression	69
7.5.4.1	Derivatives of a behaviour-expression	69
7.5.4.2	Structured labelled transition system	69
7.5.5	Formal interpretation of a canonical LOTOS specification	70

ANNEXES

A	Standard library of data types	71
A.1	Introduction	71
A.2	Syntax of the data type library	71
A.3	Semantics of the data type library	71
A.4	The Boolean data type	72
A.5	Parameterized data type definitions	73
A.5.1	Element	73
A.5.2	Set	74
A.5.3	Strings	76
A.5.3.1	Non-empty string	76
A.5.3.2	String	78
A.6	Unparameterized data type definitions	79
A.6.1	Natural number	79
A.6.1.1	Abstract definition of natural numbers	79
A.6.1.2	Representations of natural numbers	80
A.6.1.2.1	Hexadecimal representation	80
A.6.1.2.2	Decimal representation	82

A.6.1.2.3	Octal representation	83
A.6.1.2.4	Binary representation	84
A.6.2	Octet	85
A.6.3	Octet string	85
B	Equivalence relations	87
B.1	Introduction	87
B.2	Weak bisimulation	88
B.2.1	Definitions	88
B.2.2	Laws for weak bisimulation congruence	89
B.2.3	Laws for weak bisimulation equivalence	92
B.2.3.1	Notation	92
B.2.3.2	General law	92
B.2.3.3	Rules for $\approx$	92
B.3	Testing equivalence	92
B.3.1	Definitions	92
B.3.2	Laws for testing congruence	93
B.3.3	Laws for testing equivalence	93
B.4	References.	94
C	A tutorial on LOTOS	95
C.1	The specification of processes	95
C.2	Behaviour expressions in basic LOTOS	96
C.2.1	A basic process: inaction	96
C.2.2	Two basic operators	96
C.2.2.1	Action prefix	96
C.2.2.2	Choice	96
C.2.2.3	Processes as trees	97
C.2.3	Recursion	98
C.2.4	Parallelism	99
C.2.4.1	Parallelism of independent processes	99
C.2.4.2	Parallelism of dependent processes	99
C.2.4.3	The general parallel operator	100
C.2.4.4	The hiding operator	101
C.2.4.5	Reasons for the hiding operator	102
C.2.5	Nondeterminism in LOTOS	103
C.2.6	Sequential composition of processes	104
C.2.7	Disruption of processes	105
C.2.8	An example in basic LOTOS	106
C.3	LOTOS data types	107
C.3.1	Introduction	107
C.3.1.1	Basic concepts	107
C.3.1.2	Abstract Data Types versus Concrete Data Types	107
C.3.2	Concepts of LOTOS data types	108
C.3.2.1	The signature	108
C.3.2.2	Terms and equations	108
C.3.2.3	The combination	110
C.3.2.4	The parameterization	111
C.3.2.5	Renaming	113
C.3.2.6	Library invocation	114
C.4	LOTOS with structured interactions	114
C.4.1	Structured event offers	114
C.4.1.1	Value declarations	114
C.4.1.2	Variable declarations	115
C.4.1.3	Types of interaction	115
C.4.2	Conditional constructs	116
C.4.2.1	Selection predicates	116

C.4.2.2	Guarded expressions	117
C.4.3	Process abstraction with parameterization	117
C.4.4	Generalized choice and parallel expressions	118
C.4.5	Generalized sequential composition	119
C.4.5.1	Successful termination and functionality	120
C.4.5.2	Functionality of LOTOS behaviour expressions	120
C.4.5.3	Process abstraction and functionality	121
C.4.5.4	Sequential composition with value passing	122
C.4.5.5	Local variable definition	122
C.4.6	Another example in LOTOS	122
C.5	LOTOS syntax table	123
D	Syntax diagrams	127
E	Informal basis for abstract data types	135
E.1	Introduction	136
E.1.1	Representations	136
E.2	Signatures	138
E.3	Terms and expressions	139
E.3.1	Generation of terms	139
E.4	Values and algebras	140
E.4.1	Equations and quantification	141
E.5	Algebraic specification and semantics	141
E.6	Representation of values	142
 TABLES		
1	Metalanguage symbols	8
2	Actualization function	43
3	Functional structure of flattening function	56
4	Interaction types	116
5	LOTOS syntax table	123
 FIGURES		
1	Structure of clause 7	26
2	Two interacting processes	95
3	Full duplex buffer	96
4	Composition of two buffer processes	100
5	Hiding	101

This page intentionally left blank

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

0 Introduction

0.1 General

Formal description techniques (FDTs) are methods of defining the behaviour of an (information processing) system in a language with a formal syntax and semantics, instead of a natural language such as English. In the following sub-clauses of this introduction, the importance of FDTs and their standardization is discussed. The objectives that an FDT must satisfy are considered. The origin of LOTOS is discussed. Finally the structure of this document is explained.

0.2 FDTs

Formal description techniques are important tools for the design, analysis and specification of information processing systems. It is by means of formal techniques that system descriptions can be produced that are *complete, consistent, concise, unambiguous* and *precise*. This is only possible if an FDT is self-contained, so that the descriptions given in an FDT need not refer to any informal knowledge of the system that is described. An important aspect of a formal system is that it allows analysis by mathematical methods. An FDT that has such a formal, mathematical basis can be used to prove the correctness of specifications.

0.3 The requirement for standard FDTs

If an FDT is defined in an International Standard, the description is available to all who require it. The Directives for the production of such a standard require a high degree of international acceptance and technical stability. Any amendment also requires international agreement. Hence a standard FDT offers the most useful form of presentation to those who wish to apply it.

0.4 The objectives to be satisfied by an FDT

Although this document describes an FDT that is generally applicable to distributed, concurrent information processing systems, it has been developed particularly for OSI. The main objectives to be satisfied by such an FDT are that it should be

- a) *expressive*: an FDT should be able to define both the protocol specifications and the service definitions of the seven layers of OSI described in ISO 7498.
- b) *well-defined*: an FDT should have a formal mathematical model that is suitable for the analysis of these specifications and definitions. The same model should support the checking of conformance of implementations that are permitted by the OSI International Standards. This model should also support the testing of an implementation for conformance.
- c) *well-structured*: an FDT should offer means for structuring the description of a specification or definition in manner that is meaningful and intuitively pleasing. A good structure increases the readability, understandability, flexibility, and maintainability of system descriptions, and offers a better framework for their analysis.
- d) *abstract*: there are two aspects of abstraction that an FDT should offer:
 - 1) an FDT should be completely independent of methods of implementation, so that the technique itself does not provide any undue constraints on implementors
 - 2) an FDT should offer the means of abstraction from irrelevant details with respect to the context at any point in a description. Abstraction can reduce the local complexity of system descriptions considerably. In the presence of a good structure, abstraction can help even further to reduce the complexity of descriptions.

0.5 The origin of LOTOS

LOTOS (Language of Temporal Ordering Specification) was developed by FDT experts from ISO/TC97 during the years 1981-1988. The basic idea that LOTOS developed from was that systems can be described by defining the temporal relation between events in the externally observable behaviour of a system. LOTOS has two components. The first component deals with the description of process behaviours and interactions, and is based on a modification of the Calculus of Communicating Systems (CCS), which was developed at the University of Edinburgh. The modification includes elements that were introduced in other calculi, which are related to CCS, viz. CSP and CIRCAL. Among the other theories that are related to CCS, and thus to LOTOS, are SCCS, MEIJE and ACP. CCS, and the related formal systems, provide a powerful analytical theory for concurrent processes.

The second component deals with the description of data structures and value expressions and is based on the abstract data type language ACT ONE. ACT ONE was developed at the Technical University of Berlin. The part of LOTOS dealing with the description of processes, i.e. dynamic behaviours, is not dependent upon ACT ONE. Many well-defined languages for the description of data structures could, in principle, be used in combination with the process definition facilities of LOTOS.

0.6 The structure of this International Standard

This document differs in contents from most International Standards. The importance of a formal, mathematical basis of an FDT (see clauses 0.2 and 0.4 b)) makes the inclusion of mathematical material in this definition necessary. Clause 4 introduces some fundamental mathematical concepts and notations that are used in the rest of the document. Clause 5 presents the fundamental mathematical structures that provide a semantic basis for LOTOS data types, behaviour expressions, and their combination. Clause 6 presents the syntax of the language and contains, together with 7.3, the rules for producing syntactically correct specifications; this part of the document requires knowledge of only basic mathematical concepts. Clause 7 presents the semantics of a LOTOS specification, based on the semantics of data types and behaviour expressions. Annex A contains the standard library of LOTOS data types and forms a part of this International Standard. The other annexes provide more information related to LOTOS, but do not form a part of the standard. In particular, annex C contains a *tutorial* on LOTOS, which is meant to provide a guide to the features of the language, and a convenient introduction to this standard for the non-technical reader.

1 Scope and field of application

This International Standard defines the syntax and semantics of the Formal Description Technique LOTOS. LOTOS is in general used for the formal description of distributed, concurrent information processing systems. In particular LOTOS can be used to describe formally the service definitions and protocol specifications of the layers of Open Systems Interconnection (OSI) architecture described in ISO 7498, and related standards, and conformance tests for implementations of OSI protocols and/or OSI functions. It can also be applied for the formal description of other distributed systems, such as telephone switching networks.

2 References

ISO 7498, *Information processing systems - Open Systems Interconnection - Basic Reference Model*.

CCITT Recommendation Z.100, *SDL*.

3 Conformance

A formal specification written in LOTOS conforms to the requirements of this International Standard if and only if it is derivable according to the syntactic rules defined in clause 6, and unambiguously defines a behaviour according to the semantics defined in clause 7.

This page intentionally left blank

4 Basic mathematical concepts and notation

This clause contains a list of basic mathematical concepts and related notations used in clauses 5,6, and 7.

4.1 General

=df

is defined as.

iff

if and only if, i.e. double implication.

4.2 Sets

$\{a,b,c,\dots\}$

the *set* made up of elements $a,b,c,\dots$. The order in which the elements are listed is immaterial.

$\emptyset$

the *empty set*, i.e. the set having no elements.

$x \in A$

x is an *element* of the set A .

$x \notin A$

x is *not* an *element* of the set A .

$A \subseteq B$

A is a *subset* of B , i.e. all elements of A are also elements of B .

$A \cup B$

the *union* of A and B , i.e. the set which contains only all elements of A and all elements of B .

$\bigcup A$

the *union* of A , i.e. the set which contains only all elements of the elements of A (A must therefore be a set of sets).

$A \cap B$

the *intersection* of A and B , i.e. the set which contains only all elements of A which are also elements of B .

sets A and B are *disjoint* iff $A \cap B = \emptyset$.

