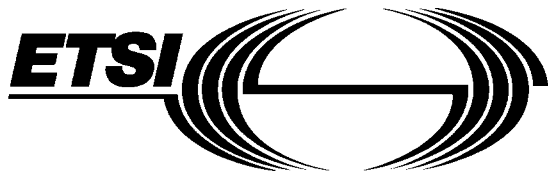

ETSI TECHNICAL REPORT

ETR 071

June 1993

Source: ETSI TC-MTS

Reference: DTR/MTS-01005

ICS: 34.040.40

Key words: TTCN, SDL, common semantics representation

Methods for Testing and Specification (MTS); Semantical relationship between SDL and TTCN A common semantics representation

ETSI

European Telecommunications Standards Institute

ETSI Secretariat

Postal address: F-06921 Sophia Antipolis CEDEX - FRANCE
Office address: 650 Route des Lucioles - Sophia Antipolis - Valbonne - FRANCE
X.400: c=fr, a=atlas, p=etsi, s=secretariat - **Internet:** secretariat@etsi.fr

Tel.: +33 92 94 42 00 - Fax: +33 93 65 47 16

Copyright Notification: No part may be reproduced except as authorized by written permission. The copyright and the foregoing restriction extend to reproduction in all media.

© European Telecommunications Standards Institute 1993. All rights reserved.

New presentation - see History box

Whilst every care has been taken in the preparation and publication of this document, errors in content, typographical or otherwise, may occur. If you have comments concerning its accuracy, please write to "ETSI Editing and Committee Support Dept." at the address shown on the title page.

Contents

Foreword 7

Introduction 7

1 Scope 9

2 References 10

3 Domains, notation, abbreviations 11
3.1 Domains 11
3.2 Functions and operators 13
3.3 Notation 14
3.4 Labelled transition systems (LTS) 15
3.5 Abbreviations 16

4 Conceptual models for SDL and TTCN 17
4.1 Conceptual model for SDL 17
4.2 Conceptual model for TTCN 18
4.2.1 TTCN tester 19
4.2.1.1 The tree process 19
4.2.1.2 The input process 19
4.2.1.3 The timer process 19
4.2.2 TTCN system 20

5 Common Semantics Representation (CSR) 21
5.1 Introduction 21
5.2 Definitions 22
5.2.1 Notation 22
5.2.2 Domains 23
5.2.3 Data terms and signal terms 24
5.2.3.1 Data terms 24
5.2.3.2 Signal terms 25
5.2.3.3 Interpretation of data terms and signal terms 25
5.2.4 Signals 26
5.2.5 Time and timers 27
5.2.6 Routes 27
5.3 Basic process 28
5.3.1 States of a basic process 28
5.3.2 Events of a basic process 29
5.3.3 Operators in the BPA 29
5.3.4 Example 30
5.3.5 Semantics of a basic process 31
5.3.5.1 Inference rule B 1 31
5.3.5.2 Inference rule B 2 and B 3 31
5.3.5.3 Inference rule B 4 and B 5 32
5.4 Timer process 32
5.4.1 States of a timer process 32
5.4.2 Events of a timer process 32
5.4.3 Semantics of a timer process 33
5.4.3.1 Inference rule T 1 33
5.4.3.2 Inference rule T 2 34
5.4.3.3 Inference rule T 3 34
5.4.3.4 Inference rule T 4 34
5.4.3.5 Inference rule T 5 35
5.5 Input port process 35
5.5.1 States of an input port process 35
5.5.2 Events of an input port process 35

5.5.3 Semantics of an input port process 35
5.5.3.1 Inference rule I 1 36
5.5.3.2 Inference rule I 2 36
5.5.3.3 Inference rule I 3 36
5.5.3.4 Inference rule I 4 37
5.5.3.5 Inference rule I 5 37
5.5.3.6 Inference rule I 6 37
5.6 Process instance 37
5.6.1 States of a process instance 37
5.6.2 Events of a process instance 38
5.6.3 Semantics of a process instance 38
5.6.3.1 Inference rule Proc 1 40
5.6.3.2 Inference rule Proc 2 40
5.6.3.3 Inference rule Proc 3 41
5.6.3.4 Inference rule Proc 4 41
5.6.3.5 Inference rule Proc 5 41
5.6.3.6 Inference rule Proc 6 42
5.6.3.7 Inference rule Proc 7 42
5.6.3.8 Inference rule Proc 8 42
5.6.3.9 Inference rule Proc 9 43
5.6.3.10 Inference rule Proc 10 43
5.6.3.11 Inference rule Proc 11 43
5.6.3.12 Inference rule Proc 12 44
5.6.3.13 Inference rule Proc 13 44
5.6.3.14 Inference rule Proc 14 44
5.6.3.15 Inference rule Proc 15 45
5.6.3.16 Inference rule Proc 16 45
5.6.3.17 Inference rule Proc 17 45
5.6.3.18 Inference rule Proc 18 46
5.6.3.19 Inference rule Proc 19 46
5.6.3.20 Inference rule Proc 20 46
5.6.3.21 Inference rule Proc 21 47
5.7 Module process 47
5.7.1 States of a module process 47
5.7.2 Events of a module process 47
5.7.3 Semantics of a module process 47
5.7.3.1 Inference rule M 1 48
5.7.3.2 Inference rule M 2 49
5.7.3.3 Inference rule M 3 49
5.7.3.4 Inference rule M 4 49
5.7.3.5 Inference rule M 5 50
5.7.3.6 Inference rule M 6 50
5.7.3.7 Inference rule M 7 50
5.7.3.8 Inference rule M 8 51
5.7.3.9 Inference rule M 9 51
5.7.3.10 Inference rule M 10 51
5.7.3.11 Inference rule M 11 52
5.7.3.12 Inference rule M 12 53
5.8 Path process 53
5.8.1 States of a path process 53
5.8.2 Events of a path process 53
5.8.3 Semantics of a path process 53
5.8.3.1 Inference rule Path 1 54
5.8.3.2 Inference rule Path 2 54
5.9 System 54
5.9.1 States of system 54
5.9.2 Events of a system 55
5.9.3 Semantics of a system 55
5.9.3.1 Inference rule S 1 55
5.9.3.2 Inference rule S 2 55
5.9.3.3 Inference rule S 3 56
5.9.3.4 Inference rule S 4 56
5.9.3.5 Inference rule S 5 56

5.9.3.6 Inference rule S 6 57
5.9.4 Derivation of the initial system state 57

Annex A: Introduction to the CSR 58

A.1 Labelled Transition Systems 58

A.2 Use of inference rules 60

A.3 Usage of the CSR 66
A.3.1 The INRES protocol 66
A.3.2 An SDL process and a TTCN test case for the INRES protocol 66
A.3.3 Evaluation of the behaviour of a process instance 68

Annex B: Definition of the transformations 72

B.1 The identification of the range 72
B.1.1 Introduction 72
B.1.2 The range 72

B.2 Transformation of SDL 72
B.2.1 Generator functions 81
B.2.2 Conversion functions 81
B.2.3 Auxiliary functions 81

B.3 Transformation of TTCN 82
B.3.1 Assumptions on TTCN 82
B.3.1.1 Assumptions on referencing parts of messages 82
B.3.1.2 Assumptions on constraints 83
B.3.2 Transformation of TTCN data types and values 87
B.3.2.1 Base data types 87
B.3.2.2 TTCN data types 88
B.3.2.2.1 ASN.1 data type definitions 88
B.3.2.2.2 Test suite type definitions 94
B.3.2.3 An example - The Verdict type 98
B.3.3 The range for TTCN 99
B.3.3.1 Sets of identifiers 99
B.3.3.2 Transformation of values, expressions, timers, and constraints 102
B.3.3.3 Storage environment 104
B.3.4 Transformation of TTCN behaviour descriptions 104
B.3.4.1 Abstract Evaluation Trees (AET) 105
B.3.4.2 Identification of the subset of the Basic Process Algebra (BPA) 105
B.3.4.3 Definition of the transformation of AETs 105

Annex C: Examples 109

C.1 Introduction 109

C.2 SDL to CSR 109
C.2.1 A simple SDL example 109
C.2.2 Its CSR representation 114

C.3 TTCN to CSR 115
C.3.1 A simple TTCN example 115
C.3.2 Its CSR representation 119

History 123

Blank page

Foreword

ETSI Technical Reports (ETRs) are informative documents resulting from studies carried out by the European Telecommunications Standards Institute (ETSI) which are not appropriate for European Telecommunication Standard (ETS) or Interim European Telecommunication Standard (I-ETS) status. An ETR may be used to publish material which is either of an informative nature, relating to the use or application of ETSs or I-ETSs, or which is immature and not yet suitable for formal adoption as an ETS or I-ETS.

This ETR was produced by the Methods for Testing and Specification (MTS) Technical Committee of ETSI. It gives a Common Semantics Representation (CSR) for Specification and Description Language (SDL) and Tree and Tabular Combined Notation (TTCN) language.

Introduction

Within the communications research community much effort has been spent to investigate the applicability of formal methods for the specification of protocols and telecommunication systems and the specification of tests. The results of these efforts are manifold. In the context of this ETR, the results of importance are the definition of Specification and Description Language (SDL) (CCITT Recommendation Z.100 [1]), a functional specification and description language for protocols and telecommunication systems, and the definition of Tree and Tabular Combined Notation (TTCN) (ISO 9646 [2], Part 3), a test notation applicable in the context of OSI (ISO 7498 [3]). The use of these formal methods is recommended by the standardisation organisations, e.g. ISO, CCITT and ETSI.

Over recent years, it has been recognised that an integrated methodology has to be established covering all aspects of protocol engineering. As part of this broader context, it seems to be reasonable to investigate a semantical relationship between SDL and TTCN. The work on a semantical relationship is motivated by the following aspects:

- computer aided test generation:

  experiences with manual test case derivation in terms of required resources, both for definition and maintenance of test suites, have shown that it seems reasonable to look for a generally applicable method to semi-automatically generate test cases from formal protocol specifications. As a prerequisite, the semantical equivalencies or in-equivalencies of SDL and TTCN have to be investigated;

- test suite validation:

  in close relation with the specification of Open Systems Interconnection (OSI) services and protocols, the development of abstract test suites is carried out. Test suites for protocols already exist, and it is an open problem on how to validate these against their specifications.

In order to contribute to these problems, ETSI has decided to define a C*ommon Semantics Representation (CSR)* for SDL and TTCN. The term CSR refers to a representation, able to represent the semantics of objects from different domains in a common model. Such a common model enables the investigation of semantical relations between objects of the different domains. Hence, a common semantics representation for SDL and TTCN enables the definition of formal semantical relations between SDL and TTCN specifications. Such relations could then act as a basis when developing tools for test generation, test validation, etc...

The CSR described in this ETR is an operational model, consistent with the semantics defined in CCITT Recommendation Z.100 [1] and ISO 9646 [2], Part 3, and is based on an existing operational model for SDL (see TFL RR 1991-2, "An operational semantic model for basic SDL" by J. Godskesen [4]). It is defined as a compositional hierarchical model, which enables reasoning about the dynamic behaviour of SDL and TTCN specifications at different levels of observability.

Related work includes the SPECS project, RACE ref. 1046: "Specification and Programming Environment for Communication Software" [5], where a common representation language (CRL) for LOTOS and SDL has been defined, and in "Translation of test specifications in TTCN to LOTOS" [6], where a direct mapping from TTCN to LOTOS is described.

Blank page

1 Scope

This ETR specifies a Common Semantics Representation (CSR) for SDL (see CCITT Recommendation Z.100 [1]) and TTCN. The CSR is defined as a model for *basic SDL* and *TTCN abstract evaluation trees*. The CSR defines an operational semantics in terms of *Labelled Transition Systems (LTS)*. The handling of data is on an abstract level and does not deal with coding and representation information as e.g. Abstract Syntax Notation no. 1 (ASN.1).

This ETR gives guidance on the transformation of basic SDL and TTCN abstract evaluation trees to the CSR model. The validation of the CSR with respect to the semantics defined for SDL and TTCN is given in-line with the definition of the CSR.

The following general restriction applies:

- The concurrent behaviour of SDL systems and TTCN systems is approximated by interleaving of events.

For TTCN, the restrictions below apply:

- a discrete time model is assumed;
- a send event makes no assignments to Abstract Service Primitive (ASP) parameters or Protocol Data Unit (PDU) fields from within the assignment list;
- constraints for receive events use only a subset of all matching constructs.

Basic SDL, as defined in CCITT Recommendation Z.100 [1], is covered by the model except for the following concepts:

- channels with no delay;
- service definitions which have become a concept in basic SDL.

The technical contents of this ETR are presented in Clauses 4 and 5:

- in Clause 4, conceptual models for SDL and TTCN are introduced to motivate the structure of the CSR;
- in Clause 5, the CSR is defined. The main characteristic of the CSR is that it has a compositional and hierarchical structure.

Annexes are provided as a guide to the understanding of the approach.

