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)

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 passsing 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 quatification 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

$\{a,b,c,...\}$	the *set* made up of elements $a,b,c,...$. 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 $<a,b>$, such that $a \in A$ and $b \in B$.
$A_1 \times A_2 \times ... \times A_n$	the *generalized Cartesian product* of $A_1,A_2,..,A_n$, i.e. the set of ordered n-tuples $<a_1,a_2,..,a_n>$ (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

$a_1,..,a_n$, or $<a_1,..,a_n>$	the finite list (or sequence, or (n-)tuple) made up of the *elements*, or *components* $a_1,..,a_n$. Unlike sets, lists may contain more than one instance of the same element, since elements are distinguished by their position in the ordering of the list; an *ordered pair* is a list of two elements (e.g. $<a,b>$); the *empty* list has no elements and is denoted by $<>$. 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.
$<a_1,..,a_n> \cup <b_1,..,b_n>$	the componentwise union of lists, defined as: $<a_1 \cup b_1,..,a_n \cup b_n>$ ($a_1,..,a_n,b_1,..,b_n$ 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

$a_1 \ldots a_n$	the *string* made up of the elements $a_1, \ldots ,a_n$. 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 $\{a \in A \mid$ there exists some $b \in B$ such that $<a, b> \in R\}$; the *range* of R is defined as $\{b \in B \mid$ there exists some $a \in A$ such that $<a, b> \in R\}$;

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 $<a,c>$ such that there exists some $b \in B$ with $<a,b> \in R$ and $<b,c> \in S$.

$f : A \rightarrow B$ — 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 $<a, b> \in f$;

if $<a,b> \in f$ 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:
$f^{-1} : B \rightarrow A =_{df} \{<b, a> \mid f(a) = b\}$;

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:

a) 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*.
b) the *x in* a *y*: is synonymous with 'the *x* of a *y*'.
c) 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.
d) an *x contained* in a *y*: refers to any *x* of a *y*, or to any *x* of a *z* contained in a *y*.
e) the *y closest-containing* an *x*: refers to that *y* which contains an *x* but does not contain another *y* containing that *x*.
f) 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	Pronounciation
"*xyz*"	terminal symbol *xyz*	*xyz*
abc	nonterminal symbol *abc*	(nonterminal) *abc*
=	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 $a_1,..., a_n$ represent the context information of *N*, then

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

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

In general the translation mapping is partial: only for correct combinations of occurrences *N* with contexts $a_1, \ldots, a_n$ the translation is defined; combinations $N, a_1, \ldots, a_n$ for which $\#N\#(a_1, \ldots, a_n)$ 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 #.# 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 = "B" DS
    then #N# = #DS#(binary)
b)  if N is a number that is a decimal-number, DS is a digit-string
    with N = DS
    then #N# = #DS#(decimal)
c)  if DS is a digit-string, d is a digit
    with N = d
    then #DS #(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#(number-system) =
                  if  number-system = decimal
                  then  #DS'#(decimal) * 10 + #d#
                  else if digit is "0" or "1" then  #DS'#(binary) * 2 + #d#
e)  if d is a digit
    then          #d# = 0           if d = "0"
                  ...
                  #d# = 9           if d = "9"
```

For each production the translation of its left-hand side is defined, with context information as parameters (i.c. *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

```
          digit-string  =  digit-string  digit .
```

would look like:

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

digit-string = digit-string' digit