$A - B$

the *difference* of A and B , i.e. the set which contains only all elements of A which are not also elements of B .

$A \times B$

the *Cartesian product* of A with B , i.e. the set of all ordered pairs $\langle a,b \rangle$, such that $a \in A$ and $b \in B$.

$A_1 \times A_2 \times \dots \times A_n$

the *generalized Cartesian product* of $A_1, A_2, \dots, A_n$, i.e. the set of ordered n -tuples $\langle a_1, a_2, \dots, a_n \rangle$ (see 4.3), such that $a_1 \in A_1$, $a_2 \in A_2$, ..., $a_n \in A_n$.

$\{x \in A \mid Q(x)\}$

the set which contains only all those elements of A which satisfy property Q (the abbreviation $\{x \mid Q(x)\}$ is used where set A may be deduced from the context).

4.3 Lists

$a_1, \dots, a_n$, or $\langle a_1, \dots, a_n \rangle$

the finite list (or sequence, or (n) -tuple) made up of the *elements*, or *components* $a_1, \dots, a_n$. Unlike sets, lists may contain more than one instance of the same element, since elements are distinguished by their position in the ordering of the list;

an *ordered pair* is a list of two elements (e.g. $\langle a, b \rangle$);

the *empty* list has no elements and is denoted by $\langle \rangle$.

a *record* is an n -tuple of which each element is *labelled* with a unique label. If *lab* is the label of element x of record y then $y.lab$ denotes x .

$\langle a_1, \dots, a_n \rangle \cup \langle b_1, \dots, b_n \rangle$

the componentwise union of lists, defined as:

$\langle a_1 \cup b_1, \dots, a_n \cup b_n \rangle$ ($a_1, \dots, a_n, b_1, \dots, b_n$ must therefore be sets or, again lists).

$\bigcup A$

The *union* (of the lists in) A , i.e. the componentwise union of only all elements of A (A must therefore be a set of n -tuples for a fixed n).

A^n

the set of all n -tuples with elements in A for fixed n .

A^*

is equivalent with $\bigcup \{A^n \mid n \in \mathbf{N}\}$, where $\mathbf{N}$ is the set of natural numbers.

4.4 Strings

$a_1 \dots a_n$

the *string* made up of the elements $a_1, \dots, a_n$. A string is formed by the juxtaposition of its elements.

$s.t$

the *concatenation* of the strings s and t . The concatenation is the string that consists of the elements of s followed by those of t in the same order.

A^*

the set of all finite strings consisting of elements in the set A . This also includes the *empty string* ϵ , that has no elements.

$s \mid A$

the *restriction* of a string s to a set A is the string that consists of only all elements of s that are in A , in the order of their occurrence in s .

4.5 Relations and functions

$R \subseteq A \times B$

R is a *binary relation* between A and B , i.e. a set of elements of $A \times B$;

the *domain* of R is defined as $\{a \in A \mid \text{there exists some } b \in B \text{ such that } \langle a, b \rangle \in R\}$;

the *range* of R is defined as $\{b \in B \mid \text{there exists some } a \in A \text{ such that } \langle a, b \rangle \in R\}$;

if R is a binary relation between A and B , and S is a binary relation between B and C , their *composition* $R.S$ is the binary relation between A and C containing all and, only the pairs $\langle a, c \rangle$ such that there exists some $b \in B$ with $\langle a, b \rangle \in R$ and $\langle b, c \rangle \in S$.

$f : A \rightarrow B$

f is a (*partial*) *function* from A to B , i.e. f is a binary relation between A and B such that for each $a \in A$, there exists at most one $b \in B$ such that $\langle a, b \rangle \in f$;

if $\langle a, b \rangle \in f$ then f is *defined* for a , and one may write $f(a) = b$;

the function f is *total* iff it is defined for all $a \in A$;

function composition is a special case of composition of binary relations, viz. when both relations are functions;

a function $f : A \rightarrow B$ is *injective* iff, for all a_1, a_2 in the domain of f , $f(a_1) = f(a_2)$ implies that $a_1 = a_2$;

the *inverse function*, $f^{-1} : B \rightarrow A$, of an injective function is defined as follows: for $b \in B$, if there exists an $a \in A$ such that $f(a) = b$, then $f^{-1}(b) = a$, otherwise f^{-1} is not defined; in symbols:

$f^{-1} : B \rightarrow A =_{df} \{ \langle b, a \rangle \mid f(a) = b \}$;

a function $f : A \rightarrow B$ is '*one-to-one*' iff it is injective and its inverse function is total.

4.6 Backus-Naur Form

The metalanguage used in this International Standard to specify the syntax of LOTOS is based on Backus-Naur Form (BNF). A BNF-like description of a language L is given by a set of *productions*, or *rewrite rules*. The meta-symbols used to compose rewrite rules are listed in table 1.

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

A rewrite rule has the format:

nonterminal-symbol = meta-expression .

where the meta-expression is an expression constructed using terminal and nonterminal symbols, and the operators listed in table 1. A meta-expression contains at least one terminal or non-terminal symbol. Adjacent terminal and/or nonterminal symbols occurring in a meta-expression denote the concatenation of the text they ultimately represent. The nature of this concatenation is subject to the rules given in 6.1.

A meta-expression denotes, depending on the operators used in it, a set S of terminal/nonterminal sequences. A rewrite rule is interpreted as follows: the nonterminal symbol at the left-hand side can be replaced by any one of the sequences of S .

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

EXAMPLE

$A = B "x" | "y" .$ is equivalent to the pair of rules .

$A = B "x" .$

$A = "y" .$

Use of the words 'of', 'in', 'containing', 'contained in', 'closest-containing', and 'outermost' when expressing a relationship between terminal or nonterminal symbols has the following meanings:

- the *x of a y*: refers to the *x* occurring directly in the meta-expression defining *y* ; if there are additional direct occurrences of *x* in a production of *y*, a unique occurrence may be referred to as the first, or second, etc. *x of a y*.
- the *x in a y*: is synonymous with 'the *x of a y*'.
- a *y containing an x*: refers to any *y* in the defining production of which either *x* occurs directly, or a *z* containing an *x* occurs directly.
- an *x contained in a y*: refers to any *x of a y*, or to any *x of a z* contained in a *y*.
- the *y closest-containing an x*: refers to that *y* which contains an *x* but does not contain another *y* containing that *x*.
- the *outermost x of a y*: refers to that *x* contained in *y* containing all other occurrences of *x* contained in *y*.

Table 1 - Metalanguage symbols

Meta-symbol	Name	Pronunciation
"xyz"	terminal symbol <i>xyz</i>	<i>xyz</i>
<i>abc</i>	nonterminal symbol <i>abc</i>	(nonterminal) <i>abc</i>
=	rewrite symbol	is defined to be, or can be rewritten as
.	termination symbol	end of rewrite rule
	alternation symbol	or, alternatively
[...]	option operator	0 or 1 instances of
{...}	repetition operator	0 or more instances of

4.7 Syntax-directed definitions

The specification of the static semantics of LOTOS is based on syntax-directed translation. A syntax-directed translation function, denoted by $\#N\#$ where *N* is a non-terminal, is a function that maps each occurrence of a non-terminal to its translation, using the BNF-production rule and context information as arguments.

Let *N* be an occurrence of a non-terminal, let $a_1, \dots, a_n$ represent the context information of *N*, then

$$\#N\#(a_1, \dots, a_n) = T_N$$

defines that the translation of *N* in a context environment $a_1, \dots, a_n$ is equal to T_N .

In general the translation mapping is partial: only for correct combinations of occurrences N with contexts $a_1, \dots, a_n$ the translation is defined; combinations $N, a_1, \dots, a_n$ for which $\#N\#(a_1, \dots, a_n)$ is not defined are static semantically incorrect.

EXAMPLE

Consider the following BNF syntax definition, that defines binary and decimal numbers. Binary numbers are prefixed with "B".

```

number = binary-number .
number = decimal-number .
binary-number = "B" digit-string .
decimal-number = digit-string .
digit-string = digit .
digit-string = digit-string digit .
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .

```

Using a translation function $\# \cdot \#$ we define the value of each number via the corresponding rules:

- a) if N is a *number* that is a *binary-number*, DS is a *digit-string* with $N = \text{"B" } DS$
then $\#N\# = \#DS\#(\text{binary})$
- b) if N is a *number* that is a *decimal-number*, DS is a *digit-string* with $N = DS$
then $\#N\# = \#DS\#(\text{decimal})$
- c) if DS is a *digit-string*, d is a *digit* with $N = d$
then $\#DS\#(\text{number-system}) =$
 if *number-system* = decimal
 then $\#d\#$
 else if d is "0" or "1" then $\#d\#$
- d) if DS, DS' are *digit-string* instances, d is a *digit* with $DS = DS' d$
then $\#DS\#(\text{number-system}) =$
 if *number-system* = decimal
 then $\#DS'\#(\text{decimal}) * 10 + \#d\#$
 else if *digit* is "0" or "1" then $\#DS'\#(\text{binary}) * 2 + \#d\#$
- e) if d is a *digit*
then $\#d\# = 0$ if $d = \text{"0"}$
 ...
 $\#d\# = 9$ if $d = \text{"9"}$

For each production the translation of its left-hand side is defined, with context information as parameters (i.e. *decimal* or *binary* for non-terminal *digit-string*). The translation of the BNF sentence B101 is:

```

#B101# =                                      (translation a))
#101#(binary) =                              (translation d) and e))
(#10#(binary)) * 2 + 1 =                    (translation d) and e))
((#1#(binary)) * 2 + 0) * 2 + 1 = 5        (translation c) and e))

```

For the BNF sentence B201 the translation, $\#B201\#$, is not defined, although the sentence is syntactically correct. However, the sentence is not static semantically correct since 201 is not a valid binary number.

NOTE - The notation of syntax-directed translation that is used in this standard is related to the formalism of attribute grammars in the following way: the result of the translation can be compared with a synthesized attribute of the left-hand side non-terminal and the context arguments can be considered as inherited attributes of the left-hand side non-terminal.

In the example above a synthesized attribute *value* can be defined for all non-terminals, an inherited attribute *number-system* for the non-terminal *digit-string* and a synthesized boolean attribute *error* for *digit-string* to indicate static semantic errors.