- Annex A informally introduces the basic modelling concepts of the CSR and gives some intuition how the CSR can be used.

- Annex B defines the range for the transformations of basic SDL and TTCN abstract evaluation trees and gives guidance on how to transform basic SDL and TTCN abstract evaluation trees to the CSR;

- Annex C discusses an example. The discussion is intended to provide guidance on how to perform a transformation from SDL or TTCN to the CSR.

The CSR defined in this ETR is applicable as a basis for further work on computer aided test generation and test suite validation.

2 References

For the purposes of this ETR, the following references apply.

[1] CCITT Recommendation Z.100 (1992): "Functional specification and description language SDL".

[2] ISO 9646-1 (1991): "Conformance testing methodology and framework - Part 1: General conecpts"..

ISO 9646-2 (1991): "Conformance testing methodology and framework - Part 2: Abstract Test Suite".

ISO 9646-3 (1991-11-22): "Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation".

[3] ISO 7498 (1984): "Information processing systems - Open Systems Interconnection - Basic Reference Model".

[4] J. Godskesen (1991) TFL RR 1991-2: "An operational semantic model for basic SDL".

[5] SPECS consortium (1990) RACE Ref. 1046: "Specification and Programming Environment for Communication Software".

[6] B. Sarikaya, Q. Gao (1988) Proceedings of PSTV VIII, North-Holland: "Translation of test specifications in TTCN to LOTOS".

[7] H. Ehrig, B. Mahr (1985) Springer-Verlag: "Fundamentals of algebraic specifications 1".

[8] ISO 8807 (1989): "LOTOS - A formal description technique based on the temporal ordering of observational behaviour".

[9] ISO 8824 (1990): "Specification of Abstract Syntax Notation 1 (ASN.1)".

[10] ISO 8825 (1990): "Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)".

[11] G. v. Bochmann, M. Deslauriers (1989) Proceedings of PSTV IX, North-Holland: "Combining ASN.1 support with the LOTOS language".

[12] D. Pitt, A. Boshier, B. Szczygiel (1990) NPL: "System Documentation for the One2One translator and user guide".

[13] D. Hogrefe (1991) IAM-91-012: "OSI formal specification case study: the Inres protocol and service".

3 Domains, notation, abbreviations

The following subclauses, except for subclause 3.5, summarise terms defined in Clause 5.

3.1 Domains

A Σ-algebra over $\Sigma = \langle Sort \cup SigSort, OP \cup SigOP \rangle$ (subclause 5.2).

BPId The set of all *basic process identifiers* (subclause 5.2). This set is ranged over by *X, Y*.

$BPId_m$ The set of *basic process identifiers* of *module m* (subclause 5.2). This set is ranged over by *X, Y*.

$BPPId_X$ The set of *process instance identifiers* of the basic process identified by *X*. (subclause 5.2) This set is ranged over by pid, pid', pid_1, pid_2.

BProc	The set of all *basic processes* (subclause 5.3.1). This set is ranged over by P, Q, R.
Env	The set of *functions* from *variables* to *locations* (subclause 5.2). This set is ranged over by $\varepsilon, \varepsilon'$.
IProc	The set of *states* of an *input port process* (subclause 5.5.1).
ModId	The set of *module identifiers* (subclause 5.2). This set is ranged over by *m*.
$ModPId_m$	The set of *process instance identifiers* of *module m* (subclause 5.2). This set is ranged over by pid, pid', pid_1, pid_2.
$ModPId_{env}$	The set of *process instance identifiers* in the *environment* of a system (subclause 5.2).
ModStor	The set of all *module storages* (subclause 5.6.3). This set is ranged over by ms, ms', ms''.
$ModStor_m$	The set of all *module storages* of *module m* (subclause 5.6.3). This set is ranged over by ms, ms', ms''.
Loc	The set of all *storage locations* (subclause 5.2). This set is ranged over by $l, l', l'', \ldots$.
OP	The set of all *operations* (of a signature) (subclause 5.2). This set is ranged over by *op.*
PathId	The set of *path identifiers* (subclause 5.2). This set is ranged over by *pth,* ...
PathProc	The set of *states* of a *path process* (subclause 5.8.1).
PId	The set of all *process instance identifiers* in all *modules* of the system and in the *enviroment* (subclause 5.2). This set is ranged over by pid, pid', pid_1, pid_2.
$PId_\perp$	The set of all *process instance identifiers* including the *undefined process instance identifier* (subclause 5.2). This set is ranged over by pid, pid', pid_1, pid_2.
PId_{spec}	The set of *process instance identifiers in the specification* (subclause 5.2).
Proc	The set of *states* of a *process instance* (subclause 5.6.1). This set is ranged over by p, p', q, q', p_1, p_2.
Routes	The of set all *routes* in a *system* (subclause 5.2.6). Each element being either a pair of process identifiers or a pair consisting of a process identifier and a path identifier. This set is ranged over by *r*.
$Routes_m$	The set of *routes* in the *module m* (subclause 5.2.6). This set is ranged over by r_m.
Sig	The set of all *signals* (subclause 5.2). This set is ranged over by *sig, sig'*.
Sig^*	The set of all *finite sequences* of *signals* (subclause 5.2).

$Sig_{\bot}$	The set of all *signals* including the *undefined signal* (subclause 5.2). This set is ranged over by *sig, sig'*.
$Sig^{*}_{\bot}$	The set of all *finite sequences* of *signals* including the *undefined signal* (subclause 5.2).
SigOP'	The set of all *operations* (of a signature) for *signal declarations* (subclause 5.2). This set is ranged over by *op'*.
SigSort	The set of all *signal sorts* (subclause 5.2). This set is ranged over by *s', ss*.
SigTerm	The set of all *signal ground terms* (subclause 5.2). This set is ranged over by *st, ...*
$SigTerm_s$	The set of *signal ground terms* of *sort s* (subclause 5.2). This set is ranged over by *st, ...*
$SigTerm^A$	The set of *interpretation* of *signal terms* (subclause 5.2). This set is ranged over by *si, ...*
$SigTerm^A_s$	The set of *interpretations* of *signal terms* of *sort s* (subclause 5.2). This set is ranged over by *si, ...*
SigTerm(Var)	The set of all *signal terms* with respect to the set of variables *Var* (subclause 5.2). This set is ranged over by *st, ...*
$SigTerm_s(Var)$	The set of *signal terms* of *sort s* with respect to the set of variables *Var* (subclause 5.2). This set is ranged over by *st, ...*
Sort	The set of all *data sorts* (subclause 5.2). This set is ranged over by *s, ...*
Sto	The set of *functions* from *locations* to *values* (subclause 5.2). This set is ranged over by ρ, ρ'
Sys	The set of *states* of a *system* (subclause 5.9.1).
Term	The set of all *data ground terms* (subclause 5.2). This set is ranged over by *bt, dt, ...*
Term(Var)	The set of all *data terms* with respect to the set of *variables Var* (subclause 5.2). This set is ranged over by *st, ...*
$Term_s$	The set of *data ground terms* of *sort s* (subclause 5.2). This set is ranged over by *st, ...*
$Term_s(Var)$	The set of data terms of sort *s* with respect to the set of variables *Var* (subclause 5.2). This set is ranged over by *st, ...*
$Time^A$	The set of *time values* (subclause 5.2).
Timers	The set of *timers* (subclause 5.2).
$Timers^*$	The set of all *finite sequences* of *timers* (subclause 5.2).
Var	The set of all *variables* (subclause 5.2).

Var_s	The set of all *variables* of *sort s* (subclause 5.2). For example, $Var_{Boolean}$, Var_{Time}, Var_{PId}.
ε_S	The set of *system events* (subclause 5.9.2).
ε_{Path}	The set of *path process events* (subclause 5.8.2).
ε_M	The set of *module process events*. (subclause 5.7.2)
ε_{PI}	The set of *process instance events* (subclause 5.6.2).
ε_B	The set of *basic process events* (subclause 5.3.2).
ε_T	The set of *timer process events* (subclause 5.4.2).
ε_I	The set of *input port process events* (subclause 5.5.2).
$\longrightarrow_S$	The *system transtion relation* (subclause 5.9.3).
$\longrightarrow_{Path}$	The *path process transition relation* (subclause 5.8.3).
$\longrightarrow_M$	The *module process transition relation* (subclause 5.7.3).
$\longrightarrow_{PI}$	The *process instance transition relation* (subclause 5.6.3).
$\longrightarrow_B$	The *basic process transition relation* (subclause 5.3.5).
$\longrightarrow_T$	The *timer process transition relation* (subclause 5.4.3).
$\longrightarrow_I$	The *input port process transition relation* (subclause 5.5.3).

3.2 Functions and operators

$e(\Theta)$	A *predicate* over timer sequences (subclause 5.4.3).
From	A function from signals to pid-values (of the sending process instances) (subclause 5.2).
Inst	A function from signals to explicit signals (instantiation) (subclause 5.2).
Intr	A function from signals to interpretations of signal ground terms (subclause 5.2).
Path	A function from signals to path identifiers (subclause 5.2).
To	A function from signals to pid-values (of the receiving process instances) (subclause 5.2).
$filter_T$	A function from timers and timer sequences to timer sequences (subclause 5.4.3). $filter_T$ removes timer entries from a timer sequence.
$filter_I$	A function from signal terms and signal sequences to signal sequences (subclause 5.5.3). $filter_I$ removes signal term entries from a signal sequence.

$insert_T$	A function from timers and timer sequences to timer sequences (subclause 5.4.3). $insert_T$ inserts a timer in a timer sequence.
$ms(...)$	A partial function from pid-values to process instances in module storage *ms* (subclause 5.6.3).
ms_{env}	A partial function from pid-values to partial functions from variables to locations (*Env*) (subclause 5.6.3).
ms_{sto}	A partial function from pid-values to partial functions from locations to values (*Sto*) (subclause 5.6.3).
$ms[...]$	A mapping of module storages to module storages (subclause 5.6.3).
r_m	A function from routes to subsets of signal interpretations (subclause 5.2.6). This function is called *route function*.
$s[...]$	A mapping from system states to system states (subclause 5.9.1).
S^*	The set of finite sequences of elements from *S* (subclause 5.2).
ε	A partial function from variables to locations (*environment*) (subclause 5.2).
$\varepsilon[...]$	A mapping from environments to environments (subclause 5.6.1).
ρ	A partial function from locations to values (*storage*) (subclause 5.2).
$\rho[...]$	A mapping from storages to storages (subclause 5.6.1).
$::$	The concatenation of finite sequences (subclause 5.2).
$\#S$	The cardinality of the set *S* (subclause 5.2).

3.3 Notation

$D : BPid \to BProc \times Inst \times Par \times (Var \times Term(Var))^*$

process and procedure declaration (subclause 5.3.1).

$op : s_1, \ldots s_n \to s$	operation definition (subclause 5.2).
$op' : s_1, \ldots, s_n \to s'$	signal definition (subclause 5.2).

$(t, pid, pid', p) \in (SigTerm^A \cup \{\perp_{st}\}) \times PId \times PId_\perp \times (PathId \cup \{\perp_{pth}\})$

signal (subclause 5.2).

$\Sigma = \langle Sort, OP \rangle$	signature (subclause 5.2).
$\varepsilon_\perp$	undefined environment (subclause 5.6.1).
$\rho_\perp$	undefined storage (subclause 5.6.1).
$\perp_{pi}$	undefined process instance (subclause 5.6.1).
$\perp_{pid}$	undefined process instance identifier (subclause 5.2).

$\perp_{pth}$	undefined path identifier (subclause 5.2).
$\perp_{st}$	undefined signal (subclause 5.2).
$\perp_{t}$	undefined (data or signal) term (subclause 5.2).
$\bigcup_{i=1,\ldots,n} S_i$	generalised union (subclause 5.2).
$\biguplus_{i=1,\ldots,n} S_i$	disjoint union (subclause 5.2).
$S_1 \times \ldots \times S_n$	cartesian product (subclause 5.2).
$\Vert\ \Vert^{A}$	interpretation of ground terms in algebra A (subclause 5.2).
$\Vert\ \Vert^{A}_{\varepsilon,\rho}$	interpretation of terms in algebra A with respect to ε and ρ (subclause 5.2).
$\Vert\ \Vert^{A}_{ms,\varepsilon,\rho}$	interpretation of terms in algebra A with respect to module storage ms, ε and ρ (subclause 5.2).
$\langle\ \rangle$	empty sequence (subclause 5.2).
$\dfrac{t_1 \ \ldots \ t_n}{t}\ C$	inference rule (generic form) (subclause 5.1).
:-	function definition (subclauses 5.4 and 5.5).