```
digit-string.error = (digit-string.number-system = binary) and (digit is not "0" or "1");
digit-string.value =    if digit-string.number-system = decimal
                                  then      digit-string'.value *10 + digit.value
                                  else      if  digit-string.error
                                            then  0
                                            else  digit-string'.value * 2 + digit.value;
digit-string'.number-system = digit-string.number-system.
```

4.8 Derivation systems

A *derivation system* is a 3-tuple $D = <A,Ax,I>$ with:

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

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

$$R: \frac{P_1,\ldots,P_n}{Q} \qquad \textit{where } P_1,..,P_n,Q \in A$$

NOTE - The informal meaning of such a rule R is that if the assertions $P_1,..,P_n$ (the *premisses* 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

$$\frac{x=y,\ y=z}{x=z}$$ has instances like

$$\frac{1+1=2\ ,\ 2=3-1}{1+1=3-1} \quad \text{and} \quad \frac{1=3,\ 3=2+1}{1=2+1}$$

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 $R \in I$

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

with $P_1,..,P_n$ 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 <*D*,*O*> 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 <*S*, *Act*, *T*, s_0>, 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 $-a\rightarrow \subseteq S \times S$ 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 <*cur*, *a*, *next*>, also represented as *cur* $-a\rightarrow$ *next*, such that <*cur*, *next*> $\in -a\rightarrow$.

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

a) $s_{11} -a\rightarrow s_{12}$ iff $f(s_{11}) -a\rightarrow f(s_{12})$ 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 $cur -a\rightarrow next$ 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 $< S, L \cup \{\mathbf{i}\}, A, T, s >$ with

a) *L* is a set of *labels*, or *gates*; and

b) $\mathbf{i} \notin L$ is an *internal event*; and

c) $A = < D, O >$ is a many-sorted algebra,

such that for $Act =_{df} \{\mathbf{i}\} \cup \{ gv \mid g \in L, v \in (\bigcup D)^* \}$

d) $< S, Act, T, s >$ 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 characterset 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 ('||','|||','|[a_1,...,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 '|[a_1,...,a_n]|') is *not* associative, e.g. in general 'B_1 || (B_2 |[a_1,...,a_n]| B_3)' defines another behaviour than '(B_1 || B_2) |[a_1,...,a_n]| B_3'.

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 .

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.

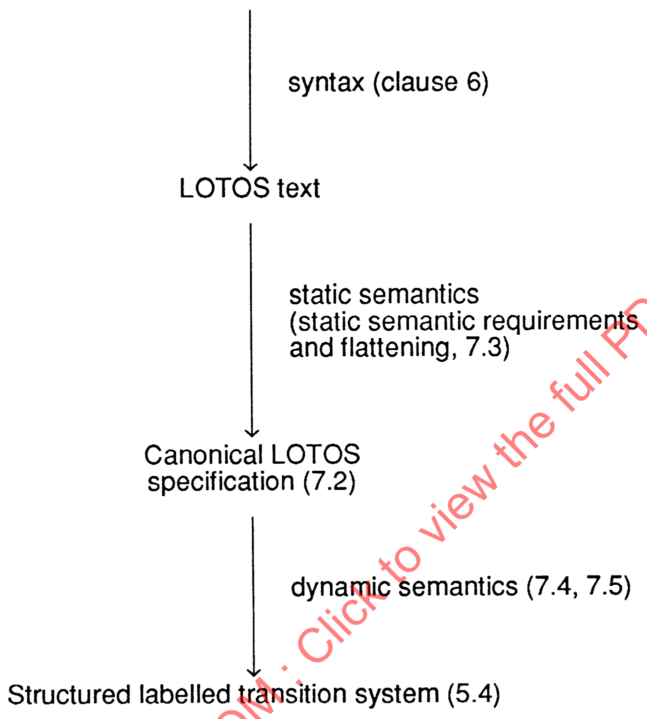
syntax (clause 6)
LOTOS text
static semantics
(static semantic requirements
and flattening, 7.3)
Canonical LOTOS
specification (7.2)
dynamic semantics (7.4, 7.5)
Structured labelled transition system (5.4)

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 $arg(op)=<>$ 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 $< S, OP >$ 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 $< S, OP >$ 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 $i = 1,..., n$, then for each $op \in OP$ with $arg(op)=<s_1,...,s_n>$ and $res(op)=s$, $op(t_1,..., t_n)$ 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 $TERM(OP, \varnothing, s)$.

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 $< S, OP >$ is a triple $< V, L, R >$, where

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

An equation $e'=<\varnothing,L',R'>$ is a *ground instance* of an equation $e=<V,L,R>$ if there exists a function $h{:}V \rightarrow \bigcup\{TERM(OP,s) \mid s \in S\}$ 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 $h{:}V \rightarrow \bigcup\{TERM(OP,s) \mid s \in S\}$.

The notation $L = R$ is used for the ground instance $< \varnothing, L, R >$ of an equation.

NOTE - Also an equation $< V, L, R >$ 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 < *S*, *OP* > is a triple < *V*, *Eq*, *e* >, where

a) *V* is a set of variables; and
b) *Eq* is a set of equations with respect to < *S*, *OP* > with variables in *V*; and
c) *e* is an equation of sort *s* with respect to < *S*, *OP* > with variables in *V*.

NOTE - A conditional equation < *V*, {e_1, .., e_n}, *e* > may be written as e_1, .., e_n => *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 < *S*, *OP*, *E* >, where

a) < *S*, *OP* > is a signature; and
b) *E* is a set of conditional equations with respect to < *S*, *OP* >.

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:

a) *sort-identifier* occurrences in an *exit-parameter* are replaced by *sort-names*;
b) *process-identifier* occurrences in a *process-instantiation* are replaced by *process-names*;
c) all occurrences of *gate-identifier* are replaced by *gate-names*;
d) if *gd* is a *gate-declaration* with *gd* = g_1, ...,g_m **in** [a_1,...a_n] then *gd* is replaced by g_1 **in** [a_1,...a_n], ..., g_m **in** [a_1,...,a_n];
e) all *value-expression* occurrences are replaced by *terms*;
f) if *id* is an *identifier-declaration* with *id* = vid_1,...,vid_n : *sid* contained in:
    1) an *identifier-declarations* then *id* is replaced by v_1,...,v_n where v_i are *variables*;
    2) an *identifier-equation id* = *v-exp* then this equation is replaced by $v_1 = t$, ..., $v_n = t$ where v_i are variables and *t* is a *term*;
    3) an *experiment-offer* ? *id* then the experiment-offer is replaced by ? v_1 ..... ? v_n where v_i are *variables*;
g) all occurrences of *simple-equation* are replaced by formal equations of the form < *V*, *L*, *R* > (see 7.2.2.4);
h) all occurrences of *boolean-expression* are replaced by formal equations of the form < *V*, *L*, true >.

7.2.3.3 Behaviour specification

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

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

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* $CLS = < AS, BS >$ consists of an algebraic specification *AS* and a behaviour specification *BS* such that the signature $<S,OP>$ 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 *#.#:LOTOS texts → 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-spid* = < *spidf*, *fg*, *fv*, *func* >

where *spidf* = < *spid*, *scp* > is the *specification-identification*, consisting of a *specification-identifier spid* and a scope *scp*;
fg = < $e\text{-}gid_1$, ..., $e\text{-}gid_n$ > is the *formal-gate-parameter-list*, consisting of *extended-gate-identifiers*;
fv = < $e\text{-}vid_1$, ..., $e\text{-}vid_m$ > 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-tid* = < *tid*, *scp* >

where *tid* is a *type-identifier*;
scp is a *scope*.

c) *Extended-sort-identifier*: *e-sid* = < *sid*, *e-tid* >

where *sid* is a *sort-identifier*;
e-tid is an *extended-type-identifier*.

d) *Extended-operation-identifier*: *e-opid* = < *opidf*, *e-args*, *e-res*, *pos* >

where *opidf* = < *opid*, *e-tid* > is an *operation-identification*, consisting of an *operation-identifier opid* and an *extended-type-identifier e-tid*;
e-args = < $e\text{-}sid_1$, ..., $e\text{-}sid_k$ > is the *argument-list*, consisting of a list of *extended-sort-identifiers*;
e-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-pid* = < *pidf*, *fg*, *fv*, *func* >
where *pidf* = < *pid*, *scp* > is the *process-identification*, consisting of *process-identifier pid* and scope *scp*;
fg = < $e\text{-}gid_1, \ldots, e\text{-}gid_n$ > is the *formal-gate-parameter-list*, consisting of *extended-gate-identifiers*;
fv = < $e\text{-}vid_1, \ldots, e\text{-}vid_m$ > 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-gid* = < *gid*, *scp* >
where *gid* is a *gate-identifier*;
scp is a *scope*.

g) *Extended-value-identifier*: *e-vid* = < < *vid*, *scp* >, *e-sid* >
where *vid* is a *value-identifier*;
scp is a *scope*;
e-sid is an *extended-sort-identifier*.

NOTE - During the static semantics phase an extended-identifier *e-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:

a) 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
b) 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-args*, the *result-sort e-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-args*=∅; the result-sort of an extended-value-identifier *<<vid,scp>,e-sid>* is *e-sid*; the position is *undefined*: *pos*=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:

a) extended-sort-identifiers correspond to sort-names; and
b) extended-operation-identifiers correspond to operation-names; and
c) 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) $arg(e\text{-}opid) =_{df} e\text{-}opid.e\text{-}args$ for each extended-operation-identifier *e-opid*; and
g) $res(e\text{-}opid) =_{df} e\text{-}opid.e\text{-}res$ for each extended-operation-identifier *e-opid*; and
h) $fg(e\text{-}pid) =_{df} e\text{-}pid.fg$ for each extended-process-identifier *e-pid*; and
j) $fv(e\text{-}pid) =_{df} e\text{-}pid.fv$ for each extended-process-identifier *e-pid*; and
k) $sort(e\text{-}vid) =_{df} e\text{-}vid.e\text{-}sid$ 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 $< e\text{-}sid_1, ..., e\text{-}sid_k >$ ($k \geq 0$), 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* = < *S* ', *OP* ', *E* '> is an algebraic specification < *S* , *OP* , *E* >, 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* = < *fpres*, *tpres* > 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* = < *S*, *OP*, *E* > is *non-overlapping* if:

a) the *sort-identifiers* of all *extended-sort-identifiers* in *S* are different; and
b) 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* = < *fpres*, *tpres* > 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 $sig_1 = < S_1, OP_1 >$ and $sig_2 = < S_2, OP_2 >$ be signatures consisting of extended-identifiers. A *signature morphism* g: $sig_1 \rightarrow sig_2$ is a pair of functions

$$g = < gs: S_1 \rightarrow S_2, gop: OP_1 \rightarrow OP_2 >$$

such that for all $e\text{-}opid_1 = < opidf_1, < e\text{-}sid_1, \ldots, e\text{-}sid_k >, e\text{-}res, pos > \in OP_1$:

$$gop\,(e\text{-}opid_1) = < opidf_2, < gs(e\text{-}sid_1), \ldots, gs(e\text{-}sid_k) >, gs(e\text{-}res), pos >$$

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 $pres_1 = < S_1, OP_1, E_1 >$ and $pres_2 = < S_2, OP_2, E_2 >$ be data-presentations.

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

$$g' = < gs: S_1 \rightarrow S_2, gop: OP_1 \rightarrow OP_2\,, geq: E_1 \rightarrow E_1' >$$

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

a) replacing all extended-sort-identifiers *e-sid* contained in *e* by *gs*(*e-sid*); and
b) replacing all extended-operation-identifiers *e-opid* contained in *e* by *gop*(*e-opid*); and
c) replacing all extended-value-identifiers $e\text{-}vid = < < vid, scp >, e\text{-}sid >$ contained in *e* by $< < vid, scp >, gs(e\text{-}sid) >$.

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

$$\{ geq(e) \mid D_{pres1} \vdash e \} \subseteq \{ e \mid D_{pres2} \vdash e \}$$

where

$\{ geq(e) \mid D_{pres1} \vdash e \}$ and $\{ e \mid D_{pres2} \vdash e \}$ are sets of ground equations;
D_{pres1} and D_{pres2} 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*: $te = < e\text{-}tid, ppres >$.

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

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

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 <identifier,scope>-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:

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

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=<<tid,scp>,ppres>*, 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:
$cpres = < < \varnothing, \varnothing, \varnothing >, < S, OP, E > >$.

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*:

$$CTE = complete(TE) = \{ < e\text{-}tid, pres > \mid < e\text{-}tid, ppres > \in TE, ppres = < < \varnothing, \varnothing, \varnothing >, \mathrm{pres} > \}$$

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*$_1$ .... *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-def*$_1$ is said to refer directly to a *data-type-definition type-def*$_2$ if the *type-identifier* in *type-def*$_2$ is equal to a *type-identifier* contained in the *p-expression* of *type-def*$_1$.

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 < *res*(e_1),.., *res*(e_n) >, where *arg*(*v-exp*) = < e_1,.., e_n >, 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* **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* = <<*tid* ,*scp* >,<*S*,*OP*,*E* >>, 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-exp*, consists of all operation-assignments that can be taken into consideration for the reconstruction of *v-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-exp*, and must bind the value-atoms of *v-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-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-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*∈ *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* | there exists *te* ∈ *TE*, *te* = <<*tid*,*scp* >,<*S*,*OP*,*E* >> with *asg*(*vat*)∈ *OP* ; or there exists an *e-vid* ∈ *V*, *e-vid* = *asg*(*vat*) = < < *vat*, *scp* >, *e-sid* > for some *e-sid* }

NOTE - Informally, with each operation-assignment *asg* and value-atom *vat* of a value-expression *v-exp* the smallest scope is associated that contains *v-exp* and belongs to a type-definition or identifier-declaration introducing *asg*(*vat*). Since *asg* is generated by the environment of *v-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-exp* generated by *TE*, *V*, and *es*. *asg* is said to be *smaller* with respect to *v-exp* than *asg'* if for all value-atoms *vat* in *v-exp*, *scp*(*asg*(*vat*)) is contained in or equal to *scp*(*asg'*(*vat*)).

An operation-assignment *asg* for a value-expression *v-exp* generated by *TE*, *V*, and *es* is *minimal* with respect to *v-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-exp* generated by *TE*, *V* and *es*, that is minimal, then *asg* is denoted by *min-asg*(*v-exp*, *TE*, *V*, *es*). If such a unique minimal assignment does not exist *min-asg*(*v-exp*, *TE*, *V*, *es*) is undefined.

NOTE - The minimal operation-assignment *asg* binds all value-atoms of a value-expression *v-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-exp*, but not for all of them.

7.3.3.10 Reconstruction of a term

Let *v-exp* be a value-expression. Then *recon*(*v-exp*, *TE*, *V*, *es*) is defined by:

a) if *min-asg*(*v-exp*, *TE*, *V*, *es*) is undefined then *recon*(*v-exp*, *TE*, *V*, *es*) is undefined ; and

b) if *min-asg*(*v-exp*, *TE*, *V*, *es*) is defined then *recon*(*v-exp*, *TE*, *V*, *es*) is the result of:

   1) the replacement in *v-exp* of each subexpression contained in it, of the form *val-expr opid s-expr* by *opid*(*val-expr* ,*s-expr*), where *s-expr* is a simple-expression, *opid* is an operation identifier, and *val-expr* is a value-expression. The resulting expression is referred to as *v-exp* '; and

   2) removing all occurrences of [*of-symbol sort-identifier*] contained in *v-exp* ' from it, yielding *v-exp*''; and

   3) replacing all occurrences of '(' *val-expr* ')' contained in *v-exp* '' by *val-expr*, where *val-expr* is a value-expression, yielding *v-exp* ''';

4) replacing in *v-exp* "' each value-atom *vat* contained in it by *min-asg*(*v-exp*, *TE*, *V*, *es*)(*vat*), yielding *recon*(*v-exp*, *TE*, *V*, *es*).

NOTE - The function *recon*, if defined, reconstructs the intended binding for value-atoms in a value-expression *v-exp* following the minimal operation-assignment. At the same time all operations are reformatted to a prefix format, and syntax that has become redundant, such as parentheses and explicit sort indications, are removed. The resulting expression is a term according to 7.2.2.2.

The requirements concerning the reconstructibility of value-expressions in LOTOS are expressed as requirements on the definedness of *recon* applications in the flattening processes described in subclauses 7.3.4.3 and 7.3.4.5. In these applications the appropriate values for the environment parameters *TE*, *V*, and *es* are furnished. Essentially, the requirements express that overloading is permitted if minimal operation-assignments exist for the value-expressions involved.

7.3.4 Flattening of a LOTOS specification

7.3.4.1 Introduction

The subclauses of 7.3.4 define the flattening function #.#: LOTOS texts → canonical LOTOS specifications. The static semantic requirements are formulated as conditions for the definedness of #.#. If these requirements are not fulfilled for a given text, this text is incorrect as part of a LOTOS specification and the result of application of #.# is not defined.

The definition of #.# is by syntactic recursion over the non-terminals of the syntax defined in 6.2., and therefore the domain of #.# consists of all texts generated by one of the non-terminals of 6.2. The range is the set of canonical LOTOS specifications and all substructures of canonical LOTOS specifications.

In the following subclauses explanatory notes have been added. Specific features are only commented on at their first occurrence. The functional structure of #.# is summarized in table 3 in 7.3.5.

NOTE - If in the definition of the flattening function #.# this function, or auxiliary functions are applied to optional parts of the specification text, the result of this application is taken to be the appropriate empty result (empty set, empty list, etc., depending on the expected type of the result) in the case that the optional part is not present.

7.3.4.2 Flattening of specification

a) if
spec is a specification,
spid is a specification-identifier,
fp is a formal-parameter-list,
types is a global-type-definitions,
db is a definition-block

with
spec = **specification** *spid fp types* **behaviour** *db* **endspec**

then
#*spec*# = *CLS*

where
CLS = <*AS*,*BS* > with
AS = *pres*; and
BS = <*P*,p_0 >; and
pres = *combine* (*complete* (*global-TE*)) ∪ *pres* '
global-TE = #*types* #(∅,*scp* (*spec*))
p_0 = <*e-spid* ,*B* '>
P = {p_0} ∪ *P* '
e-spid = <<*spid* ,*scp* (*spec*)>, #*fp* #(*global-TE* ,*scp* ')>

$\#db\ \#(global\text{-}TE, \varnothing, e\text{-}spid) = <B', pres', P'>$
$scp' = scp\,(beh)$ with *beh* is the behaviour-expression of *db*.

NOTE - The result of flattening a specification is a canonical LOTOS specification *CLS*, consisting of a algebraic specification *AS* and a behaviour specification *BS*. *AS* is the *combination* (see 7.3.2.10) of all *complete* (7.3.2.12) data-presentations in the specification, both of global data-type-definitions (*global-TE*) and of local data-type-definitions (*pres'*). *BS* is the set of all local process-definitions in the specification (*P'*) united with p_0, the process-definition of the specification itself.

b) if *db* is a definition-block,
beh is a behaviour-expression,
ldefs is a local-definitions,
types is the data-type-definitions that is the concatenation of all data-type-definition occurrences in *ldefs*,
procs is the process-definitions that is the concatenation of all process-definition occurrences in *ldefs* with process-definitions = { process-definition } .

with

db = *beh* **where** *ldefs*

then

$\#db\ \#(TE, PE, e\text{-}pid) = <B, pres, P>$

where

$B = \#beh\ \#(complete\,(TE \cup local\text{-}TE), PE \cup local\text{-}PE, e\text{-}pid.fg, e\text{-}pid.fv)$
$pres = combine\,(complete\,(local\text{-}TE)) \cup pres'$
$<pres', P> = \#procs\ \#(TE \cup local\text{-}TE, PE \cup local\text{-}PE)$
$local\text{-}TE = \#types\ \#(TE, scp\,(db))$
$local\text{-}PE = e\text{-}pids\,(procs, TE \cup local\text{-}TE, scp\,(db))$

Requirement b1: $e\text{-}pid.func = func\,(beh, complete\,(TE \cup local\text{-}TE), PE \cup local\text{-}PE, e\text{-}pid.fv)$.

NOTE - The result of flattening a definition-block is a triple, consisting of a behaviour-expression-structure *B* (7.2.3.2), a complete data-presentation *pres* and a set of process-definitions *P*. *pres* is the *combination* of all *complete* data-presentations of the data-type-definitions in the local-definitions *ldefs*, united with the data-presentation of all data-type-definitions defined locally in one of the local process-definitions in *ldefs* (*pres'*). *Process-definitions* is a non-terminal symbol not found in clause 6.2, but introduced here for ease of definition. The requirement b1) states that the functionality (7.3.2.4) specified in the header of a process-definition shall be equal to the functionality of the corresponding behaviour-expression.

7.3.4.3 Flattening of data-type-definitions

a) if *types* is a data-type-definitions,
$type_1, \ldots, type_n$ are data-type-definition occurrences

with

$type_1\ \ldots.\ type_n$ is in valid dependence order

and

types is a permutation of $type_1\ \ldots.\ type_n$

then

$\#types\ \#(TE, scp) = \bigcup\{TE_i \mid 1 \le i \le n\}$

where

$TE_i = \#type_i\#(TE \cup \bigcup\{TE_j \mid 1 \le j \le i\text{-}1\}, scp)\ 1 \le i \le n$

Requirement a1: There shall be a valid dependence order for *types*;
Requirement a2: For all $<<tid_i, scp>, ppres_i> \in \#types\ \#(TE, scp)$, the type-identifiers tid_i shall be pairwise different;
Requirement a3: $combine\ (complete\ (\#types\ \#(TE, scp)))$ shall be a non-overlapping data-presentation.

NOTE - The result of flattening a data-type-definitions is a *type-environment* (7.3.2.10), which is the union of all type-environments of the constituting data-type-definition occurrences. The data-type-definition occurrences may be written in any order, but there shall be no circular definition of any of the data-type-definitions [requirement a1)]. All type-identifiers of the defined types shall be different [requirement a2)] and there shall be no doubly defined sorts or operations in types that may be used in behaviour-expressions [requirement a3)].

b) if *type* is a data-type-definition,
tid$_1$, ..., *tid*$_n$ are type-identifier occurrences
with
type = **library** *tid*$_1$, ..., *tid*$_n$ **endlib**
then
$\#type\,\#(TE\,,scp\,) = \{\, <<tid_i\,,scp\,>,ppres_i> \mid ppres_i = \#tid_i\,\#(LIB),\ 1 \le i \le n\,\}$

NOTE - A library-declaration specifies a subset of the type-environment *LIB* (the type-environment of the standard library, see 7.3.2.11), where the scope of the extended-type-identifiers is replaced by the local scope *scp*.

c) if *type* is a data-type-definition,
tid is a type-identifier,
pexp is a p-expression
with
type = **type** *tid* **is** *pexp* **endtype**
then
$\#type\,\#(TE\,,scp\,) = \{<<tid\,,scp\,>,\ \#pexp\,\#(TE\,,<tid\,,scp\,>)>\}$

NOTE - The result of the flattening is a type-environment with one element, viz. the pair of the extended-type-identifier and data-presentation defined by this data-type-definition.