The attribute equations for the production rule

$$\text{digit-string} = \text{digit-string} \text{ digit} .$$

would look like:

synthesized attribute *value* for *digit*, *digit-string*
 inherited attribute *number-system* for *digit-string*
 synthesized attribute *error* for *digit-string*

$$\text{digit-string} = \text{digit-string}' \text{ digit}$$

$$\text{digit-string.error} = (\text{digit-string.number-system} = \text{binary}) \text{ and } (\text{digit is not "0" or "1"});$$

$$\begin{aligned} \text{digit-string.value} = & \text{ if } \text{digit-string.number-system} = \text{decimal} \\ & \text{ then } \text{digit-string'.value} * 10 + \text{digit.value} \\ & \text{ else } \text{ if } \text{digit-string.error} \\ & \quad \text{ then } 0 \\ & \quad \text{ else } \text{digit-string'.value} * 2 + \text{digit.value}; \end{aligned}$$

$$\text{digit-string'.number-system} = \text{digit-string.number-system}.$$

4.8 Derivation systems

A *derivation system* is a 3-tuple $D = \langle A, Ax, I \rangle$ with:

- A a set, the elements of which are called *assertions*,
- $Ax \subseteq A$, the set of *axioms*,
- I a set of *inference rules*.

Each inference rule $R \in I$ has the following format:

$$R: \frac{P_1, \dots, P_n}{Q} \quad \text{where } P_1, \dots, P_n, Q \in A$$

NOTE - The informal meaning of such a rule R is that if the assertions $P_1, \dots, P_n$ (the *premises* of R) are true, then also the assertion Q (the *conclusion* of R) is true.

In many derivation systems the sets Ax and I of axioms and inference rules are not defined directly by describing its individual elements, but by a set of axiom and inference rule schemata. An axiom or inference rule schema is a general rule, of which each instance is an axiom or inference rule, respectively.

EXAMPLE

In a derivation system dealing with equality between simple expressions in integer arithmetic the inference schema

$$\begin{array}{ccc}
 \frac{x=y, y=z}{x=z} & \text{has instances like} & \\
 \\
 \frac{1+1=2, 2=3-1}{1+1=3-1} & \text{and} & \frac{1=3, 3=2+1}{1=2+1}
 \end{array}$$

The latter rule cannot be applied in a derivation system consistent with simple arithmetic, as the truth of the first premiss $1=3$ cannot be established.

A *derivation* of an assertion P in a derivation system D is a finite sequence s of assertions satisfying the following conditions:

- a) the last element of s is P ,
- b) if Q is an element of s , then either $Q \in Ax$, or there exists a rule $R \in I$

$$R: \frac{P_1, \dots, P_n}{Q}$$

with $P_1, \dots, P_n$ elements of s preceding Q .

If there exists a derivation of P in a derivation system D , this is written $D \vdash P$. If D is uniquely determined by the context this may be abbreviated to $\vdash P$.

This page intentionally left blank

5 Model

5.1 Introduction

This clause defines the fundamental mathematical structures that are used in clause 7 to construct the semantic model for LOTOS.

In 5.2 the concept of a *many-sorted algebra* is introduced as a basic model for data types. It is used in 7.4 to define the formal interpretation of the *algebraic specification* (see 7.2.3.4) of a LOTOS specification, which is obtained from a LOTOS specification in 7.3.4.2 a), and is a formal syntactic representation of the data type definitions in that specification. The resulting many-sorted algebra is defined by 7.4.4.

In 5.3 the concept of a *labelled transition system* is presented, which can be used to model the behaviour of processes. In 5.4 this model is refined by integrating labelled transition systems and many-sorted algebras, thus yielding the proper structure for the formal interpretation of LOTOS expressions of behaviour, viz. the *structured labelled transition system*. This structure is used in 7.5.4.2 to define the formal interpretation of instantiated LOTOS specifications.

5.2 Many-sorted algebras

A *many-sorted algebra* A is a 2-tuple $\langle D, O \rangle$ where

- a) D is a set of sets, and the elements of D are referred to as the data carriers (of A); the elements of a data-carrier dc are referred to as *data-values*; and
- b) O is a set of total functions, where the domain of each function is a cartesian product of data carriers of A and the range is one of the data carriers.

NOTE - A data type consists of a set of data domains or data carriers, together with a set of operations. Each operation takes values from the data carriers and returns a value in one of the data carriers. The data carriers contain the actual values of (parts of) the data types that are specified. Operations related to a data type are considered part of the definition of that data type.

EXAMPLES

A Boolean type involves a data carrier {true,false}, or a record structure with an integer field involves a data carrier that represents the integer numbers.

In a data type representing a stack, the operations "push" and "pop" would be considered part of the data type, instead of operations that exist in the environment of that type.

5.3 Labelled transition systems

A *labelled transition system* Sys is a 4-tuple $\langle S, Act, T, s_0 \rangle$, where

- a) S is a non-empty set, whose elements are referred to as *states*; and
- b) Act is a set, whose elements are referred to as *actions*; and
- c) T is a set of *transition relations*, which contains precisely one relation $-a\delta \subseteq S \times S$ for each $a \in Act$; and
- d) $s_0 \in S$ is the *initial state* of Sys .

A *transition* of a labelled transition system is a 3-tuple $\langle cur, a, next \rangle$, also represented as $cur - a \rightarrow next$, such that $\langle cur, next \rangle \in -a\delta$.

Two labelled transition systems with identical action sets $Sys_1 = \langle S_1, Act, T_1, s_{01} \rangle$, and $Sys_2 = \langle S_2, Act, T_2, s_{02} \rangle$ are (*state-space*) *isomorphic* if there exists a one-to-one function $f: S_1 \rightarrow S_2$ with:

- a) $s_{11} \xrightarrow{a} s_{12}$ iff $f(s_{11}) \xrightarrow{a} f(s_{12})$ for all $a \in A$, $s_{11}, s_{12} \in S_1$;
- b) $f(s_{01}) = s_{02}$

A (*state-space*) *isomorphism class* of labelled transition systems is a class of labelled transition systems that are pair-wise isomorphic. Each member of such a class is a representing element of its class.

NOTE - Isomorphism classes of transition systems are models for the behaviour of communicating processes. A process is an abstract entity that is able to perform *internal events*, and to communicate with other processes in its environment via *communication events*. Communication events are primitive interactions which may be thought of as occurring at *interaction points*, or *gates*. The occurrence of a communication event involves participation of two or more processes in that event. Processes can be combined to form new processes.

Occurrences of events are modelled by transitions. The set of actions Act of a transition system contains both observable actions and internal actions. If cur represents the current state of a process, and $cur \xrightarrow{a} next$ is a transition in the associated labelled transition system, then we say that this system (or the process it models) is ready to *execute* (or *offer*) action a .

When a is an observable action, it is understood that the event associated with a may occur if both the transition system and its environment are ready to execute a . When a is an internal action, then an internal event may occur with no participation by the environment.

5.4 Structured labelled transition systems

A *structured labelled transition system* $Struc$ is a 5-tuple $\langle S, L \cup \{i\}, A, T, s \rangle$ with

- a) L is a set of *labels*, or *gates*; and
- b) $i \notin L$ is an *internal event*; and
- c) $A = \langle D, O \rangle$ is a many-sorted algebra, such that for $Act =_{df} \{i\} \cup \{gv \mid g \in L, v \in (\bigcup D)^*\}$
- d) $\langle S, Act, T, s \rangle$ is a labelled transition system.

To emphasize the particular many-sorted algebra A of a structured labelled transition system $Struc$, this system is also referred to as a labelled transition system *over* A .

NOTE - A structured labelled transition system is a special case of a labelled transition system. Its actions are structured, and consist of a label, or gate, followed by a finite string of data-values taken from the union of the domains of a given many-sorted algebra. Intuitively, the labels or gates represent the locality of an action, and the strings represent the data-values that are transferred as a result of an action. The special internal event i has no data-values concatenated to it.

6 Formal Syntax

6.1 Lexical tokens

6.1.1 General

Lexical tokens are formations of characters that are separated by token separators. They are the basic building blocks of a specification text. The lexical tokens used to construct a LOTOS specification are defined in 6.1.2 to 6.1.6. In these clauses the concatenation of text represented by a sequence of terminal and non-terminal symbols in a production rule is direct, there are no separating spaces or other separating symbols.

The definition of token separators is given in 6.1.7; 6.1.7 and 6.1.8 define the separation rules for tokens.

6.1.2 Basic characters

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

NOTE - Formally, in LOTOS no distinction is made between different versions of the same character, such as capitalized, bold, italic etc. It is understood, however, that such versions may be used to convey informally information that can be used for the correct interpretation of the text. In this International Standard boldface type is used for LOTOS keywords.

6.1.3 Reserved symbols

6.1.3.1 Word-symbols

A word-symbol is any of the productions defined in this clause.

specification-symbol =	"specification" .
end-specification-symbol =	"endspec" .
behaviour-symbol =	"behaviour" "behavior" .
type-symbol =	"type" .
end-type-symbol =	"endtype" .
is-symbol =	"is" .
actualization-symbol =	"actualizedby" .
using-symbol =	"using" .
library-symbol =	"library" .

end-library-symbol =	"endlib" .
rename-symbol =	"renamedby" .
formal-sorts-symbol =	"formalsorts" .
formal-operations-symbol =	"formalopns" .
formal-equations-symbol =	"formaleqns" .
sorts-symbol =	"sorts" .
operations-symbol =	"opns" .
of-sort-symbol =	"ofsort" .
for-all-symbol =	"forall" .
equations-symbol =	"eqns" .
sort-names-symbol =	"sortnames" .
operation-names-symbol =	"opnnames" .
for-symbol =	"for" .
process-symbol =	"process" .
end-process-symbol =	"endproc" .
where-symbol =	"where" .
stop-symbol =	"stop" .
exit-symbol =	"exit" .
noexit-symbol =	"noexit" .
any-symbol =	"any" .
internal-event-symbol =	"i" .
let-symbol =	"let" .
in-symbol =	"in" .
par-symbol =	"par" .
accept-symbol =	"accept" .
choice-symbol =	"choice" .
hide-symbol =	"hide" .
of-symbol =	"of" .

6.1.3.2 Special-symbols

A *special-symbol* is any of the productions defined in this clause.

equal-symbol =	"=" .
implication-symbol =	"=>" .
definition-symbol =	"::=" .
enable-symbol =	">>" .
disable-symbol =	"[>" .
synchronization-symbol =	" " .
interleaving-symbol =	" " .
left-par-symbol =	"[" .
right-par-symbol =	"]" .
fat-bar-symbol =	"[]" .
arrow-symbol =	"->" .
semi-colon-symbol =	";" .
comma-symbol =	"," .
colon-symbol =	":" .
open-bracket-symbol =	"[" .
close-bracket-symbol =	"]" .
open-parenthesis-symbol =	"(" .
close-parenthesis-symbol =	)" .
query-symbol =	"?" .
exclam-symbol =	"!" .
underscore-symbol =	"_" .