3.4 Labelled transition systems (LTS)

System	$S = \left(Sys, \varepsilon_S, \longrightarrow_S, s_0\right)$ (subclause 5.9.3).
Path process	$Path = \left(PathProc, \varepsilon_{Path}, \longrightarrow_{Path}\right)$ (subclause 5.8.3).
Module process	$M = \left(ModStor, \varepsilon_M, \longrightarrow_M\right)$ (subclause 5.7.3).
Process instance	$PI = \left(Proc, \varepsilon_{PI}, \longrightarrow_{PI}, ModStor\right)$ (subclause 5.6.3).
Basic process	$B = \left(BProc, \varepsilon_B, \longrightarrow_B\right)$ (subclause 5.3.5).
Timer process	$T = \left(Timers^*, \varepsilon_T, \longrightarrow_T\right)$ (subclause 5.4.3).
Input port process	$I = \left(IProc, \varepsilon_I, \longrightarrow_I\right)$ (subclause 5.5.3).

3.5 Abbreviations

For the purposes of this ETR, the following abbreviations apply:

ADT	Abstract Data Type

ASP	Abstract Service Primitive
BPA	Basic Process Algebra
CRL	Common Representation Language
CSR	Common Semantics Representation
FIFO	First In, First Out
IUT	Implementation Under Test
LTS	Labelled Transition System
PCO	Point of Control and Observation
PDU	Protocol Data Unit
SDL	CCITT Specification and Description Language
SUT	System Under Test
TTCN	Tree and Tabular Combined Notation

4 Conceptual models for SDL and TTCN

This Clause introduces two models whose purpose is to identify the main concepts of SDL and TTCN. The models are intended as an aid to motivate the structure of the CSR model. The CSR is introduced in Clause 5.

4.1 Conceptual model for SDL

The definition of a conceptual model for SDL is straightforward. On a high level of abstraction the CCITT Recommendation Z.100 [1] defines an *SDL system* as an entity which may receive signals from the environment and may send signals to the environment via *channels*. On a more detailed level an SDL system is substructured in blocks connected by channels. *Blocks* are refined in *process instances* and *signal routes*. Block partitioning and channel partitioning are not considered as the CSR is only defined for *basic SDL*. An information item communicated by a process instance to another process instance or to the environment is called a *signal instance*.

It is assumed that, in the environment of an SDL system, at least one process instance exists which behaves obeying the rules defined for an SDL system.

Channels are either uni- or bi-directional. Channels introduce an arbitrary but finite delay on signal instances conveyed from sender to receiver. At any instance in time more than one signal instance can be conveyed by a channel in either direction of communication. The signal instances are delivered in the order they are put on the channel. A channel can be regarded as a First-In, First Out (FIFO)-queue or a pair of two FIFO-queues if the channel is uni- or bi-directional.

The dynamic semantics of a block is determined by the process instances of the block. From an engineering point of view, blocks can be interpreted as modules or even indicate the distribution of parts of a system over different locations.

Communication in SDL is asynchronous, thus the sending and the consumption of a signal instance do not coincide. Process instances are equipped with an *input port* for the temporary storage of signal instances not yet processed.

Process instances are the only parts of an SDL system whose behaviour are totally determined by the specifier. Even timing constraints are expressible. Therefore, every process instance may use timers as it has access to a mechanism which counts, in terms of ticks, global time.

Referring to CCITT Recommendation Z.100 [1] the behaviour of channels and input ports is not explicitly specified. For example, for channels only their main characteristic (FIFO-queue) is noted. The idea to define an operational semantics for an SDL system is to explicitly define the operational semantics of all components of an SDL system (figure 1): channels, blocks, process instances, input ports, and timers. The details are described in Clause 5.

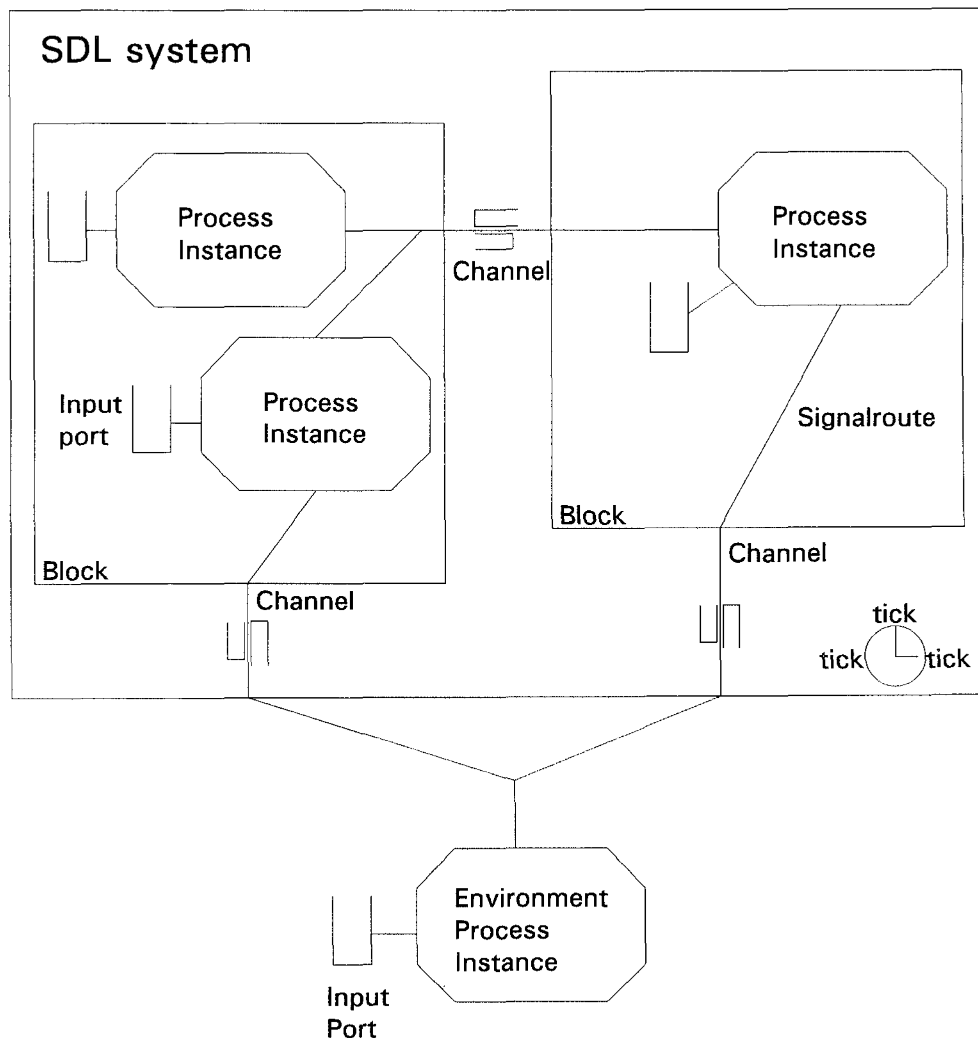

Figure 1: Conceptual model for SDL

4.2 Conceptual model for TTCN

A model for TTCN is not as well established as the one for SDL. The international standard ISO 9646 [2], Parts 1 to 3, provides an implicit description of the model introduced hereafter and which is called *TTCN tester* (figure 2). A TTCN tester executes a test case. The model described in the following is invented to aid the understanding of TTCN concepts and TTCN semantics, and is not intended as an implementation description.

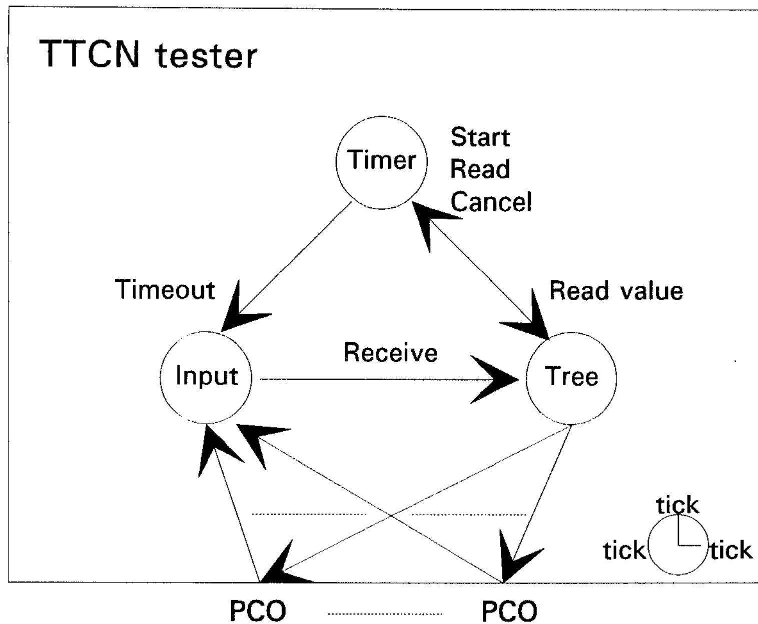

Figure 2: Conceptual model for a TTCN tester

4.2.1 TTCN tester

A TTCN tester consists of three communicating processes:

- the *tree process* executes a transformation (see below) of a TTCN behaviour tree of a test case;
- the *input process* deals with all incoming Abstract Service Primitives (ASPs) or Protocol Data Units (PDUs) and time-out signals. This process maintains a data structure referred to as *input stream* which contains all ASPs or PDUs of all P*oints of Control and Observation (PCO)* interleaved with time-out indications in the order of arrival. Only if a signal is in the input stream can it be processed by the TTCN tester.
  The PCO queues for incoming ASPs or PDUs and the time-out list mentioned in ISO 9646 [2], Part 3, are modelled by the input stream;
- the *timer process* keeps track of all running timers.

It is important to note that the tree process is the variant part of the model which changes from test case to test case. Associated with the three processes is a storage environment for values of variables and constants.

The internal structure of a TTCN tester and the communication relations between the three processes are shown in figure 2 above. In the sequel each of the processes is discussed on a more detailed level. The description also indicates how to relate these processes to concepts of ISO 9646 [2], Part 3.

4.2.1.1 The tree process

The tree process executes the events as indicated by an *abstract evaluation tree* (ISO 9646 [2], Part 3, Annex B). The abstract evaluation tree is the behaviour tree with all defaults attached and repeats eliminated. Furthermore, all GOTOs and tree attachments are assumed to be eliminated. This may result in a potentially infinite tree.

The tree process evaluates a set of alternatives with respect to the current contents of the input stream. The contents of the input stream represents a *snapshot*. In this model a snapshot is not *explicitly* taken. If a signal is received from a PCO or the timer process, the input stream is updated and thus may represent a new snapshot. While a set of alternatives is evaluated the contents of the input stream do not change. In case no matching signal can be found in the input stream the tree

process has to waits until a signal arrives from one of the PCOs or a timer expires and a time-out signal is put in the input stream.

4.2.1.2 The input process

The tree process need not immediately deal with an incoming message, but may delay this until it has finished its current task(s). This applies to ASPs or PDUs received from the PCOs and time-out signals emitted by the timer process. All inputs are temporarily stored by the input process in the input stream. The stored messages are in the possession of the TTCN tester. The tree process takes into account only a subset of all stored messages when evaluating a set of alternatives. This subset includes all ASPs or PDUs which have arrived first for each PCO and all time-out signals (which comprises the time-out list ISO 9646 [2], Part 3.

If the reception of more messages (ASPs, PDUs or time-out signals) is possible then these messages are put in the input stream in an arbitrary order.

4.2.1.3 The timer process

In TTCN (ISO 9446 [2], Part 3) time is counted as integer multiples of fixed (real) time units. In the conceptual model this is interpreted as *discrete time* counted in ticks. A tick represents a (fixed) duration of time. It is assumed that there is a mechanism that provides the update of the global time.

If a timer expires the timer process sends a signal, which identifies the expired timer, to the input process. If the tree process requests the elapsed time since a timer was started, the timer process returns the number of ticks that have been counted from the time the timer was started.

4.2.2 TTCN system

As a conceptual model for TTCN the TTCN tester is sufficient. However, in defining a common semantics representation for TTCN and SDL there obviously are some significant differences. A TTCN tester can be related to a process instance in SDL, thus no equivalent concepts for system, channel and block exist for TTCN. To overcome these differences the concepts TTCN system and abstract tester are introduced and the concept of PCO is modified. These changes are motivated in the following paragraphs.

A *TTCN system* (figure 3) is an entity which may receive ASPs or PDUs from the environment and send ASPs or PDUs to the environment via P*oints of Control and Observation (PCO)*. The environment of a TTCN system is formed by a *service provider* and an I*mplementation Under Test (IUT)* (see ISO 9646 [2], Part 1) which may be embedded in a S*ystem Under Test (SUT)*.

A PCO, as indicated in figure 3, provides a TTCN system with an interface to the environment. Communication between a TTCN tester and the environment is asynchronous. The following temporal relation between sending (by a TTCN tester) of an ASP or PDU and receiving (by the environment) of an ASP or PDU and vice versa can be established:

$$t_{sending} \le t_{receiving}$$

In general, a PCO can arbitrarily delay an ASP or PDU. Conceptually, each PCO has associated two queues: one for intermediate storage of incoming ASPs or PDUs and one for intermediate storage of outgoing ASPs or PDUs.