d) if *pexp* is a p-expression,
tun is a type-union,
pspec is a p-specification
with
pexp = *tun pspec*
then
$\#pexp\#(TE\,,e\text{-}tid\,) = \#pspec\,\#(e\text{-}tid\,,\#tun\,\#(TE\,))$

NOTE - The result of flattening a p-expression is a parameterized data-presentation (7.3.2.6).

e) if *pexp* is a p-expression,
tid is a type-identifier,
tun is a type-union,
rep is a replacement
with
pexp = *tid* **actualizedby** *tun* **using** *rep*
then
$\#pexp\#(TE,e\text{-}tid\,) = <fpres_2,\ tpres_2 \cup g'(tpres_1\text{-}fpres_1)>$
where
$ppres_1 = <fpres_1,tpres_1> = <<FS_1,FOP_1,FE_1>,<S_1,OP_1,E_1>> = \#tid\,\#(TE\,)$
$ppres_2 = <fpres_2,tpres_2> = <<FS_2,FOP_2,FE_2>,<S_2,OP_2,E_2>> = \#tun\,\#(TE\,)$
g': $tpres_1 \rightarrow g'(tpres_1)$ is the data-presentation morphism defined by the signature morphism g: $<S_1,OP_1> \rightarrow <gs(S_1),gop(OP_1)>$ and *gs* and *gop* defined as follows:

if *sid*$_1$, ..., *sid*$_n$, *sid*$_1$', ..., *sid*$_n$' are sort-identifier occurrences,
opid$_1$, ..., *opid*$_m$, *opid*$_1$', ..., *opid*$_m$' are operation-identifier occurrences
with

rep = **sortnames** sid_1 **for** sid_1'
...
sid_n **for** sid_n'
opnnames $opid_1$ **for** $opid_1'$
...
$opid_m$ **for** $opid_m'$

then $gs(<sid,t>)$ $= <sid_i,t'>$ if $<sid,t> \in FS_1$ and $sid = sid_i'$ $(1 \le i \le n)$ where $<sid_i,t'> \in S_2$

$= <sid,t'>$ if $<sid,t> \in FS_1$ and $sid \ne sid_i'$ $(1 \le i \le n)$ where $<sid,t'> \in S_2$

$= <sid_i,e\text{-}tid>$ if $<sid,t> \in S_1\text{-}FS_1$ and $sid = sid_i'$ $(1 \le i \le n)$

$= <sid,t>$ if $<sid,t> \in S_1\text{-}FS_1$ and $sid \ne sid_i'$ $(1 \le i \le n)$

$=$ undefined otherwise

$gop(<<opid,t>,e\text{-}args,e\text{-}res,pos>)$

$= <<opid_j,t'>,gs'(e\text{-}args),gs(e\text{-}res),pos>$
if $<<opid,t>,e\text{-}args,e\text{-}res,pos> \in FOP_1$,
and $opid = opid_j'$ $(1 \le j \le m)$
where $<<opid_j,t'>,gs'(e\text{-}args),gs(e\text{-}res),pos> \in OP_2$

$= <<opid,t'>,gs'(e\text{-}args),gs(e\text{-}res),pos>$
if $<<opid,t>,e\text{-}args,e\text{-}res,pos> \in FOP_1$
and $opid \ne opid_j'$ $(1 \le j \le m)$
where $<<opid,t'>,gs'(e\text{-}args),gs(e\text{-}res),pos> \in OP_2$

$= <<opid_j,e\text{-}tid>,gs'(e\text{-}args),gs(e\text{-}res),pos>$
if $<<opid,t>,e\text{-}args,e\text{-}res,pos> \in OP_1\text{-}FOP_1$,
and $opid = opid_j'$ $(1 \le j \le m)$

$= <<opid,t>,e\text{-}args,e\text{-}res,pos>$
if $<<opid,t>,e\text{-}args,e\text{-}res,pos> \in OP_1\text{-}FOP_1$,
and $gs'(e\text{-}args) = e\text{-}args$ and $gs(e\text{-}res) = e\text{-}res$,
and $opid \ne opid_j'$ $(1 \le j \le m)$

$= <<opid,e\text{-}tid>,gs'(e\text{-}args),gs(e\text{-}res),pos>$
if $<<opid,t>,e\text{-}args,e\text{-}res,pos> \in OP_1\text{-}FOP_1$,
and $gs'(e\text{-}args) \ne e\text{-}args$ or $gs(e\text{-}res) \ne e\text{-}res$,
and $opid \ne opid_j'$ $(1 \le j \le m)$

$=$ undefined otherwise

where gs' is the elementwise application of gs.

Requirement e1: All sid_i' shall be pairwise different $(1 \le i \le n)$;
Requirement e2: All $opid_j'$ shall be pairwise different $(1 \le j \le m)$;
Requirement e3: For all sid_i', $(1 \le i \le n)$, there shall be an $e\text{-}sid_i \in S_1$, with $e\text{-}sid_i = <sid_i', e\text{-}tid_i>$ for some $e\text{-}tid_i$;
Requirement e4: For all $opid_j'$ $(1 \le j \le m)$, there shall be an $e\text{-}opid_j \in OP_1$, with $e\text{-}opid_j = <<opid_j', e\text{-}tid_j>, e\text{-}args_j, e\text{-}res_j, pos_j>$ for some $e\text{-}tid_j$, $e\text{-}args_j$, $e\text{-}res_j$ and pos_j.
Requirement e5: If $e\text{-}sid \in FS_1$ then $gs(e\text{-}sid) \in S_2$ ($gs(e\text{-}sid)$ defined);
Requirement e6: If $e\text{-}opid \in FOP_1$ then $gop(e\text{-}opid) \in OP_2$ ($gop(e\text{-}opid$ defined);
Requirement e7: $gs : (S_1\text{-}FS_1) \rightarrow gs(S_1\text{-}FS_1)$ shall be injective;
Requirement e8: $gop : (OP_1\text{-}FOP_1) \rightarrow gop(OP_1\text{-}FOP_1)$ shall be injective;

Requirement e9: The result $<fpres_2, tpres_2 \cup g' (tpres_1\text{-}fpres_1)>$ shall be a non-overlapping parameterized data-presentation;
Requirement e10: h: $fpres_1 \rightarrow tpres_2$, being the restriction of g': $tpres_1 \rightarrow g'(tpres_1)$ to $fpres_1$, shall be a well-defined data-presentation morphism (see note of 7.3.2.9).

NOTE - The result of flattening an actualization is a parameterized data-presentation that is the result of substituting the actual data-presentation (*ppres*$_2$) for the formal part of the formal data-presentation (*fpres*$_1$ of *ppres*$_1$). The result of applying the signature morphism *gs* to an extended-sort-identifier *e-sid* of the formal data-presentation *ppres* is shown in table 2. The presence/absence of *e-sid* in the *replacement* of the data-type-definition is defined by the vertical axis; the distinction between *e-sid* being *formal* or *non-formal* is made along the horizontal axis.

Table 2 - Actualization function

***gs* (*e-sid*)**	***e-sid* formal**	***e-sid* not formal**
sid of *e-sid* in replacement	take corresponding *e-sid'* in actual data-presentation *ppres*$_2$	*renaming* of non-formal sort: a new sort is defined here
sid of *e-sid* not in replacement	take sort with same name in actual data-presentation *ppres*$_2$	nothing happens to the sort

In table 2 we can see: formal sorts are always replaced by actual sorts (note that actual sorts may again be formal sorts); non-formal sorts may be *renamed*, exactly the same as in the *renaming* data-type-definition, [see 7.3.4.3 f)].

The function *gop* is defined analogously. There is one extra case: if the argument- or result-sorts of an operation do change as a result of renaming of sorts, a new operation is defined even if the non-formal operation is not explicitly renamed [see also 7.3.4.3.f)].

Requirements e1)-e4) guarantee that the replacement defines a *function* with *domain ppres*$_1$. The function *g* applied to non-formal sorts and operations (the renaming part) shall also be *injective* according to requirements e7)-e8) [compare requirements f5)-f6)]. Requirements e5)-e6) state that all formal sorts and operations in *ppres*$_1$ shall be actualized by a sort, respectively operation of *ppres*$_2$. Well-definedness of the data-presentation morphism *h* [requirement e10)], meaning that all equations of *fpres*$_1$ can be derived from equations in *tpres*$_2$, is formally required, but this property is algorithmically undecidable (see 7.3.2.9).

f) if *pexp* is a p-expression,
tid is a type-identifier,
rep is a replacement

with

pexp = *tid* **renamedby** *rep*

then

$\#pexp\#(TE,e\text{-}tid) = <g'(fpres),g'(tpres)>$

where

$<fpres,tpres> = <<FS,FOP,FE>,<S,OP,E>> = \#tid\#(TE)$;
g': $tpres \rightarrow g'(tpres)$ is the data-presentation morphism defined by the signature morphism $g: <S,OP> \rightarrow <gs(S),gop(OP)>$ and *gs* and *gop* defined as follows:

if $sid_1, ..., sid_n, sid_1', ..., sid_n'$ are sort-identifier occurrences,
$opid_1, ..., opid_m, opid_1', ..., opid_m'$ are operation-identifier occurrences

with

rep = **sortnames** sid_1 **for** sid_1'
...
sid_n **for** sid_n'

opnnames $opid_1$ **for** $opid_1'$
...
$opid_m$ **for** $opid_m'$

then

$gs(<sid,t>) = <sid_i,e\text{-}tid>$ if $sid = sid_i'$ $(1 \le i \le n)$
$= <sid,t>$ otherwise

$gop\,(<<opid,t>,e\text{-}args,e\text{-}res,pos>)$
$= <<opid_j,e\text{-}tid>,gs'(e\text{-}args),gs\,(e\text{-}res),pos>$
if $opid = opid_j'$ $(1 \le j \le m)$
$= <<opid,t>,e\text{-}args,e\text{-}res,pos>$
if $opid \ne opid_j'$ $(1 \le j \le m)$
and $gs'(e\text{-}args) = e\text{-}args$, $gs\,(e\text{-}res) = e\text{-}res$
$= <<opid,e\text{-}tid>,gs'(e\text{-}args),gs\,(e\text{-}res),pos>$
otherwise

where *gs'* is the elementwise application of *gs*.

Requirement f1: All sid_i' shall be pairwise different, $(1 \le i \le n)$;
Requirement f2: All $opid_j'$ shall be pairwise different, $(1 \le j \le m)$;
Requirement f3: For all sid_i' $(1 \le i \le n)$, there shall be an $e\text{-}sid_i \in S$, with $e\text{-}sid_i = <sid_i',e\text{-}tid_i>$ for some $e\text{-}tid_i$;
Requirement f4: For all $opid_j'$ $(1 \le j \le m)$, there shall be an $e\text{-}opid_j \in OP$, with $e\text{-}opid_j{=}<<opid_j',e\text{-}tid_j>, e\text{-}args_j,e\text{-}res_j,pos_j>$ for some $e\text{-}tid_j$, $e\text{-}args_j$, $e\text{-}res_j$ and pos_j.
Requirement f5: $gs : S \rightarrow gs\,(S)$ shall be injective;
Requirement f6: $gop : OP \rightarrow gop\,(OP)$ shall be injective;
Requirement f7: $g'(tpres)$ shall be a non-overlapping data-presentation.

NOTE - The result of flattening a renaming is a parameterized data-presentation that is a renamed copy of the original parameterized data-presentation. During renaming new sorts are defined for some of the original sorts, while other sorts are identified with their renamed version. The same applies to operations. Definition of a new sort or operation means that the extended-type-identifier *t* of the original data-type-definition in the extension of the extended-sort/operation-identifier is changed into the extended-type-identifier *e-tid* of this (renaming) data-type-definition. If a sort is renamed explicitly in the replacement a new sort is defined; otherwise the renamed sort is identified with its original. Operations are identified with their originals if also all argument- and result sorts are identified with their respective originals. The requirements f1)-f6) state that a renaming shall be an *injective function defined on* the original data-presentation *<fpres,tpres>*.

g) if *tun*, tun_1 are type-union occurrences,
tid is a type-identifier

with

$tun = tid$, tun_1

then

$\#tun\#(TE) = \#tid\#(TE) \cup \#tun_1\#(TE)$

Requirement g1: *#tun #(TE)* shall be a non-overlapping parameterized data-presentation.

h) if *pspec* is a p-specification,
fsl, *sl* are occurrences of sort-list,
fopl, *opl* are occurrences of operation-list,
fels, *els* are occurrences of equation-lists

with

pspec = **formalsorts** *fsl* **formalopns** *fopl* **formaleqns** *fels* **sorts** *sl* **opns** *opl* **eqns** *els*

then

$\#pspec\#(e\text{-}tid,ppres_1) = ppres'$

where

$ppres_1 = <fpres_1, tpres_1>$
$= <<FS_1, FOP_1, FE_1>, <FS_1 \cup S_1, FOP_1 \cup OP_1, FE_1 \cup E_1>>$
$ppres_2 = <fpres_2, tpres_2>$
$= <<FS_2, FOP_2, FE_2>, <FS_2 \cup S_2, FOP_2 \cup OP_2, FE_2 \cup E_2>>$
$ppres' = <fpres', tpres'>$
$= <<FS', FOP', FE'>, <FS' \cup S', FOP' \cup OP', FE' \cup E'>>$

$FS_2 = \#fsl\#(e\text{-}tid)$
$S_2 = \#sl\#(e\text{-}tid)$
$FOP_2 = \#fopl\#(e\text{-}tid, FS')$
$OP_2 = \#opl\#(e\text{-}tid, FS' \cup S')$
$FE_2 = \#fels\#(FS', FOP')$
$E_2 = \#els\#(FS' \cup S', FOP' \cup OP')$

$FS' = FS_1 \cup FS_2$
$S' = S_1 \cup S_2$
$FOP' = FOP_1 \cup FOP_2$
$OP' = OP_1 \cup OP_2$
$FE' = FE_1 \cup FE_2$
$E' = E_1 \cup E_2$

Requirement h1: FS_2 and S_2 shall be disjoint.
Requirement h2: FOP_2 and OP_2 shall be disjoint.
Requirement h3: *ppres*' shall be a non-overlapping parameterized data-presentation.

NOTES
1 - $ppres_2$ is not a 'real' presentation: neither all sorts contained in operations and equations, nor all operations contained in equations, are necessarily contained in the sets of sorts and operations, respectively.

2 - The result of flattening a p-specification is a parameterized data-presentation that is an enumeration of the sorts, operations and equations defined in the p-specification, together with the sorts, operations and equations imported via the type-union of the p-expression containing this p-specification. The imported sorts, operations and equations are represented by $ppres_1$. The requirements guarantee that the names of defined sorts and operations are unique, and that there is no overlap with the imported sorts and operations.

j) if *sl* is a sort-list,
$sid_1, ..., sid_n$ are sort-identifier occurrences
with
$sl = sid_1, ..., sid_n$
then
$\#sl\#(e\text{-}tid) = \{ <sid_i, e\text{-}tid> \mid 1 \leq i \leq n \}$

Requirement j1: The sort-identifiers $sid_1, ..., sid_n$ shall be pairwise different.

k) if *opl* is an operation-list,
$opn_1, ..., opn_n$ are operation occurrences
with
$opl = opn_1 \; .... \; opn_n$
then
$\#opl\#(e\text{-}tid, S) = \bigcup \{\#opn_i\#(e\text{-}tid, S) \mid 1 \leq i \leq n\}$

Requirement k1: For all $1 \leq i < j \leq n$ $\#opn_i\#(e\text{-}tid, S) \cap \#opn_j\#(e\text{-}tid, S) = \emptyset$.

NOTE - Overloading of operation-identifiers is allowed: two operations may have the same identifier if they can be distinguished by argument-list, result-sort, or position, i.e. if the complete extended-operation-identifiers are different.

m) if opn is an operation,
$od_1, \ldots, od_n$ are operation-descriptor occurrences,
$sid_1, \ldots, sid_k$, $sid\text{-}r$ are sort-identifier occurrences

with

$opn = od_1 , \ldots , od_n : sid_1 , \ldots , sid_k \rightarrow sid\text{-}r$

then

$\#opn\#(e\text{-}tid,S) = OP$

where

$OP = \{<<opid_i,e\text{-}tid>,<e\text{-}sid_1,\ldots,e\text{-}sid_k>,e\text{-}res,pos> \mid$
$opid_i$ is the operation-identifier in od_i ($1 \le i \le n$);
$e\text{-}sid_j = \#sid_j\#(S)$ ($1 \le j \le k$);
$e\text{-}res = \#sid\text{-}r\#(S)$;
pos = infix if $od_i = \_\ opid_i\ \_$
= prefix otherwise }

Requirement m1: If pos=infix for some extended-operation-identifier $op \in OP$ then k shall be equal to 2;
Requirement m2: All od_i ($1 \le i \le n$) shall be pairwise different.

n) if els is an equation-lists,
ifd is an identifier-declarations,
$el_1, \ldots, el_n$ are occurrences of equation-list

with

$els =$ **forall** $ifd\ el_1 \ldots el_n$

then

$\#els\#(S,OP) = \bigcup \{\#el_i\#(S,OP,V) \mid 1 \le i \le n\}$

where

$V = \#ifd\#(S,scp(el_1 \ldots el_n))$.

NOTE - The result of flattening an equation-lists is a set of conditional equations (7.2.2.4) with respect to $<S,OP>$, which is the union of all sets of conditional equations generated by the equation-list occurrences in the equation-lists.

p) if el is an equation-list,
sid is a sort-identifier,
ifd is an identifier-declarations,
$eqn_1, \ldots, eqn_n$ are occurrences of equation

with

$el =$ **ofsort** sid **forall** $ifd\ eqn_1 \ \ldots \ eqn_n$

then

$\#el\#(S,OP,V) = \{ceqn_i \mid ceqn_i = \#eqn_i\#(S,OP,V',e\text{-}sid\text{-}e), 1 \le i \le n\}$

where

$V' = V \cup \#ifd\#(S,scp(eqn_1 \ldots eqn_n))$
$e\text{-}sid\text{-}e = \#sid\#(S)$

q) if eqn is an equation,
$prem_1, \ldots, prem_n$ are occurrences of premiss,
$seqn$ is a simple-equation

with

$eqn = prem_1 \ldots prem_n \Rightarrow seqn$

then

$\#eqn\#(S,OP,V,e\text{-}sid\text{-}e) = <V,\{ prm_i \mid 1 \le i \le n\},\#seqn\#(S,OP,V,e\text{-}sid\text{-}e)>$

where

if $seqn_i$ is a simple-equation
with
$prem_i = seqn_i$

then

$prm_i = \#seqn_i\#(S,OP,V,undef)$

if $bexp_i$ is a boolean-expression
with

$prem_i = bexp_i$

then

$prm_i = \#bexp_i\#(S,OP,V)$

NOTE - The result of flattening an equation is a conditional equation of sort *e-sid-e* with respect to *<S,OP>*. Distinction is made for a premiss occurrence being a simple-equation or a boolean-expression; the result is always an equation in the sense of 7.2.2.3.

r) if *seqn* is a simple-equation,
vexp$_1$ and *vexp*$_2$ are occurrences of value-expression

with

$seqn = vexp_1 = vexp_2$

then

$\#seqn\#(S,OP,V,e\text{-}sid\text{-}e) = < V, term_1, term_2 >$

where

$term_1 = recon\,(vexp_1, TE, V, e\text{-}sid\text{-}e)$
$term_2 = recon\,(vexp_2, TE, V, e\text{-}sid\text{-}e)$
$TE = \{ <e\text{-}tid,<S,OP,\varnothing>> \}$ for any extended-type-identifier *e-tid*

Requirement r1: $recon\,(vexp_1,TE,V,e\text{-}sid\text{-}e)$ and $recon\,(vexp_2,TE,V,e\text{-}sid\text{-}e)$ shall be defined;
Requirement r2: $sort(term_1) = sort(term_2)$.

NOTE - *term*$_1$ and *term*$_2$ are terms (7.2.2.2) reconstructed from value-expressions *vexp*$_1$, respectively *vexp*$_2$, according to the application of the function *recon*, see 7.3.3.10. Definedness of application of *recon* means that every operation- and value-identifier contained in the value-expression can be uniquely bound to an extended-operation-identifier defined in *OP*, or to an extended-value-identifier defined in *V*. In this case the type-environment *TE* is a 'dummy' type-environment constructed from *S* and *OP*, to be used in the application of *recon* only.

s) if *bexp* is a boolean-expression,
vexp is a value-expression

with

$bexp = vexp$

then

$\#bexp\#(S,OP,V) = <V,term_1,term_2>$

where

$term_1 = recon\,(vexp, TE, V, undef)$
$term_2 = recon\,(\text{'true'}, TE, \varnothing, sort(term_1))$
$TE = \{ <e\text{-}tid,<S,OP,\varnothing>> \}$ for any extended-type-identifier *e-tid*

Requirement s1: $recon\,(vexp,TE,V,undef)$ and $recon\,(\text{'true'}, TE, V, sort(term_1))$ shall be defined.

NOTE - A boolean-expression is transformed into the equivalent equation '*boolean-expression = true*'.

t) if *ifd* is an identifier-declarations contained in an equation-lists,
$vid_{i,j}$, $1 \leq i \leq n$, $1 \leq j \leq m_i$, are value-identifier occurrences,
$sid_1, \ldots, sid_n$ are sort-identifier occurrences

with

$ifd = vid_{1,1}, \ldots, vid_{1,m1} : sid_1,$
.....
$vid_{n,1}, \ldots, vid_{n,mn} : sid_n$

then

$\#ifd\#(S,scp) = \{e\text{-}vid_{i,j} \mid 1 \le i \le n,\ 1 \le j \le m_i\}$

where

$e\text{-}vid_{i,j} = \langle\langle vid_{i,j},scp\rangle, \#sid_i\#(S)\rangle,\ 1 \le i \le n,\ 1 \le j \le m_i$

Requirement t1: All $vid_{i,j}$ shall be pairwise different, $1 \le i \le n$, $1 \le j \le m_i$.

7.3.4.4 Flattening of process definitions

a) if

procs is a process-definitions
where process-definitions = { process-definition } .
$proc_1, \ldots, proc_n$ are occurrences of process-definition

with

$procs = proc_1 \ldots. proc_n$

then

$\#procs\#(TE,PE) = \bigcup \{\langle pres_i,P_i\rangle \mid 1 \le i \le n\}$
$e\text{-}pids\,(procs,TE_1,scp) = \{ep \mid ep = e\text{-}pid(proc_i,TE_1,scp),\ 1 \le i \le n\}$

where

$\langle pres_i,P_i\rangle = \#proc_i\#(TE,PE),\ 1 \le i \le n$

Requirement a1: For all $\langle\langle pid_i,scp\rangle,fg_i,fv_i,func_i\rangle = e\text{-}pid(proc_i,TE_1,scp)$ the process-identifiers pid_i shall be pairwise different, $1 \le i \le n$.

NOTE - The result of flattening a *process-definitions* is a pair $\langle pres,P\rangle$, where $pres = \bigcup\{pres_i \mid 1 \le i \le n\}$ is the union of all data-presentations defined locally in the process-definition occurrences of the *process-definitions*, and $P = \bigcup\{P_i \mid 1 \le i \le n\}$ is the union of all process-definitions $proc_1 \ldots proc_n$, together with the process-definitions occurring locally in $proc_1 \ldots proc_n$. The definition of union of structures is given in clause 4.3. *TE* and *PE* are the environments of data-type-definitions and process-definitions that are used in *procs*. The set *e-pids* is the set of extended-process-identifiers in *procs*. The set *e-pids* appears as part of *PE* via the flattening of *definition-block*, see 7.3.4.2 b). The definition of *e-pids* is necessary to avoid a circular definition of $\#procs\#(TE,PE)$. In *e-pids* all process-identifiers shall be different according to requirement a1).

b) if

proc is a process-definition,
pid is a process-identifier,
fp is a formal-parameter-list,
db is a definition-block

with

proc = **process** *pid fp* := *db* **endproc**

then

$\#proc\#(TE,PE) = \langle pres,P\rangle$
$e\text{-}pid\,(proc,TE_1,scp) = e\text{-}pid$

where

$pres = pres'$
$P = \{\langle e\text{-}pid,B'\rangle\} \cup P'$
$e\text{-}pid = \langle\langle pid,scp\rangle, \#fp\#(TE_1,scp')\rangle$
$\#db\#(TE,PE,e\text{-}pid) = \langle B',pres',P'\rangle$
$scp' = scp\,(beh)$ with *beh* the behaviour-expresssion of *db*.

This page intentionally left blank

Besides the behaviour-expression-structure this subclause defines a functionality *func(beh,TE,PE,VE)* for each behaviour-expression, (see 7.3.2.4). The functionality is needed for the expression of static constraints only, and does not appear in the behaviour-expression-structure that is the result of the flattening of the behaviour-expression.

b) if *beh* is a sum-expression,
beh' is a behaviour-expression,
gd is a gate-declarations,

with

beh = **choice** *gd* [] *beh'*

then

$\#beh\#(TE, PE, GE, VE)$ = **choice** $\#gd\#(GE, scp(beh'))$ [] $\#beh'\#(TE, PE, GE \cup G, VE)$
func(*beh*, *TE*, *PE*, *VE*) = func(*beh'*, *TE*, *PE*, *VE*)

where

$G = \{e\text{-}gid \mid {}'e\text{-}gid$ **in** $[e\text{-}gid_1, .., e\text{-}gid_n]' \in \#gd\#(GE, scp(beh'))$
for some extended-gate-identifiers $e\text{-}gid_1, .., e\text{-}gid_n\}$

if *ifd* is an identifier-declarations,

with

beh = **choice** *ifd* [] *beh'*

then

$\#beh\#(TE, PE, GE, VE)$ = **choice** $\#ifd\#(TE, scp(beh'))$ [] $\#beh'\#(TE, PE, GE, VE \cup V)$
func(*beh*, *TE*, *PE*, *VE*) = *func*(*beh'*, *TE*, *PE*, $VE \cup V$)

where

$V = \{e\text{-}vid \mid e\text{-}vid \in \#ifd\#(TE, scp(beh'))\}$

c) if *beh* is a par-expression,
beh' is a behaviour-expression,
gd is a gate-declarations,
op is parallel-operator

with

beh = **par** *gd op beh'*

then

$\#beh\#(TE, PE, GE, VE)$ = **par** $\#gd\#(GE,scp(beh'))\#op\#(GE)\ \#beh'\#(TE, PE, GE \cup G, VE)$
func(*beh*, *TE*, *PE*, *VE*) = *func*(*beh'*, *TE*, *PE*, *VE*)

where

$G = \{e\text{-}gid \mid {}'e\text{-}gid$ **in** $[e\text{-}gid_1, .., e\text{-}gid_n]' \in \#gd\#(GE, scp(beh'))$
for some extended-gate-identifiers $e\text{-}gid_1, .., e\text{-}gid_n\}$

d) if *beh* is a hiding-expression,
beh' is a behaviour-expression,
$gid_1, .., gid_n$ are gate-identifier occurrences,

with

beh = **hide** $gid_1, ..., gid_n$ **in** *beh'*,

then

$\#beh\#(TE, PE, GE, VE)$ =
hide $< gid_1, scp(beh') >, .., < gid_n, scp(beh') >$ **in** $\#beh'\#(TE, PE, GE \cup G, VE)$
func(*beh*, *TE*, *PE*, *VE*) = *func*(*beh'*, *TE*, *PE*, *VE*)

where

$G = \{< gid_i, scp(beh')> \mid 1 \leq i \leq n\}$

Requirement d1: $gid_1, ... ,gid_n$ shall be pairwise different.

e) if beh is an enable-expression,
beh_1 is a disable-expression,
beh_2 is an enable-expression,
ifd is an identifier-declarations,

with

$beh = beh_1$ >> **accept** ifd **in** beh_2,

then

$\#beh\#(TE, PE, GE, VE) = \#beh_1\#(TE, PE, GE, VE)$ >> **accept** $\#ifd\#(TE, scp(beh_2))$ **in** $\#beh_2\#(TE, PE, GE, VE \cup V)$

$func(beh, TE, PE, VE) = func(beh_2, TE, PE, VE \cup V)$

where

$V = \{e\text{-}vid \mid e\text{-}vid \in \#ifd\#(TE, scp(beh_2))\}$

Requirement e1: $func(beh_1, TE, PE, VE) =< e\text{-}vid_1.e\text{-}sid, ..., e\text{-}vid_n.e\text{-}sid >$
if $\#ifd\#(TE, scp(beh_2)) = < e\text{-}vid_1, .., e\text{-}vid_n >$

f) if beh is a disable-expression,
beh_1 is a parallel-expression,
beh_2 is a disable-expression,

with

$beh = beh_1$ [> beh_2

then

$\#beh\#(TE, PE, GE, VE) = \#beh_1\#(TE, PE, GE, VE)$ [> $\#beh_2\#(TE, PE, GE, VE)$