6.1.4 Identifiers

identifier =	letter [[normal-character underscore-symbol] normal-character] .
special-identifier =	special-character {special-character} digit [[normal-character underscore-symbol] normal-character] .

6.1.5 Requirement

An *identifier*, or a *special-identifier* shall not have the same spelling as any reserved symbol.

6.1.6 Comments

Let any-string-of-text represent any string of text not containing " *) " .

comment = " (* " any-string-of-text " *) " .

Comments are not part of the formal text of a LOTOS specification.

NOTE - It is understood that the character set used in *any-string-of-text* may extend the explicitly defined characters in this clause to allow for comments of a special nature (e.g. non-English text, pictures).

6.1.7 Token separators

Spaces , comments, and the separation of consecutive lines are token separators.

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

6.1.8 Requirement

There shall be at least one token separator between any pair of consecutive tokens if the concatenation of their texts can also be interpreted as the concatenated texts of a different sequence of tokens.

6.2 Specification text

NOTE - This subclause contains the formal context-free syntax of the LOTOS language. The syntax follows the structure of a specification starting from the outermost level.

6.2.1 specification

specification =
specification-symbol specification-identifier formal-parameter-list
global-type-definitions behaviour-symbol definition-block
end-specification-symbol .

global-type-definitions = data-type-definitions .

6.2.2 definition-block

definition-block = behaviour-expression [local-definitions] .

local-definitions = where-symbol local-definition { local-definition } .

local-definition = data-type-definition | process-definition .

6.2.3 data-type-definitions

data-type-definitions =	{ data-type-definition } .
data-type-definition =	type-symbol type-identifier is-symbol p-expression end-type-symbol library-declaration .
library-declaration =	library-symbol type-identifier { comma-symbol type-identifier } end-library-symbol .

6.2.4 p-expressions

p-expression =	[type-union] p-specification type-identifier actualization-symbol type-union using-symbol replacement type-identifier rename-symbol replacement .
type-union =	type-identifier [comma-symbol type-union] .
p-specification =	[formal-sorts] [formal-operations] [formal-equations] [sorts] [operations] [equations] .
replacement =	[sort-names-symbol sort-pair-list] [operation-names-symbol operation-pair-list] .

6.2.5 sorts, operations, and equations

sort-pair-list =	sort-pair [sort-pair-list] .
operation-pair-list =	operation-pair [operation-pair-list] .
sort-pair =	sort-identifier for-symbol sort-identifier .
operation-pair =	operation-identifier for-symbol operation-identifier .
formal-sorts =	formal-sorts-symbol sort-list .
formal-operations =	formal-operations-symbol operation-list .
formal-equations =	formal-equations-symbol equation-lists .
sorts =	sorts-symbol sort-list .
sort-list =	sort-identifier { comma-symbol sort-identifier } .
operations =	operations-symbol operation-list .
operation-list =	operation { operation } .
operation =	operation-descriptor { comma-symbol operation-descriptor } colon-symbol [argument-list] arrow-symbol result .

operation-descriptor =	operation-identifier underscore-symbol operation-identifier underscore-symbol .
argument-list =	sort-list .
result =	sort-identifier .
equations =	equations-symbol equation-lists .
equation-lists =	[for-all-symbol identifier-declarations] equation-list { equation-list } .
equation-list =	of-sort-symbol sort-identifier [for-all-symbol identifier-declarations] equation { semi-colon-symbol equation } [semi-colon-symbol] .
equation =	[premisses implication-symbol] simple-equation .
premisses =	premiss { comma-symbol premiss } .
premiss =	simple-equation boolean-expression .
simple-equation =	value-expression equal-symbol value-expression .

6.2.6 process-definitions

process-definition =	process-symbol process-identifier formal-parameter-list definition-symbol definition-block end-process-symbol .
formal-parameter-list =	[gate-parameter-list] [value-parameter-list] functionality-list .
gate-parameter-list =	gate-tuple .
value-parameter-list =	open-parenthesis-symbol identifier-declarations close-parenthesis-symbol .
functionality-list =	colon-symbol exit-list colon-symbol noexit-symbol .
exit-list =	exit-symbol [open-parenthesis-symbol sort-list close-parenthesis-symbol] .

6.2.7 behaviour-expressions

NOTE - The association of binary LOTOS behaviour operators is to the right, unless the use of parentheses indicates otherwise. The LOTOS behaviour operators for disabling ('>') and choice ('|') are associative, as are all instances of the parallel-operator ('||', '|||', '[a₁,...,a_n]||'), so that repeated applications of any of these operators are equivalent for any order of association. Note that in general the mixed use of different instantiations of the parallel-operator (e.g. '||' and '[a₁,...,a_n]||') is *not* associative, e.g. in general 'B₁ || (B₂ || [a₁,...,a_n] B₃)' defines another behaviour than '(B₁ || B₂) || [a₁,...,a_n] B₃'.

6.2.7.1. general structure

behaviour-expression =
 local-definition-expression
 | sum-expression
 | par-expression
 | hiding-expression
 | enable-expression .

6.2.7.2 local-definition-expressions

local-definition-expression = let-symbol identifier-equations in-symbol behaviour-expression .
 identifier-equations = identifier-equation { comma-symbol identifier-equation } .
 identifier-equation = identifier-declaration equal-symbol value-expression .

6.2.7.3 sum-expressions

sum-expression = choice-symbol sum-domain-expression choice-operator behaviour-expression .
 sum-domain-expression = identifier-declarations | gate-declarations .
 choice-operator = fat-bar-symbol .

6.2.7.4 par-expressions

par-expression = par-symbol par-domain-expression parallel-operator behaviour-expression .
 par-domain-expression = gate-declarations .
 parallel-operator =
 synchronization-symbol
 | interleaving-symbol
 | left-par-symbol [gate-identifier-list] right-par-symbol .

6.2.7.5 hiding-expressions

hiding-expression = hiding-operator behaviour-expression .
 hiding-operator = hide-symbol gate-identifier-list in-symbol .

6.2.7.6 enable-expressions

enable-expression = disable-expression [enable-operator enable-expression] .
 enable-operator =
 enable-symbol
 [accept-symbol identifier-declarations in-symbol] .

6.2.7.7 disable-expressions

disable-expression = parallel-expression [disable-operator disable-expression] .
disable-operator = disable-symbol .

6.2.7.8 parallel-expressions

parallel-expression = choice-expression [parallel-operator parallel-expression] .

6.2.7.9 choice-expressions

choice-expression = guarded-expression [choice-operator choice-expression] .

6.2.7.10 guarded-expressions

guarded-expression = guard-operator guarded-expression | action-prefix-expression .
guard-operator = guard arrow-symbol .
guard = open-bracket-symbol premiss close-bracket-symbol .

6.2.7.11 action-prefix-expressions

action-prefix-expression = action-denotation action-prefix-operator action-prefix-expression
| atomic-expression .
action-denotation = gate-identifier
[experiment-offer { experiment-offer } [selection-predicate]]
| internal-event-symbol .
experiment-offer = query-symbol identifier-declaration
| exclam-symbol value-expression .
selection-predicate = guard .
action-prefix-operator = semi-colon-symbol .

6.2.7.12 atomic-expressions

atomic-expression = stop-symbol
| exit-symbol [exit-parameter-list]
| process-instantiation
| open-parenthesis-symbol behaviour-expression
close-parenthesis-symbol .
exit-parameter-list = open-parenthesis-symbol exit-parameter
{ comma-symbol exit-parameter }
close-parenthesis-symbol .
exit-parameter = value-expression | any-symbol sort-identifier .

process-instantiation =	process-identifier [actual-gate-list] [actual-parameter-list] .
actual-gate-list =	gate-tuple .
actual-parameter-list =	value-expression-list .

6.2.8 value-expressions

value-expression =	[value-expression operation-identifier] simple-expression .
simple-expression =	term-expression [of-symbol sort-identifier] .
term-expression =	value-identifier operation-identifier [value-expression-list] open-parenthesis-symbol value-expression close-parenthesis-symbol .
value-expression-list =	open-parenthesis-symbol value-expression { comma-symbol value-expression } close-parenthesis-symbol .
boolean-expression =	value-expression .

NOTE - In the production rule for value-expression, the operation-identifier either appears in front of its arguments (e.g. $f(v1,v2)$), or in the middle of two arguments (e.g. $v1+v2$). These notations are known as prefix and (binary) infix notation, respectively. Alternative forms in which the operator and arguments occur in a mixed fashion, so-called mixfix notation (e.g. if $x>3$ then 1 else 0) are not allowed.

6.2.9 declarations

identifier-declarations =	identifier-declaration { comma-symbol identifier-declaration } .
identifier-declaration =	value-identifier-list colon-symbol sort-identifier .
value-identifier-list =	value-identifier { comma-symbol value-identifier } .
gate-declarations =	gate-declaration { comma-symbol gate-declaration } .
gate-tuple =	open-bracket-symbol gate-identifier-list close-bracket-symbol .
gate-declaration =	gate-identifier-list in-symbol gate-tuple .
gate-identifier-list =	gate-identifier { comma-symbol gate-identifier } .

6.2.10 special-identifiers

specification-identifier =	identifier .
type-identifier =	identifier .
sort-identifier =	identifier .

ISO 8807 : 1989 (E)

operation-identifier = identifier | special-identifier .

process-identifier = identifier .

gate-identifier = identifier .

value-identifier = identifier .

7 Semantics

7.1 Introduction

This clause deals with the formal interpretation, or semantics, of LOTOS specifications. The definition of this formal interpretation is substructured into two main phases: the static semantics phase and the dynamic semantics phase. The starting point is a LOTOS text, i.e. a text syntactically generated by the non-terminal *specification* as defined in 6.2. The result is the interpretation of this text in terms of *structured labelled transition systems*, as defined in 5.4.

In this subclause we explain the general structure of the definition of the static semantics in 7.1.1 and that of the dynamic semantics in 7.1.2. Subclause 7.1.3 explains the organization of 7.2 to 7.5.

7.1.1 Structure of static semantics definition

The static semantics phase serves two purposes:

- a) to define all static semantic requirements; and
- b) to define the transformation of a LOTOS text into an abstract syntactic structure, called a *canonical LOTOS specification*.

The definition of these two aspects of the static semantics phase is combined in the definition of a syntax-directed translation function, called the *flattening function*. The flattening function is a partial function that maps LOTOS texts to canonical LOTOS specifications. The function is partial because only LOTOS texts that conform to the static semantic requirements are mapped to a canonical LOTOS specification. The result of flattening a LOTOS text that does not conform to the static semantic requirements is undefined.