The concept of an *Abstract tester* is introduced to provide, in the conceptual model for TTCN, a concept similar to blocks in SDL. Mapping TTCN to the CSR presupposes the existence of exactly one abstract tester.

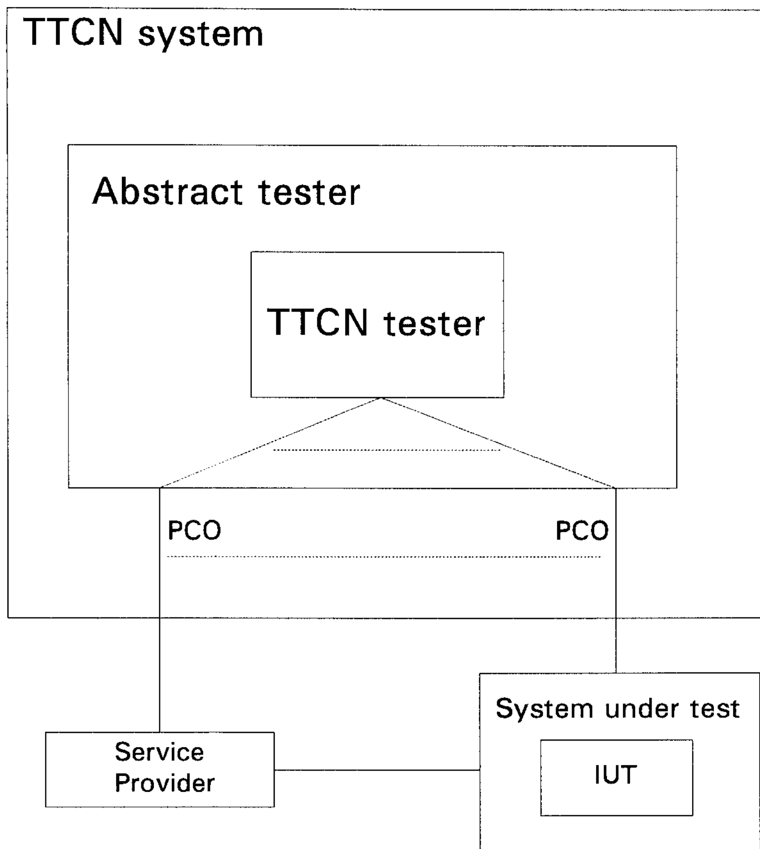

Figure 3: Conceptual model for a TTCN system

5 Common Semantics Representation (CSR)

5.1 Introduction

The CSR enables the representation of the dynamic behaviour of syntactically well-formed specifications in basic SDL and TTCN in a common model. To make the representation useful for reasoning about properties of such systems at different levels of abstraction, it is defined as a compositional hierarchical model. The structure of the CSR (figure 4) is defined to reflect the conceptual models as described in the previous Clause.

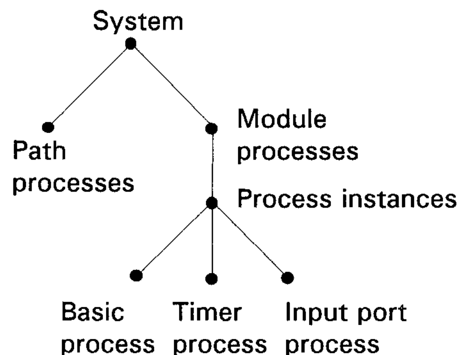

Figure 4: The structure of the CSR

The behaviour of each higher level entity can be derived from the behaviour of the entities at the next lower level. This property allows for an analysis of the behaviour of a system at different levels of observability. Observability is determined with respect to where an observer is placed. An observer in the environment of a system (figures 1 and 3) can only observe signals received from or sent to the system. If the observer is placed in the system, then the signals received from or sent to the environment and the signals exchanged between the components (blocks in case of SDL and the abstract tester in case of TTCN) can be observed.

The entities in the CSR are:

- **Basic process**
  A *basic process* results from a syntactical transformation of an *SDL process graph* or *TTCN abstract evaluation tree*. The range of this transformation is defined over a B*asic Process Algebra (BPA)*. The BPA identifies a set of events and a set of operators. The set of events is the union of a set of events common to SDL and TTCN, a set of SDL specific events, and a set of TTCN specific events. The events are considered to be atomic. The operators in the BPA are *prefixing* and *choice*. Prefixing and choice are used to combine events to compound behaviour descriptions.

- **Timer process**
  A *timer process* models the handling of SDL *active timers* and TTCN *running timers*. Expired timers are modelled as signals, which are transferred from the timer process to the input port process.

- **Input port process**
  An *input port process* models the reception and removal of signals to and from an input queue. For SDL this process models the behaviour of the *input port*, and for TTCN the input port process is equivalent to the *input process* in the conceptual model for TTCN.

- **Process instance**
  A *process instance* is composed of a basic process, a timer process, an input port process and a storage environment for variables and values. Hence, the behaviour of a process instance is based on the behaviour of these entities. Furthermore, the behaviour of a process instance can also depend on the value of variables in other process instances in the same

module. A Process instance is used to model the behaviour of an *SDL process instance* or the behaviour of the *TTCN tester.*

- **Module process**
  The behaviour of a *module process* is derived from the behaviour of the process instances residing in the module process. Module processes are used to model the behaviour of *SDL blocks* and the behaviour of a *TTCN abstract tester*.

- **Path process**
  A *path process* models the behaviour of an *SDL channel* and the behaviour of a *TTCN PCO*.

- **System**
  The behaviour of a *system* is derived from the behaviour of its path processes and its module processes. A system is used to model the behaviour of an *SDL system* and a *TTCN system*.

For each entity in the model the operational semantics is defined by L*abelled Transition Systems (LTS)*. This approach is convenient for the definition of compositional structures and allows for formal reasoning on temporal properties. The transition relation of each LTS is determined by a set of inference rules of the following form:

$$\frac{t_1 \;\ldots\; t_n}{t} \quad C$$

where t_1, ..., t_n and t are transitions. The set of premises of an inference rule, t_1, ..., t_n or the empty set, are transitions possible by the components of the entity or the entity itself. For example, in an inference rule for a process instance the premises are transitions of a basic process, a timer process and an input port process. The conclusion, transition t, is the inferred behaviour of the entity under consideration. The compositional structure of the model is expressed in terms of such inference rules. C defines side-conditions on the applicability of the inference rule: the inference rule can be applied only if all side-conditions hold.

Using Labelled Transition Systems implies that the concurrency of an SDL system and TTCN system is represented by interleaving of events. True concurrency is not part of the model. The discrete time model of SDL is preserved in the CSR. As outlined in subclause 4.4.2.3, on a certain level of abstraction the discrete time model is sufficient to model timing requirements specified in TTCN test cases.

As data influences the behaviour of a system, a model for data is also implicitly part of the CSR. The data model is a many-sorted Σ-algebra in which data terms are evaluated. The differences in both languages with respect to data typing become obvious if the common representation of data is investigated. The abstract model of a Σ-algebra is close to the data model defined for SDL. TTCN makes use of a concrete implementation-oriented data model known from many programming languages.

Each of the entities of the CSR model are defined in the following subclauses.

5.2 Definitions

Throughout the rest of this Clause, a fixed SDL specification or TTCN test suite specification *Spec* is assumed.

5.2.1 Notation

If S is a set, then S^* denotes the set of all *finite sequences*, including the *empty sequence* $\langle\rangle$, of elements of S. If $w, w' \in S^*$, $w = a_n a_{n-1} \ldots a_1$ and $w' = b_m b_{m-1} \ldots b_1$, $n,m > 0$, then $w::w' = a_n a_{n-1} \ldots a_1 b_m b_{m-1} \ldots b_1$ denotes the *concatenation* of w and w', and $\langle\rangle::w = w::\langle\rangle = w$ the concatenation of w and the empty sequence. In the sequence $w::w'$, b_1 is called the *first element* of the sequence, and a_n is called the *last element* of the sequence. The cardinality operation $\#S$ denotes the number of elements of the set S.

Given the sets $S_1,...,S_n$ then $S_1 \times ... \times S_n$ denotes the cartesian product.

Given the sets $S_1,...,S_n$ then $S_1 \cup ... \cup S_n$ is denoted by $\bigcup_{i=1,...,n} S_i$

Given the sets S_1, ..., S_n, the *disjoint union* $\biguplus_{i=1,...,n} S_i$ is defined as $\bigcup_{i=1,...,n} \{i\} \times S_i$

5.2.2 Domains

In the CSR, a module process is identified by a *module identifier*, and a path process is identified by a *path identifier*.

Definition: *ModId* is the set of all module identifiers in *Spec*. □

Definition: *PathId* is the set of all path identifiers in *Spec*. □

Since a module is related to a block in SDL, a module identifier is related to a block name. In TTCN, a module is related to an abstract tester, hence, in this case, there exists always only one module. Since the concept of an abstract tester is not explicitly represented in a TTCN test suite, there is no relation between any object in a TTCN test suite to a module identifier. However, an abstract tester in the CSR shall be identified by the module identifier *m*. Two path processes are related to a bi-directional channel in SDL, hence a path identifier is determined from a channel name. In case of TTCN, two path processes are related to a PCO and a path identifier is determined from a PCO identifier.

For each module process there exists a set of related basic processes. Each basic process is identified by a *basic process identifier*.

Definition: $BPId_m$ is the set of basic process identifiers related to a module *m*. □

It is assumed that the sets $BPId_m$ for all modules *m* are disjoint.

Definition: the set of all basic process identifiers in *Spec* is then given by

$$BPId = \bigcup_{m \in ModId} BPId_m$$

□

For SDL, a basic process is related to a process graph and a basic process identifier is related to a process name. For TTCN, a basic process is related to an abstract evaluation tree, in the conceptual model represented by the tree process. Hence, a basic process identifier is related to a test case identifier.

In the CSR, the behaviour of a process instance is determined by its related basic process and there may exist more than one process instance related to the same basic process. Each process instance is identified by a *process instance identifier*, and there exists disjoint infinite sets of process instance identifiers for each basic process identifier, denoted by $BPPId_X$ where $X \in BPId$.

Definition: $ModPId_m$ is the set of process instance identifiers for a module *m* defined by

$$ModPId_m = \bigcup_{X \in BPId_m} BPPId_X$$

□

Definition: PId_{spec} is the set of process instance identifiers in the system *Spec* defined by

$$PId_{spec} = \bigcup_{m \in ModId} ModPId_m$$

□

Definition: let $ModPId_{env}$ denote the set of process instance identifiers in the environment of a system, to which any signal communicated out of the system is sent and from which any signal to the system is sent. Then

$$PId = PId_{spec} \cup ModPId_{env}$$

is the set of all process instance identifiers in all modules of the system together with all process instance identifiers in the environment of the system. □

Definition: Let $\perp_{pid}$ be an element not in *PId*. $\perp_{pid}$ denotes the *undefined process instance identifier*, *i.e.* it does not identify any process instance in the system or in the environment. Then

$$PId_{\perp} = PId \cup \{\perp_{pid}\}$$

□

A process instance identifier is related to a pid-value in SDL. In the case of TTCN, it is assumed that there exist unique identifiers for the tester, the IUT or SUT and the service provider.

5.2.3 Data terms and signal terms

Data is handled in the CSR on an abstract level. Data types are specified algebraically. An algebraic specification of a data type defines a syntax for the denotation of data terms and an interpretation of data terms in an algebra. This subclause introduces only parts of the theory of algebraic specification necessary for the understanding of the subsequent subclauses. More detailed introductions to algebraic specifications of abstract data types (ADT) are given in "Fundamentals of algebraic specifications 1" [7], ISO 8807 [8] and CCITT Recommendation Z.100 [1].

5.2.3.1 Data terms

It is assumed that a signature $\Sigma = \langle Sort, OP \rangle$ is given with *Sort* a set of *sorts* and *OP* a set of *operations*. If *s* is a sort in *Sort* then this is denoted by $s \in Sort$. If $op : s_1,\ldots,s_n \rightarrow s$ is an operation definition in *OP* then this is denoted by $op : s_1,\ldots,s_n \rightarrow s \in OP$.

Definition: $Term_s$ is the set of *data ground terms* of sort *s*, inductively defined by:

a) $op : \rightarrow s \in OP$ *implies* $op \in Term_s$

b) $op : s_1,\ldots,s_n \rightarrow s \in OP, t_i \in Term_{s_i}$ *for all* $i = 1,\ldots,n$
implies $op(t_1,\ldots,t_n) \in Term_s$

□

For each sort *s* a set of variables Var_s exists. Whenever x_s is an element of Var_s this is denoted by $x_s \in Var_s$.

Definition: *Var* is the set of all variables so *Var* is the union of the sets Var_s:

$$Var = \bigcup_{s \in Sort} Var_s$$

□

Every variable keeps its sort which makes them unique in the set *Var*. In SDL and TTCN, variables are used for the storage of values. The set *Var* is the set of all variables and constants in *Spec*.