$func(beh, TE, PE, VE) = max(func(beh_1, TE, PE, VE), func(beh_2, TE, PE, VE))$

Requirement f1: $max(func(beh_1, TE, PE, VE), func(beh_2, TE, PE, VE))$ shall be defined.

NOTE - The maximum *max(func1,func2)* and minimum *min(func1,func2)* for functionalities are defined in 7.3.2.4.

g) if beh is a parallel-expression,
beh_1 is a choice-expression,
beh_2 is a parallel-expression,
op is a parallel-operator

with

$beh = beh_1$ op beh_2,

then

$\#beh\#(TE, PE, GE, VE) = \#beh_1\#(TE, PE, GE, VE)\ \#op\#(GE)\ \#beh_2\#(TE, PE, GE, VE)$

$func(beh, TE, PE, VE) = min(func(beh_1, TE, PE, VE), func(beh_2, TE, PE, VE))$

Requirement g1: $min(func(beh_1, TE, PE, VE), func(beh_2, TE, PE, VE))$ shall be defined.

h) if beh is a choice-expression,
beh_1 is a guarded-expression,
beh_2 is a choice-expression

with

$beh = beh_1$ [] beh_2

then

$\#beh\#(TE, PE, GE, VE) = \#beh_1\#(TE, PE, GE, VE)$ [] $\#beh_2\#(TE, PE, GE, VE)$

$func(beh, TE, PE, VE) = max(func(beh_1, TE, PE, VE), func(beh_2, TE, PE, VE))$

Requirement h1: $max(func(beh_1, TE, PE, VE), func(beh_2, TE, PE, VE))$ shall be defined.

j) if beh is a guarded-expression,
beh' is a guarded-expression,
g is a guard,

with

$beh = g \text{ -> } beh'$

then

$\#beh\#(TE, PE, GE, VE) = \#g\#(TE, VE) \rightarrow \#beh'\#(TE, PE, GE, VE)$
$func(beh, TE, PE, VE) = func(beh', TE, PE, VE)$

k) if

beh is an action-prefix-expression,
beh' is an action-prefix-expression,
a is an action-denotation

with

$beh = a \,;\, beh'$,

then

$\#beh\#(TE, PE, GE, VE) = \#a\#(TE, GE, VE, \mathrm{scp}\,(beh'))\,;\,\#beh'\#(TE, PE, GE, VE \cup V)$
$func(beh, TE, PE, VE) = func(beh', TE, PE, VE \cup V)$

where

$V = \{e\text{-}vid \mid \text{'}?e\text{-}vid\text{'} \text{ contained in } \#a\#(TE, GE, VE, scp\,(beh'))\}$

m) if

beh is an atomic-expression

with

1) $beh =$ **stop** (inaction)

then

$\#beh\#(TE, PE, GE, VE) =$ **stop**
$func(beh, TE, PE, VE) = 0$

2) $exen_1, .., exen_n$ are occurrences of exit-parameter,
sid is a sort-identifier

with

$beh =$ **exit**$(exen_1, .., exen_n)$ (termination)

then

$\#beh\#(TE, PE, GE\,, VE) =$ **exit**$(exen_1', ..., exen_n')$
$func(beh, TE, PE, VE) = <e\text{-}sid_1, ..., e\text{-}sid_n>$

where

$exen_i' = recon(exen_i, TE, VE, \mathrm{undef})$ if $exen_i$ is a value-expression $(1 \le i \le n)$
$=$ **any** $\#sid\#(TE)$ if $exen_i$ is '**any** *sid*' $(1 \le i \le n)$
$e\text{-}sid_i = sort(recon(exen_i, TE, VE, undef))$ if $exen_i$ is a value-expression $(1 \le i \le n)$
$= \#sid\#(TE)$ if $exen_i$ is '**any** *sid*' $(1 \le i \le n)$

Requirement m1: $recon(exen_i, T, VE, undef)$ shall be defined, if $exen_i$ is a value-expression $(1 \le i \le n)$.

NOTE - The result of $recon(exen_i, TE, VE, undef)$ is a term reconstructed from the value-expression $exen_i$, see 7.3.3.10. Definedness of application of *recon* means that every operation- and value-identifier contained in the value-expression can be uniquely bound to an extended-operation-identifier defined in *TE* or to an extended-value-identifier defined in *VE*.

with $beh =$ **exit**

then

$\#beh\#(TE, PE, GE, VE) =$ **exit**
$func(beh, TE, PE, VE) = <>$

3) *pid* is a process-identifier,
$gid_1, ..., gid_n$ are occurrences of gate-identifier,
$E_1, ..., E_m$ are re occurrences of value-expression

with

$beh = pid\,[gid_1, ..., gid_n](E_1, ..., E_m)$ (process-instantiation)

then

$\#beh\,\#(TE, PE, GE, VE) = \#pid\,\#(PE)[\#gid_1\#(GE), ..., \#gid_n\#(GE)]$
$(recon(E_1, TE, VE, undef), ..., recon(E_m, TE, VE, undef))$
$func(beh, TE, PE, VE) = \#pid\#(PE).func$

Requirement m2: $recon(E_j, TE, VE, undef)$ $(1 \le j \le m)$ shall be defined;
Requirement m3: $\#pid\,\#(PE).fg$ has n elements;
Requirement m4: actual-sorts = formal-sorts
where
actual-sorts = $<sort\,(recon(E_1, TE, VE, undef)), ..., sort\,(recon(E_m, TE, VE, undef))>$,
formal-sorts = $< e\text{-}vid_1.e\text{-}sid, ..., e\text{-}vid_m.e\text{-}sid >$
$\#pid\#(PE).fv = < e\text{-}vid_1, ..., e\text{-}vid_m >$.

4) *beh* is a behaviour-expression,
beh' is a behaviour-expression
with
$beh = (\ beh'\ )$ (priority-parentheses)
then
$\#beh\,\#(TE, PE, GE, VE) = (\ \#beh'\#(TE, PE, GE, VE)\ )$
$func(beh, TE, PE, VE) = func(beh', TE, PE, VE)$

n) if *e* is an identifier-equations,
$id_1, ..., id_n$ are identifier-declaration occurrences,
$E_1, ..., E_n$ are value-expression occurrences
with
$e = \ id_1 = E_1, ..., id_n = E_n$
then
$\#e\,\#(TE, VE, scp) = e\text{-}vid_{1,1} = t_1, ..., e\text{-}vid_{1,m1} = t_1$
...
$e\text{-}vid_{n,1} = t_n, ..., e\text{-}vid_{n,mn} = t_n$
where
$\#id_i\,\#(TE, scp) = <e\text{-}vid_{i,1}, ..., e\text{-}vid_{i,mi}>\ (\ 1 \le i \le n\ )$,
$recon(E_i, TE, VE, e\text{-}vid_{i,1}.e\text{-}sid) = t_i\ \ (\ 1 \le i \le n\ )$,

Requirement n1: $recon(E_i, TE, VE, e\text{-}vid_{i,1}.e\text{-}sid)$ $(\ 1 \le i \le n\ )$ shall be defined;
Requirement n2: $vid_{i,j}$ shall be pairwise different for $e\text{-}vid_{i,j}$=$<<vid_{i,j},scp>,e\text{-}sid_{i,j}>$ $(\ 1 \le i \le n, 1 \le j \le m_i\ )$.

p) if *ifd* is an identifier-declarations contained in a behaviour-expression or formal-parameter-list;
$id_1, .., id_n$ are identifier-declaration occurrences,
with
$ifd = id_1, ..., id_n$
then
$\#ifd\,\#(TE, scp) = < e\text{-}vid_1, ..., e\text{-}vid_m >$
where
$< e\text{-}vid_1, ..., e\text{-}vid_m >$ is the concatenation of $\#id_i\,\#(TE, scp)$ $(\ 1 \le i \le n\ )$.

Requirement p1: vid_j shall be pairwise different for $e\text{-}vid_j = <<vid_j,scp>,e\text{-}sid_j>$ $(\ 1 \le j \le m\ )$.

q) if *id* is an identifier-declaration,
$vid_1, ..., vid_n$ are value-identifier occurrences,
sid is a sort-identifier,
with
$id = vid_1, ..., vid_n : sid$
then
$\#id\,\#(TE, scp) = < <<vid_1, scp>, \#sid\,\#(TE)>, ..., <<vid_n, scp>, \#sid\#(TE)> >$

r) if gd is a gate-declarations,
$gid_{i,j}$, ($1 \le i \le n$, $1 \le j \le m_i$) are gate-identifier occurrences,
gt_1, ..., gt_n are gate-tuple occurrences

with

$gd =$ $gid_{1,1}$, ..., $gid_{1,m1}$ **in** gt_1,
...
$gid_{n,1}$, ..., $gid_{n,mn}$ **in** gt_n

then

$\#gd\#(GE, scp) =$
$< gid_{1,1}, scp >$ **in** $\#gt_1\#(GE)$, ..., $< gid_{1,m1}, scp >$ **in** $\#gt_1\#(GE)$
...
$< gid_{n,1}, scp >$ **in** $\#gt_n\#(GE)$, ..., $< gid_{n,mn}, scp >$ **in** $\#gt_n\#(GE)$

Requirement r1: $gid_{1,1}$, ..., $gid_{n,mn}$ shall be pairwise different.

s) if gt is a gate-tuple
gid_1, ..., gid_n are gate-identifier occurrences,

with

$gt = [\ gid_1, ..., gid_n\ ]$

then

$\#gt\#(GE) = [\ \#gid_1\#(GE), ..., \#gid_n\#(GE)\ ]$

t) if op is a parallel-operator,
gid_1, ..., gid_n are gate-identifier occurrences

with

$op = |[gid_1, ..., gid_n]|$

then

$\#op\#(GE) = |[\ \#gid_1\#(GE), ..., \#gid_n\#(GE)\ ]|$

with

$op = ||$ or $op = |||$

then

$\#op\#(GE) = op$

u) if a is an action-denotation,
gid is a gate-identifier,
d_1, ..., d_n are experiment-offer occurrences,
P is a guard

with

$a = gid\ d_1 \ldots d_n\ P$

then

$\#a\#(TE, GE, VE, scp) = \#gid\#(GE)\ d_1' \ldots d_n'\ \#P\#(TE, VE \cup V)$

where

d_i' = ! recon$(E, TE, VE, undef)$
if $d_i = !E$ ($1 \le i \le n$) where E is a value-expression,
= ? $e\text{-}vid_1$ ... ? $e\text{-}vid_m$ with $< e\text{-}vid_1 \ldots e\text{-}vid_m > = \#id\#(TE, scp)$
if $d_i = ?id$ ($1 \le i \le n$) where id is an identifier-declaration
$V = \{\ e\text{-}vid \mid e\text{-}vid \in \#id\#(TE,scp),\ d_i = ?\ id,\ 1 \le i \le n\ \}$

Requirement u1: d_i' shall be defined for all i, $1 \le i \le n$,
Requirement u2: all vid with $e\text{-}vid = <<vid,scp>, e\text{-}sid> \in \#id\#(TE,scp)$ $d_i = ?\ id$ for some i ($1 \le i \le n$) shall be pairwise different.

v) if g is a guard,
L, R are value-expression occurrences,
E is a boolean-expression

with
  $g = [\,L{=}R\,]$
then
  $\#g\#(TE, VE) = [\,eqn\,]$
where
  $eqn = <VE, term_1, term_2>$
  $term_1 = recon(L, TE, VE, undef)$
  $term_2 = recon(R, TE, VE, undef)$

Requirement v1: $recon(L, TE, VE, undef)$ and $recon(R, TE, VE, undef)$ shall be defined;
Requirement v2: $sort(term_1) = sort(term_2)$.

with
  $g = [\,E\,]$
then
  $\#g\#(TE, VE) = [\,eqn\,]$
where
  $eqn = <VE, term_3, term_4>$
  $term_3 = recon(E, TE, VE, undef)$
  $term_4 = recon(\text{'true'}, TE, \varnothing, sort(term_3))$

Requirement v3: $recon(E, TE, VE, undef)$ and $recon(\text{'true'}, TE, \varnothing, sort(term_3))$ shall be defined.

7.3.4.6 Flattening of identifiers

a) if *tid* is a type-identifier
then
  $\#tid\#(TE) = ppres$
where
  $<< tid, scp>, ppres> \in SCOPES(TE, tid)$;
  $SCOPES(TE, tid) = \{<< tid, scp'>, ppres'> \mid << tid, scp'>, ppres'> \in TE\}$;
  the scope denoted by *scp* is contained in, or equal to, *scp'* for all $<< tid, scp'>, ppres'> \in SCOPES(TE, tid)$.

Requirement a1: $SCOPES(TE, tid)$ is not empty.

NOTE - Flattening of identifiers is defined for applied occurrences of identifiers. The result of the flattening is an extended-identifier from an environment of defined extended-identifiers, except for type-identifier where the result is the parameterized data-presentation of that type-identifier. The extended-identifier that is selected from the environment must have the same (unextended) name as the argument identifier. The requirement states that at least one such extended-identifier shall exist in the environment. If more extended-identifiers are found the one with the smallest scope containing the applied occurrence is chosen. The requirements on constructing environments in the other clauses of the flattening definition guarantee that this extended-identifier is unique.

b) if *sid* is a sort-identifier
then
  $\#sid\#(S) = <sid, e\text{-}tid>$
where
  $<sid, e\text{-}tid> \in S$

Requirement b1: There shall be exactly one $<sid, e\text{-}tid> \in S$.

if *sid* is a sort-identifier
then
  $\#sid\#(TE) = e\text{-}sid$

where

$< e\text{-}sid, scp > \in SCOPES(TE, sid);$

$SCOPES(TE, sid) = \{< e\text{-}sid', scp' > \mid e\text{-}sid' = < sid, e\text{-}tid'' >,$
there exists S with $< sid, e\text{-}tid'' > \in S$,
and $<< tid', scp' >, < S, OP, E >> \in TE$
for some tid', $e\text{-}tid''$, OP and $E\}$;

the scope denoted by *scp* is contained in, or equal to, *scp'* for all $< e\text{-}sid', scp' > \in SCOPES(TE, sid)$.

Requirement b2: $SCOPES(TE, sid)$ is not empty.

c) if *pid* is a process-identifier
then
$\#pid\#(PE) = e\text{-}pid$
where
$e\text{-}pid = << pid, scp >, fg, fv, func > \in SCOPES(PE, pid);$
$SCOPES(PE, pid) = \{<< pid, scp' >, fg', fv', func' > \mid << pid, scp' >, fg', fv', func' > \in PE\}$
the scope denoted by *scp* is contained in, or equal to, *scp'*
for all $<< pid, scp' >, fg', fv', func' > \in SCOPES(PE, pid)$.

Requirement c1: $SCOPES(PE, pid)$ is not empty.

d) if *gid* is a gate-identifier
then
$\#gid\#(GE) = e\text{-}gid$
where
$e\text{-}gid = < gid, scp > \in SCOPES(GE, gid);$
$SCOPES(GE, gid) = \{< gid, scp' > \mid < gid, scp' > \in GE\};$
the scope denoted by *scp* is contained in, or equal to,
scp' for all $< gid, scp' > \in SCOPES(GE, gid)$.

Requirement d1: $SCOPES(GE, gid)$ is not empty.

7.3.5 Functional structure of the flattening function

Table 3 lists the argument and result structures of the flattening function #.# defined in 7.3.4.

NOTE - Table 3 does not define new requirements, but summarizes the functional structure of the flattening function for easier reference.

Table 3 - Functional structure of flattening function

Non-terminal	**Flattening**	**Reference**
action-denotation *a*	$\#a\#(TE,GE,VE,scp) = ad$ where *TE*: type-environment *GE*: gate-environment *VE*: value-environment *scp*: scope *ad*: flattened action-denotation	(7.3.4.5 u)

Table 3 (continued)

behaviour-expression *beh*	(or local-definition-expression, sum-expression, par-expression, hiding-expression, enable-expression, disable-expression, parallel-expression, choice-expression, guarded-expression, action-prefix-expression, atomic-expression) *#beh* #(*TE*,*PE*,*GE*,*VE*) = *B* (7.3.4.5 a-m) *func* (*beh*,*TE*,*PE*,*VE*) = *f* where *TE*: type-environment *PE*: process-environment *GE*: gate-environment *VE*: value-environment *B*: behaviour-expression-structure *f*: functionality
boolean-expression *bexp*	*#bexp* #(*S*,*OP*,*V*) = *e* (7.3.4.3 s) where *S*: set of extended-sort-identifiers *OP*: set of extended-operation-identifiers *V*: set of extended-value-identifiers *e*: equation
data-type-definition *type*	*#type* #(*TE*,*scp*) = *TE'* (7.3.4.3 b-c) where *TE*, *TE'*: type-environment *scp*: scope
data-type-definitions *types*	*#types* #(*TE*,*scp*) = *TE'* (7.3.4.3 a) where *TE*, *TE'*: type-environment *scp*: scope
definition-block *db*	*#db* #(*TE*,*PE*,*e-pid*) = <*B*,*pres*,*P*> (7.3.4.2 b) where *TE*: type-environment *PE*: process-environment *e-pid*: extended-process- or specification-identifier *B*: behaviour-expression-structure *pres*: complete data-presentation *P*: set of flattened process-definitions
equation *eqn*	*#eqn* #(*S*,*OP*,*V*,*e-sid-e*) = *ce* (7.3.4.3 q) where *S*: set of extended-sort-identifiers *OP*: set of extended-operation-identifiers *V*: set of extended-value-identifiers *e-sid-e*: extended-sort-identifier *ce*: conditional equation
equation-list *el*	*#el* #(*S*,*OP*,*V*) = *E* (7.3.4.3 p) where *S*: set of extended-sort-identifiers *OP*: set of extended-operation-identifiers *V*: set of extended-value-identifiers *E*: set of conditional equations
equation-lists *els*	*#els* #(*S*,*OP*) = *E* (7.3.4.3 n) where *S*: set of extended-sort-identifiers *OP*: set of extended-operation-identifiers *E*: set of conditional equations

Table 3 (continued)

formal-parameter-list *fp*	*# fp* #(*TE*,*scp*) = <*fg*,*fv*,*func*> (7.3.4.4 c) where *TE*: type-environment *scp*: scope *fg*: list of extended-gate-identifiers *fv*: list of extended-value-identifiers *func*: functionality
gate-declarations *gd*	*#gd* #(*GE*,*scp*) = *gds* (7.3.4.5 r) where *GE*: gate-environment *scp* : scope *gds*: list of flattened gate-declarations
gate-identifier *gid*	*#gid* #(*GE*) = *e-gid* (7.3.4.6 d) where *GE*: gate-environment *e-gid*: extended-gate-identifier
gate-tuple *gt*	*#gt* #(*GE*) = *egs* (7.3.4.5 s) where *GE*: gate-environment *egs*: list of extended-gate-identifiers
guard *g*	*#g* #(*TE*, *VE*) = *e* (7.3.4.5 v) where *TE*: type-environment *VE*: value-environment *e*: equation
identifier-declarations *ifd*	*#ifd* #(*S*,*scp*) = *V* (7.3.4.3 t) where *S*: set of extended-sort-identifiers *scp*: scope *V*: set of extended-value-identifiers
identifier-declarations *ifd*	*#ifd* #(*TE*,*scp*) = *V* (7.3.4.5 p) where *TE*: type-environment *scp*: scope *V*: list of extended-value-identifiers
identifier-declaration *id*	*#id* #(*TE*,*scp*) = *V* (7.3.4.5 q) where *TE*: type-environment *scp*: scope *V*: list of extended-value-identifiers
identifier-equations *e*	*#e* #(*TE*, *VE*,*scp*) = *ies* (7.3.4.5 n) where *TE*: type-environment *VE*: value-environment *scp*: scope *ies*: list of flattened identifier-equations
operation *opn*	*#opn* #(*e-tid*,*S*) = *OP* (7.3.4.3 m) where *e-tid*: extended-type-identifier *S*: set of extended-sort-identifiers *OP*: set of extended-operation-identifiers
operation-list *opl*	*#opl* #(*e-tid*,*S*) = *OP* (7.3.4.3 k) where *e-tid*: extended-type-identifier *S*: set of extended-sort-identifiers *OP*: set of extended-operation-identifiers
parallel-operator *op*	*#op* #(*GE*) = *op'* (7.3.4.5 t) where *GE*: gate-environment *op'*: parallel-operator with extended-gate-identifiers

Table 3 (concluded)

p-expression *pexp*	*#pexp #(TE,e-tid) = ppres* where *TE*: type-environment *e-tid*: extended-type-identifier *ppres*: parameterized data-presentation	(7.3.4.3 d-e-f)
process-definition *proc*	*#proc #(TE,PE) = <pres,P >* *e-pid (proc,TE*$_1$*,scp) = ep* where *TE,TE*$_1$: type-environment *PE*: process-environment *pres*: complete data-presentation *P*: set of flattened process-definitions *scp*: scope *ep*: extended-process-identifier	(7.3.4.4 b)
process-definitions *procs*	*#procs #(TE,PE) = <pres,P >* *e-pids (procs,TE*$_1$*,scp) = eps* where *TE,TE*$_1$: type-environment *PE*: process-environment *pres*: complete data-presentation *P*: set of flattened process-definitions *scp*: scope *eps*: set of extended-process-identifiers	(7.3.4.4 a)
process-identifier *pid*	*#pid #(PE) = e-pid* where *PE*: process-environment *e-pid*: extended-process-identifier	(7.3.4.6 c)
p-specification *pspec*	*#pspec #(e-tid,ppres) = ppres'* where *e-tid*: extended-type-identifier *ppres, ppres'*: parameterized data-presentation	(7.3.4.3 h)
simple-equation *seqn*	*#seqn #(S,OP,V,e-sid-e) = e* where *S*: set of extended-sort-identifiers *OP*: set of extended-operation-identifiers *V*: set of extended-value-identifiers *e-sid-e*: extended-sort-identifier *e*: equation	(7.3.4.3 r)
sort-identifier *sid*	*#sid #(S) = e-sid* *#sid #(TE) = e-sid* where *S*: set of extended-sort-identifiers *TE*: type-environment *e-sid*: extended-sort-identifier	(7.3.4.6 b)
sortlist *sl*	*#sl #(e-tid) = S* where *e-tid*: extended-type-identifier *S*: set of extended-sort-identifiers	(7.3.4.3 j)
specification *spec*	*#spec # = CLS* where *CLS*: canonical LOTOS specification	(7.3.4.2 a)
type-identifier *tid*	*#tid #(TE) = ppres* where *TE*: type-environment *ppres*: parameterized data-presentation	(7.3.4.6 a)
type-union *tun*	*#tun #(TE) = ppres* where *TE*: type-environment *ppres*: parameterized data-presentation	(7.3.4.3 g)

7.4 Semantics of data-presentations

7.4.1 General

This subclause contains the definition of the semantics of a data-presentation *pres* = < *S*, *OP*, *E* >.

The semantic model for *pres* is the quotient term algebra *Q*(*PRES*) defined in 7.4.4. In order to define this algebra a derivation system associated with *pres* is used. This derivation system is defined by 7.4.2.

NOTE - The quotient term algebra *Q*(*PRES*) is a many-sorted algebra as defined in 5.2. It is used in the definition of a labelled transition system in 7.5, for which the quotient term algebra of the algebraic specification *AS* of a canonical LOTOS specification is constructed.

7.4.2 The derivation system of a data-presentation

7.4.2.1 Axioms generated by equations

Let *ceq* be a conditional equation. The set of axioms generated by *ceq*, notation *Ax*(*ceq*), is defined as follows:

a) if $ceq = < V, Eq, e >$ with $Eq \neq \varnothing$, then $Ax(ceq) = \varnothing$; and
b) if $ceq = < V, \varnothing , e >$, then *Ax*(*ceq*) is the set of all ground instances of *e*.

7.4.2.2 Inference rules generated by equations

Let *ceq* be a conditional equation. The set of inference rules generated by *ceq*, notation *Inf*(*ceq*), is defined as follows:

a) if $ceq = < V, \varnothing , e >$, then $Inf(ceq) = \varnothing$, and
b) if $ceq = < V, \{e_1, .., e_n\}, e >$ with $n > 0$, then *Inf*(*ceq*) contains all rules of the form

$$\frac{e_1\,', .., e_n\,'}{e\,'}$$

where $e_1\,', .., e_n\,', e\,'$ are ground instances of $e_1, .., e_n, e$ respectively, that are obtained using the same function $h : V \rightarrow \bigcup\{TERM(OP,s) \mid s \in S\}$ for the substitution of the variables.

7.4.2.3 Generated derivation system

The derivation system $D = < A, Ax, I >$ (see 4.8) generated by a data-presentation $pres = < S, OP, E >$ is defined as follows:

a) *A* is the set of all ground instances of equations *w.r.t.* < *S*, *OP* >; and
b) $Ax = \bigcup \{Ax(ceq) \mid ceq \in E\} \cup ID$, with $ID = \{t = t \mid t$ is a ground term$\}$; and
c) $I = \bigcup \{Inf(ceq) \mid ceq \in E\} \cup SI$, where *SI* is given by the following schemata

1) $$\frac{t_1 = t_2}{t_2 = t_1}$$ for all ground terms t_1, t_2; and

2) $$\frac{t_1 = t_2 ,\ t_2 = t_3}{t_1 = t_3}$$ for all ground terms t_1, t_2, t_3; and

3) $$\frac{t_1 = t_1', \ldots, t_n = t_n'}{op(t_1, \ldots, t_n) = op(t_1', \ldots, t_n')}$$

for all operation-names $op{:}s_1, .., s_n \rightarrow s \in OP$ with $n > 0$ and all ground terms t_i, t_i' of sort s_i for $i = 1, .., n$.

7.4.3 Congruence relation induced by a data-presentation

Let D be the derivation system generated by a data-presentation $pres = < S, OP, E >$. Two ground terms t_1 and t_2 are called *congruent* with respect to *pres*, notation $t_1 \equiv_{pres} t_2$, or $t_1 \equiv t_2$ for short, iff $D \vdash t_1 = t_2$.

The *pres-congruence* class $[\,t\,]$ of a ground term t is the set of all terms congruent to t with respect to *pres*, i.e.

$$[\,t\,] = \{t' \mid t \equiv_{pres} t'\}.$$

7.4.4 Quotient term algebra

The semantical interpretation of a data-presentation $pres = < S, OP, E >$ is the many-sorted algebra $Q(pres) = < D_Q, O_Q >$, called the quotient term algebra, where

a) D_Q is the set $\{Q(s) \mid s \in S\}$, where $Q(s) = \{[\,t\,] \mid t$ is ground term of sort $s\}$ for each $s \in S$; and
b) O_Q is the set of functions $\{Q(op) \mid op \in OP\}$, where the $Q(op)$ are defined by $Q(op)([\,t_1\,], .., [\,t_n\,]) = [\,op(t_1, .., t_n)\,]$.

NOTE - Ground terms denote values. Congruent ground terms are different denotations for the same value (e.g. '4' , '3+1' , '2+2' , '1+3' , '0+4' etc.). The 'trick' used in this subclause is to represent each data value by the set of all its denotations (e.g. the number 4 is represented by the set of terms {'4', '3+1', '1', '2+2', '1+3', '0+4', etc.}). In this way all specified data carriers are obtained in the quotient term algebra.

7.5 Semantics of a canonical LOTOS specification

7.5.1 General

This subclause contains the definition of the semantics of a canonical LOTOS specification $CLS = < AS, BS >$.

The semantic model for CLS is the function defined by subclause 7.5.5. This function maps each instance of the initial process definition p_0 of BS to a transition system as defined by subclause 7.5.4. In order to define this transition system a derivation system associated with CLS is used. This derivation system is defined by subclause 7.5.3. Some auxiliary concepts and definitions are contained in subclause 7.5.2.

7.5.2 Auxiliary definitions

7.5.2.1 Notation

Throughout 7.5 the following notational conventions are maintained:

a) P is a set of *process-names*;
b) G is a set of *gate-names*;
c) V is a set of *variables*;
d) OP is a set of *operation-names*;
e) S is a set of *sort-names*.

It is assumed that these sets are sufficiently large, and contain all names that occur either directly or indirectly in the definition of the semantics of *CLS*.

Throughout 7.5 $Q(AS) = < \{Q(s) \mid s \in S\}, \{Q(op) \mid op \in OP\} >$ and $DD = \bigcup \{Q(s) \mid s \in S\}$.

7.5.2.2 Extended behaviour-expressions

The set of extended behaviour-expression-structures *BE* consists of all behaviour-expression-structures according to the definition in 7.2.3.2 following the syntax as defined in clause 6 with the following modifications:

a) the production for atomic-expression is extended to:

```
atomic-expression =   stop-symbol
                     | exit-symbol [ exit-parameter-list ]
                     | process-instantiation
                     | open-parenthesis-symbol behaviour-expression
                       close-parenthesis-symbol
                     | relabelling-expression .
```

b) the following productions are added:

```
relabelling-expression = open-parenthesis-symbol behaviour-expression
                                   close-parenthesis-symbol relabelling .

relabelling = open-bracket-symbol replacements close-bracket-symbol .

replacements = gate-name "/" gate-name
                     [ comma-symbol  replacements ]  .
```

NOTE - The notion of extended behaviour-expression is needed for the proper definition of the derivation system, which must include the postfix operator p [g_1/a_1, .., g_n/a_n] ('relabelling') for the proper treatment of actual gate-name parameters in process-instantiation instances, and the sum- and par-expression instances.

7.5.2.3 The simplification of sum- and par-expressions

A sum-expression B_1 is a simplification of a sum-expression B_2 if B_1 can be obtained from B_2 by applying the following rule a finite number of times:

(Sum) If B is a sum-expression,
B' is a behaviour-expression,
$d_1, ..., d_n$ are instances of gate-declaration, or variable,

with

B = **choice** $d_1, ..., d_n$ [] B'

then

replace B by **choice** d_1 [] ... **choice** d_n [] B'

A par expression B_1 is a simplification of a par-expression B_2 if B_1 can be obtained from B_2 by applying the following rule a finite number of times:

(Par) If B is a par-expression,
B' is a behaviour-expression,
$g_1, ..., g_n$ are instances of gate-declaration,
op is a parallel-operator

with

B = **par** $g_1, ..., g_n$ *op* B'

then

replace B by **par** g_1 *op* ... **par** g_n *op* B'

A sum- or par-expression is in *normal form* if none of the above rules can be applied to it.

NOTES

1 - Note that a first simplification of gate-declaration occurrences already takes place as part of the flattening process in 7.3.4.5 r).

2 - Every sum- or par-expression has a simplification that is in normal form.

7.5.2.4 Substitution

Let B be an extended behaviour-expression-structure, $x_1, ..., x_n$ variable instances, and $t_1, ..., t_n$ term instances of the same sort as $x_1, ..., x_n$ respectively.

The (literal) substitution $[\, t_1/x_1, ..., t_n/x_n \,]\, B$ is the result of the simultaneous replacement of all occurrences of $x_1, ..., x_n$ in B by $t_1, ..., t_n$ respectively.

NOTE - This simple notion of substitution suffices since we may assume all variables to have a unique name in *BS*.

7.5.3 Transition derivation system

This subclause defines a derivation system for transitions (see 5.3) that is used for the definition of labelled transition systems in 7.5.4.2.

7.5.3.1 General framework

The transition derivation system of a canonical LOTOS specification $CLS = < AS, BS >$ is the triple $D_{CLS} = < As, Ax, I >$, with

a) $As = \{B \text{–}a\!\!\rightarrow B' \mid B, B' \in BE, a = \mathbf{i} \text{ or } a = gv \text{ with } g \in G \cup \{\delta\}, v \in DD^*\}$;

b) *Ax*: the axioms defined by subclause 7.5.3.2;

c) *I*: the inference rules defined by subclause 7.5.3.3;

NOTES

1 - The special label δ is added to the set of gate-names G for the representation of *successful termination*, i.e. the event expressed by the **exit**-construct, see 7.5.3.2 c).

2 - The definition of some axioms and inference rules depends on the evaluation of terms t, i.e. their *AS*-congruence class $[t]$. If t is not a ground term then neither $[t]$, nor the axiom or inference rule depending on $[t]$ is defined. See also the note to 7.5.4.2.

7.5.3.2 Axioms of transition

The following schemata generate the axioms:

a) if B is an action-prefix-expression,
B' is an action-prefix-expression,

with

$B = \mathbf{i}\ ; B'$

then

$B \xrightarrow{\mathbf{i}} B'$ is an axiom,

b) if B is an action-prefix-expression,
B' is a action-prefix-expression,
g is a gate-name,
$d_1, ..., d_n$ are experiment-offer instances

with

$B = g\ d_1 \ldots d_n\ ; B'$

then

$B \xrightarrow{g v_1 \ldots v_n} [\, ty_1/y_1, ..., ty_m/y_m \,]\, B'$ is an axiom

iff

$v_i = [\ t_i\ ]$ if $d_i = !\ t_i$ ($1 \le i \le n$) and t_i is a ground term,
$v_i \in Q(s_i)$ if $d_i = ?\ x_i$ ($1 \le i \le n$) with $sort(x_i) = s_i$,
$ty_1, ..., ty_m$ are term instances with $v_i = [ty_j]$ if $d_i = ?\ y_j$ ($1 \le i \le n$, $1 \le j \le m$)
and $\{y_1,...,y_m\} = \{x_i \mid d_i = ?\ x_i,\ 1 \le i \le n\}$.

if $[\ SP\ ]$ is a selection-predicate
with

$B = g\ d_1 .. d_n\, [\ SP\ ]\ ; B'$

then

$B \xrightarrow{g v_1 .. v_n} [\, ty_1/y_1, ..., ty_m/y_m \,]\, B'$ is an axiom

iff

$v_i = [\ t_i\ ]$ if $d_i = !\ t_i$ ($1 \le i \le n$) and t_i is a ground term,
$v_i \in Q(s_i)$ if $d_i = ?\ x_i$ ($1 \le i \le n$) with $sort(x_i) = s_i$,
$ty_1, ..., ty_m$ are term instances with $v_i = [ty_j]$ if $d_i = ?\ y_j$ ($1 \le i \le n$, $1 \le j \le m$)
and $\{y_1,...,y_m\} = \{x_i \mid d_i = ?\ x_i,\ 1 \le i \le n\}$.

and

$D \vdash SP'$

where

D is the derivation system generated by AS (see 7.4.2.3);
SP' is the ground equation that is the result of the simultaneous replacement in SP of all occurrences of variables x_i in SP that also occur contained in a $d_i = ?\ x_i$ ($1 \le i \le n$), by a term $t \in v_i$.

c) if B is an atomic-expression,
$E_1, ..., E_n$ are instances of exit-parameter

with

$B = \mathbf{exit}(E_1\ ,..., E_n)$

then

$B \xrightarrow{\delta v_1 \ldots v_n} \mathbf{stop}$ is an axiom

iff

$v_i = [\ E_i\ ]$ if E_i is a ground term ($1 \le i \le n$)
$v_i \in Q(s_i)$ if $E_i = \mathbf{any}\ s_i$ ($1 \le i \le n$)

if $B = \mathbf{exit}$
then

$B \xrightarrow{\delta} \mathbf{stop}$ is an axiom

7.5.3.3 Inference rules of transition

In the definition of the inference rules the syntactic function name(a) is used; for each transition label a, name(a) yields the gate-name in a, or δ, or **i**:

name($gv_1 \ldots v_n$) = g
name($\delta v_1 \ldots v_n$) = δ
name(**i**) = **i**

a) if B is a local-definition-expression
B', B'' are behaviour-expression instances,
$x_1 = t_1, \ldots, x_n = t_n$ are identifier-equation instances

with

B = **let** $x_1 = t_1, \ldots, x_n = t_n$ **in** B'

then

$$\frac{[\, t_1/x_1, \ldots, t_n/x_n \,]\, B' \xrightarrow{a} B''}{B \xrightarrow{a} B''}$$

is an inference rule

b) if B is a sum-expression,
B', B'' are behaviour-expression instances,
$g, g_1, \ldots, g_n$ are gate-name instances

with

B = **choice** g **in** [$g_1, \ldots, g_n$] [] B'

then

$$\frac{(B')[\, g_i/g \,] \xrightarrow{a} B''}{B \xrightarrow{a} B''}$$

is an inference rule for each $g_i \in \{g_1, \ldots, g_n\}$

if x is a variable with $sort(x) = s$,

with

B = **choice** x [] B'

then

$$\frac{[\, t/x \,]\, B' \xrightarrow{a} B''}{B \xrightarrow{a} B''}$$

is an inference rule

iff

t is a ground term with $[\, t \,] \in Q(s)$

if B_1 is a simplification of B in normal form,
B_1' is a behaviour-expression

then

$$\frac{B_1 \xrightarrow{a} B_1'}{B \xrightarrow{a} B_1'}$$

is an inference rule

c) if B is a par-expression,
B', B'' are instances of behaviour-expression,
$g, g_1, \ldots, g_n$ are gate-name instances,
op is a parallel-operator

where

B = **par** g **in** [$g_1, \ldots, g_n$] op B'

then

$$\frac{(B')[g_1/g]\ op \ldots op\ (B')[g_n/g] -a\rightarrow B''}{B -a\rightarrow B''}$$

is an inference rule.

if B_1 is a simplification of B in normal form,
B_1 ' is a behaviour-expression

then

$$\frac{B_1 -a\rightarrow B_1'}{B -a\rightarrow B_1'}$$

is an inference rule

d) if B is a hiding-expression,
B', B'' are behaviour-expression instances,
$g_1, \ldots, g_n$ are gate-name instances,

with

$B =$ **hide** $g_1, \ldots, g_n$ **in** B'

then

$$\frac{B' -a\rightarrow B''}{B -a\rightarrow \textbf{hide}\ g_1,\ldots, g_n\ \textbf{in}\ B''} \qquad \text{if name}(a) \notin \{g_1, \ldots, g_n\}$$

$$\frac{B' -a\rightarrow B''}{B \textbf{-i}\rightarrow \textbf{hide}\ g_1,\ldots, g_n\ \textbf{in}\ B''} \qquad \text{if name}(a) \in \{g_1, \ldots, g_n\}$$

are inference rules

e) if B is an enable-expression,
B_1 is a disable-expression,
B_2 is an enable-expression,
$x_1, \ldots, x_n$ are variable instances,
B_1 ' is a behaviour-expression

with

$B = B_1 >>$ **accept** $x_1,\ldots, x_n$ **in** B_2 ,

then

$$\frac{B_1 -a\rightarrow B_1'}{B -a\rightarrow B_1' >> \textbf{accept}\ x_1,\ldots, x_n\ \text{in}\ B_2} \qquad \text{name}(a) \neq \delta$$

$$\frac{B_1 -\delta v_1\ldots v_n \rightarrow B_1'}{B \textbf{-i}\rightarrow [\, t_1/x_1,\ldots, t_n/x_n \,]B_2}$$

are inference rules

where

$t_1, \ldots, t_n$ are ground terms with $[\, t_1 \,] = v_1, \ldots, [\, t_n \,] = v_n$

f) if B is a disable-expression,
B_1 is a parallel-expression,
B_2 is a disable-expression,

B_1 ', B_2 ' are behaviour-expression instances

with

$B = B_1$ [> B_2,

then

$$\frac{B_1 \xrightarrow{a} B_1'}{B \xrightarrow{a} B_1' \text{ [> } B_2} \qquad \text{name}(a) \neq \delta$$

$$\frac{B_1 \xrightarrow{\delta v_1 \ldots v_n} B_1'}{B \xrightarrow{\delta v_1 \ldots v_n} B_1'}$$

$$\frac{B_2 \xrightarrow{a} B_2'}{B \xrightarrow{a} B_2'}$$

are inference rules,

g) if

B is a parallel-expression,
B_1 is a choice-expression,
B_2 is a parallel-expression,
$g_1, \ldots, g_n$ is a (possibly empty) list of gate-name instances,
B_1 ', B_2 ', B ' are behaviour-expression instances

with

$B = B_1$ |[$g_1, \ldots, g_n$]| B_2,

then

$$\frac{B_1 \xrightarrow{a} B_1'}{B \xrightarrow{a} B_1' \text{ |[} g_1, \ldots, g_n \text{]| } B_2} \qquad \text{name}(a) \notin \{g_1, \ldots, g_n, \delta\}$$

$$\frac{B_2 \xrightarrow{a} B_2'}{B \xrightarrow{a} B_1 \text{ |[} g_1, \ldots, g_n \text{]| } B_2'} \qquad \text{name}(a) \notin \{g_1, \ldots, g_n, \delta\}$$

$$\frac{B_1 \xrightarrow{a} B_1', \ B_2 \xrightarrow{a} B_2'}{B \xrightarrow{a} B_1' \text{ |[} g_1, \ldots, g_n \text{]| } B_2'} \qquad \text{name}(a) \in \{g_1, \ldots, g_n, \delta\}$$

are inference rules

if

$B = B_1$ ||| B_2

then

$$\frac{B_1 \text{ |[]| } B_2 \xrightarrow{a} B'}{B \xrightarrow{a} B'}$$

is an inference rule

if

$B = B_1$ || B_2

then

$$\frac{B_1 \text{ |[} g_1, \ldots, g_n \text{]| } B_2 \xrightarrow{a} B'}{B \xrightarrow{a} B'}$$

is an inference rule

where

$\{g_1, \ldots, g_n\} = G$.

h) if B is a choice-expression,
B_1 is a guarded-expression,
B_2 is a choice-expression,
B_1 ', B_2 ' are behaviour-expression instances

with

$B = B_1 \, [] \, B_2$

then

$$\frac{B_1 \xrightarrow{a} B_1'}{B \xrightarrow{a} B_1'}$$

$$\frac{B_2 \xrightarrow{a} B_2'}{B \xrightarrow{a} B_2'}$$

are inference rules

j) if B is a guarded-expression,
B ' is a guarded-expression,
[SP] is a guard,
B '' is a behaviour-expression

with

$B = [\ SP\ ] \mathbin{\text{->}} B'$,

then

$$\frac{B' \xrightarrow{a} B''}{B \xrightarrow{a} B''}$$

is an inference rule

iff

SP is a ground equation and $D \vdash SP$

where

D is the derivation system generated by AS (see 7.4.2.3)

k) if B is an atomic-expression

with

$B =$ **stop** (inaction)

then

no inference rules are generated

if p is a process-name,
$g_1, \ldots, g_n$ are gate-name instances,
$t_1, \ldots, t_m$ are terms,
B ' is a behaviour-expression

with

$B = p\,[\ g_1, \ldots, g_n\ ](t_1, \ldots, t_m)$ (process-instantiation)

then

$$\frac{([\ t_1/x_1, \ldots, t_m/x_m\ ]B_p)[\ g_1/h_1, \ldots, g_n/h_n\ ] \xrightarrow{a} B'}{B \xrightarrow{a} B'}$$

where

$fg(p) = < h_1,..., h_n >$,
$fv(p) = < x_1,..., x_m >$,

is an inference rule

iff

$< p, B_p > \in BS.PDEFS$

if B', B'' are behaviour-expression instances
with

$B = (B')$ (priority parentheses)

then

$$\frac{B' -a\rightarrow B''}{B -a\rightarrow B''}$$

is an inference rule

m) if B, B', B'' are behaviour-expression instances, $h_1, ..., h_n, g_1, ..., g_n$ are gate-name instances,

with

$B = (B')[g_1/h_1,..., g_n/h_n]$

then

$$\frac{B' -a\rightarrow B''}{B -a'\rightarrow (B'')[g_1/h_1,..., g_n/h_n]}$$

with

$a = g\ v_1...v_m$
$a' = g\ v_1...v_m$ if $g \notin \{h_1,...,h_n\}$
$= g_i\ v_1...v_m$ if $g = h_i\ (1 \leq i \leq n)$

is an inference rule.

7.5.4 Structured labelled transition system of a behaviour-expression

This subclause defines for each behaviour-expression *B* the structured labelled transition system that is its formal interpretation (see 5.4). This interpretation is relative to a canonical LOTOS specification *CLS*, which is needed for the generation of the transition derivation system.

7.5.4.1 Derivatives of a behaviour-expression

The set $Der_{CLS}(B)$, the set of derivatives of a behaviour-expression *B*, is the smallest set satisfying:

a) $B \in Der_{CLS}(B)$; and
b) if $B' \in Der_{CLS}(B)$ and $D_{CLS} \vdash B' -a\rightarrow B''$ for some *a*, then $B'' \in Der_{CLS}(B)$.

7.5.4.2 Structured labelled transition system

The structured labelled transition system $TS_{CLS}(B)$ of a behaviour-expression *B* relative to a canonical LOTOS specification $CLS = < AS, BS >$ is the tuple $< S, G \cup \{\mathbf{i}, \delta\}, AS, T, s_0 >$, with

a) $S = Der_{CLS}(B)$;
b) $T = \{ -a\rightarrow \mid a \in Act \}$ with $-a\rightarrow = \{< B_1, B_2 > \mid D_{CLS} \vdash B_1 -a\rightarrow B_2\}$,
where $Act = \{\mathbf{i}\} \cup \{ gv \mid g \in G \cup \{\delta\}, v \in DD^* \}$, and D_{CLS} is the derivation system defined in 7.5.3;

c) $s_0 = B$

NOTE - Formally, this subclause also interprets *non-closed* behaviour-expressions , i.e. behaviour-expressions in which not all variables are bound by defining occurrences. Therefore, in 7.5.5 the formal interpretation is only defined relative to a function in which the free variables are replaced by ground terms.

7.5.5 Formal interpretation of a canonical LOTOS specification

Let $CLS = < AS, BS >$ be a canonical LOTOS specification with $BS.pdef_0 = < p_0, B_0 >$;
let $fg(p_0) = < g_1,\ldots, g_n >$;
let $fv(p_0) = < x_1,\ldots, x_m >$;
let $< s_1,\ldots, s_m > = < sort(x_1),\ldots, sort(x_m) >$.

The formal interpretation [*CLS*] of *CLS* is a function defined as follows:

$$[\, CLS\,] : G^n x Q(s_1) x \ldots x Q(s_m) \rightarrow \{ TS_{CLS}(B) \mid B \in BE \}$$

with

$$[\, CLS\,](h_1,\ldots, h_n, v_1,\ldots, v_m) = TS_{CLS}(([\, t_1/x_1,\ldots, t_m/x_m\,]B_0)[\, h_1/g_1,\ldots, h_n/g_n\,])$$

where
$t_i \in v_i \qquad (1 \leq i \leq m)$.

NOTE - The function $[CLS]$ yields a labelled transition system for each correct instantiation of the LOTOS specification. As *BS* can be assumed to be correct according to the static semantics, all free variables of the specification are elements of $\{x_1,\ldots,x_m\}$.

Annex A

Standard library of data types

(*This annex is an integral part of the body of this International Standard.*)

A.1 Introduction

This annex defines the standard library of LOTOS data type definitions.

This annex provides the users of the language with a number of definitions which are deemed to be of general use. These definitions may be referenced in specifications via their type identifiers in library invocations (see 6.2.3). No other reference is required in order to make use of these definitions in specifications.

The rest of this annex is organized as follows:

Clause A.2 refers to the syntax of the present data type library definition.

Clause A.3 gives the semantics of the present data type library definition.

Clause A.4 defines formally the standard type "Boolean".

Clause A.5 defines the "parameterized", i.e. generic, data types "Element", "Set", "String".

Clause A.6 defines the "unparameterized", i.e. specific, data types "NaturalNumber", "NatRepresentations" (viz. decimal, binary, octal and hexadecimal representations of natural numbers), "Bit", "Octet", "Bitstring", "Octetstring".

NOTE - The definitions in this annex are *not* listed in a valid dependence order (see 7.3.2.13), but this property can be obtained by a suitable permutation of the definitions.

A.2 Syntax of the data type library

Each data type definition listed in this library is defined according to the formal syntax defined in clause 6, and is an instance of the non-terminal symbol *data-type-definition*. The concatenation of all data type definitions listed in this library is an instance of the non-terminal *data-type-definitions*.

A.3 Semantics of the data type library

The semantics of the data type definitions of this library that are contained in clauses A.4 and A.6, as well as those that are actualizations of the data types in clause A.5, are obtained according to the interpretation of LOTOS data type definitions that is defined in clause 7 of this International Standard.

A.4 The Boolean data type

```
type        Boolean is
sorts       Bool
opns        true, false                                         : -> Bool
            not                                                 : Bool -> Bool
            _and_, _or_, _xor_, _implies_, _iff_, _eq_, _ne_    : Bool, Bool -> Bool

eqns        forall x, y : Bool

            ofsort Bool

            not(true)       = false                             ;
            not(false)      = true                              ;

            x and true      = x                                 ;
            x and false     = false                             ;

            x or true       = true                              ;
            x or false      = x                                 ;

            x xor y         = (x and not(y)) or (y and not(x))  ;

            x implies y     = y or not(x)                       ;

            x iff y         = (x implies y) and (y implies x)   ;

            x eq y          = x iff y                           ;

            x ne y          = x xor y                           ;

endtype
```

A.5 Parameterized data type definitions

All the parameterized type definitions presented in this clause require that boolean operators of equality and inequality be defined in the parameter type(s). The definition in clause A.5.1 is therefore a basic building block for all of the subsequent definitions, that introduces the formal requirements on boolean equality and inequality.

A.5.1 Element

```
type          FBoolean is
formalsorts   FBool
formalopns    true             : -> FBool
              not              : FBool -> FBool

formaleqns    forall x: FBool

              ofsort FBool

              not(not(x))      = x             ;

endtype


type          Element is FBoolean
formalsorts   Element
formalopns    _eq_, _ne_ : Element, Element -> FBool

formaleqns    forall x, y: Element

              ofsort Element

                x eq y =>
              x                = y             ;

              ofsort FBool

                x = y =>
              x eq y           = true          ;

              x ne y           = not(x eq y)   ;

endtype
```

A.5.2 Set

```
type     Set is Element, Boolean, NaturalNumber
sorts    Set
opns     {}                                       : -> Set
         Insert, Remove                           : Element, Set -> Set
         _IsIn_, _NotIn_                          : Element, Set -> Bool
         _Union_, _Ints_, _Minus_                 : Set, Set -> Set
         _eq_, _ne_, _Includes_, _IsSubsetOf_     : Set, Set -> Bool
         Card                                     : Set -> Nat

eqns     forall x, y: Element, s, t: Set

         ofsort Set

         Insert(x, Insert(x, s))      = Insert(x, s)                 ;
         Insert(x, Insert(y, s))      = Insert(y, Insert(x, s))      ;

         Remove(x, {})                = {}                           ;
           x eq y =>
         Remove(x, Insert(y, s))      = Remove(x, s)                 ;
           x ne y =>
         Remove(x, Insert(y, s))      = Insert(y, Remove(x, s))      ;

         {} Union s                   = s                            ;
         Insert(x, s) Union t         = Insert(x, s Union t)         ;

         {} Ints s                    = {}                           ;
           x IsIn t =>
         Insert(x, s) Ints t          = Insert(x, s Ints t)          ;
           x NotIn t =>
         Insert(x, s) Ints t          = s Ints t                     ;

         s Minus {}                   = s                            ;
         s Minus Insert(x, t)         = Remove(x, s) Minus t         ;
```

```
     ofsort Bool

     x IsIn {}                       = false                          ;
        x eq y =>
     x IsIn Insert (y, s)            = true                           ;
        x ne y =>
     x IsIn Insert (y, s)            = x IsIn s                       ;

     x NotIn s                       = not(x IsIn s)                  ;

     s Includes {}                   = true                           ;
     s Includes Insert(x,t)          = (x IsIn s) and (s Includes t)  ;

     s IsSubsetOf t                  = t Includes s                   ;

     s eq t                          = (s Includes t) and (t Includes s) ;

     s ne t                          = not(s eq t)                    ;

     ofsort Nat

     Card ({})                       = 0                              ;
        x NotIn s =>
     Card(Insert(x,s))               = Succ(Card(s))                  ;

endtype
```

A.5.3. Strings

Two parameterized definitions are given in this clause that both specify generic strings. The definition of NonEmptyString differs from that of String only because the empty string is represented in the latter but not in the former.

A.5.3.1 Non-empty string

```
type       BasicNonEmptyString is Element
sorts      NonEmptyString
opns       String          : Element -> NonEmptyString
           _+_             : Element, NonEmptyString -> NonEmptyString
endtype

type       RicherNonEmptyString is BasicNonEmptyString, NaturalNumber
opns       _+_             : NonEmptyString, Element -> NonEmptyString
           _++_            : NonEmptyString, NonEmptyString -> NonEmptyString
           Reverse         : NonEmptyString -> NonEmptyString
           Length          : NonEmptyString -> Nat

eqns       forall s, t : NonEmptyString, x, y : Element

           ofsort NonEmptyString

           String(x) + y             = x + String(y)              ;
           (x + s) + y               = x + (s + y)                ;

           String(x) ++ s            = x + s                      ;
           (x + s) ++ t              = x + (s ++ t)               ;

           Reverse(String(x))        = String(x)                  ;
           Reverse(x + s)            = Reverse(s) + x             ;

           ofsort Nat

           Length(String(x))         = Succ(0)                    ;
           Length(x + s)             = Succ(Length(s))            ;

endtype
```

```
type      NonEmptyString is RichierNonEmptyString, Boolean
opns      _eq_,_ne_: NonEmptyString, NonEmptyString -> Bool

eqns      forall s, t: NonEmptyString, x, y: Element

          ofsort Bool

            x eq y =>
          String(x) eq String(y)        = true
            x ne y =>
          String(x) eq String(y)        = false
          String(x) eq (y + s)          = false
          (x + s) eq String(y)          = false
            x eq y =>
          (x + s) eq (y + t)            = s eq t
            x ne y =>
          (x + s) eq (y + t)            = false

          s ne t                        = not(s eq t)

endtype
```

A.5.3.2 String

```
type        String0 is RicherNonEmptyString renamedby
sortnames   String for NonEmptyString
endtype


type        String1 is String0
opns        <> : -> String

eqns        forall s : String, x : Element

            ofsort String

            String(x)       = x + <>        ;

            <> + x          = x + <>        ;

            <> ++ s         = s             ;

            Reverse(<>)     = <>            ;

            ofsort Nat

            Length(<>)      = 0             ;
endtype


type        String is String1, Boolean
opns        _eq_,_ne_      : String, String -> Bool

eqns        forall s, t : String, x, y: Element

            ofsort Bool

            <> eq <>                       = true                    ;
            <> eq (x + s)                  = false                   ;
            (x + s) eq <>                  = false                   ;
              x eq y =>
            (x + s) eq (y + t)             = s eq t                  ;
              x ne y =>
            (x + s) eq (y + t)             = false                   ;

            s ne t                         = not(s eq t)             ;
endtype
```

A.6 Unparameterized data type definitions

A.6.1 Natural number

A.6.1.1 Abstract definition of natural numbers

```
type       BasicNaturalNumber is
sorts      Nat
opns       0                   : -> Nat
           Succ                : Nat -> Nat
           _+_, _*_, _**_      : Nat, Nat -> Nat

eqns       forall m, n: Nat

           ofsort Nat

           m + 0               = m                                      ;
           m + Succ(n)         = Succ(m) + n                            ;

           m * 0               = 0                                      ;
           m * Succ(n)         = m + (m * n)                            ;

           m ** 0              = Succ(0)                                ;
           m ** Succ(n)        = m * (m ** n)                           ;

endtype


type       NaturalNumber is BasicNaturalNumber, Boolean
opns       _eq_, _ne_, _lt_, _le_, _ge_, _gt_ : Nat, Nat -> Bool

eqns       forall m, n: Nat

           ofsort Bool

           0 eq 0                        = true                         ;
           0 eq Succ(m)                  = false                        ;
           Succ(m) eq 0                  = false                        ;
           Succ(m) eq Succ(n)            = m eq n                       ;

           m ne n                        = not(m eq n)                  ;

           0 lt 0                        = false                        ;
           0 lt Succ(n)                  = true                         ;
           Succ(n) lt 0                  = false                        ;
           Succ(m) lt Succ(n)            = m lt n                       ;

           m le n                        = (m lt n) or (m eq n)         ;
           m ge n                        = not(m lt n)                  ;
           m gt n                        = not(m le n)                  ;

endtype
```

A.6.1.2. Representations of natural numbers

```
type        NatRepresentations is
            HexNatRepr, DecNatRepr, OctNatRepr, BitNatRepr
endtype
```

A.6.1.2.1 Hexadecimal representation

```
type        HexNatRepr is HexString
opns        NatNum : HexString -> Nat

eqns        forall hs : HexString, h : HexDigit

            ofsort Nat

            NatNum(Hex(h))          = NatNum(h)                                  ;
            NatNum(h + hs)          = (NatNum(h) * (Succ(NatNum(F)) ** Length(hs)))
                                      + NatNum(hs)                               ;

endtype


type        HexString is NonEmptyString actualizedby HexDigit using
sortnames   HexDigit      for Element
            Bool          for FBool
            HexString     for NonEmptyString
opnames     Hex           for String
endtype
```

```
type      HexDigit is Boolean, NaturalNumber
sorts     HexDigit
opns      0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F      : -> HexDigit
          _eq_, _ne_, _lt_, _le_, _ge_, _gt_                   : HexDigit, HexDigit -> Bool
          NatNum                                               : HexDigit -> Nat
eqns      forall x, y: HexDigit

          ofsort Nat

          NatNum(0)     = 0                                    ;
          NatNum(1)     = Succ(NatNum(0))                      ;
          NatNum(2)     = Succ(NatNum(1))                      ;
          NatNum(3)     = Succ(NatNum(2))                      ;
          NatNum(4)     = Succ(NatNum(3))                      ;
          NatNum(5)     = Succ(NatNum(4))                      ;
          NatNum(6)     = Succ(NatNum(5))                      ;
          NatNum(7)     = Succ(NatNum(6))                      ;
          NatNum(8)     = Succ(NatNum(7))                      ;
          NatNum(9)     = Succ(NatNum(8))                      ;
          NatNum(A)     = Succ(NatNum(9))                      ;
          NatNum(B)     = Succ(NatNum(A))                      ;
          NatNum(C)     = Succ(NatNum(B))                      ;
          NatNum(D)     = Succ(NatNum(C))                      ;
          NatNum(E)     = Succ(NatNum(D))                      ;
          NatNum(F)     = Succ(NatNum(E))                      ;

          ofsort Bool

          x eq y        = NatNum(x) eq NatNum(y)               ;
          x ne y        = NatNum(x) ne NatNum(y)               ;
          x lt y        = NatNum(x) lt NatNum(y)               ;
          x le y        = NatNum(x) le NatNum(y)               ;
          x ge y        = NatNum(x) ge NatNum(y)               ;
          x gt y        = NatNum(x) gt NatNum(y)               ;

endtype
```

A.6.1.2.2 Decimal representation

```
type        DecNatRepr is DecString
opns        NatNum : DecString -> Nat

eqns        forall ds : DecString, d : DecDigit

            ofsort Nat

            NatNum(Dec(d))          = NatNum(d)                                ;
            NatNum(d+ds)            = (NatNum(d) * (Succ(NatNum(9)) ** Length(ds)))
                                       + NatNum(ds)                            ;
endtype


type        DecString is NonEmptyString actualizedby DecDigit using
sortnames   DecDigit     for Element
            Bool         for FBool
            DecString    for NonEmptyString
opnnames    Dec          for String
endtype


type        DecDigit is NaturalNumber, Boolean
sorts       DecDigit
opns        0, 1, 2, 3, 4, 5, 6, 7, 8, 9          -> DecDigit
            _eq_, _ne_, _lt_, _le_, _ge_, _gt_    : DecDigit, DecDigit -> Bool
            NatNum                                : DecDigit -> Nat

eqns        forall x, y: DecDigit

            ofsort Nat

            NatNum(0)    = 0                      ;
            NatNum(1)    = Succ(NatNum(0))        ;
            NatNum(2)    = Succ(NatNum(1))        ;
            NatNum(3)    = Succ(NatNum(2))        ;
            NatNum(4)    = Succ(NatNum(3))        ;
            NatNum(5)    = Succ(NatNum(4))        ;
            NatNum(6)    = Succ(NatNum(5))        ;
            NatNum(7)    = Succ(NatNum(6))        ;
            NatNum(8)    = Succ(NatNum(7))        ;
            NatNum(9)    = Succ(NatNum(8))        ;

            ofsort Bool

            x eq y       = NatNum(x) eq NatNum(y) ;
            x ne y       = NatNum(x) ne NatNum(y) ;
            x lt y       = NatNum(x) lt NatNum(y) ;
            x le y       = NatNum(x) le NatNum(y) ;
            x ge y       = NatNum(x) ge NatNum(y) ;
            x gt y       = NatNum(x) gt NatNum(y) ;

endtype
```

A.6.1.2.3 Octal representation

```
type        OctNatRepr is OctString
opns        NatNum : OctString -> Nat

eqns        forall os: OctString, o: OctDigit

            ofsort Nat

            NatNum(Oct(o))          = NatNum(o)                          ;
            NatNum(o + os)          = (NatNum(o) * (Succ(NatNum(7)) ** Length(os)))
                                      + NatNum(os)                       ;

endtype


type        OctString is NonEmptyString actualizedby OctDigit using
sortnames   OctDigit        for Element
            Bool            for FBool
            OctString       for NonEmptyString
opnnames    Oct             for String
endtype


type        OctDigit is NaturalNumber, Boolean
sorts       OctDigit
opns        0, 1, 2, 3, 4, 5, 6, 7                      : -> OctDigit
            _eq_, _ne_, _lt_, _le_, _ge_, _gt_          : OctDigit, OctDigit -> Bool
            NatNum                                      : OctDigit -> Nat

eqns        forall x, y: OctDigit

            ofsort Nat

            NatNum(0)     = 0                           ;
            NatNum(1)     = Succ(NatNum(0))             ;
            NatNum(2)     = Succ(NatNum(1))             ;
            NatNum(3)     = Succ(NatNum(2))             ;
            NatNum(4)     = Succ(NatNum(3))             ;
            NatNum(5)     = Succ(NatNum(4))             ;
            NatNum(6)     = Succ(NatNum(5))             ;
            NatNum(7)     = Succ(NatNum(6))             ;

            ofsort Bool

            x eq y        = NatNum(x) eq NatNum(y)      ;
            x ne y        = NatNum(x) ne NatNum(y)      ;
            x lt y        = NatNum(x) lt NatNum(y)      ;
            x le y        = NatNum(x) le NatNum(y)      ;
            x ge y        = NatNum(x) ge NatNum(y)      ;
            x gt y        = NatNum(x) gt NatNum(y)      ;
endtype
```

A.6.1.2.4 Binary representation