A canonical LOTOS specification, which is denoted by *CLS*, is an abstract representation of a flattened specification, in which all structures are defined at one, global level, all nesting of definitions has been removed and all identifiers have been made unique by relating them to their corresponding defining occurrences. A canonical LOTOS specification *CLS* consists of two related parts:

- a) an *algebraic specification*, denoted by *AS*, that contains representations of the data types in LOTOS; and
- b) a *behaviour specification*, denoted by *BS*, that contains representations of the behaviour definitions in LOTOS.

7.1.2 Structure of dynamic semantics definition

In the dynamic semantics phase a canonical LOTOS specification is transformed into a class of structured labelled transition systems: it defines a structured labelled transition system for each substitution of actual parameters for the formal parameters of the specification.

For the dynamic semantics a separation can be made between the data type part *AS* and the behaviour part *BS* of a canonical LOTOS specification *CLS*. The interpretation of the algebraic specification *AS*, i.e. the formal model for LOTOS data structures and operations on them, is a particular *many-sorted algebra* $Q(AS)$, the *quotient term algebra* of *AS* (see 7.4.4). This algebra is subsequently used in the interpretation of the behaviour part *BS* of *CLS*. The interpretation of *BS* is a class of structured labelled transition systems over $Q(AS)$, or more accurately, a function with that class as its range. This function maps each correct substitution of actual values for the formal parameters of the specification to a structured labelled transition system. This transition system serves as a model for the dynamic behaviour of the corresponding instance of the LOTOS specification.

7.1.3 Structure of clause 7

The rest of clause 7 is organized as follows: subclause 7.2 gives some general concepts and definitions resulting in the definition of the structure of a canonical LOTOS specification in 7.2.3. Subclause 7.3 defines the static semantics, i.e. the flattening function that maps LOTOS texts to canonical LOTOS specifications. Subclause 7.4 defines the interpretation of an algebraic specification AS as a many-sorted algebra. Subclause 7.5 defines the interpretation of a canonical LOTOS specification in terms of structured labelled transition systems, resulting in 7.5.5: "Formal interpretation of a canonical LOTOS specification".

Figure 1 gives an overview of the different phases of the definition of the semantics of a LOTOS specification.

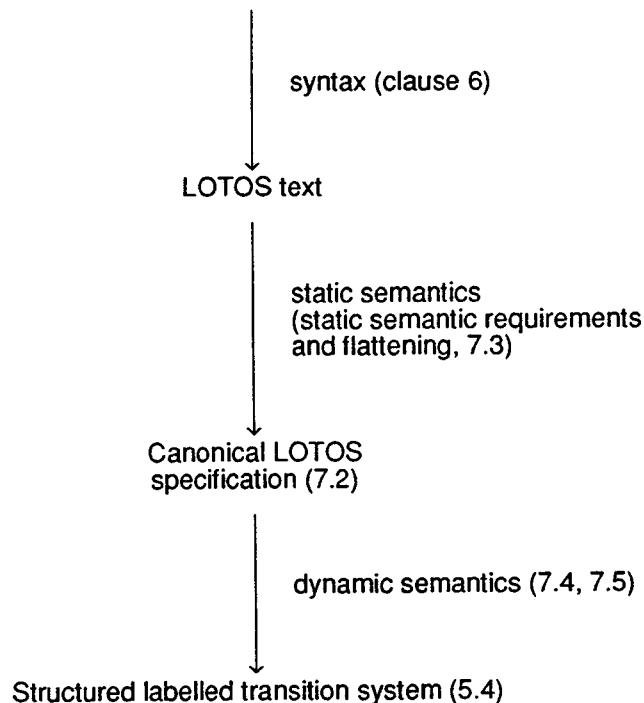


Figure 1 - Structure of clause 7

7.2 General structures and definitions

7.2.1 Names and related functions

Throughout 7.2 the existence of the following sets of names is assumed:

- a) the set *SV* of *sort-names*; and
- b) the set *OV* of *operation-names*; and
- c) the set *PV* of *process-names*; and
- d) the set *GV* of *gate-names*; and
- e) the set *VV* of *value-names*, usually referred to as *variables*.

Also the existence of the following total functions is assumed:

- f) *arg*: $OV \rightarrow SV^*$, yielding for each operation-name *op* its list of *argument* sort-names *arg*(*op*); and
- g) *res*: $OV \rightarrow SV$, yielding for each operation-name *op* its *result* sort-name *res*(*op*); and
- h) *fg*: $PV \rightarrow GV^*$, yielding for each process-name *p* its *formal-gate-parameter-list* *fg*(*p*); and
- j) *fv*: $PV \rightarrow VV^*$, yielding for each process-name *p* its *formal-value-parameter-list* *fv*(*p*); and
- k) *sort*: $VV \rightarrow SV$, yielding for each variable *x* its sort-name *sort*(*x*).

If op is an operation-name with $arg(op)=\langle \rangle$ and $res(op)=s$, then op is referred to as a *nullary* operation-name or *constant* of sort s . If op is an operation-name with $arg(op)=a$ and $res(op)=s$ this is sometimes written as $op : a \rightarrow s$.

If x is a variable with $sort(x)=s$ then this is sometimes indicated by writing $x:s$ instead of x .

7.2.2 Algebraic specifications, terms, and equations

7.2.2.1 Signature

A *signature* sig is a tuple $\langle S, OP \rangle$ where

- a) S is a set of *sort-names*; and
- b) OP is a set of *operation-names*.

7.2.2.2 Terms

Let V be any set of variables, and let $\langle S, OP \rangle$ be a signature. The sets $TERM(OP, V, s)$ of *terms* of sort $s \in S$ with operations in OP and variables in V , are defined inductively by the following steps:

- a) each variable $x:s \in V$ is in $TERM(OP, V, s)$;
- b) each nullary operation-name $op \in OP$ with $res(op)=s$ is in $TERM(OP, V, s)$;
- c) if the terms t_i of sort s_i are in $TERM(OP, V, s_i)$ for $i = 1, \dots, n$, then for each $op \in OP$ with $arg(op)=\langle s_1, \dots, s_n \rangle$ and $res(op)=s$, $op(t_1, \dots, t_n)$ is in $TERM(OP, V, s)$.

If term t is an element of $TERM(OP, V, s)$ then s is called *the sort of t* , denoted as $sort(t)$.

The set $TERM(OP, s)$ of *ground terms* of sort $s \in S$ is defined as the set $TERM(OP, \emptyset, s)$.

NOTE - For the sake of convenience we will write terms with their (argument and result) sorts omitted if this introduces no ambiguity. For example, we write $not(x)$ instead of $not.Bool \rightarrow Bool(x: Bool)$ in the case of a boolean data type with an operator not .

7.2.2.3 Equations

An *equation* of sort s with respect to a signature $\langle S, OP \rangle$ is a triple $\langle V, L, R \rangle$, where

- a) V is a set of variables; and
- b) $L, R \in TERM(OP, V, s)$; and
- c) $s \in S$.

An equation $e'=\langle \emptyset, L', R' \rangle$ is a *ground instance* of an equation $e=\langle V, L, R \rangle$ if there exists a function $h: V \rightarrow \bigcup \{ TERM(OP, s) | s \in S \}$ with $h(v:s) \in TERM(OP, s)$ for all $v:s \in V$ such that L', R' can be obtained from L, R by replacing each occurrence in L and R of a variable v by $h(v)$ for each $v:s$ in V . In this case e' is said to be obtained from e via $h: V \rightarrow \bigcup \{ TERM(OP, s) | s \in S \}$.

The notation $L = R$ is used for the ground instance $\langle \emptyset, L, R \rangle$ of an equation.

NOTE - Also an equation $\langle V, L, R \rangle$ may be written $L = R$ if V is the set of variables occurring in L or R .

7.2.2.4 Conditional equations

A *conditional equation* of sort s with respect to a signature $\langle S, OP \rangle$ is a triple $\langle V, Eq, e \rangle$, where

- V is a set of variables; and
- Eq is a set of equations with respect to $\langle S, OP \rangle$ with variables in V ; and
- e is an equation of sort s with respect to $\langle S, OP \rangle$ with variables in V .

NOTE - A conditional equation $\langle V, \{e_1, \dots, e_n\}, e \rangle$ may be written as $e_1, \dots, e_n \Rightarrow e$ if this does not introduce ambiguities. If Eq is the empty set one may write simply e , under the same conditions.

7.2.2.5 Algebraic specifications

An *algebraic specification SPEC* is a triple $\langle S, OP, E \rangle$, where

- $\langle S, OP \rangle$ is a signature; and
- E is a set of conditional equations with respect to $\langle S, OP \rangle$.

7.2.3 Canonical specifications

7.2.3.1 Introduction

This subclause defines the structure of a canonical LOTOS specification CLS, using the definitions of names in 7.2.1 and of algebraic specification, terms and equations in 7.2.2.

The set of canonical LOTOS specifications is the range of the flattening function. A canonical LOTOS specification is an abstract representation of a LOTOS specification. This subclause only gives the structure of a canonical LOTOS specification; how to obtain a canonical LOTOS specification from a LOTOS text is described in 7.3.

7.2.3.2 Behaviour-expression-structure

A *behaviour-expression-structure B* is an instance of behaviour-expression as defined in sub-clause 6.2.7, with the following modifications:

- sort-identifier* occurrences in an *exit-parameter* are replaced by *sort-names*;
- process-identifier* occurrences in a *process-instantiation* are replaced by *process-names*;
- all occurrences of *gate-identifier* are replaced by *gate-names*;
- if gd is a *gate-declaration* with $gd = g_1, \dots, g_m \text{ in } [a_1, \dots, a_n]$ then gd is replaced by $g_1 \text{ in } [a_1, \dots, a_n], \dots, g_m \text{ in } [a_1, \dots, a_n]$;
- all *value-expression* occurrences are replaced by *terms*;
- if id is an *identifier-declaration* with $id = vid_1, \dots, vid_n : sid$ contained in:
 - an *identifier-declarations* then id is replaced by $v_1, \dots, v_n$ where v_i are *variables*;
 - an *identifier-equation* $id = v\text{-exp}$ then this equation is replaced by $v_1 = t, \dots, v_n = t$ where v_i are *variables* and t is a *term*;
 - an *experiment-offer* $? id$ then the experiment-offer is replaced by $? v_1 \dots ? v_n$ where v_i are *variables*;
- all occurrences of *simple-equation* are replaced by formal equations of the form $\langle V, L, R \rangle$ (see 7.2.2.4);
- all occurrences of *boolean-expression* are replaced by formal equations of the form $\langle V, L, \text{true} \rangle$.

7.2.3.3 Behaviour specification

A *behaviour specification* BS is a set of flattened process-definitions $PDEFS$ with an initial process-definition $pdef_0 \in PDEFS$: $BS = \langle PDEFS, pdef_0 \rangle$.