Definition: $Term_s(Var)$ is the set of *data terms* of sort *s* over the set of variables Var_s inductively defined by:

a) $x_s \in Var_s$ *implies* $x_s \in Term_s(Var)$

b) $op : \rightarrow s \in OP$ *implies* $op \in Term_s(Var)$

c) $op : s_1,\dots,s_n \rightarrow s \in OP, t_i \in Term_{s_i}(Var)$ *for all* $i = 1,\dots,n$
implies $op(t_1,\dots,t_n) \in Term_s(Var)$

□

In SDL a variable may be owned by another SDL process instance in the same block. These variables are called *revealed variables*. If x_s is a variable in Var_s and t a term which denotes a process instance identifier, then the set of data terms is supposed to additionally contain the following terms:

$$x_s \in Var_s, t \in Term_{PId}(Var) \text{ implies } x_{s,t} \in Term_s(Var)$$

Definition: *Term(Var)* is the set of all data terms over *Var*, defined by

$$Term(Var) = \bigcup_{s \in Sort} Term_s(Var)$$

and *Term* is the set of all data ground terms, defined by

$$Term = \bigcup_{s \in Sort} Term_s .$$

□

5.2.3.2 Signal terms

Signal ground terms and signal terms are defined similarly to data ground terms and data terms. Let *SigSort* and *SigOP* be the set of sorts and operations, disjoint from *Sort* and *OP,* to be used for signal declarations.

Definition: $SigTerm_{s'}$ is the set of *signal ground terms* of sort $s' \in SigSort$, inductively defined by:

a) $op' : \rightarrow s' \in SigOP$ *implies* $op' \in SigTerm_{s'}$
b) $op' : s_1,\dots,s_n \rightarrow s' \in SigOP, t_i \in Term_{s_i}$ *for all* $i = 1,\dots,n$
implies $op'(t_1,\dots,t_n) \in SigTerm_{s'}$

□

Definition: $SigTerm_{s'}(Var)$ is the set of *signal terms* of sort s' over the set of variables *Var*, inductively defined by:

a) $op' : \rightarrow s' \in SigOP$ *implies* $op' \in SigTerm_{s'}(Var)$
b) $op' : s_1,\dots,s_n \rightarrow s' \in SigOP, t_i \in Term_{s_i}(Var)$ *for all* $i = 1,\dots,n$
implies $op'(t_1,\dots,t_n) \in SigTerm_{s'}(Var)$

□

Definition: *SigTerm* is the set of all signal ground terms and *SigTerm(Var)* is the set of all signal terms and they are defined like the sets *Term* and *Term(Var)* respectively. □

For SDL, a signal term is related to an SDL signal identifier followed by a sequence of actual parameters. For TTCN, a signal term is related to a constraint.

5.2.3.3 Interpretation of data terms and signal terms

Let $\Sigma' = \langle Sort \cup SigSort, OP \cup SigOP \rangle$. The interpretation of data terms and signal terms is given in an $\Sigma'-$*algebra*, denoted by *A*, relative to a storage and environment where the values assigned to variables can be found. For each sort of the signature a carrier set exists in the algebra. The algebra *A* is assumed to contain an error element $error_s$ for each carrier set. The derivation of a Σ'*-algebra* from a signature Σ' is described in ISO 8807 [8] and CCITT Recommendation Z.100 [1].

Definition: the storage environment is defined for every process instance (refer to subclause 5.6) and is comprised of a set *Env* of partial functions ε from variables *Var* to locations *Loc* and a set *Sto* of partial functions ρ from locations *Loc* to *A*:

$$Env = \{\varepsilon \mid \varepsilon : Var \to Loc\}$$
$$Sto = \{\rho \mid \rho : Loc \to A\}$$

□

Definition: the interpretation of a data term *t* in the algebra *A* is given by the following inductive definition relative to functions ε and ρ

a) $\|x\|^{A}_{ms,\,\varepsilon,\,\rho} = \rho(\varepsilon(x))$ for $x \in Var$

b) $\|op(t_1, \ldots, t_n)\|^{A}_{ms,\,\varepsilon,\,\rho} = op^{A}\left(\|t_1\|^{A}_{ms,\,\varepsilon,\,\rho}, \ldots, \|t_n\|^{A}_{ms,\,\varepsilon,\,\rho}\right)$ for $n \geq 0$

op^A denotes the interpretation of operation *op* in algebra *A*. The index *ms* refers to a module storage where the values of revealed variables are stored. □

Definition: in order to resolve references to revealed variables, the interpretation of a revealed variable $x_{s,t}$ is defined by:

$$\|x_{s,t}\|^{A}_{ms,\,\varepsilon,\,\rho} = \|x_s\|^{A}_{ms,\,\varepsilon',\,\rho'}$$

where ε', ρ' are the environment and storage of the process instance referred to by $\|t\|^{A}_{ms,\,\varepsilon,\,\rho}$. □

If the process instance has stopped then

$\|x_{s,t}\|^{A}_{ms,\,\varepsilon,\,\rho} = \bot_t$ where $\bot_t$ denotes the undefined term.

For signal terms the following condition applies:

$$\|st\|^{A}_{ms,\,\varepsilon,\,\rho} = \|st'\|^{A}$$

The condition assures that the interpretation of signal term *st* in algebra *A* with respect to a storage and environment equals the interpretation of some signal ground term *st'*. This means, that all variables used in the signal term *st* are bound to values.

Definition: $SigTerm^{A}_{s'}$ is the set of interpretations of signal ground terms of sort *s'*. □

Definition: $SigTerm^A$ is the set of interpretations of signal ground terms. □

Whenever it is clear from the context *A*, *ms*, ε and ρ may be omitted when denoting interpretations.

5.2.4 Signals

Definition: a *signal* is a tuple $(t, pid, pid', p) \in \left(SigTerm^A \cup \{\bot_{st}\}\right) \times PId \times PId_{\bot} \times \left(PathId \cup \{\bot_{pth}\}\right)$. □

In the tuple *t* is the interpretation of a signal ground term, *i.e.* $t \in SigTerm^A$ or $t = \bot_{st} \notin SigTerm^A$. A signal where $t = \bot_{st}$ will be called an *undefined signal*, and is introduced in order to model an unforeseen message of TTCN. Moreover *pid* is the process instance identifier of the process instance which has sent the signal, *pid'* is the process instance identifier of the process instance

that is to receive the signal. p denotes the path on which the signal has been conveyed where $\perp_{pth}$ denotes the undefined path. $\perp_{pth}$ is used to indicate that the signal has not been conveyed by any path.

Definition: *Sig* is the set of all signals except for undefined signals. Sig^* denotes the set of all finite sequences of signals of *Sig*. □

Definition: $Sig_\perp$ ($Sig_\perp \supset Sig$) is the set of all signals including the undefined signal, and $Sig_\perp^*$ denotes the set of all finite sequences of such signals. □

In order to have a more convenient way of referring to the components of a signal, the following shorthands are introduced: whenever $sig = (t, pid, pid', p)$ then $Intr(sig) = t$, $From(sig) = pid$, $To(sig) = pid'$, and $Path(sig) = p$.

Definition: if $To(sig) \neq \perp_{pid}$ then *sig* is called an *explicit signal*. Otherwise *sig* is called an *implicit signal*. □

Definition: an *instantiation* of a signal is an explicit signal. The set of instantiations, *Inst(sig)*, of a signal *sig*, is defined as:

$$\text{Inst(sig)} = \begin{cases} \{\text{sig}\} & \text{if sig is an explicit signal ,} \\ \left\{ \text{sig'} \middle| \begin{array}{l} \text{Intr(sig')} = \text{Intr(sig)}, \text{From(sig')} = \text{From(sig)}, \\ \text{To(sig')} \in \text{PId}, \text{Path(sig')} = \text{Path(sig)} \end{array} \right\} & \text{if sig is an implicit signal .} \end{cases}$$

□

Intuitively, *Inst(sig)* denotes the set of all possible explicit signals of signal *sig*, *i.e.* *Inst(sig)* is the set of all signals *sig'* equal to *sig* except for that $To(sig') \in PId$. Hence, *Inst(sig)* where *sig* is an explicit signal is the set consisting of only *sig*.

A signal is related to a signal instance in SDL and a message in TTCN.

5.2.5 Time and timers

Definition: the time domain $Time^A$ is associated with a total ordering $\leq$, *i.e.*

$$\forall a_1, a_2 \in Time^A : (a_1 \leq a_2) \vee (a_2 \leq a_1)$$

There also exists a strict ordering $<$, *i.e.*

$$\forall a_1, a_2 \in Time^A : a_1 < a_2 \Leftrightarrow (a_1 \leq a_2) \wedge (a_1 \neq a_2)$$

□

There also exists a *global clock* with some time origin $a_0 \in Time^A$ and some positive time unit. The global clock can be read at any time and its value is referred to by *time*.

Definition: *Timers* is the set of all timer descriptions defined by

$$Timers = \left\{ \Theta \middle| \, \Theta \in Time^A \times Time^A \times SigTerm^A \right\}$$

□

Then a *timer* is a tuple $(t_1, t_2, si) \in Timers$, where t_1 is the time when the timer was started, t_2 is the time when the timer is to expire and $si = \| st \|$ is the interpretation of some signal ground term *st* identifying the timer. *si* is the signal term interpretation to be put into the input port process when the timer expires.

Definition: *Timers** is the set of all sequences of elements of the set *Timers*. □

A timer is related to an active timer in SDL and a running timer in TTCN.

5.2.6 Routes

Definition: the routes inside a module *m* is defined by the set:

$$Routes_m = \{(x,y) | x, y \in (BPId_m \cup PathId) \text{ and there is a connection leading from } x \text{ to } y\}$$

□

Definition: there also exists a set:

$$Routes_{env} = \{(pth, TOENV) | pth \in PathId \text{ and the path pth leads to the environment}\} \cup \{(FROMENV, pth) | pth \in PathId \text{ and the path pth leads from the environment}\}$$

representing the routes between a system and its environment. □

Definition: the set

$$Routes = \bigcup_{m \in ModId} Routes_m \cup Routes_{env}$$

is the set of all routes in the system. □

If there is a route leading from basic process X to basic process Y in the module *m*, then the pair (X,Y) is in the set $Routes_m$ (and by definition also in the set *Routes*). Similarly, if there is a route leading from basic process X in module *m* to the path *p* (or in the opposite direction) then the pair (X,p) (or (p,X)) is in the set $Routes_m$. In terms of SDL, the set $Routes_m$ represents the SDL signal routes in a module *m*. In terms of TTCN, the set $Routes_m$ represents the relation between an abstract tester and its environment expressed in terms of PCOs.

Definition: *route function*, to each module identified by $m \in ModId$ there is also associated a *route function* r_m such that

$$r_m : Routes_m \rightarrow 2^{SigTerm^A \cup \{\perp_{st}\}}$$

□

If $(X,Y) \in Routes_m$, then $\|s\| \in r_m(X,Y)$ means that a signal *sig* with $Intr(sig) = \|s\|$ may be conveyed by the route (X,Y). The route function $r_m(X,Y)$ for a module *m*, defines the signals that may be conveyed from X to Y. This models the SDL signal identifier set associated to signal routes, which specifies the sorts of the signals that may be conveyed by the signal route. In TTCN, a route function models the relation between an ASP or PDU and a PCO type.

NOTE: Every route may always convey an undefined signal in case of TTCN.

5.3 Basic process

A basic process results from the syntactical transformation of an SDL process, an SDL procedure or a TTCN abstract evaluation tree. A basic process is defined over a B*asic Process Algebra (BPA)* that consists of a set of *events* and a set of *operators*. The events are used to represent SDL actions and TTCN statements. For example, the SDL action *set(t, s)* and the TTCN operation *START T (S)* are represented in the BPA by the event *set(t, s)*. In some cases an SDL action or a TTCN statement is represented as a sequence of events in the BPA. The operators in the BPA are *event prefixing* and *choice*. The operators are used to reflect the structure of SDL process graphs and TTCN abstract evaluation trees as a basic process.

5.3.1 States of a basic process

The set of states of a basic process, denoted by *BProc*, consists of all basic process expressions which can be derived from the following grammar:

$$P ::= nil \mid e; P \mid P + P \mid P \oplus P$$

where P is a basic process, e is an event, and ";", "+" and "$\oplus$" are operators of the BPA. The terminal symbol *nil* is used to denote a basic process that cannot perform any actions.