```
type            BitNatRepr is BitString
opns            NatNum : BitString -> Nat

eqns            forall bs: BitString, b: Bit

                ofsort Nat

                NatNum(Bit(b))           = NatNum(b)                             ;
                NatNum(b + bs)           = (NatNum(b) * (Succ(NatNum(1)) ** Length(bs)))
                                           + NatNum (bs)                         ;

endtype


type            BitString is NonEmptyString actualizedby Bit using
sortnames       Bit             for Element
                Bool            for FBool
                BitString       for NonEmptyString
opnnames        Bit             for String
endtype


type            Bit is NaturalNumber, Boolean
sorts           Bit
opns            0, 1                                         : -> Bit
                _eq_, _ne_, _lt_, _le_, _ge_, _gt_           : Bit, Bit -> Bool
                NatNum                                       : Bit -> Nat

eqns            forall x, y: Bit

                ofsort Nat

                NatNum(0)      = 0             ;
                NatNum(1)      = Succ(0)       ;

                ofsort Bool

                x eq y         = NatNum(x) eq NatNum(y)        ;
                x ne y         = NatNum(x) ne NatNum(y)        ;
                x lt y         = NatNum(x) lt NatNum(y)        ;
                x le y         = NatNum(x) le NatNum(y)        ;
                x ge y         = NatNum(x) ge NatNum(y)        ;
                x gt y         = NatNum(x) gt NatNum(y)        ;

endtype
```

A.6.2 Octet

```
type        Octet is Bit
sorts       Octet
opns        Octet                                       : Bit, Bit, Bit, Bit, Bit, Bit, Bit, Bit -> Octet
            Bit1, Bit2, Bit3, Bit4, Bit5, Bit6, Bit7, Bit8  : Octet -> Bit
            _eq_, _ne_                                  : Octet, Octet -> Bool

eqns        forall b1, b2, b3, b4, b5, b6, b7, b8: Bit, x, y: Octet

            ofsort Bit

            Bit1(Octet(b1,b2,b3,b4,b5,b6,b7,b8))        = b1              ;
            Bit2(Octet(b1,b2,b3,b4,b5,b6,b7,b8))        = b2              ;
            Bit3(Octet(b1,b2,b3,b4,b5,b6,b7,b8))        = b3              ;
            Bit4(Octet(b1,b2,b3,b4,b5,b6,b7,b8))        = b4              ;
            Bit5(Octet(b1,b2,b3,b4,b5,b6,b7,b8))        = b5              ;
            Bit6(Octet(b1,b2,b3,b4,b5,b6,b7,b8))        = b6              ;
            Bit7(Octet(b1,b2,b3,b4,b5,b6,b7,b8))        = b7              ;
            Bit8(Octet(b1,b2,b3,b4,b5,b6,b7,b8))        = b8              ;

            ofsort Bool

            x eq y      = (Bit1(x) eq Bit1(y)) and (Bit2(x) eq Bit2(y)) and
                          (Bit3(x) eq Bit3(y)) and (Bit4(x) eq Bit4(y)) and
                          (Bit5(x) eq Bit5(y)) and (Bit6(x) eq Bit6(y)) and
                          (Bit7(x) eq Bit7(y)) and (Bit8(x) eq Bit8(y))   ;

            x ne y      = not(x eq y)                                     ;
endtype
```

A.6.3 Octet String

```
type        OctetString is String actualizedby Octet using
sortnames   Octet        for Element
            Bool         for FBool
            OctetString  for String
opnnames    Octet        for String
endtype
```

ANNEX B

Equivalence relations

(*This annex is not an integral part of the body of this International Standard.*)

B.1 Introduction

The formal model of LOTOS provides a rigorous mathematical basis for the analysis of LOTOS specifications. This basis enables the properties of specifications to be studied by using only relevant mathematical theory, provided that it can be applied to this formal model.

Algebraic transformation laws enable subexpressions of expressions to be replaced by equivalent expressions without change of meaning. Such laws simplify the analysis of algebraic specifications. In the analysis of LOTOS specifications this approach can be applied both to the behaviour expressions and value expressions (ACT ONE terms). In this annex only the transformation laws for behaviour expressions will be considered. More work on the verification of behaviour expressions, and work on the verification of abstract data types can be found in the references of this annex.

Using algebraic laws behaviour expressions may be transformed into further equivalent expressions by repeated substitution. This method may be applied to demonstrate that two descriptions of a system are equivalent. For example one description might be the original specification and the other a description in LOTOS of the implementation. The other description might also be a more explicit description of the behaviour of the systems of which some properties can be more readily verified.

In clauses B.2 and B.3 transformation laws are given for LOTOS behaviour expressions. These laws are attained by identifying labelled transition systems, being the formal interpretation of closed behaviour expressions, following certain equivalence relations that are defined over such transition systems. The equivalence relations used in clauses B.2 and B.3 both relate transition systems that have indistinguishable observable behaviour, i.e. by observing (=interacting with) such systems an experimentor is unable to distinguish one system from an equivalent one.

The equivalence relations used in clauses B.2 and B.3 are termed 'weak bisimulation equivalence' and 'testing equivalence', respectively. They are different in the way in which the concept of 'observable equivalence' is formalized. Two expressions that are 'weak bisimulation equivalent' are also 'testing equivalent', but not vice versa. In other words, the formalization of 'bisimulation' distinguishes between more systems than 'testing'. The choice between these two concepts of equivalence must be made by those who wish to use the transformation laws, depending on which relation reflects their technical criteria for identification best.

Unfortunately, neither the weak bisimulation relation nor the testing equivalence possess the substitution property on LOTOS behaviour expressions described above in all contexts. When considered as systems in isolation there are behaviour expressions that are equivalent, but which will behave differently when they are part of a larger system. This may occur, for example, if they form a component of a choice expression.

Therefore, stronger relations between behaviour expressions are required that do have the full substitution property. Such relations are termed congruence relations. Clauses B.2.2 and B.3.2 contain the laws for the bisimulation congruence relation, and the testing congruence relation, respectively.

The sets of laws that are presented in the subsequent parts of this annex are not 'complete', i.e. if two expressions are in a certain equivalence/congruence relation, this does not imply that one expression can be transformed into the other using the relevant set of laws. Of course, the converse implication does hold: the sets of laws are 'sound'.

B.2 Weak bisimulation

B.2.1 Definitions

Let $Sys = < S,A,T,s_0 >$ be a labelled transition system.

Let s, $s' \in S$, $a_1,..,a_n \in A$, then the following notation is defined:

$s - a_1,..,a_n \rightarrow s'$ iff there exist $s_1,..,s_{n+1} \in S$ with $s = s_1$, $s' = s_{n+1}$, and $s_i \text{-} a_i \rightarrow s_{i+1}$ for all $1 \leq i \leq n$

Let s, $s' \in S$, $a \in A$, k be a natural number then the following notation is defined:

$s - a^k \rightarrow s'$ iff $s - a,a,...,a \rightarrow s'$ where $a,a,...,a$ is a list of k occurrences of a

Let s, $s' \in S$, $a_1,..,a_n \in A\text{-}\{\mathbf{i}\}$, then the following notation is defined:

$s = a_1,..,a_n \Rightarrow s'$ iff there exist natural numbers $k_0,..,k_n$ with

$$s - \mathbf{i}^{k_0}, a_1, \mathbf{i}^{k_1},...,a_n, \mathbf{i}^{k_n} \rightarrow s'$$

$s = \varepsilon \Rightarrow s'$ iff either $s = s'$ or there exists a natural number n with

$$s - \mathbf{i}^n \rightarrow s'$$

Let S be a set of states. A relation $R \subseteq S \times S$ is a *weak bisimulation relation* iff:

If $< s_1,s_2 > \in R$ then for all $t \in (A\text{-}\{\mathbf{i}\})^*$

a) there exists an $s_1' \in S$ with $s_1 = t \Rightarrow s_1'$ implies there exists an $s_2' \in S$ with $s_2 = t \Rightarrow s_2'$ and $< s_1',s_2' > \in R$;

b) there exists an $s_2' \in S$ with $s_2 = t \Rightarrow s_2'$ implies there exists an $s_1' \in S$ with $s_1 = t \Rightarrow s_1'$ and $< s_1',s_2' > \in R$;

Let Sys_1 and Sys_2 be labelled transition systems, with state sets S_1 and S_2, and initial states s_{01} and s_{02}, respectively. Sys_1 and Sys_2 are *weak bisimulation equivalent* iff there exists a weak bisimulation relation $R \subseteq S \times S$, with

a) $S = S_1 \cup S_2$;

b) $< s_{01},s_{02} > \in R$

Two closed LOTOS behaviour expressions, in the context of an algebraic specification and a complete process environment, are weak bisimulation equivalent iff their transition systems as defined in 7.5.4. are weak bisimulation equivalent.

Two LOTOS behaviour expressions B_1, B_2 with all their free value-identifiers contained in $\{x_1,..,x_n\}$ are weak bisimulation equivalent, iff all instances $[E_1/x_1,..E_n/x_n]B_1$ and $[E_1/x_1,..,E_n/x_n]B_2$ are weak bisimulation equivalent, where $E_1,..,E_n$ are closed value expressions of the same sort as $x_1,..,x_n$, respectively.

A LOTOS *context* C [ ] is a LOTOS behaviour expression with a formal process parameter ' [ ] '. If C [ ] is a context and B is a behaviour expression then C [B] is the behaviour expression that is the result of replacing all ' [ ] ' occurrences in C [ ] by B.

Two LOTOS behaviour expressions B_1, B_2 are *weak bisimulation congruent* iff for all LOTOS contexts C [ ], C [B_1] is weak bisimulation equivalent with C [B_2].

B.2.2 Laws for weak bisimulation congruence.

If two LOTOS behaviour expressions B_1 and B_2 are weak bisimulation congruent this is written $B_1 = B_2$, where the use of ' = ' is justified by the fact that B1 may be replaced by B2 and vice versa in all LOTOS contexts without changing the meaning. Equations of the form $B_1 = B_2$ involving variables for behaviour expressions are called (weak bisimulation congruence) *laws*. The most useful laws for weak bisimulation congruence are listed below. The formulation of some of these laws depends on the *label sets* of some of the involved behaviour expressions. The label set $L(B)$ of a behaviour expression B is defined as the set of all gate-identifiers that occur in B that are not bound by a hiding-operator, or a sum-domain- or par-domain-expression.

a) *Action-prefix*

Let $g..?\ x : t..$ be an action-denotation with an experiment-offer $?\ x : t$, and let z be a value-identifier that does not occur in $g \ldots ?\ x : t \ldots [\ E\ ];\ B$.

1) $g \ldots ?\ x : t \ldots [\ E\ ];\ B = g \ldots ?\ z : t \ldots [\ z / x\ ][\ E\ ];\ [\ z / x\ ]\ B$
2) $g \ldots ?\ x : t \ldots ;\ B =$ **choice** $x : t\ [\,]\ g \ldots !\ x \ldots ;\ B$
3) $g\ !\ E_1 \ldots !\ E_n\ [\ E\ ];\ B = [\ E\ ] \text{->}\ g\ !\ E_1 \ldots !\ E_n;\ B$

b) *Choice*

1) $B_1\ [\,]\ B_2 = B_2\ [\,]\ B_1$
2) $B_1\ [\,]\ (B_2\ [\,]\ B_3) = (B_1\ [\,]\ B_2)\ [\,]\ B_3$
3) B [] **stop** $= B$
4) $B\ [\,]\ B = B$

5) $[\ E/x\ ]B\ [\,]$ (**choice** $x{:}t\ [\,]\ B$) = **choice** $x{:}t\ [\,]\ B$ if $[\ E\ ] \in Q(t)$
6) **choice** $x{:}t\ [\,]\ B = B$ if x is not free in B
7) **choice** $x{:}t$ [] **exit**(... ,x, ...) = **exit**(... ,**any** t, ...)

c) *Parallel*

If a law holds for all of the parallel operators '| ' is used to denote any of them (using the same instance throughout the law)

1) $B_1 \mid B_2 = B_2 \mid B_1$
2) $B_1 \mid (B_2 \mid B_3) = (B_1 \mid B_2) \mid B_3$
3a) **exit**($E_1, \ldots ,E_n$) | **exit**($E_1', \ldots ,E_m'$) = **exit**($E_1, \ldots ,E_n$) if $n = m$ and ($[E_i] = [E_i']$ or (E_i' = **any** t and $sort(E_i) = t$)) for all i with $1 \le i \le n$
= **stop** otherwise

3b) **exit**(...) | **stop** = **stop**

Let A, A' be lists of gate-identifiers

4) $B_1\ |[\ A\ ]|\ B_2 = B_1\ |[\ A'\ ]|\ B_2$ where A' is any list containing the same elements as A
5) $B_1\ |[\ A\ ]|\ B_2 = B_1\ |[\ A'\ ]|\ B_2$ where $A' = A \cap (L(B_1) \cup L(B_2))$
6) $B_1\ |[\ A\ ]|\ B_2 = B_1\ ||\ B_2$ if $(L(B_1) \cup L(B_2)) \subseteq A$
7) $B_1\ |[\ ]|\ B_2 = B_1\ |||\ B_2$

d) *Enabling*

Let >>* denote any instance of the enable operator.

1) $\textbf{stop} >>^* B = \textbf{stop}$
2a) $\textbf{exit} >> B = \textbf{i} \,;\, \mathrm{B}$
2b) $\textbf{exit}(E_1, \ldots, E_n) >> \textbf{accept}\ x_1{:}t_1, \ldots, x_n{:}t_n\ \textbf{in}\ B = \textbf{i}\,;[\,E_1/x_1, \ldots, E_n/x_n\,]B$
3) $(B_1 >> B_2) >> B_3 = B_1 >> (B_2 >> B_3)$
4) $B >>^* \textbf{stop} = B\ |||\ \textbf{stop}$

e) *Disabling*

1) $B_1 [> (B_2 [> B_3) = (B_1 [> B_2) [> B_3$
2) $B [> \textbf{stop} = B$
3) $(B_1 [> B_2)\ [\,]\ B_2 = B_1 [> B_2$
4) $\textbf{stop} [> B = B$
5) $\textbf{exit}(\ldots) [> B = \textbf{exit}(\ldots)\ [\,]\ B$

f) *Hiding*

Let *A*,*A* ',*A*" be lists of gate-identifiers, >>* any instance of the enable-operator

1) $\textbf{hide}\ A\ \textbf{in}\ B = \textbf{hide}\ A'\ \textbf{in}\ B$ — if *A* ' is a list containing the same elements as A
2) $\textbf{hide}\ A\ \textbf{in}\ B = \textbf{hide}\ A'\ \textbf{in}\ B$ — if $A' = A \cap L(B)$
3) $\textbf{hide}\ A\ \textbf{in hide}\ A'\ \textbf{in}\ B = \textbf{hide}\ A''\ \textbf{in}\ B$ — where $A'' = A \cup A'$
4) $\textbf{hide}\ A\ \textbf{in}\ B = B$ — if $A \cap L(B) = \varnothing$
5a) $\textbf{hide}\ A\ \textbf{in}\ a!E_1 \ldots !E_n;B = \textbf{i};(\textbf{hide}\ A\ \textbf{in}\ B)$ — if $a \in A$
5b) $\textbf{hide}\ A\ \textbf{in}\ g;B = g;(\textbf{hide}\ A\ \textbf{in}\ B)$ — if $\mathrm{name}(g) \notin A$
6) $\textbf{hide}\ A\ \textbf{in}\ B_1\ [\,]\ B_2 = (\textbf{hide}\ A\ \textbf{in}\ B_1)\ [\,]\ (\textbf{hide}\ A\ \textbf{in}\ B_2)$
7) $\textbf{hide}\ A\ \textbf{in}\ (B_1\ |[\,A'\,]|\ B_2) = (\textbf{hide}\ A\ \textbf{in}\ B_1)\ |[\,A'\,]|\ (\textbf{hide}\ A\ \textbf{in}\ B_2)$ — if $A \cap A' = \varnothing$
8) $\textbf{hide}\ A\ \textbf{in}\ (B_1 >>^* B_2) = (\textbf{hide}\ A\ \textbf{in}\ B_1) >>^* (\textbf{hide}\ A\ \textbf{in}\ B_2)$
9) $\textbf{hide}\ A\ \textbf{in}\ (B_1 [> B_2) = (\textbf{hide}\ A\ \textbf{in}\ B_1) [> (\textbf{hide}\ A\ \textbf{in}\ B_2)$
10) $\textbf{hide}\ A\ \textbf{in}\ [\,E\,] \rightarrow B = [\,E\,] \rightarrow (\textbf{hide}\ A\ \textbf{in}\ B)$

g) *Guarding*

1a) $[\,L = R\,] \rightarrow B = B$ — if $L = R$
$[\,L = R\,] \rightarrow B = \textbf{stop}$ — otherwise
1b) $[\,BE\,] \rightarrow B = [\,BE = \mathrm{true}\,] \rightarrow B$ — if *BE* is a value-expression

h) *Instantiation*

$b[\,a_1, \ldots, a_n\,](E_1, \ldots, E_m) = ([\,E_1/x_1, \ldots, E_m/x_m\,]B_b)[\,a_1/g_1, \ldots, a_n/g_n\,]$

if $\textbf{process}\ b[\,g_1, \ldots, g_n\,](x_1, \ldots, x_m){:}f := B_b\ \textbf{endproc}$ is the format of the corresponding process abstraction for the process-identifier *b*.

j) *Local definition*

$\textbf{let}\ x_1{:}t_1 = E_1, \ldots, x_n{:}t_n = E_n\ \textbf{in}\ B = [\,E_1/x_1, \ldots, E_n/x_n\,]B$

k) *Relabelling*

Let [S] be any instance [$a_1/g_1, \ldots, a_n/g_n$] of the (post-fix) relabelling operator. We associate with [S] the function S on gate-identifiers defined by
$S(g_i) = a_i \quad (1 \le i \le n)$
$S(g) = g \quad$ if $g \ne g_i \; (1 \le i \le n)$
S can be extended to lists, sets, strings etc. containing gate-identifiers, in which case the application of S yields an object of the same category in which all occurrences g have been replaced by $S(g)$ for all gate-identifiers g.

1) **stop**[S] = **stop**
2) **exit**(...)[S] = **exit**(...)
3) $(a;B)[\,S\,] = S(a);B[\,S\,]$
4) $(B_1 \; [] \; B_2)[\,S\,] = B_1[\,S\,] \; [] \; B_2[\,S\,]$
5) $(B_1 \; |[\,A\,]| \; B_2)[\,S\,] = B_1[\,S\,] \; |[\,S(A)\,]| \; B_2[\,S\,]$ if S is injective on $L(B_1) \cup L(B_2) \cup A$
6) $(B_1 \gg^* B_2)[\,S\,] = B_1[\,S\,] \gg^* B_2[\,S\,]$
7) $(B_1 \,[>\, B_2)[\,S\,] = B_1[\,S\,] \,[>\, B_2[\,S\,]$
8) (**hide** A' **in** B)[S] = **hide** A **in** B[S] if S is injective on $L(B) \cup A'$, and $S(A') = A$
9) $B[\,S\,] = B$ if S is the identity on $L(B)$
10) $B[\,S_1\,] = B[\,S_2\,]$ if $S_1(a) = S_2(a)$ for all $a \in L(B)$
11) $B[\,S_1\,][\,S_2\,] = B[\,S_2 \bullet S_1\,]$

m) *Internal action*

1) $a;$**i**$;B = a;B$
2) $B \; [] \;$ **i**$;B =$ **i**$;B$
3) $a;(B_1 \; [] \;$ **i**$;B_2) \; [] \; a;B_2 = a;(B_1 \; [] \;$ **i**$;B_2)$
4) [E/x]B [] (**choice** $x{:}t$ [] **i**;B) = **choice** $x{:}t$ [] **i**;B if [E] $\in Q(t)$

n) *Expansion theorems*

For the sake of convenience, the following notational convention is introduced:

$B_1 \; [] \; B_2 \; [] \ldots [] \; B_n$	is written $[]\{B_1, \ldots, B_n\}$
choice $x{:}t$ [] B	is written $[]\{[\,E/x\,]B \mid [\,E\,] \in Q(t)\}$
choice g **in** [$a_1, \ldots, a_n$] [] B	is written $[]\{B\,[a_i/g\,] \mid a_i \in \{a_1, \ldots, a_n\}\}$

In this way we have introduced the general format $[]S$ where S is a set of behaviour expressions. It is assumed that the elements of S can always be enumerated by some suitably chosen index set. If the elements of S are all of the form $b_i;B_i$ then the following useful laws apply

Let $B = []\{b_i;B_i \mid i \in I\}$, $C = []\{c_j;C_j \mid j \in J\}$

1) $B \; |[\,A\,]| \; C =$
$[]\{b_i;(B_i|[\,A\,]|C) \mid name(b_i) \notin A, \; i \in I\}$
$[] \; []\{c_j;(B|[\,A\,]|C_j) \mid name(c_j) \notin A, \; j \in J\}$
$[] \; []\{a;(B_i|[\,A\,]|C_j) \mid a = b_i = c_j, \; name(a) \in A, \; i \in I, \; j \in J\}$

if all $b_i \; (i \in I)$, $c_j \; (j \in J)$ are of the form $g!E_1 \ldots !E_n$

2) $B \,[>\, C =$
C
$[] \; [] \; \{b_i;(B_i \,[>C) \mid i \in I\}$

3) **hide** A **in** $B =$ $[]\{b_i;$**hide** A **in** $B_i \mid name(b_i) \notin A,\ i \in I\}$
$[]\ []\{$**i**;**hide** A **in** $B_i \mid name(b_i) \in A,\ i \in I\}$

if all b_i $(i \in I)$ are of the form $g!E_1 \ldots !E_n$

4) $B[\,S\,] = []\{S(b_i);B_i[\,S\,] \mid i \in I\}$

B.2.3 Laws for weak bisimulation equivalence

B.2.3.1. Notation

If two behaviour expression B_1 and B_2 are weak bisimulation equivalent this is denoted by $B_1 \approx B_2$.

B.2.3.2 General law

$B \approx$ **i**; B

B.2.3.3 Rules for $\approx$

1) Let $C[\ ]$ be a LOTOS context of the following forms:
   a) $g;[\ ]$
   b) $[\ ]\,|[A]|\,B$, or $B\,|[A]|\,[\ ]$
   c) $[\ ] >>^* B$, or $B >>^* [\ ]$
   d) $[\ ]\,[> B$
   e) **hide** A **in** $[\ ]$
   f) $[E] \rightarrow [\ ]$
   g) $[\ ][S]$
   h) **let** ... **in** $[\ ]$

then if $B_1 \approx B_2$ then $C[\,B_1\,] \approx C[\,B_2\,]$

2) if $B \approx C$ then $a;B = a;C$ for all action-denotations 'a'

3) if $B = C$ then $B \approx C$

B.3 Testing equivalence

B.3.1 Definitions

Let $Sys = < S,A,T,s_0 >$ be a labelled transition system.

Let $s \in S$, then the set (s **after** t) $\subseteq S$, for $t \in (A\text{-}\{\mathbf{i}\})^*$ is defined by s **after** $t = \{s' \mid s =t\Rightarrow s'\}$

Let $L \subseteq A\text{-}\{\mathbf{i}\}$, then the predicate ($s$ **must** L) is defined by
s **must** L iff $s =\varepsilon\Rightarrow s'$ implies that there exists an s'' with $s' =a\Rightarrow s''$ for some $a \in L$

Let $R \subseteq S$, then the predicate (R **must** L) is defined by
R **must** L iff s **must** L for all $s \in R$

Let $Sys_1 = < S_1,A_1,T_1,s_{01} >$ and $Sys_2 = < S_2,A_2,T_2,s_{02} >$, and extend them to the label set $A = A_1 \cup A_2$, then the predicate (Sys_1 **red** Sys_2) is defined by

Sys_1 **red** Sys_2 iff
(s_{02} **after** t) **must** L implies (s_{01} **after** t) **must** L for every $t \in (A\text{-}\{\mathbf{i}\})^*$ for every $L \subseteq (A\text{-}\{\mathbf{i}\})$

For closed LOTOS behaviour expressions B_1, B_2, in the context of an algebraic specification and a complete process environment, B_1 **red** B_2 iff for their transition systems Sys_1, Sys_2 respectively, as defined in 7.5.4, Sys_1 **red** Sys_2.

For LOTOS behaviour expressions B_1, B_2 with all their free value-identifiers contained in $\{x_1, \ldots, x_n\}$
B_1 **red** B_2 iff
$[E_1/x_1, \ldots E_n/x_n]B_1$ **red** $[E_1/x_1, \ldots, E_n/x_n]B_2$ for all $E_1, \ldots, E_n$, where $E_1, \ldots, E_n$ are closed value expressions of the same sort as $x_1, \ldots, x_n$, respectively.

Two LOTOS behaviour expressions B_1,B_2 are *testing equivalent* iff B_1 **red** B_2 and B_2 **red** B_1.

For LOTOS behaviour expressions B_1, B_2 B_1 **cred** B_2 iff for all LOTOS contexts $C[\ ]$ $C[B_1]$ **red** $C[B_2]$.

Two LOTOS behaviour expressions B_1,B_2 are *testing congruent* iff B_1 **cred** B_2 and B_2 **cred** B_1.

B.3.2 Laws for testing congruence

The laws for testing congruence and equivalence will be expressed in the pre-orders cred and red that generate them. The reason for this is that cred and red are interesting in their own right. B_1 **red** B_2 expresses the fact that B_1 is a (deterministic) reduction of B_2, i.e. B_1 is less nondeterministic than B_2. Since this reduction leaves deadlock properties intact, i.e. B_1 responds to interactions with the environment (tests) as B_2 might respond, B_1 **red** B_2 can be interpreted as 'B_1 implements B_2'. **cred** may thus be used to characterize the class of valid implementations B_1 of a specification B_2. **cred** is the largest sub-relation of **red** that has the substitution property with respect to LOTOS contexts.

1) $\vdash B_1 = B_2$ implies B_1 **cred** B_2

NOTE - This implies that **cred** inherits all the laws listed in B.2.2.

2) B **cred** **i**;B
3) $g;(B_1 [] B_2)$ **cred** $g;B_1 [] g;B_2$
4) $g;B_1$ **cred** $g;B_1 [] g;B_2$
5) B_1 **cred** B_2 & B_2 **cred** B_3 implies B_1 **cred** B_3
6) B_1 **cred** B_3 & B_2 **cred** B_3 implies $(B_1 [] B_2)$ **cred** B_3

B.3.3 Laws for testing equivalence

1) $B_1 \approx B_2$ implies B_1 **red** B_2

2) Let $C[\ ]$ be a LOTOS context of the following forms:
   a) $g;[\ ]$
   b) $[\ ] \,|[A]|\, B$, or $B \,|[A]|\, [\ ]$
   c) $[\ ] >>^* B$, or $B >>^* [\ ]$
   d) $[\ ] [> B$
   e) $[E] \text{->} [\ ]$
   f) $[\ ][S]$
   g) **let** ... **in** [ ]
   then if B_1 **red** B_2 then $C[B_1]$ **red** $C[B_2]$

3) B_1 **red** B_2 & B_2 **red** B_3 implies B_1 **red** B_3
4) B_1 **red** B_3 & B_2 **red** B_3 implies $(B_1 [] B_2)$ **red** B_3

B.4 References

The following references may be found useful for the further study of verification issues.

[1] T.Bolognesi, S.Smolka, Fundamental results for the verification of observational equivalence: a survey, in: H.Rudin, C.West (eds.), Proc. 7th IFIP/WG6.1 Int. Workshop on Protocol Specification, Testing and Verfication, North-Holland, Amsterdam 1988, pp. 165-180.

[2] E.Brinksma, G.Scollo, C.Steenbergen, LOTOS specifications, their implementations and their tests, in: G.Bochmann, B.Sarikaya (eds.), Proc. 6th IFIP/WG6.1 Int. Workshop on Protocol Specification, Testing and Verfication, North-Holland, Amsterdam 1986.

[3] R. De Nicola, M.Hennessy, Testing equivalences for processes, Th.Comp.Science 34 (1984), pp. 83-133.

[4] H.Ehrig, B.Mahr, Fundamentals of algebraic specification 1, EATCS, Springer-Verlag, Berlin 1985

[5] R.Milner, A calculus of communicating systems, LNCS 92, Springer-Verlag, Berlin, 1980.

[6] E.Najm, A verification oriented specification in LOTOS of the transport protocol, in: H.Rudin, C.West (eds.), Proc. 7th IFIP/WG6.1 Int. Workshop on Protocol Specification, Testing and Verfication, North-Holland, Amsterdam 1988, pp. 181-203.

Annex C
A tutorial on LOTOS

(*This annex is not an integral part of the body of this International Standard.*)

C.1 The specification of processes

In LOTOS distributed systems are described in terms of *processes*. A system as a whole is a single process that may consist of several interacting subprocesses. These subprocesses may in turn be refined into sub-subprocesses etc., so that a specification of a system in LOTOS is essentially a hierarchy of process definitions.

In LOTOS the static picture of a process can be imagined as that of a *black box* that is capable of communication with its environment. The mechanisms inside this box are not observable therefore, in principle, not part of the model. The way in which a process may thus be described is by specification of its ability to communicate with its environment. A process communicates with its environment by means of interactions. The atomic form of interaction is an *event*. An event is a unit of synchronized communication that may exist between two processes that can both perform that event.

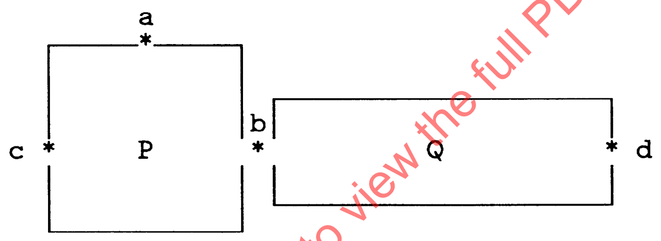

Figure 2 - Two interacting processes

The format of a *process definition* or *process abstraction* in LOTOS is

> **process** <process-identifier><parameter-list> :=
> <behaviour-expression>
> **endproc**

Here the process-identifier is the name by which the process can be referenced and the behaviour-expression is the LOTOS expression defining the observable behaviour of the process. The contents of the parameter-list qualifies the type of the process; in the case of basic LOTOS this is a list of the potential external events of the process.

EXAMPLE

The configuration drawn in figure 2 when represented in LOTOS could be based on the following LOTOS process abstractions:

> **process** P[a,b,c] := ... **endproc**

> **process** Q[b,d] := ... **endproc**

C.2 Behaviour expressions in basic LOTOS

As was already indicated above in LOTOS observable behaviour is expressed by behaviour expressions. These expressions formally relate the order in which events may occur. Consequently, the formation rules for these expressions are an essential part of LOTOS.

C.2.1 A basic process: inaction

The completely inactive process is represented by **stop**. It cannot perform any event, and is as basic in LOTOS as the number zero is in arithmetic.

C.2.2 Two basic operators

C.2.2.1 Action prefix

This is a construct which produces a new behaviour expression out of an existing one by prefixing it with an event name. If *B* is the existing expression and *a* is the event name the result is written as *a;B*.

The interpretation of *a;B* is straightforward: the process it describes first offers participation in event *a*; if *a* takes place the resulting behaviour is given by *B*. The operator ';' thus clearly implies sequentiality.

EXAMPLE