A *process-definition* $pdef$ is a pair consisting of a process-name p and a behaviour-expression-structure B : $pdef = \langle p, B \rangle$.

NOTE - $PDEFS$ is the representation of all process-definitions contained in a specification. The behaviour of the specification itself is also considered as a process-definition, viz. the process-definition $pdef_0$.

7.2.3.4 Canonical LOTOS specification

A *canonical LOTOS specification* $CLS = \langle AS, BS \rangle$ consists of an algebraic specification AS and a behaviour specification BS such that the signature $\langle S, OP \rangle$ of AS contains all sort-names and operation-names occurring in BS .

7.3 Static semantics

7.3.1 Introduction

This subclause defines the flattening function, which is a syntax-directed translation function from LOTOS texts to canonical LOTOS specifications, and is denoted by $\#.\# : \text{LOTOS texts} \rightarrow \text{canonical LOTOS specifications}$. A LOTOS text is a text as generated by the non-terminal *specification* according to the syntactical rules of 6.2. A canonical LOTOS specification is defined in 7.2.3.

The function $\#.\#$ specifies implicitly the static semantics of a LOTOS specification, i.e. all static conformance requirements other than syntactic requirements. These requirements are defined using the fact that $\#.\#$ is a *partial* function: only LOTOS texts that conform to the static semantic requirements can be mapped to a canonical LOTOS specification. For all statically non-conforming LOTOS texts the result of applying the flattening function is undefined.

The flattening function $\#.\#$ is defined by syntactic recursion over LOTOS texts using the notational conventions introduced in 4.7. For each syntactic construct defined in 6.2 the result of applying the flattening function to such a construct is defined in 7.3.4.

Moreover, 7.3.4 contains with each construct those static semantic requirements that shall be fulfilled when applying the flattening function to such a construct. Thus, all requirements of all subclauses of 7.3.4 together specify the static semantics requirements of a LOTOS text.

As indicated in 7.1.1, the major difference between a syntactic structure N and the corresponding structure $\#N\#$ that results from the application of the flattening function, is the renaming of elements of N such that all names in $\#N\#$ become globally unique. Also, in a few places there is a compactification of information in $\#N\#$. As the essential structure of N is preserved in $\#N\#$, from this point onwards we write N instead of $\#N\#$ where this is convenient and does not cause ambiguity.

Subclauses 7.3.2 (general structures and definitions) and 7.3.3 (reconstruction of terms) define concepts and intermediate structures as a prerequisite for the definition of $\#.\#$ in 7.3.4. Subclause 7.3.5 contains a table for reference purposes, summarizing the structure of the arguments and results of the flattening function.

NOTE - The definition of the static semantics is by syntactic recursion, analogous to a definition by attribute grammars. This means that all properties are described locally to non-terminal symbols. On a global level this defines a "conventional" static semantics with properties like:

- a) Occurrences of identifiers can be divided into *defining* and *applied* occurrences. Within the same scope defining occurrences of identifiers of the same class shall not have the same spelling, except for operation-identifiers (for operation-identifiers overloading is allowed if they can be distinguished by argument-sorts or result-sort). Classes of identifiers are specification-, type-, sort-, operation-, process-, gate- and value-identifier.
- b) Each applied occurrence of identifier shall be bound to a defining occurrence of identifier of the same class with the same spelling. If there are more possibilities the identifier is bound to the defining occurrence in the closest containing scope.
- c) Data-type-definitions shall be non-circular; renamings shall define injective copies of data types; actualizations shall actualize completely and consistently all formal sorts and operations; all defined sorts and operations shall be uniquely identifiable.
- d) In process-instantiations formal and actual gate- and value-parameters shall match.
- e) The functionality of a process-definition and its corresponding behaviour-expression shall match. The same applies to the functionalities of the two operands of binary behaviour operators.

7.3.2 General structures and definitions

7.3.2.1 Scope

A *scope scp* is a part of a LOTOS text. If *x* is a scope then *scp(x)* denotes some unique identification of this scope.

7.3.2.2 Extended identifiers

The *extended identifier e-id* of an occurrence of an *identifier id*, is an extension of the identifier with all relevant static information of that identifier. The following classes of extended identifiers exist:

- a) *Extended-specification-identifier* : $e\text{-}spid = \langle spidf, fg, fv, func \rangle$
 where $spidf = \langle spid, scp \rangle$ is the *specification-identification*, consisting of a *specification-identifier spid* and a *scope scp*;
 $fg = \langle e\text{-}gid_1, \dots, e\text{-}gid_n \rangle$ is the *formal-gate-parameter-list*, consisting of *extended-gate-identifiers*;
 $fv = \langle e\text{-}vid_1, \dots, e\text{-}vid_m \rangle$ is the *formal-value-parameter-list*, consisting of *extended-value-identifiers*;
 $func$ is the *functionality* (see 7.3.2.4).
- b) *Extended-type-identifier* : $e\text{-}tid = \langle tid, scp \rangle$
 where tid is a *type-identifier*,
 scp is a *scope*.
- c) *Extended-sort-identifier* : $e\text{-}sid = \langle sid, e\text{-}tid \rangle$
 where sid is a *sort-identifier*,
 $e\text{-}tid$ is an *extended-type-identifier*.
- d) *Extended-operation-identifier* : $e\text{-}opid = \langle opidf, e\text{-}args, e\text{-}res, pos \rangle$
 where $opidf = \langle opid, e\text{-}tid \rangle$ is an *operation-identification*, consisting of an *operation-identifier opid* and an *extended-type-identifier e-tid*;
 $e\text{-}args = \langle e\text{-}sid_1, \dots, e\text{-}sid_k \rangle$ is the *argument-list*, consisting of a list of *extended-sort-identifiers*;
 $e\text{-}res$ is the *result-sort*, which is an *extended-sort-identifier*;
 pos denotes the nature of the operator: *prefix* or *infix*.

- e) *Extended-process-identifier*. $e\text{-pid} = \langle \text{pidf}, fg, fv, func \rangle$
 where $\text{pidf} = \langle \text{pid}, scp \rangle$ is the *process-identification*, consisting of *process-identifier* pid and *scope* scp ;
 $fg = \langle e\text{-gid}_1, \dots, e\text{-gid}_n \rangle$ is the *formal-gate-parameter-list*, consisting of *extended-gate-identifiers*;
 $fv = \langle e\text{-vid}_1, \dots, e\text{-vid}_m \rangle$ is the *formal-value-parameter-list*, consisting of *extended-value-identifiers*;
 $func$ is the *functionality* (see 7.3.2.4).
- f) *Extended-gate-identifier*. $e\text{-gid} = \langle \text{gid}, scp \rangle$
 where gid is a *gate-identifier*,
 scp is a *scope*.
- g) *Extended-value-identifier*. $e\text{-vid} = \langle \langle \text{vid}, scp \rangle, e\text{-sid} \rangle$
 where vid is a *value-identifier*,
 scp is a *scope*;
 $e\text{-sid}$ is an *extended-sort-identifier*.

NOTE - During the static semantics phase an extended-identifier $e\text{-id}$ is assigned to every occurrence of an identifier id . The extended-identifier of an identifier contains all static information that is relevant for that identifier, in such a way that all extended-identifiers are globally unique, i.e. if two extended-identifiers are identical then they denote the same object. The most important part of this extra static information is the *scope*: since it is not allowed to define two identifiers with the same spelling within one scope, scope information is sufficient to identify an object uniquely except for *extended-operation-identifiers*. For operation-identifiers *overloading* is allowed, i.e. different operations may have the same name, even within one scope. Distinction between different operations is made via *argument-list*, *result-sort* and *position* (whether the operation is a prefix or an infix operation).

Although in most cases scope information is sufficient for unique identification of identifiers, also other static information is included in extended-identifiers. For *extended-specification-identifier* and *extended-process-identifier* this is the information in the specification- or process-definition heading: the *formal gate-parameterlist*, *formal value-parameterlist* and the *functionality*. In these structures, the identifiers are replaced by their corresponding extended-identifiers, too. Since extended-specification-identifiers and extended-process-identifiers have the same structure they are sometimes identified.

For *extended-sort-identifier* and *extended-operation-identifier* not only the scope of definition of the identifier is added to the extended-identifier, but also the *extended-type-identifier* of the data-type-definition of the original definition of the sort or operation. In this way it is possible to distinguish between two cases:

- the identification of sorts (or operations) that were originally defined in the same data-type-definition but that were used in different other data-type-definitions; and
- the definition of two different sorts (or operations) with the same name, in different data-type-definitions.

For a *value-identifier* the *sort* (extended-sort-identifier) assigned to it in its corresponding identifier-declaration is added to the *extended-value-identifier*.

In 7.3.3 the *argument-list* $e\text{-args}$, the *result-sort* $e\text{-res}$, and the *position* pos , which are only explicitly defined for an extended-operation-identifier, are also introduced for *extended-value-identifier*. The following standard values are assigned for an extended-value-identifier: the argument-list of an extended-value-identifier is *empty*: $e\text{-args} = \emptyset$; the result-sort of an extended-value-identifier $\langle \langle \text{vid}, scp \rangle, e\text{-sid} \rangle$ is $e\text{-sid}$; the position is *undefined*: $pos = \text{undef}$.

7.3.2.3 The interpretation of extended identifiers

Some of the classes of extended identifiers are concrete instances of the sets of names defined in 7.2.1. The correspondence is as follows:

- extended-sort-identifiers correspond to sort-names; and
- extended-operation-identifiers correspond to operation-names; and
- extended-process-identifiers and extended specification-identifiers correspond to process-names; and

- d) extended-gate-identifiers correspond to gate-names; and
- e) extended-value-identifiers correspond to variables.

The related functions are defined as follows:

- f) $arg(e-opid) =_{df} e-opid.e-args$ for each extended-operation-identifier $e-opid$; and
- g) $res(e-opid) =_{df} e-opid.e-res$ for each extended-operation-identifier $e-opid$; and
- h) $fg(e-pid) =_{df} e-pid.fg$ for each extended-process-identifier $e-pid$; and
- j) $fv(e-pid) =_{df} e-pid.fv$ for each extended-process-identifier $e-pid$; and
- k) $sort(e-vid) =_{df} e-vid.e-sid$ for each extended-value-identifier $e-vid$.

NOTE - The flattening function $\#$ in the static semantics phase, which is defined in 7.3.4, uses the concept of extended-identifiers to identify identifiers uniquely and to relate a defining occurrence of an identifier with each applied occurrence of an identifier with the same spelling. The result is a canonical LOTOS specification in which all objects are denoted by different extended-identifiers. In the dynamic semantics phase the extended-identifiers of the canonical LOTOS specification are considered to be names, as defined in 7.2.1. The sophisticated structure of extended-identifiers is not needed once all objects have a unique representation. The only structuring information that is needed in the dynamic semantics phase is the information represented by the functions arg , res , fg , fv and $sort$, as defined in f), g), h), j) and k).