A *declaration D* of a basic process is defined as:

$$D : BPid \rightarrow BProc \times Inst \times Par \times (Var \times Term(Var))^*$$

where:

$$Inst = N \times N_\infty$$

$$Par = Var^* \times Var^*$$

and $\forall (x,t) \in (Var \times Term(Var)) : x \in Var_s \Rightarrow t \in Term_s(Var) \wedge t \neq error_s$

If the declaration D of a basic process X is:

$$D(X) = (P, (init, max), ((x_1, \ldots, x_k), (y_1, \ldots, y_m)), ((z_1, t_1), \ldots, (z_n, t_n)))$$

then, in the case of SDL, $P \in BProc$ is the representation of an SDL process graph. *init* is the initial number of process instances and *max* is the maximal number of simultaneously existing process instances. If X is an SDL procedure, *init* is always *zero* and *max* is always ∞ as an SDL procedure can be called an infinite number of times. The parameters x_i are call-by-reference parameters, y_i are call-by-value parameters, and (z_i, t_i) are tuples of local variables and data terms. Data term t_i is the initial value of local variable z_i. For an SDL process declaration the tuple $(x_1, \ldots, x_k)$ is empty as call-by-reference parameters cannot be used in SDL process declaration.

In the case of TTCN, $P \in BProc$ is the representation of a TTCN abstract evaluation tree. *init* and *max* are always 1. The tuples $(x_1, \ldots, x_k)$ and $(y_1, \ldots, y_m)$ are empty because for the test case root tree no parameters are defined and all attached trees have been expanded. The tuple $((z_1, t_1), \ldots, (z_n, t_n))$ are local variables e.g., the pre-defined variable R which holds the preliminary verdict.

5.3.2 Events of a basic process

The events in the BPA are of three categories: SDL specific events, TTCN specific events and events that are common to SDL and TTCN. The SDL events, ε_{SDL}, are events that are used to represent SDL specific operations, *i.e.* represent operations that occur only in SDL. Similarly, the TTCN events, ε_{TTCN}, are events that are used to represent TTCN specific statements. Operations and statements that occur in both SDL and TTCN are represented as common events, ε_{Common}. The events ε_B a basic process can perform are defined as:

$$\varepsilon_B = \varepsilon_{SDL} \cup \varepsilon_{TTCN} \cup \varepsilon_{Common}$$

where

$$\varepsilon_{SDL} = \{input(s), save(ss), create\ X(t_1, \ldots, t_n), output(s)\ to(pid), output(s)\ to(pid)\ via\ r, call\ X(v_1, \ldots, v_n)(t_1, \ldots, t_n), return\}$$

$$\varepsilon_{TTCN} = \{input(s)\ [bt], input(s)\ via\ p\ [bt], read(v, s), otherwise\ p\ [bt]\}$$

$$\varepsilon_{Common} = \{stop, v := t, [bt], set(t, s), reset(s), output(s)\ via\ r\}$$

and

$$s \in SigTerm(Var)$$

$$ss \in SigSort$$

$$v, v_1, \ldots, v_n \in Var$$

$$t, t_1, \ldots, t_n \in Term(Var)$$

$bt \in Term_{Boolean}(Var)$
$pid \in Var_{PId}$
$r \in Routes$
$p \in PathId$
$X \in BPId$

The names of the events indicate to which operations or statements they correspond. For example, the event *input(s)* in ε_{SDL} is used to represent the SDL action *Input s*. The event *input(s) via p [bt]* in ε_{TTCN} is used to represent the TTCN *Receive* event p?s [bt].

5.3.3 Operators in the BPA

There exists three operators in the BPA: ";", "+" and "⊕". The operator ";" denotes sequential composition, *e.g.* if $P = e_1; P'$, the basic process P performs event e_1 before it can behave as specified by P'. The operators + and ⊕ respectively denote a choice and a priority choice in a basic process. If $P = e_1; Q + e_2; R$ then process P may perform event e_1 or event e_2 and then behaves like either Q or R. If both events are possible, then the selection of either e_1 or e_2 is made non-deterministic. If $P = e_1; Q \oplus e_2; R$ then process P may perform event e_1 or event e_2 and then behave like either Q or R. If both events are possible, then P performs e_1, i.e. e_1 takes priority over e_2.

The reason for having two different choice operators in the BPA, is that there is a difference in how choices are resolved in SDL and TTCN. The +-operator is used to represent the SDL decision action and the selection of an input in an SDL state, while the ⊕-operator is used to represent the selection of an event line in a set of alternatives in TTCN.

5.3.4 Example

This example illustrates the transformation of an SDL process and a TTCN behaviour tree into a basic process definition. The SDL process graph in figure 6 is represented in the CSR as

$$D(SRV) = \left(P, (1,100), ((),()), ((i, \bot_t))\right)$$

where

$P ::= i := 1;\ output(a)\ via\ C;\ S1$
$S1 ::= (input(b);\ (i := i + 1);\ S1) + (input(c);\ output(d(i))\ via\ C;\ stop;\ nil)$

Process SRV(1, 100)
DCL i integer;

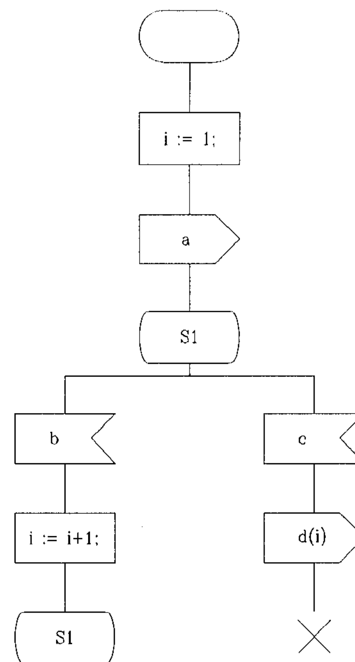

Figure 6: An example of an SDL process graph

As another example, the TTCN test case in table 1 is represented in the CSR as:

$$D(TREE_EX) = \left(P, (1,1), ((),()), ((final, \perp_t))\right)$$

where

$P ::=$ *output(cr1) via L;*
(input(cc1) via L [true];
output(dtr1) via L;
(input(dti1) via L [true];
output(dscr1) via L; final := pass; stop; nil)
$\oplus$ *(input(dsci1) via L [true]; final := inconc; stop; nil)))*
$\oplus$ *(input(dsci1) via L [true]; final := inconc; stop; nil)*

Table 1: An example of a TTCN Test Case

Test Case Dynamic Behaviour

Test Case Name: TREE_EX
Group: group
Purpose:
Default:
Comments:

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		L ! CONNECTrequest	CR1		
2		L ? CONNECTconfirm	CC1		
3		L ! DATArequest	DTR1		
4		L ? DATAindication	DTI1		
5		L ! DISCONNECTrequest	DSCR1	Pass	
6		L ? DISCONNECTindication	DSCI1	Inconc	
7		L ? DISCONNECTindication	DSCI1	Inconc	

Detailed Comments:

5.3.5 Semantics of a basic process

The semantics of a basic process is given by a labelled transition system

$$B = \left(BProc, \varepsilon_B, \longrightarrow_B\right)$$

where $\longrightarrow_B$ is the transition relation associated with the set of inference rules given below.

Each rule is of the form $\frac{b'}{b}$ C stating that if b' is a transition in B then b is a transition in B provided that side-condition C holds.

5.3.5.1 Inference rule B 1

A basic process consisting of a sequence of events can always perform the first event in the sequence.

Let $e \in \varepsilon_B$ and $P \in BProc$ and $e; P$ a basic process

$$\frac{}{e;P \xrightarrow{e}_B P}$$

A transition inferred from this rule models the behaviour when an SDL process instance executes an action of a transition, or a TTCN tester executes a statement of an event line, or performs an assignment of a verdict.

5.3.5.2 Inference rule B 2 and B 3

If it can be inferred that a basic process in a choice expression can execute an event then the choice expression may also be resolved by execution of this event.

Let $e \in \varepsilon_B$ and $P, P', Q, Q' \in BProc$

$$\frac{P \xrightarrow{e}_B P'}{P + Q \xrightarrow{e}_B P'}$$

and

$$\frac{Q \xrightarrow{e}_B Q'}{P + Q \xrightarrow{e}_B Q'}$$

The +-operator is used to model the structuring of an SDL process graph when using the decision or the state construct. A transition inferred from these rules models the behaviour when a decision is made or an input is performed.

5.3.5.3 Inference rule B 4 and B 5

In a priority choice expression if it can be inferred that a basic process can execute an event then this event can also be executed in the choice expression.

Let $e \in \varepsilon_B$ and $P, P', Q, Q' \in BProc$

$$\frac{P \xrightarrow{e}_B P'}{P \oplus Q \xrightarrow{e}_B P'}$$

and

$$\frac{Q \xrightarrow{e}_B Q'}{P \oplus Q \xrightarrow{e}_B Q'}$$

The ⊕-operator is used to model the structuring of a TTCN behaviour tree at the different levels of indentation.

At the basic process level the behaviour of a basic process is the same with respect to the +- and ⊕-operator as the prioritising of events is dependent on information in the environment of a basic process. The full semantics for the ⊕-operator is defined at the process instance level.

5.4 Timer process

A timer process is used to model the handling of SDL active timers and TTCN running timers.

5.4.1 States of a timer process

A state of a timer process consists of an ordered finite sequence of *active timers*, $Timers^*$. The first timer in the sequence is the timer that is to expire next. Timers are identified by the interpretation of a signal ground term specified when the timer was activated.

5.4.2 Events of a timer process

The set of events ε_T a timer process can perform is defined by:

$$\varepsilon_T = Start \uplus SI_{tstop} \uplus SI_{expired} \uplus Request \uplus \{notexpired\}$$

where

$$Start = \left(Time^A \times SigTerm^A\right)$$

$$SI_{tstop} = SigTerm^A$$

$$SI_{expired} = SigTerm^A$$

$$Request = \left(Time^A \times SigTerm^A\right)$$

An event $(t, s) \in Start$ is denoted $tstart(t, s)$. The event models the start of a timer. An event $si \in SI_{tstop}$ is denoted $tstop(si)$ and represents the stopping of the specified timer. An event $si \in SI_{expired}$ is denoted $expired(si)$ and models the expiration of the specified timer. An event $(t, si) \in Request$ is denoted $request(t, si)$. $request(t, si)$ models the request of the elapsed time since the specified timer was started. Finally the set $\{notexpired\}$ represents the event that no timer is to expire, and is denoted *notexpired*.

5.4.3 Semantics of a timer process

The semantics of a timer process is given by a labelled transition system:

$$T = \left(Timers^*, \varepsilon_T, \longrightarrow_T\right)$$

where $\longrightarrow_T$ is defined by the inference rules given below. Each rule is of the form $\frac{C}{t}$ stating that t is a transition in T provided that condition C holds. $\Theta \in Timers^*$ will be used to denote the state of a timer process, and whenever $(\Theta, e, \Theta') \in \longrightarrow_T$, the notation $\Theta \xrightarrow{e}_T \Theta'$ is used. In the definition of the rules the function *insert* is used to insert a timer in a sequence of timers. The function is defined such that the list is always sorted with the timer to expire next as the first element of the list and is defined as:

$$insert : Timers \times Timers^* \rightarrow Timers^*$$

$$insert(t, \langle\rangle) :- \langle t\rangle$$

$$\begin{aligned} &insert((t_1, t_2, si), (t_3, t_4, si')::\Theta) :- \\ &\quad t_2 < t_4 \rightarrow (t_3, t_4, si')::insert((t_1, t_2, si), \Theta) \\ &\quad t_2 \geq t_4 \rightarrow (t_1, t_2, si)::(t_3, t_4, si')::\Theta \end{aligned}$$

The function $filter_T$ is used to remove all occurrences of timers, identified by the interpretation $si = \|st\|$ of some signal ground term st, within a sequence of timers and is defined as:

$$filter_T : SigTerm^A \times Timers^* \rightarrow Timers^*$$

$$filter_T(t, \langle\rangle) :- \langle\rangle$$

$$\begin{aligned} &filter_T(si, (t_1, t_2, si')::\Theta) :- \\ &\quad si = si' \rightarrow filter_T(si, \Theta) \\ &\quad si \neq si' \rightarrow (t_1, t_2, si')::filter_T(si, \Theta) \end{aligned}$$

NOTE: Even if this function is defined such that every occurrence of a specific timer is removed in the current model it is the case that there is at most one occurrence of a timer in the list.

Finally, the predicate $e : Timers^*$ is used to determine if a timer is to expire and is defined as:

$$e(\langle\rangle) = false.$$

$$e(\Theta::(t_1, t_2, si)) = \begin{cases} true & \text{if } time \geq t_2 \\ false & \text{if } time < t_2 \end{cases}$$

where *time* is the current time.

5.4.3.1 Inference rule T 1

A timer process may always start a timer. The result of the event is that all occurrences of active timers of the specified timer are removed from the timer list and then the timer instance is inserted in the list.

Let $t \in Time^A$ and $si \in SigTerm^A$

$$\frac{}{\Theta \xrightarrow{tstart(t, si)}_T \Theta'}$$

where $\Theta' = insert((time, t, si), filter_T(si, \Theta))$.

A transition inferred from this rule models the behaviour of a set timer action performed in SDL or a start timer event performed in TTCN.