```
process one_time_buffer[in_data,out_data] :=
                    in_data
          ;         out_data
          ;         stop
endproc
```

C.2.2.2 Choice

If B_1 and B_2 are existing behaviour expressions then (B_1[]B_2) denotes a process that behaves either like B_1 or like B_2. The choice offered is resolved in the interaction of the process with the environment. If (another process in) the environment offers an initial event of B_1, B_1 may be selected, if the environment offers an initial event of B_2, B_2 may be selected. If an event is offered that is initial to both B_1 and B_2 the outcome is not determined.

EXAMPLE

Consider a process that is a pair of buffer processes as defined in C.2.2.1, used for transmitting data full duplex (but only once) between two endpoints, e.g.

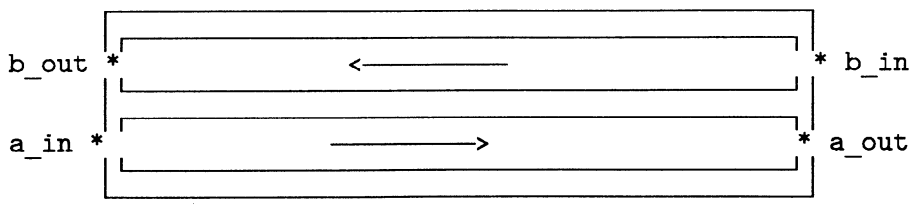

Figure 3 - Full duplex buffer

In LOTOS we could describe the observable behaviour of this pair of buffers by:

```
process simple_duplex_buffer[in_a,in_b,out_a,out_b] :=
    in_a ; (   in_b ; (   out_a ; out_b ; stop
                       []  out_b ; out_a ; stop )
            []  out_a ; in_b ; out_b ; stop )
 [] in_b ; (   in_a ; (   out_b ; out_a ; stop
                       []  out_a ; out_b ; stop )
            []  out_b ; in_a ; out_a ; stop )
endproc
```

Describing the behaviour using only the operators ';' and '[]' makes the definition of the process a straightforward summation of all possible behaviours that exist for the given systems. This is of course a rather clumsy solution in the case of large component processes, and we will show later how the same can be done in a more concise and structured way. Any finite behaviour can in principle always be specified in a normal form that uses only these two operators and **stop**, which is our reason for calling them basic.

C.2.2.3 Processes as trees

The behaviour of a process may be thought of as having a tree-like structure. The root of the tree represents the initial state of the process. The edges in the tree are labelled by the names of events. In this way the labels on the outgoing edges of each node represent possible next steps of the process. The tree structure thus represents process behaviour as a sequence of possible choices ordered in time according to the depth of the nodes in the tree. In the finite case of course, such a tree nicely coincides with the structure of a normal form expression describing that behaviour. It is important to emphasize that this tree structure must not be regarded as a state machine in disguise under the usual interpretation as a string acceptor (see e.g. [15]). In the case of LOTOS a tree is not just a characterization of a number of paths or traces. For example, the two following trees model distinct behaviours:

```
P:      •                    Q:      •
       a|                          a/  \a
        •                          •    •
      b/ \c                        b|    |c
      •   •                        •    •
```

although they both have the same set of traces, viz. {*ab*,*ac*}. The essence of the interpretation as a tree structure resides in the fact that the choice that is represented by branching nodes of a tree is resolved in the interaction between the process and its evironment. The difference between *P* and *Q* above can be illustrated as follows. Suppose both *P* and *Q* are placed in an environment that wants to perform experiment *a* followed by *b*. With process *P* this will be possible under all circumstances. For interaction with *P* at the branching node will result in the only experiment that is possible for both *P* and the experimenter, viz. *b*. In the case of *Q* the choice is made in the initial state. The fact that the environment wants to perform *a* does not resolve the choice. A nondeterministic choice is made between *a* with possible future *b* and *a* with possible future *c* beyond the influence of the environment. Thus it is possible that the 'wrong' branch, offering *c*, is selected, which results in deadlock. The fact that there exists a possible behaviour of *Q* that distinguishes it from *P* forces us to accept the difference between them.

EXAMPLE

The following processes are different:

```
process exam1[ConRes,DatReq,DisInd] :=
    ConRes ;    (  DatReq ; stop
                [] DisInd ; stop )
endproc

process exam2[ConRes,DatReq,DisInd] :=
        ConRes ; DatReq ; stop
    []  ConRes ; DisInd ; stop
endproc
```

C.2.3 Recursion

Infinite behaviour can be defined using the recursive occurrence of process-identifiers in behaviour expressions. In this way we can define a more useful version of a buffer than before:

EXAMPLE

```
    process buffer[in_data,out_data] :=
                in_data
        ;       out_data
        ;       buffer[in_data,out_data]
    endproc
```

The recursion can also be mutual, i.e. process abstractions referring to the names of other process abstractions. We can also define a process like buffer in this way:

```
    process inbuffer[in_data,out_data] :=
                in_data
        ;       outbuffer[in_data,out_data]
    endproc

    process outbuffer[in_data,out_data] :=
                out_data
        ;       inbuffer[in_data,out_data]
    endproc
```

The process defined by inbuffer here is equal to the process buffer defined above. Note that this definition is like a state transition definition where the applied occurrences of inbuffer and outbuffer can be interpreted as states. In applied occurrences of behaviour-identifiers actual names may be substituted for the formal event names in the definition heading. Using this facility we can give a rather judicious definition that is also equivalent with buffer, e.g.

```
    process new_buffer[in_data,out_data] :=
                in_data
        ;       new_buffer[out_data,in_data]
    endproc
```

We shall need this feature later for more useful applications.

C.2.4 Parallelism

C.2.4.1 Parallelism of independent processes

In this case we consider processes without a causal relation between events in one subprocess and those in another. In the case of two independent subprocesses given by B_1 and B_2 we can write $B_1|||B_2$ for their independent composition. In the case that B_1 and B_2 share event names this must be interpreted as the ability of both subprocesses to synchronize on these events with their common environment, but not with each other. In other words, $B_1|||B_2$ can participate in an event if B_1 can participate in it, or B_2 can participate in it. On the other hand, $B_1|||B_2$ refuses participation in any event that is refused by both B_1 and B_2.

An example of a parallel system without internal communication is the process simple_duplex_buffer in C.2.2.2. Using the new operator we can define it with more structure and more concisely as:

```
process new_simple_duplex_buffer[in_a,in_b,out_a,out_b] :=
    one_time_buffer[in_a,out_a]
 ||| one_time_buffer[in_b,out_b]
where
    process one_time_buffer[inp,outp] :=
        inp ; outp ; stop
    endproc
endproc
```

The meaning of this definition is exactly the same as that given in C.2.2.2. The operator ||| defines the behaviour of the composition as the arbitrary interleaving of the events of the subprocesses, where events that belong to the same subprocess remain in the specified order w.r.t. the other events of that subprocess but may have any relative occurrence w.r.t. events in the other subprocess(es). This interpretation of parallelism is critically based on the assumption of the atomic nature of events.

C.2.4.2 Parallelism of dependent processes

There exists also an operator to reflect the parallel composition of dependent subprocesses of a system. If the subprocesses are given by B_1 and B_2 then $B_1||B_2$ represents their parallel composition, in which B_1 and B_2 must synchronize w.r.t. to all events. This operator enjoys the dual properties of those of the parallel operator of C.2.4.1. $B_1||B_2$ can participate in an event only if both B_1 and B_2 can participate in it; it refuses any event that is refused by B_1 or B_2.

EXAMPLE

A typical example of the use of this parallel operator is when the capabilities of a process are determined by two or more of its subprocesses.

```
process produce[a,b,c,d] :=
    item_available[a,b,c,d]
 || item_acceptable[a,b,c,d]
where
    process item_available[a,b,c,d] :=
        a ; (   b ; item_available[a,b,c,d]
            []  c ; item_available[a,b,c,d]
            )
    endproc
    process item_acceptable[a,b,c,d] :=
        a ; (   b ; item_acceptable[a,b,c,d]
            []  d ; item_acceptable[a,b,c,d]
            )
    endproc
endproc
```

In this simple example we can check that produce[a,b,c,d] has the same behaviour as buffer[a,b] of the example in C.2.3.

C.2.4.3 The general parallel operator

Although in many cases of parallel systems the two operators presented above are sufficiently expressive, there exist examples that require a more general approach. If two subprocesses are given by B_1 and B_2 then the expression $B_1 |[\ a_1,..,a_n\ ]| B_2$ describes a composition in which B_1 and B_2 must synchronize w.r.t. the events $a_1,..,a_n$. In this case the list of synchronization events is explicitly given, and not implicitly as in the other operators (in ||| this list is empty, in || it is the complete alphabet of events).

EXAMPLE

In figure 4 we show how to make a two slot shift register out of two one_slot buffers. We do this by synchronizing the out_offer of one buffer with the in_offer of the other. The combined event middle is offered to the environment, and allows the transfer of contents from the first into the second buffer.

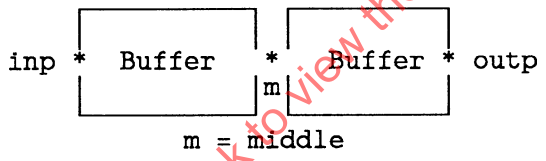

Figure 4 - Composition of two buffer processes

In LOTOS we could represent this design by:

```
process shift[inp,middle,outp] :=
        buffer[inp,middle]
    |[middle]|
        buffer[middle,outp]
where
    process buffer[inp,outp] :=
        inp ; outp ; buffer[inp,outp]
    endproc
endproc
```

If we try to represent a finite part of the behaviour of this process using only the operators ; and [] we get

```
inp ; middle ; (   outp ; shift ...
               []  inp ; outp ; middle ;     ( outp ; shift ...
                                             [] inp ; ......
```

C.2.4.4 The hiding operator

If we want to build a two-place buffer out of the elementary buffer process, the process shift defined above is not sufficient. In this case we want synchronization between the buffer processes only, without interference from their environment, so that the transfer from one buffer to the next occurs 'invisibly'. Another operator is needed to shield off this synchronization from the environment. It is known as the *hiding* operator: if *B* is a behaviour expression and $a_1,..,a_n$ are event names, then '**hide** $a_1,,..,a_n$ **in** *B*' represents a behaviour like *B* in which $a_1,..,a_n$ have become *internal events*. They are not observable and occur spontaneously without the participation of the environment.

EXAMPLE

In figure 5 an arrangement is sketched suggesting how one could make a two slot buffer out of two one-slot buffers. It is like the shift process above, but with the middle-event shielded from the environment.

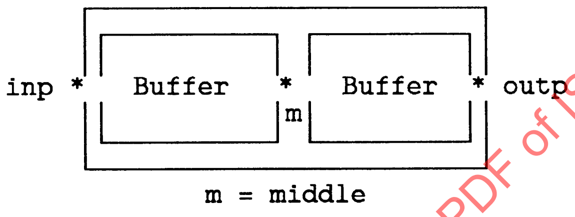

Figure 5 - Hiding

Using the hiding operator we can write this as

```
process two_slot_buffer[inp,outp] :=
    hide middle in
        Buffer[inp,middle] |[middle]| Buffer[middle,outp]
where
    process Buffer[inp,outp] := ...
    endproc
endproc
```

Again, representing a finite part of the behaviour of this process using only the operators ';' and '[]' we get

```
inp ; i ; (     outp ; two_slot_buffer
          []    inp ; outp ; i ; (  outp ; two_slot_buffer
                                 []     inp ; ......
```

Here we have represented the synchronization of the two subprocesses w.r.t. the event middle with the symbol **i**: the *internal event*. Since it is unobservable for the environment and we have adopted the black box approach for the modelling of our processes we are justified in neglecting it, since it does not effect the course of events in any way. The observable behaviour of the two-slot buffer is represented equally well by writing:

```
inp ; (     outp ; two_slot_buffer
      []    inp ; outp ; (      outp ; two_slot_buffer
                         []     inp ; ......
```

Note, incidentally, that we can represent this process completely using ';', '[ ]' and recursion as follows

```
process new_two_slot_buffer[inp,outp] :=
        inp
    ;   (   outp ; new_two_slot_buffer[inp,outp]
        []  inp  ; two_in_buffer[inp,outp]
        )
where
    process two_in_buffer[inp,outp] :=
            outp
        ;   (   outp ; new_two_slot_buffer[inp,outp]
            []  inp  ; two_in_buffer[inp,outp]
            )
    endproc
endproc
```

As we shall see in C.2.5 we cannot ignore the presence of internal events under all circumstances. They play a crucial role in the language for the representation of nondeterminism.

C.2.4.5 Reasons for the hiding operator

At first sight it would be desirable to merge the parallel operators with the hiding operator so that events are made inaccessible to the environment immediately upon synchronization. This option makes it impossible however, to define multi-way synchronizations. By multi-way synchronization we mean the possibility to synchronize two and more processes by a single signal or event. There are both technical and methodological reasons that make this an attractive feature.

The technical reasons have to do with specific applications. In some applications the structure of interprocess communication is reflected best by specifying a multi-way synchronization between processes, in situations where a message is broadcast, for example.

The methodological reasons are related to the fact that where many processes synchronize on a single event, each of these processes may add constraints with respect to the occurrence of that event. This can already be seen in the example of C.2.4.2. If the argument is inverted this means that a complicated constraint w.r.t. the occurrence of a number of events may be decomposed as the conjunction of several simpler constraints, each of which may be reflected by a simple process definition. The complicated constraint can then be expressed by the parallel composition of all these simpler processes. This method is referred to as *constraint-oriented specification*. We illustrate this with a small example:

EXAMPLE

Let *C* be the condition that event *d* should only occur after the events *a*, *b* and *c* have occurred in any possible order. We could specify this by defining a process that explicitely offers all the possible interleavings of *a*, *b* and *c* followed by *d*.

If however, we have a parallel operator |[*d*]| that forces a multi-way synchronization on event *d*, we have a different option. The constraint *C* is logically equivalent with:

a precedes *d* and *b* precedes *d* and *c* precedes *d*

The simpler constraints of the conjunction are reflected respectively by the processes:

a ; *d* ; **stop**   *b* ; *d* ; **stop**   *c* ; *d* ; **stop**

The constraint *C* is now reflected by the process:

C := (*a* ; *d* ; **stop**)|[*d*]|(*b* ; *d* ; **stop**)|[*d*]|(*c* ; *d* ; **stop**)

This approach of 'logical modularity' is worthwhile especially in cases where we cannot be sure that more constraints will not be added later. If after a while it has become necessary that also '*e* precedes *d* ' giving a new constraint *C'*, we can write down

C' := *C* |[*d*]|(*e* ; *d* ; **stop**)

If we have to list the interleaving of *a*, *b* , *c* and e explicitely in our reflection of *C* ', it is impossible to profit from the work already invested in expressing *C*.

C.2.5 Nondeterminism in LOTOS

In C.2.2 we have already seen an example of nondeterminism, viz. in

```
    a ; b ; stop
[]  a ; c ; stop
```

where the result of performing *a* is not determined. This is a special case of the following more general rule: if there exists more than one alternative in a process that starts with an event that is also enabled by the environment then the choice is nondeterministically resolved between such alternatives. E.g. in the case of a process given by

```
    a ; b ; stop
[]  c ; d ; stop
```

the choice between the two alternatives is resolved by the environment if it offers either *a* or *c*; in case the environment offers both *a and c* the outcome is not determined. Another sort of nondeterminism can be modelled using the *internal event* **i**. We will illustrate this using a small example.

EXAMPLE

We want to model a vending machine. After inserting a coin it will offer some candy. The user can obtain the candy by pulling a drawer.