7.3.2.4 Functionality

A *functionality func* is either a k -tuple of *extended-sort-identifiers*, denoted with $\langle e-sid_1, \dots, e-sid_k \rangle$ ($k \geq 0$), or the special symbol 0.

The functionality of a *behaviour-expression B* in a context of a *type-environment TE*, *process-environment PE* and *value-environment VE* (see 7.3.2.10) is denoted as $func(B, TE, PE, VE)$.

For pairs of functionalities the partial functions *min* and *max* are defined:

$min(func_1, func_2)$	$=_{df} 0$	if $func_1 = 0$ or $func_2 = 0$
	$=_{df} func_1$	if $func_1 = func_2$
	undefined	otherwise
$max(func_1, func_2)$	$=_{df} func_1$	if $func_2 = 0$
	$=_{df} func_2$	if $func_1 = 0$
	$=_{df} func_1$	if $func_1 = func_2$
	undefined	otherwise

7.3.2.5 Data-presentation

A *data-presentation pres* = $\langle S', OP', E' \rangle$ is an algebraic specification $\langle S, OP, E \rangle$, with *extended-sort-identifiers* in S' instead of *sort-names* in S , *extended-operation-identifiers* in OP' instead of *operation-names* in OP , and *extended-value-identifiers* in E' instead of *variables* in E .

7.3.2.6 Parameterized data-presentation

A *parameterized data-presentation ppres* = $\langle fpres, tpres \rangle$ is a pair of data-presentations consisting of a *formal* data-presentation $fpres$ and a *target* data-presentation $tpres$, where the formal data-presentation is included componentwise in the target data-presentation.

7.3.2.7 Non-overlapping data-presentation

A data-presentation $pres = \langle S, OP, E \rangle$ is non-overlapping if:

- the sort-identifiers of all extended-sort-identifiers in S are different; and
- there are not two different extended-operation-identifiers in OP that differ only in the extended-type-identifier of their respective operation-identifications.

A parameterized data-presentation $ppres = \langle fpres, tpres \rangle$ is non-overlapping if $tpres$ is non-overlapping.

NOTE - In a non-overlapping data-presentation all sorts have different, unextended identifiers and all operations are characterized completely by their name, argument sorts, result sort and position (prefix- or infix-operation). If it is required that a given data-presentation is non-overlapping this implies that it shall not contain identical, unextended sorts and/or operations that have different extensions, i.e. are declared as part of different data-type-definitions.

7.3.2.8 Signature morphism

Let $sig_1 = \langle S_1, OP_1 \rangle$ and $sig_2 = \langle S_2, OP_2 \rangle$ be signatures consisting of extended-identifiers. A signature morphism $g: sig_1 \rightarrow sig_2$ is a pair of functions

$$g = \langle gs: S_1 \rightarrow S_2, gop: OP_1 \rightarrow OP_2 \rangle$$

such that for all $e-opid_1 = \langle opidf_1, \langle e-sid_1, \dots, e-sid_k \rangle, e-res, pos \rangle \in OP_1$:

$$gop(e-opid_1) = \langle opidf_2, \langle gs(e-sid_1), \dots, gs(e-sid_k) \rangle, gs(e-res), pos \rangle$$

for some operation-identification $opidf_2$.

NOTE - A signature morphism maps a signature to another signature. The argument- and result-sorts of the operations should be consistently mapped, i.e. the argument/result sort of the image of an operation should be equal to the image of the argument/result sort of that operation.

7.3.2.9 Data-presentation morphism

Let $pres_1 = \langle S_1, OP_1, E_1 \rangle$ and $pres_2 = \langle S_2, OP_2, E_2 \rangle$ be data-presentations.

A signature morphism $g: \langle S_1, OP_1 \rangle \rightarrow \langle S_2, OP_2 \rangle$ defines a data-presentation morphism $g': pres_1 \rightarrow pres_2$ as a 3-tuple of functions

$$g' = \langle gs: S_1 \rightarrow S_2, gop: OP_1 \rightarrow OP_2, geq: E_1 \rightarrow E_2 \rangle$$

where gs and gop are the component functions of g , and for each equation $e \in E_1$ $geq(e)$ is defined by:

- replacing all extended-sort-identifiers $e-sid$ contained in e by $gs(e-sid)$; and
- replacing all extended-operation-identifiers $e-opid$ contained in e by $gop(e-opid)$; and
- replacing all extended-value-identifiers $e-vid = \langle \langle vid, scp \rangle, e-sid \rangle$ contained in e by $\langle \langle vid, scp \rangle, gs(e-sid) \rangle$.

A data-presentation morphism $g': pres_1 \rightarrow pres_2$ is well-defined if

$$\{ geq(e) \mid D_{pres_1} \vdash e \} \subseteq \{ e \mid D_{pres_2} \vdash e \}$$

where

$\{ geq(e) \mid D_{pres_1} \vdash e \}$ and $\{ e \mid D_{pres_2} \vdash e \}$ are sets of ground equations;
 D_{pres_1} and D_{pres_2} are the derivation systems generated by $pres_1$ and $pres_2$, respectively (see 7.4).

NOTE - A data-presentation morphism extends a signature morphism to equations. Besides the consistency requirement of the signature morphism there is a requirement concerning semantic well-definedness : each equation valid in the original specification must also be valid in the image specification. Unfortunately, this well-definedness property is algorithmically undecidable. A data-presentation morphism is used in the definition of renaming (by construction: well-definedness is automatically implied) and in the definition of actualization (as morphism from formal- to actual sorts and operations, where the well-definedness property is required), see 7.3.4.3 e) and f).

7.3.2.10 Environments

A *type-environment* TE is a set of pairs, each pair $te \in TE$ consisting of an *extended-type-identifier* and a *parameterized data-presentation*: $te = \langle e\text{-}tid, ppres \rangle$.

The *combination* of a type-environment TE , $combine(TE)$, is the union of all parameterized data-presentations in TE :

$$combine(TE) = \bigcup \langle ppres \mid \langle e\text{-}tid, ppres \rangle \in TE \rangle$$

A *type-environment* TE is *non-overlapping* if $combine(TE)$ is non-overlapping.

A *process-environment* PE is a set of *extended-process-identifiers*.

A *gate-environment* GE is a set of *extended-gate-identifiers*.

A *value-environment* VE is a set of *extended-value-identifiers*.

NOTE - An environment is a set of extended-identifiers of defined objects. The identifiers in the extended-identifiers of the elements of an environment, together with suitable *scope* information, should identify the objects uniquely, such that environments define functions from $\langle \text{identifier}, \text{scope} \rangle$ -pairs to their corresponding extended-identifiers. The requirements in the definition of the flattening function guarantee that environments are constructed such that all identifiers of all extended-identifiers with the same scope are different. Application of the functions defined by environments is defined in 7.3.4.6.

A type-environment not only contains the extended-type-identifiers, but also the related parameterized data-presentations. Moreover, it is sometimes necessary to generate one large parameterized data-presentation from all parameterized data-presentations contained in a type-environment TE . This is called the *combination* of TE : $combine(TE)$. This only makes sense if every object in the resulting parameterized data-presentation can still be identified uniquely, i.e. if $combine(TE)$ is non-overlapping. In this way the definition of non-overlapping data-presentations is extended to type-environments.

7.3.2.11 Standard library

The standard library is the instance of *data-type-definitions* that consists of the concatenation of *data-type-definition* occurrences defined in ANNEX A of this standard. The standard library generates a type-environment, denoted with LIB and defined as:

$$LIB =_{df} \#type\text{-}defs\#(\emptyset, lib\text{-}scp)$$

By convention *type-defs* as defined by ANNEX A is a scope for all LOTOS texts, and has the unique identification *lib-scp*.

NOTE - The standard library of data types contains common, generally used data-type-definitions. They can be included in the specification text by a *library-declaration* as if the definition were in the specification text at the place of that declaration. The standard library is an occurrence of *data-type-definitions* (containing no library-declaration occurrences), and can be flattened according to 7.3.4.3 yielding a type-environment denoted by *LIB*. The scope *lib-scp* is the scope that is used in all extended-sort-identifiers and all extended-operation-identifiers contained in the parameterized data-presentations of *LIB*. The scope of the place of the occurrence of *library-declaration* is substituted for the scope *scp* in the extended-type-identifier of a $te \in LIB$, $te = \langle \langle tid, scp \rangle, ppres \rangle$, to make it possible to declare locally the standard types of the library, see 7.3.4.3 b).

7.3.2.12 Complete data-presentation

A *complete data-presentation* *cpres* is a parameterized data-presentation with an empty formal data-presentation:

$cpres = \langle \langle \emptyset, \emptyset, \emptyset \rangle, \langle S, OP, E \rangle \rangle$.

The function *complete* applied to a type-environment *TE*, is defined as the restriction of *TE* to all complete data-presentations and delivers a complete type-environment *CTE*:

$$CTE = complete(TE) = \{ \langle e-tid, pres \rangle \mid \langle e-tid, ppres \rangle \in TE, ppres = \langle \langle \emptyset, \emptyset, \emptyset \rangle, pres \rangle \}$$

NOTE - In behaviour-expressions only data items of completely defined data types may be used, i.e. data types in which all formal parts have been substituted by actual types, and which do not contain formal sorts or formal operations. Parameterized data-presentations that satisfy this requirement are called *complete*.

In the sequel data-presentation and complete parameterized data-presentation are used synonymously.

7.3.2.13 Valid dependence order

A *data-type-definitions* *type-defs* with $type-defs = type-def_1 \dots type-def_n$ is in a *valid dependence order* if there is no *type-def_i* ($1 \leq i \leq n$) that refers directly to *type-def_j* with $j \geq i$.

A *data-type-definition* *type-def₁* is said to refer directly to a *data-type-definition* *type-def₂* if the *type-identifier* in *type-def₂* is equal to a *type-identifier* contained in the *p-expression* of *type-def₁*.

NOTE - A list of data-type-definitions is in valid dependence order if each definition in the list only depends on preceding definitions in that list. Therefore, any permutation of such a list cannot contain circularly defined types.

7.3.3 Reconstruction of terms

This subclause defines the resolution procedure for the overloading of value- and operation-identifiers in value-expressions in LOTOS specifications. As the result of this procedure each value-expression *v-exp* is transformed into a term *recon(v-exp,...)*, which depends also on parameters defined by the syntactic environment of *v-exp*. In this term all value- and operation-identifiers have been replaced by unique extended identifiers. If not all value- and operation-identifiers can be bound uniquely to a corresponding defining occurrence then *recon(v-exp,...)* is undefined.