5.4.3.2 Inference rule T 2

An active timer may be removed from the sequence of active timers. This occurs when the timer process perform the event *tstop(si)*. The result of the stop timer event is that all occurrences of timers identified by *si*, are removed from the list of timers.

Let $si \in SigTerm^A$

$$\frac{}{\Theta \xrightarrow{tstop(si)}_T \Theta'}$$

where $\Theta' = filter_T(si, \Theta)$.

A transition inferred from this rule models the behaviour when a reset timer action is performed in SDL or a cancel timer event is performed in TTCN.

5.4.3.3 Inference rule T 3

A timer process may provide information about the amount of time elapsed for an active timer. When the requested timer is not active the elapsed time value is 0.

Let $t \in Time^A$ and $si \in SigTerm^A$

$$\frac{}{\Theta \xrightarrow{request(t, si)}_T \Theta}$$

where $t = \begin{cases} time - t_1 & if\ \exists tim = (t_1, t_2, si) : \Theta = \Theta' :: tim :: \Theta'' \\ 0 & otherwise \end{cases}$

A transition inferred from this rule models the behaviour of a timer process when a TTCN tester performs a read timer event.

5.4.3.4 Inference rule T 4

A timer process may perform the event *expired*(*si*) when the first timer in the timer list is to expire, *i.e.* the predicate *e* is *true*. The result of the event is that the expired timer is removed from the list of active timers.

Let $si \in SigTerm^A$ and $(t_1, t_2, si) \in Timers$

$$\frac{}{\Theta :: (t_1, t_2, si) \xrightarrow{expired(si)}_T \Theta} \quad e(\Theta :: (t_1, t_2, si))$$

A transition inferred from this rule models the behaviour of the timer process when a timer in SDL or TTCN expires.

5.4.3.5 Inference rule T 5

A timer process may perform the event *notexpired* when the first timer of the timer list is not to expire, *i.e.* the predicate *e* is *false*.

$$\frac{}{\Theta \xrightarrow{notexpired}_T \Theta} \quad \neg e(\Theta)$$

A transition inferred from this rule models the behaviour of the timer process when the local time of a process instance is updated.

5.5 Input port process

The input port process models the behaviour of an SDL input port and the behaviour of the input queue of a TTCN tester.

5.5.1 States of an input port process

An input port process consists of two queues of signals, denoted by σ and σ'. The queue σ is termed *input queue* and the queue σ' is termed *saved signals queue*. The latter queue is used to hold all *saved* signals.

The state of an input port process is an element in the set

$$IProc = \{(\sigma, \sigma') \mid \sigma, \sigma' \in Sig_{\perp}^{*}\}$$

5.5.2 Events of an input port process

The set of events an input port process can perform is defined by:

$$\varepsilon_I = A \cup Sig_{deliver,p} \cup Sig_{timeout} \cup SigSort_{preserve} \cup SigTerm_{filter}^{A}$$

where

$$A = Sig_{receive} \cup Sig_{deliver}$$
$$Sig_{receive} = Sig_{\perp}$$
$$Sig_{deliver} = Sig_{timeout} = Sig,$$
$$Sig_{deliver,p} = (Sig_{\perp} \times PathId),$$
$$Sig_{preserve} = SigSort$$

An event $sig \in Sig_{receive}$ is denoted by *receive(sig)*. *receive(sig)* models the reception of a signal by an input port process. An event $sig \in Sig_{deliver}$ is denoted by *deliver(sig)*. *deliver(sig)* models the delivery of a signal to a process instance. An event $(sig, p) \in Sig_{deliver,p}$ is denoted by *deliver(sig, p)*. *deliver(sig, p)* models the delivery of a signal conveyed by a particular path process to a process instance. An event $sig \in Sig_{timeout}$ is denoted by *timeout(sig)*. *timeout(sig)* models the delivery of a time-out signal to a process instance. An event $s' \in SigSort_{preserve}$ is denoted by *preserve(s')*. *preserve(s')* models the saving of a signal of sort *s'* in the saved signals queue. Finally, an event $si \in SigTerm_{filter}^{A}$ is denoted by $filter_I(si)$. $filter_I(si)$ models the deletion of signals from either the input queue or the saved signals queue.

5.5.3 Semantics of an input port process

The semantics of an input port process is given by a labelled transition system

$$I = (IProc, \varepsilon_I, \longrightarrow_I)$$

where $\longrightarrow_I$ is the transition relation associated with the set of inference rules given below. Each inference rule is of the form $\frac{}{i}\ C$ stating that *i* is a transition in I provided that side-condition *C* holds. $(\sigma, \sigma') \in IProc$ shall be used to denote a state of an input port process, and whenever $((\sigma, \sigma'), e, (\sigma'', \sigma''')) \in \longrightarrow_I$ this is stated as $(\sigma, \sigma') \xrightarrow{e}_I (\sigma'', \sigma''')$.

In the definition of the rules the function $filter_I$ is used to remove every occurrence of entries identified by $si \in SigTerm^{A}$ from a queue. $filter_I$ is defined as:

$$filter_I : SigTerm^{A} \times Sig_{\perp}^{*} \rightarrow Sig_{\perp}^{*}$$

$$filter_I(si, \langle\rangle) :- \langle\rangle$$
$$filter_I(si, (si', pid, pid', p) :: \sigma) :-$$
$$si = si' \rightarrow filter_I(si, \sigma)$$
$$si \neq si' \rightarrow (si', pid, pid', p) :: filter_I(si, \sigma)$$

In the following inference rules let $(\sigma, \sigma') \in IProc$.

5.5.3.1 Inference rule I 1

An input port process can always receive a signal. The reception causes the signal to be appended to the input queue.

Let $sig \in Sig_\perp$

$$\frac{}{(\sigma, \sigma') \xrightarrow{receive(sig)}_I (sig :: \sigma, \sigma')}$$

A transition inferred from this rule models the behaviour of an SDL input port or a TTCN input queue when a signal instance or a message is enqueued.

5.5.3.2 Inference rule I 2

An input port process can always deliver the first signal from the input queue. After removal of the delivered signal the saved signals queue is inserted in front of the input queue.

Let $sig \in Sig$

$$\frac{}{(\sigma :: sig, \sigma') \xrightarrow{deliver(sig)}_I (\sigma :: \sigma', \langle\rangle)}$$

A transition inferred from this rule models the behaviour of an SDL input port when a signal instance is consumed.

5.5.3.3 Inference rule I 3

An input port process can always deliver a signal from the input queue which has been received from a particular path process. The first such signal from the input queue is delivered.

Let $sig \in Sig_\perp$ and $p \in PathId$

$$\frac{}{(\sigma :: sig :: \sigma', \sigma'') \xrightarrow{deliver(sig, p)}_I (\sigma :: \sigma' :: \sigma'', \langle\rangle)} \quad \begin{array}{l} Path(sig) = p \\ \forall sig' \in \sigma' : Path(sig') \neq p \end{array}$$

A transition inferred from this rule models the behaviour of a TTCN tester when a message is consumed from a PCO queue.

5.5.3.4 Inference rule I 4

An input port process can always deliver a time-out signal. A signal from the input queue with the undefined path identifier is delivered.

Let $sig \in Sig$

$$\frac{}{(\sigma :: sig :: \sigma', \sigma'') \xrightarrow{timeout(sig)}_I (\sigma :: \sigma' :: \sigma'', \langle\rangle)} \quad Path(sig) = \perp_{pth}$$

A transition inferred from this rule models the behaviour of a TTCN input queue when a time-out message is consumed.

5.5.3.5 Inference rule I 5

An input port process can always move a signal from the input queue to the saved signals queue.

Let $sig \in Sig$ and $s' \in SigSort$,

$$\frac{}{(\sigma::sig,\sigma') \xrightarrow{preserve(s')}_I (\sigma,sig::\sigma')} \quad Intr(sig) \in SigTerm_{s'}^{A},$$

A transition inferred from this rule models the behaviour of an SDL input port when a signal instance is saved.

5.5.3.6 Inference rule I 6

A signal can always be removed from the input queue and saved signals queue.

Let $si \in SigTerm^A$

$$\frac{}{(\sigma,\sigma') \xrightarrow{filter(si)}_I (\sigma'',\sigma''')}$$

where $\sigma'' = filter_I(si,\sigma)$ and $\sigma''' = filter_I(si,\sigma')$.

A transition inferred from this rule models the behaviour of an SDL input port when a reset timer action is performed or of a TTCN input queue when a cancel timer event is performed.

5.6 Process instance

A process instance models the behaviour of an SDL process instance or a TTCN tester.

5.6.1 States of a process instance

A state of a process instance is an element of the set

$$Proc = \left\{ \langle \pi, (\sigma,\sigma'), \Theta \rangle \middle| \pi \in (BProc \times Env \times Sto)^*, (\sigma,\sigma') \in IProc, \Theta \in Timers^* \right\} \cup \{\bot_{pi}\}$$

A process instance is modelled as a tuple $\langle \pi, (\sigma,\sigma'), \Theta \rangle$, where π is a sequence of basic processes, each associated with an *environment* and *a storage*. (σ,σ') is an input port process state and Θ is a timer process state. $\bot_{pi}$ denotes the undefined process instance.

The sequence of basic processes is introduced to model the procedure call mechanism in SDL.

The environment and the storage of a basic process is comprised of the functions $\varepsilon \in Env$ and $\rho \in Sto$. The domain of ε for a process instance $pi \in Proc$ is comprised of the set of variables declared for the SDL process or TTCN test case in *Spec* together with the following set of predefined variables:

self	contains the process instance identifier of *p*;
parent	contains the process instance identifier of the process instance that created *p*;
offspring	contains the process instance identifier of the most recently created process instance;

sender contains the process instance identifier of the sender of the signal most recently consumed by p;

now contains the local time of the process retrieved from the global clock *time*;

$active_{si}$ for each signal instance $si \in SigTerm^A$ a variable $active_{si}$ contains the value *true* if there exists an active timer identified by *si* and *false* otherwise.

The variables are defined over the domains: $self, parent, offspring, sender \in Var_{PId}$, $now \in Var_{Time}$ and $active_{si} \in Var_{Boolean}$.

The notation $\varepsilon[x \mapsto loc]$ is used to denote the environment equal to ε except for the mapping of variable x to location *loc*. $\varepsilon_\perp$ denotes the undefined environment. The notation $\rho[loc \mapsto a]$ denotes the storage equal to storage ρ except for the mapping of location *loc* to the value *a*. The undefined storage is denoted by $\rho_\perp$. The notation $\pi::P_{\varepsilon,\rho}$ is used to denote a sequence of basic processes where basic process P associated with environment ε and storage ρ is the first element in the sequence.

5.6.2 Events of a process instance

The set of events ε_{PI} a process instance can perform is defined by:

$$\varepsilon_{PI} = A \cup Create \cup \{\tau\}$$

where

$$A = Receive \uplus Output \uplus Output_{via}$$
$$Create = \biguplus_{X \in BPId} Create_X$$

and

$$Receive = Sig_\perp$$
$$Output = Sig$$
$$Output_{via} = Sig \times Routes$$
$$Create_X = \left(\{X\} \times \left(Term^A\right)^*\right)$$

An event $sig \in Receive$ is denoted by *receive(sig)*. *receive(sig)* models the reception of a signal by the process instance. An event $sig \in Output$ is denoted *output(sig)*. *output(sig)* models the sending of an explicit signal. An event $(sig, r) \in Output_{via}$ is denoted by *output(sig) via r*. *output(sig) via r* models the sending of an explicit or implicit signal via a specific route. An event $(X, a_1::\ldots::a_n) \in Create_X$ is denoted by *create* $X(a_1, \ldots, a_n)$. *create* $X(a_1, \ldots, a_n)$ models the creation of a process instance. Finally, τ denotes an internal event not observable in the environment of the process instance.

5.6.3 Semantics of a process instance

The behaviour of a process instance cannot generally be defined independently of other process instances in the same module process due to the possibility of revealed variables. Information about the values of revealed variables are needed to determine the transition relation of a process instance. Therefore the concept of module storage is introduced.

A *module storage* of a module process m is a tuple $\langle P_{pid_1}, \ldots, P_{pid_n} \rangle$, $P_{pid_i} \in Proc$. A module storage is indexed by process instance identifiers $pid_i \in ModPid_m$. Intuitively, a module storage determines the state of all process instances in the same module process. The set of all module storages $ModStor_m$ of a module process m is defined as:

$$ModStor_m = \{\langle P_{pid_1}, \ldots, P_{pid_n} \rangle | P_{pid_i} \in Proc, pid_i \in ModPid_m\}$$

The set of all module storages is defined as:

$$ModStor = \biguplus_{m \in ModId} ModStor_m$$

The following notation is introduced to deal with module storages. For $ms \in ModStor$, $pid \in PId_{Spec}$, $pi \in Proc$, $\varepsilon \in Env$, and $\rho \in Sto$.