```
process Vending_machine[coin,candy1,candy2] :=
   coin
    ; (   candy1 ; Vending_machine[coin,candy1,candy2]
      []  candy2 ; Vending_machine[coin,candy1,candy2]
      )
endproc
```

Now suppose the system also contains a little devil that can try at any time to pull a drawer (before the user) and consume the candy.

```
process Devil[candy] :=
   candy ; Devil [candy]
endproc
```

The total system as observed by a client is defined by

```
process System[coin,candy] :=
    hide candybar in
            Vending_machine[coin,candy,candybar]
        |[candybar]|
            Devil[candybar]
endproc
```

Rewriting the behaviour of this system in a normal form we get (replacing synchronization w.r.t. candybar by **i**)

```
coin ; (    candy ; System[coin,candy]
        []  i ; System[coin,candy]
        )
```

The first alternative represents 'normal' behaviour, the second is the devil's alternative. It is important to observe that in the above case one cannot ignore the occurrence of **i** and write

```
coin ; (    candy ; System[coin,candy]
        []  System[coin,candy]
        )
```

because this describes a different system! The client can choose between getting his candy and inserting a new coin. In the original description the occurrence of **i** in the second alternative is not at the client's discretion; it may simply happen unobservedly and the client is confronted afterwards with only one possible course of action, viz. System.

In the above example the environment (the client) still has some influence on the choice: if it refuses to act after inserting the coin the devil's alternative will eventually be selected. A case in which the environment has no influence at all on the selection of the alternatives is the case where all alternatives start with an internal event, e.g.

```
    i ; alternative_1
[]  i ; alternative_2
[]  ...
[]  i ; alternative_n
```

We can conclude that the presence of the internal event **i** is relevant in a []-context. We can use this to model special situations. In an OSI context a variant of the following choice will often exist:

```
    normal_course_of_action
[]  i ; disconnect_indication ; ...
```

where a process may be forced to accept a disconnect_indication although, in principle, other alternatives exist.

C.2.6 Sequential composition of processes

So far, all sequentiality in our specifications must be based on the use of the action prefix operator if we want to represent it directly, or by defining parallel compositions where one subprocess synchronizes at its termination with the beginning of the other subprocess, if we do it indirectly. It is desirable to have the option to model the sequential composition of processes directly, thus reflecting the structure of a system in the structure of a specification.

If two processes are given by B_1 and B_2 then their sequential composition is given by $B_1 >> B_2$. The intended interpretation is that if B_1 terminates successfully, i.e. not because of a premature deadlock situation, the execution of B_2 is enabled. For this reason '>>' is also known as the enabling operator.

To mark the place of successful termination in a process there exist a special basic process in LOTOS representing it, viz. **exit**. If control has reached an exit process of the first process in a sequence then this marks the transfer of control to the initial state of the second process.

EXAMPLE

```
process Sender[ConReq,ConCnf,DatReq,DisReq] :=
            Connection_Phase[ConReq,ConCnf]
    >>      Data_Phase[DatReq,DisReq]
where
    process Connection_Phase[ConReq,ConCnf] :=
        ConReq ; ConCnf ; exit
    endproc
    process Data_Phase[DatReq,DisReq] :=
        (   DataReq ; Data_Phase[DatReq,DisReq]
        []  DisReq ; stop
        )
    endproc
endproc
```

In the simple case above a direct definition of the same behaviour may still be preferable:

```
ConReq ; ConCnf ; Data_Phase[DatReq,DisReq]
```

In the case of a behaviour expression that is a parallel composition, the successful termination of the composition is defined as the successful termination of all components. To give a negative example:

(*a* ; *b* ; **exit** ||| *a* ; *c* ; **stop**) >> second_process[...]

is equivalent to

(*a* ; *b* ; **stop** ||| *a* ; *c* ; **stop**)

since **stop** does not represent successful termination, but inaction.

C.2.7 Disruption of processes

In almost any OSI connection oriented protocol or service it is the case that the 'normal' course of events can be disrupted at any point in time by events signalling disconnection or abortion of a connection. This has led to the definition in LOTOS of an 'application generated' operator, viz. the *disabling operator*. If two processes are given by B_1 and B_2 then $B_1[>B_2$ defines a process where at each point in the execution of B_1, an initial event of B_2 can occur. If such an event occurs control is transferred to B_2, leaving B_1. If such an event occurs before an initial event of B_1 only B_2 will be executed. Once B_1 has terminated sucessfully B_2 can no longer disrupt B_1.

EXAMPLE

Define the following processes

```
process Activity[a,b,c] :=
    a ; b ; c ; Activity[a,b,c]
endproc
```

```
    process Disrupt[discon,reason] :=
        discon ; reason ; stop
    endproc
```

then the expression

```
        Activity[a,b,c] [> Disrupt[discon,reason]
```

is equivalent with

```
    (   discon ; reason ; stop
    []  a ; (   discon ; reason ; stop
            []  b ; (   discon ; reason ; stop
                    [] c ; (    Activity[a,b,c]
                            [>  Disrupt[discon,reason]
    )       )       )       )
```

C.2.8 An example in basic LOTOS

In all OSI protocol specifications one can distinguish parts that are responsible for the management of the connections in the underlying service, i.e. the setting up, using and breaking off of the logical communication channels that exist between service users. Here, we present a small and simplified portion of the manager of a transport service (which would typically be a part of a session protocol). We do not discuss the transport service here; the uninitiated reader is referred to [7] for more information.

```
    process Handler[ConReq,ConInd,ConRes,ConCnf,DatReq,DatInd,DisReq,DisInd] :=
            Connection-phase[ConReq,ConInd,ConRes,ConCnf,DisReq,DisInd]
        >>  (       Data-phase[DatReq,DatInd]
            [>      Termination-phase[DisReq,DisInd]
            )
        >>  Handler[ConReq,ConInd,ConRes,ConCnf,DatReq,DatInd,DisReq,DisInd]
    where
        process Connection-phase[CRq,CI,CR,CC,DR,DI] :=
            ( Calling[CRq,CI,CR,CC,DR,DI]
            [] Called[CRq,CI,CR,CC,DR,DI]
            )
        where
            process Calling[CRq,CI,CR,CC,DR,DI] :=
                CRq ; ( CC ; exit
                        [] DI; Connection_phase[CRq,CI,CR,CC,DR,DI]
                        )
            endproc
            process Called[CRq,CI,CR,CC,DR,DI] :=
                CI ; ( i ; CR ; exit
                    [] i ; DR ; Connection_phase[CRq,CI,CR,CC,DR,DI]
                    )
            endproc
        endproc
        process Data_phase[DtR,DtI] :=
            ( i ; DtR ; Data_phase[DtR,DtI]
            [] DtI ; Data_phase[DtR,DtI]
            )
        endproc
        process Termination_phase[DR,DI] :=
            ( i ; DR ; exit
            [] DI ; exit
            )
        endproc
    endproc
```

C.3. LOTOS data types

C.3.1 Introduction

C.3.1.1 Basic Concepts

The representation of values, value expressions and data structures in LOTOS are derived from the specification language for abstract data types (ADT) ACT ONE [11]. ACT ONE is an algebraic specification method to write unparameterized as well as parameterized specifications. ACT ONE, and thus LOTOS, includes the following features for the production of structured specifications:

a) Use of a library of predefined data types
b) Combination of specifications
c) Renaming of specifications
d) Parameterization of specifications.
e) Actualization of parameterized specifications.
f) Extensions of specifications with operations and sorts.

C.3.1.2 Abstract Data Types versus Concrete Data Types

A distinction can be made between concrete and abstract data types. Concrete data types are data types as they are used in most programming languages. Algorithms or programs written in a programming language normally manipulate some data objects. These programs can be conceived of as operations that act on, or generate such data objects. The understanding of concrete data types is only possible with respect to their implementation. One is only able to describe e.g. a PASCAL queue, or a FORTRAN queue, but not simply a queue without respect to a programming language. This is a motivation to introduce a programming language independent data type description. Since we abstract from the aspects of implementations we will call these programming language independent data type description abstract data types.

Before we show *how* we can define abstract data types, we like to clarify *why* one needs abstract data types.

Let us consider a queue which may be characterized essentially by its two operations 'add' and 'first'. Both are implemented by procedures which operate in most cases on linear lists, which are again data structures. Whereas the operation 'add' inserts elements at one end of a list, the operation 'first' produces the element at the other end of a list. If these operations are implemented in different programming languages one will get different procedures. But all these procedures should have the same effect. In order to check whether an implemented procedure has the intended effect or not, one needs a correct and precise description to enable a comparison between the intended effect of the specified operation and the implemented procedures. Hence, the abstract data types are needed to prove the correctness of a class of implemented data structures. Consequently, abstract data types provide a mathematical model of concrete data types. Sets of data objects, or *data carriers*, together with their manipulating operations, form an abstract data type and, in a mathematical sense, an *algebra*.

C.3.2 Concepts of LOTOS data types

C.3.2.1 The Signature

In order to define an abstract data type one needs the possibility to define names of data carriers and operations. Such a definition or declaration must also include the domains and ranges of the operations. The names of the data carriers are referred to as *sorts*. The sorts and operations of a data type are referred to as the signature of that data type. It defines the syntax of a data type. For syntactic definitions one can provide a grammar which formally produces signatures with a finite number of sorts and operations.

Below we have listed a type definition of the natural numbers. The definition has the name 'Nat_numbers', so that it may be referred to. This is important when we want to combine different definitions. The signature of Nat_numbers consists of the single sort 'nat', and the operations '0' and 'succ'. 'succ' can be applied to a single element of sort 'nat', and yields also an element of 'nat' as its result. The intended interpretation of 'succ' is that succ(x) is the successor number of x (x+1). '0' is an operation that has no arguments, and therefore does not depend on any value. Such operations are called *constants.* The intended interpretation of '0' is of course the ordinary number zero of the natural numbers.

```
type Nat_numbers is
    sorts nat
    opns  0: -> nat
          succ: nat -> nat
endtype
```

An additional example of a complete data type definition that consists only of a signature is the definition of a set of characters {a_1,..,a_n}, as this algebra contains only constants.

```
type Characters is
    sorts char
    opns a1, ... ,an,e : -> char
endtype
```

Note that there is a special symbol 'e' which is used in the next chapter to represent an error that is of sort 'char'.

C.3.2.2 Terms and Equations

The names of the sorts and operations defined in a signature must correspond with data carriers and operations of that algebra that serves as the formal semantic model of the specification. All the elements of data carriers can be produced by repeated application of the operations. The result of such a combination of operation applications is denoted by a *term.* Terms represent elements of a sort. If the denoted element belongs to sort *s* it is said that a term is of sort *s* and, is referred to as an *s-term.* For example, if we have a specification of the natural numbers represented by the sort 'nat', a constant '0: ->nat' and an unary operation 'succ:nat -> nat' then we can produce the following nat-terms:

0,succ(0),succ(succ(0)),...

Each of these nat-terms can be interpreted as one element of the algebra of the natural numbers. Using the information of the signature all elements of an algebra can be represented. But what happens if we like to calculate with our natural numbers. We have to introduce additional operation symbols, e.g. the plus operator:

+:nat,nat->nat.

With additional operators additional terms may be produced, e.g. succ(0)+succ(succ(0)). To interpret the new nat-terms correctly, we need a new construct to express properties of operations. This construct is the *equation*.

Let us use a shorthand notation for a sequence of subsequent calls of the operation 'succ' in the following manner:

succ(succ(... succ(0) ...)) is equivalent to $succ^{n}(0)$,

where '*n*' indicates the number of calls of the operator 'succ'. The intended semantics of the plus operation is as follows:

$succ^{n}(0) + succ^{m}(0) = succ^{(n+m)}(0)$.

Note, that the result of the addition is represented by the sum of the calls of both summands (the +-operator used in the superscript is not the +-operator of our type definition, but a meta-linguistic operator used by us to express the desired properties in a convenient manner). To express this property we have to write infinitely many equations, viz. the above for every *n* and *m*, since we can only add constant values e.g. $succ^{3}+succ^{5}=succ^{8}$. Therefore, we introduce *variables* to define equations that range over an entire sort. For example:

$x + 3 = succ^{3}(x)$

The problem of addition of the constant '3' to each other natural number is now resolved by calling 3 times the operation 'succ'. Syntactically, the variable 'x' in the nat-term 'succ(succ(succ(x)))' belongs to the same category as the null-ary operations, i.e. the constants, like '0'.

Note that the denotations of '3', '5' and '8' are not part of the syntactic definition, instead they are values, i.e. interpretations, of terms, viz.

'succ(succ(succ(0)))',
'succ(succ(succ(succ(succ(0)))))', etc.

With the introduction of equations, and terms with variables, we now have the tools to specify properly the properties of operations. For example, a correct definition of the '+'-operator is given by:

```
eqns forall x,y : nat
     ofsort nat
     x + 0 = x ;
     x + succ(y) = succ(x+y) ;
```

The first equation expresses the behaviour of the plus operator when it is combined with the constant '0'. The addition with a non-zero number is given by the second equation. Note that the term succ(x) is used to denote a non-zero number. With induction on the structure of terms it can be easily proved that any term containing one or more plus operations is equal to a term containing only '0' and 'succ'. This means that by introducing the plus operator we have not introduced terms that would denote 'new' elements of 'nat'. This is also expressed by saying that the new algebra is a *complete extension* of Nat_numbers.

Note that the list of equations is preceded by the declaration of the variables x and y in '**forall** x,y : nat'. The sort of the left- and right-hand sides of the equations is indicated by '**ofsort** nat'.

Observe , that the sign of equality used in the equations does not mean the equality of the terms on the left-hand and right-hand side, but the equality of the *interpretation* of both terms if we replace the variables by actual values of the sorts of the variables.

The format of ordinary equations suffices in most cases to define the data types that are required in a convenient way. Sometimes, however, the use of *conditional equations* can improve a definition. An example (in the presence of a sufficiently rich signature) of two such conditional equations is:

```
x>=0 = true  =>  abs(x) = x  ;
x<0  = true  =>  abs(x) = -x ;
```

The general formal of a conditional equation is

$Eq_1,...,Eq_n \Rightarrow Eq$

where $Eq_1,...,Eq_n,Eq$ are all unconditional equations of the type presented so far. The intended interpretation is that the equation following '=>' is true if all equations preceding it (premisses) are true. To promote conciseness premiss equations of the form *E* = true may be abbreviated to *E*. In the above example this would yield:

```
x>=0  =>  abs(x) = x  ;
x<0   =>  abs(x) = -x ;
```

In the definition of LOTOS data types operators may be *overloaded*, i.e. operators with different argument and/or result sorts may have identical names. In this way the symbol '0' may be used to represent both zero of the natural numbers and of the integers, and in the same way the operations succ:nat –> nat and succ:int –> int can be defined. In the scope of an overloaded signature ambiguous expressions such as succ(0) can often be interpreted correctly using contextual information, e.g. because it is one of the two terms in an equation of sort int. If the context cannot be used to resolve the overloading the '**of**'-construct can be used to indicate the result sort of an expression. By writing

succ(0) **of** nat

we would indicate that succ has result sort nat, and therefore succ must be the operation succ:nat –> nat, and therefore 0 is 0:–> nat.

C.3.2.3. The Combination

To specify systems with a large number of operations and equations we need language concepts to extend parts of a specification with additional operations, and combine operators and equations logically belonging together. Therefore, a concept of combination exists which allows bulky specifications to be split into smaller parts.

In the previous section we introduced the type definition 'Nat_numbers', which contained only a signature, and after that the +-operator together with the equations that define its effect on the natural numbers. We can merge these two elements in a new data-type definition as follows:

```
type Extended_nat is
    sorts  nat
    opns   0: –> nat
           succ: nat –> nat
           _+_: nat, nat –> nat
    eqns forall x,y:nat
           ofsort nat
           x + 0 = x ;
           x + succ(y) = succ(x+y) ;
endtype
```

The +-operator is declared in the signature using underscore symbols to indicate that it is an infix operator. The same definition can also be given in a different manner:

```
type Enriched_nat is Nat_numbers
   opns _+_: nat, nat -> nat
   eqns forall x,y:nat
         ofsort nat
         x + 0 = x ;
         x + succ(y) = succ(x+y) ;
endtype
```

In 'Enriched_nat' we have imported the whole definition 'Nat_numbers' by referencing it in the heading of the definition. Consequently, 'Enriched_nat' and 'Extended_nat' define the same algebra. In general, a combination of type definitions involves a list of references to other types, e.g.

type T **is** $T_1,..,T_n$
...
endtype

C.3.2.4 The Parameterization

So far, a single data type definition describes a single data type algebra. But if we look at more complex data types it can be the case that the general structure of a data type does not depend on some of the objects that are part of it. Well-known examples of such data types are arrays, stacks and queues.

If we like to specify a queue of natural numbers and a queue of characters we can do this using the combination construction. Let us assume that the queue is characterized among other things by the operators 'first', 'remove', and 'add'. 'first(queue)' produces the first element at one end of the queue, and 'remove(queue)' describes the queue after this element has been removed from the queue; 'add(element,queue)' adds an element at the other end to the queue. Now we are able to specify the 'Character_queue' and the 'Nat_number_queue' in the following manner:

```
type Nat_number_queue is Nat_numbers
   sorts queue
   opns create: -> queue
         add: nat,queue -> queue
         first: queue -> nat
         remove: queue -> queue
   eqns forall x,y:nat, z:queue
         ofsort nat
         first(create) = 0 ;
         first(add(x,create)) = x ;
         first(add(x,add(y,z))) = first(add(y,z)) ;
         ofsort queue
         remove(create) = create ;
         remove(add(x,create)) = create ;
         remove(add(x,add(y,z))) = add(x,remove(add(y,z))) ;
endtype

type Character_queue is Characters
   sorts queue
   opns create: -> queue
         add: char,queue -> queue
         first: queue -> char
         remove: queue -> queue
```

```
    eqns forall x,y:char, z:queue
        ofsort char
        first(create) = e ;
        first(add(x,create)) = x ;
        first(add(x,add(y,z))) = first(add(y,z)) ;
        ofsort queue
        remove(create) = create ;
        remove(add(x,create)) = create ;
        remove(add(x,add(y,z))) = add(x,remove(add(y,z))) ;
  endtype
```

The constants '0' and 'e' were already introduced respectively in the type definitions 'Nat_numbers', and 'Characters' in C.3.2.1. They are used in respectively the Character_queue, and the Nat_number_queue to indicate an error when the 'first' operation is applied to a new, i.e. empty, queue. It is clear that the above definitions are almost the same. To avoid such duplication in the definitions of types, we can make the subtype of the queue that is variable a *formal part* of a parameterized type specification. In this case we formalize the element type of the queue:

```
  type Queue is
    formalsorts data
    formalopns d0: –> data
    sorts queue
    opns create: –> queue
        add: data,queue –> queue
        first: queue –> data
        remove: queue -> queue
    eqns forall x,y:data, z:queue
        ofsort data
        first(create) = d0 ;
        first(add(x,create)) = x ;
        first(add(x,add(y,z))) = first(add(y,z)) ;
        ofsort queue
        remove(create) = create ;
        remove(add(x,create)) = create ;
        remove(add(x,add(y,z))) = add(x,remove(add(y,z))) ;
  endtype
```

The queue is now equipped with formal elements 'data' and 'd0', which can be actualized by the 'Natural_numbers' or 'Characters' as follows:

```
  type Nat_number_queue is
    Queue actualizedby Nat_numbers using
    sortnames nat for data
    opnnames 0 for d0
  endtype

  type Character_queue is
    Queue actualizedby Characters using
    sortnames char for data
    opnnames e for d0
  endtype
```

The formal part of a parameterized data type definition must describe a *formal data type*, and therefore consists of the three usual elements of a data type specification, viz. a formal signature, and possibly some formal equations with respect to that signature. Such formal equations can be interpreted as requirements that must be fulfilled by an actual type that is substituted for it. For example we could have defined:

```
type Extra_queue is
    formalsorts data
    formalopns d0: -> data
               _*_: data,data -> data
    formaleqns forall x,y:data
               ofsort data
               x * y = y * x
    sorts queue
    opns create: -> queue
         ......
endtype
```

When a parameterized data type is actualized, the actual parameter must be a data type that is compatable with the formal data type. Therefore, it must satisfy two conditions:

- it must provide actual sorts and operations for the elements of the formal signature; this is defined by **sortnames** ... **for** ... and **opnnames** ... **for** ... clauses in the actualization; and
- the actual parameter must 'satisfy' the formal equations; this means that the equations of the actual parameter must imply those of the formal parameter.

These conditions imply that we could actualize Extra_queue with Enriched_nat, using the +-operator for *, but not with Characters.

C.3.2.5 Renaming

Renaming of data type specifications is useful during the development of a specification in the case where an already defined abstract data type is needed in a specific environment, but without any changes in the intended semantics. Therefore, renaming may be done explicitly by rewriting the data type definition with new sorts and operations. Changes in the signature imply changes in the declaration of variables and equations. Especially for long definitions this can be a cumbersome task.

The renaming operation avoids this drawbacks. Let us assume the data type definition Queue of C.3.2.4 is to be used in the OSI transport service environment, which deals with objects like connections and some data objects to be transferred. Then the definition of Queue can be conveniently renamed as follows:

```
type connection is
    Queue renamedby
    sortnames   channel for queue
                objects for data
    opnnames    send for add
                receive for first
endtype
```

Note that the remove operation is not explicitly renamed, and therefore the operation remove: channel -> channel is in the signature of the type connection.

There is an important difference between renaming and actualization of parameterized data types, viz. that in actualizations the signature of the data type can not only be renamed, but it can also be extended with new signature elements. The Nat_number_queue above, for example, contains the successor operation via the Nat_numbers data type, whereas this operation is not in the signature of Queue. The only effect of renaming is that elements of the original signature are given new names, thus creating a new data type that is isomorphic to the original one.