The definition of the reconstruction function *recon* is contained in 7.3.3.10; 7.3.3.1 to 7.3.3.9 contain definitions of auxiliary concepts.

NOTE - Informally, the general approach that is followed by the resolution of overloading that is defined by this subclause is as follows:

for each operation-identifier in a value-expression *v-exp*

- a) determine the set of all extended-operation-identifiers with the same name in the environment of *v-exp*; and

- b) from the set constructed under a) select the subset of those extended-operation-identifiers that have argument- and result-sorts that yield a correctly typed instance of *v-exp*; and
- c) from the set constructed under b) select those extended-operation-identifiers as binding occurrences such that the scopes that correspond with the bindings are minimized simultaneously.

The existence of the minimal solution in step c) is guaranteed by making it a requirement. In other words, overloading is only permitted if the obvious reconstruction, viz. the minimal one generating a correctly typed value-expression, is unique.

The consideration of value-identifiers in the resolution procedure is necessary because nullary operation-identifiers and value-identifiers may occur at the same places in value-expressions.

7.3.3.1 Value-atoms

A *value-atom* is an identifier occurrence contained in a value-expression that is either a value-identifier occurrence, or an operation-identifier occurrence.

7.3.3.2 Value-atom, position, and argument-list of a value-expression

The value-atom *vat(v-exp)*, position *pos(v-exp)*, and argument-list *arg(v-exp)* of a value-expression or term-expression *v-exp* are defined by

- a) if *v-exp* = *val-expr opid s-expr*
 where *s-expr* is a *simple-expression*,
opid is an *operation-identifier*,
val-expr is a *value-expression*
 then *vat(v-exp)* =_{df} *opid*
arg(v-exp) =_{df} < *val-expr* , *s-expr* >
pos(v-exp) =_{df} *infix*
- b) if *v-exp* = *t-expr expl-sort*
 where *t-expr* is a *term-expression*,
expl-sort is a production of [of-symbol sort-identifier],
 then *vat(v-exp)* =_{df} *vat(t-expr)*
arg(v-exp) =_{df} *arg(t-expr)*
pos(v-exp) =_{df} *pos(t-expr)*
- c) if *v-exp* = *vid*
 where *vid* is a *value-identifier*
 then *vat(v-exp)* =_{df} *vid*
arg(v-exp) =_{df} < >
pos(v-exp) =_{df} *undef*
- d) if *v-exp* = *opid arg-list*
 where *opid* is an *operation-identifier*,
arg-list is a production of [*value-expression-list*]
 then *vat(v-exp)* =_{df} *opid*
arg(v-exp) is the list of value-expressions in *arg-list*
pos(v-exp) =_{df} *prefix*
- e) if *v-exp* = (*val-expr*)
 where *val-expr* is a *value-expression*
 then *vat(v-exp)* =_{df} *vat(val-expr)*
arg(v-exp) =_{df} *arg(val-expr)*
pos(v-exp) =_{df} *pos(val-expr)*

7.3.3.3 Operation-assignment

An *operation-assignment* *asg* is a function that maps value-atoms to extended-operation-identifiers or extended-value-identifiers.

7.3.3.4 Consistent operation-assignment

An *operation-assignment* *asg* is *consistent* with respect to a value-expression *v-exp* if the following conditions hold:

- a) the *argument-list* *e-args* of *asg(vat(v-exp))* is identical to the list $\langle res(e_1), \dots, res(e_n) \rangle$, where $arg(v-exp) = \langle e_1, \dots, e_n \rangle$, and $res(e_i)$ is the result-sort of *asg(vat(e_i))* for $1 \leq i \leq n$; and
- b) the position of *asg(vat(v-exp))* is identical to *pos(v-exp)*; and
- c) *asg* is consistent with respect to each value-expression in *arg(v-exp)*.

NOTE - Informally, an operation-assignment *asg* is consistent with respect to a value-expression *v-exp* if the information in the extended-identifiers to which the value-atoms of *v-exp* are mapped, matches the structure of *v-exp*. In particular, the number and sorts of the arguments and the infix/prefix position of each operator must match.

7.3.3.5 Explicit sort indication

A value-expression *v-exp* has an *explicit sort indication* *es* (which is an extended-sort-identifier) with respect to a type-environment *TE*, if $v-exp = t-exp \text{ of } sid$, where *t-exp* is a term-expression, and $es = \#sid\#(TE)$ (see 7.3.4.6 b)).

7.3.3.6 Sound operation-assignment

An *operation-assignment* *asg* is *sound* with respect to a value-expression *v-exp* and a type-environment *TE* if the following conditions hold:

- a) either *v-exp* has no explicit sort indication, or the result-sort of *asg(vat(v-exp))* is identical to the explicit sort indication of *v-exp* with respect to *TE*; and
- b) *asg* is sound with respect to all value-expressions in *arg(v-exp)* with respect to *TE*.

NOTE - Informally, an operation-assignment *asg* is sound with respect to a value-expression *v-exp* if its value-atom is mapped to an extended-identifier whose sort corresponds with the explicit sort indication of *v-exp*.

7.3.3.7 Generated operation-assignments

Let *V* be a set of extended-value-identifiers.

Let *es* either be an extended-sort-identifier, or be 'undef'.

An operation-assignment *asg* for a value-expression *v-exp* is *generated* by *TE*, *V*, and *es* if

- a) *asg* is consistent w.r.t. *v-exp*; and
- b) *asg* is sound w.r.t. *v-exp* and *TE*; and
- c) for all value-atoms *vat* contained in *v-exp*, *asg(vat)* is equal to either
 - 1) an extended-operation-identifier contained in the *OP* of a $te \in TE$ with $te = \langle \langle tid, scp \rangle, \langle S, OP, E \rangle \rangle$, whose operation-identifier has an identical spelling as *vat*; or
 - 2) an extended-value-identifier *e-vid* in *V*, with a value-identifier *vid* that has the same spelling as *vat*;
 and
- d) if *es* is defined then the result-sort of *asg(vat(v-exp))* is identical to *es*.

NOTE - The set of generated operation-assignments for a value-expression $v\text{-exp}$, consists of all operation-assignments that can be taken into consideration for the reconstruction of $v\text{-exp}$, given its environment, which consists of a type-environment, a value-environment, and an explicit sort indication. All elements of this set must therefore be consistent and sound w.r.t. $v\text{-exp}$, and must bind the value-atoms of $v\text{-exp}$ to extended-identifiers in their environment that contain identical value- or operation-identifiers. 7.3.3.7 d) is needed to cater for those cases in which the sort indication of $v\text{-exp}$ is not defined explicitly by an 'of sid ' construct as in 7.3.3.5, but determined implicitly by the environment (e.g. by the sort of an identifier-declaration contained in a local-definition-expression). The value of the parameter es is generated from the context of $v\text{-exp}$ (see also 7.3.3.10).

7.3.3.8 Scopes of an operation-assignment

Let vat be a value-atom, and asg an operation-assignment generated by TE, V and es . The associated scope of $asg(vat)$, $scp(asg(vat))$, is the smallest scope in the set $SCP(asg(vat))$, i.e. that scope $scp \in SCP(asg(vat))$ that contains vat and that is contained in, or equal to all other scopes in $SCP(asg(vat))$, where

$$SCP(asg(vat)) = \{ scp \mid \text{there exists } te \in TE, te = \langle \langle tid, scp \rangle, \langle S, OP, E \rangle \rangle \\ \text{with } asg(vat) \in OP; \text{ or there exists an } e\text{-vid} \in V, e\text{-vid} = \\ asg(vat) = \langle \langle vat, scp \rangle, e\text{-sid} \rangle \text{ for some } e\text{-sid} \}$$

NOTE - Informally, with each operation-assignment asg and value-atom vat of a value-expression $v\text{-exp}$ the smallest scope is associated that contains $v\text{-exp}$ and belongs to a type-definition or identifier-declaration introducing $asg(vat)$. Since asg is generated by the environment of $v\text{-exp}$ such a scope always exists.

7.3.3.9 Minimal operation-assignment

Let asg and asg' be two operation-assignments for a value-expression $v\text{-exp}$ generated by TE, V , and es . asg is said to be *smaller* with respect to $v\text{-exp}$ than asg' if for all value-atoms vat in $v\text{-exp}$, $scp(asg(vat))$ is contained in or equal to $scp(asg'(vat))$.

An operation-assignment asg for a value-expression $v\text{-exp}$ generated by TE, V , and es is *minimal* with respect to $v\text{-exp}$ if asg itself is the only element generated by TE, V , and es that is smaller than asg .

If there exists a unique operation-assignment asg for a value-expression $v\text{-exp}$ generated by TE, V and es , that is minimal, then asg is denoted by $min\text{-}asg(v\text{-exp}, TE, V, es)$. If such a unique minimal assignment does not exist $min\text{-}asg(v\text{-exp}, TE, V, es)$ is undefined.

NOTE - The minimal operation-assignment asg binds all value-atoms of a value-expression $v\text{-exp}$ to the closest possible identifier-declarations or type-definitions possible. Such a minimal operation-assignment need not always exist, as it is possible that only generated assignments exist that minimize bindings for some value-atoms in $v\text{-exp}$, but not for all of them.

7.3.3.10 Reconstruction of a term

Let $v\text{-exp}$ be a value-expression. Then $recon(v\text{-exp}, TE, V, es)$ is defined by:

- a) if $min\text{-}asg(v\text{-exp}, TE, V, es)$ is undefined then $recon(v\text{-exp}, TE, V, es)$ is undefined ; and
- b) if $min\text{-}asg(v\text{-exp}, TE, V, es)$ is defined then $recon(v\text{-exp}, TE, V, es)$ is the result of:
 - 1) the replacement in $v\text{-exp}$ of each subexpression contained in it, of the form $val\text{-expr } opid \text{ } s\text{-expr}$ by $opid(val\text{-expr}, s\text{-expr})$, where $s\text{-expr}$ is a simple-expression, $opid$ is an operation identifier, and $val\text{-expr}$ is a value-expression. The resulting expression is referred to as $v\text{-exp}'$; and
 - 2) removing all occurrences of [*of-symbol sort-identifier*] contained in $v\text{-exp}'$ from it, yielding $v\text{-exp}''$; and
 - 3) replacing all occurrences of '($val\text{-expr}$)' contained in $v\text{-exp}''$ by $val\text{-expr}$, where $val\text{-expr}$ is a value-expression, yielding $v\text{-exp}'''$;