$ms[pid \mapsto pi]$ denotes the module storage equal to *ms* except for the process instance indexed by *pid* which has been updated by process instance *pi.*

$ms(pid)$ denotes the process instance indexed by *pid* in module storage *ms*. If there is no process instance *pi* with process instance identifier *pid* in *ms* then *ms(pid)* is undefined.

$ms_{env}(pid)$ denotes the environment ε of the process instance indexed by *pid* in module storage *ms*.

$ms_{sto}(pid)$ denotes the storage ρ of the process instance indexed by *pid* in module storage *ms*.

$ms_{env}(pid)$ and $ms_{sto}(pid)$ are undefined if *ms(pid)* is undefined.

The semantics of a process instance is then defined by the *extended* labelled transition system

$$PI = \left(Proc, \varepsilon_{PI}, \longrightarrow_{PI}, ModStor\right)$$

The transition relation $\longrightarrow_{PI} \subseteq (ModStor \times Proc) \times \varepsilon_{PI} \times (ModStor \times Proc)$ ranges over tuples composed of a module storage and a state of a process instance, and events. Whenever $((ms, p), e, (ms, p')) \in \longrightarrow_{PI}$ this is denoted by $ms \vdash p \xrightarrow{e}_{PI} p'$. This notation is used to stress the fact that the behaviour of a process instance is determined with respect to a fixed module storage. Assuming a fixed module storage is reasonable as:

- process instance *p* does only change its local variables, and hence module storage ms does not change with respect to values of revealed variables;
- no other process instance in the module storage concurrently performs an event. The concurrent execution of process instances in a module is modelled by interleaving of their events;
- $\longrightarrow_{PI}$ does not define how a module storage change due to events performed by process instance *p;*
- finally, the behaviour of a process instance can be determined in every module storage context even if knowledge about the other processes in the module makes it possible to exclude some of these module storage contexts. At the module level the behaviour is restricted to include only the behaviour possible taking into account the restrictions due to mutual dependencies of processes within the module.

The generic structure of an inference rule is given as:

$$\frac{P \xrightarrow{e_b}_{B} P' \quad (\sigma, \sigma') \xrightarrow{e_i}_{I} (\sigma'', \sigma''') \quad \Theta \xrightarrow{e_t}_{T} \Theta'}{ms \vdash \left\langle \pi :: P_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \right\rangle \xrightarrow{e_p}_{PI} \left\langle \pi :: P'_{\varepsilon',\rho'}, (\sigma'', \sigma'''), \Theta' \right\rangle} \quad C$$

stating that $\langle \pi::P_{\varepsilon,\rho}, (\sigma,\sigma'), \Theta \rangle \xrightarrow{e_p}_{PI} \langle \pi::P'_{\varepsilon',\rho'}, (\sigma'',\sigma'''), \Theta' \rangle$ is a transition in PI derivable from module storage $ms \in ModStor$, provided that $P \xrightarrow{e_b}_B P'$ is a transition of the basic process P and $(\sigma,\sigma') \xrightarrow{e_i}_I (\sigma'',\sigma''')$ is a transition in I and $\Theta \xrightarrow{e_t}_T \Theta'$ is a transition in T, and provided that the condition C holds.

In a basic process the priority choice operator $\oplus$ is used to define the selection among a set of TTCN events. The full semantics of this operator can be defined only at the process instance level. So the following meta rule applies for all inference rules defined below which includes a transition of the basic process as part of the premise. Let the structure of a basic process be $P = Q \oplus (e_b; P')$ and $e_b, e_b' \in \varepsilon_B$ and $e_p, e_p' \in \varepsilon_{PI}$

$$\frac{P \xrightarrow{e_b}_B P' \quad (\sigma,\sigma') \xrightarrow{e_i}_I (\sigma'',\sigma''') \quad \Theta \xrightarrow{e_t}_T \Theta'}{ms \vdash \langle \pi::P_{\varepsilon,\rho}, (\sigma,\sigma'), \Theta \rangle \xrightarrow{e_p}_{PI} \langle \pi::P'_{\varepsilon',\rho'}, (\sigma'',\sigma'''), \Theta' \rangle} \quad C$$

where

$$C \equiv \forall e_b' : \begin{cases} e_b' \neq [bt] \Rightarrow \neg\left(Q \xrightarrow{e_b'}_B Q' \wedge (\sigma,\sigma') \xrightarrow{e_i}_I (\sigma'',\sigma''') \wedge \Theta \xrightarrow{e_t}_T \Theta'\right) \\ e_b' = [bt] \Rightarrow \neg\left(Q \xrightarrow{[bt]}_B Q' \wedge \|bt\| = true\right) \end{cases}$$

$Q \xrightarrow{e_b'}_B Q'$ is a transition of the basic process Q (part of P). The transitions of the input port process and the timer process are optional.

Intuitively this rule defines that a process instance can only perform the event e_p if it cannot perform an event derived from Q.

Throughout the rest of this subclause let $P, P' \in BProc$, $(\sigma,\sigma'), (\sigma'',\sigma''') \in IProc$, $\Theta, \Theta' \in Timers^*$, $\varepsilon, \varepsilon' \in Env$, $\rho, \rho' \in Sto$, $now \in Var_{Time}$ and $time \in Time^A$.

5.6.3.1 Inference rule Proc 1

A process instance performs an internal event when the local time *now* is updated because its value is less than the current time value of the global clock and no timer has expired.

$$\frac{\Theta \xrightarrow{notexpired}_T \Theta}{ms \vdash \langle \pi::P_{\varepsilon,\rho}, (\sigma,\sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi::P_{\varepsilon,\rho'}, (\sigma,\sigma'), \Theta \rangle} \quad \|now\|_{\varepsilon,\rho} < time$$

where $\rho' = \rho[\varepsilon(now) \mapsto time]$.

5.6.3.2 Inference rule Proc 2

A process instance performs an internal event if a timer has expired and the value of variable *now* is less than the value *time* of the global clock.

Let $sig \in Sig$ with $sig = (si, pid, pid, \perp_{pth})$ where $si \in SigTerm^A$, $pid = \rho(\varepsilon(self))$ where $self \in Var_{PId}$, and $\perp_{pth}$ denotes the undefined path.

$$\frac{(\sigma,\sigma') \xrightarrow{receive(sig)}_I (\sigma'',\sigma''') \quad \Theta \xrightarrow{expired(si)}_T \Theta'}{ms \vdash \langle \pi::P_{\varepsilon,\rho}, (\sigma,\sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi::P_{\varepsilon,\rho}, (\sigma'',\sigma'''), \Theta' \rangle} \quad \|now\|_{\varepsilon,\rho} < time$$

A transition inferred from this rule models the behaviour when an SDL active timer expires and a signal instance is put into the input port or when a TTCN running timer expires and is added to the time-out list.

5.6.3.3 Inference rule Proc 3

A process instance performs an internal event when a timer is started.

Let $si = \|st\|_{ms,\varepsilon,\rho}$ where $si \in SigTerm^A$ and $st \in SigTerm(Var)$, and let $ti = \|dt\|_{ms,\varepsilon,\rho}$ where $ti \in Time^A$ and $dt \in Term_{Time}(Var)$, and $active_{si} \in Var_{Boolean}$

$$\frac{P \xrightarrow{set(dt,st)}_B P' \quad (\sigma,\sigma') \xrightarrow{filter(si)}_I (\sigma'',\sigma''') \quad \Theta \xrightarrow{tstart(ti,si)}_T \Theta'}{ms \vdash \langle \pi::P_{\varepsilon,\rho}, (\sigma,\sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi::P'_{\varepsilon,\rho'}, (\sigma'',\sigma'''), \Theta' \rangle} \quad \|now\|_{\varepsilon,\rho} = time$$

where $\rho' = \rho[\varepsilon(active_{si}) \mapsto true]$.

A transition inferred from this rule models the behaviour when an SDL process instance sets a timer or a TTCN tester *STARTs* a timer.

5.6.3.4 Inference rule Proc 4

A process instance performs an internal event when a timer is reset.

Let $si = \|st\|_{ms,\varepsilon,\rho}$ where $si \in SigTerm^A$ and $st \in SigTerm(Var)$ and let $active_{si} \in Var_{Boolean}$

$$\frac{P \xrightarrow{reset(st)}_B P' \quad (\sigma,\sigma') \xrightarrow{filter(si)}_I (\sigma'',\sigma''') \quad \Theta \xrightarrow{tstop(si)}_T \Theta'}{ms \vdash \langle \pi::P_{\varepsilon,\rho}, (\sigma,\sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi::P'_{\varepsilon,\rho'}, (\sigma'',\sigma'''), \Theta' \rangle} \quad \|now\|_{\varepsilon,\rho} = time$$

where $\rho' = \rho[\varepsilon(active_{si}) \mapsto false]$.

A transition inferred from this rule models the behaviour when an SDL process instance performs a reset of a timer, or a TTCN tester *CANCELs* a timer.

5.6.3.5 Inference rule Proc 5

The process instance performs an internal event when the elapsed time of a running timer is retrieved.

Let $x \in Var_{Time}$, $t \in Time^A$, and $si = \|st\|_{ms,\varepsilon,\rho}$ where $si \in SigTerm^A$ and $st \in SigTerm(Var)$

$$\frac{P \xrightarrow{read(x,st)}_B P' \quad \Theta \xrightarrow{request(t,si)}_B \Theta'}{ms \vdash \langle \pi::P_{\varepsilon,\rho}, (\sigma,\sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi::P'_{\varepsilon,\rho'}, (\sigma,\sigma'), \Theta' \rangle} \quad \|now\|_{\varepsilon,\rho} = time$$

where $\rho' = \rho[\varepsilon(x) \mapsto t]$.

A transition inferred from this rule models the behaviour when a TTCN tester *READs* a timer value.

5.6.3.6 Inference rule Proc 6

A process instance can always receive a signal that is addressed to the process instance. The receive event is an observable event.

Let $sig \in Sig_{\perp}$ and $self \in Var_{PId}$

$$\frac{(\sigma,\sigma') \xrightarrow{receive(sig)}_{I} (\sigma'',\sigma''')}{ms \vdash \langle \pi::P_{\varepsilon,\rho},(\sigma,\sigma'),\Theta\rangle \xrightarrow{receive(sig)}_{PI} \langle \pi::P_{\varepsilon,\rho},(\sigma'',\sigma'''),\Theta\rangle} \quad \begin{array}{l} \|now\|_{\varepsilon,\rho} = time \\ To(sig) = \rho(\varepsilon\,(self)) \end{array}$$

A transition inferred from this rule models the behaviour when an SDL process instance receives a signal instance, or when a TTCN tester receives a message.

5.6.3.7 Inference rule Proc 7

A process instance performs an internal event when a signal is consumed from the input port process. The condition for this event to occur is that the interpretation of the received signal and the input signal corresponds.

Let $st = op'(x_1,\ldots,x_m) \in SigTerm_{ss}(Var)$ where $op':s_1,\ldots,s_m \to ss \in SigOp$, $s_i \in Sort, i = 1,\ldots,m$, $ss \in SigSort$ and $x_i \in Var_{s_i}, i = 1,\ldots,m$. Let $sender \in Var_{PId}$, $active_{si} \in Var_{Boolean}$ and $sig \in Sig$

$$\frac{P \xrightarrow{input(st)}_{B} P' \quad (\sigma,\sigma') \xrightarrow{deliver(sig)}_{I} (\sigma'',\sigma''')}{ms \vdash \langle \pi::P_{\varepsilon,\rho},(\sigma,\sigma'),\Theta\rangle \xrightarrow{\tau}_{PI} \langle \pi::P'_{\varepsilon,\rho'},(\sigma'',\sigma'''),\Theta\rangle} \quad C$$

$$C \equiv \begin{array}{l} \|now\|_{\varepsilon,\rho} = time \\ \exists\, t_i \in Term_{s_i}(Var), i = 1,\ldots,m : \|op'(t_1,\ldots,t_m)\|_{ms,\varepsilon,\rho} = Intr(sig) \end{array}$$

where $\rho' = \rho\begin{bmatrix} \varepsilon(sender) \mapsto From(sig) \\ \varepsilon(x_i) \mapsto \|t_i\|, i = 1,\ldots,m \\ \varepsilon(active_{si}) \mapsto false \end{bmatrix}$

A transition inferred from this rule models the behaviour when an SDL process instance performs an input.

5.6.3.8 Inference rule Proc 8

A process instance performs an internal event when a signal is consumed. The conditions for the event to take place are that the signal instance to be consumed from the input port process and the signal term in the basic process corresponds w.r.t. every requirement specified for each parameter and the Boolean guard shall evaluate to *true.*

Let $st = op'(t_1,\ldots,t_m) \in SigTerm_{ss}(Var)$ where $op':s_1,\ldots,s_m \to ss \in SigOp$, $s_i \in Sort, i = 1,\ldots,m$, $ss \in SigSort$, $t_i \in Term_{s_i}(Var), i = 1,\ldots,m$, $sig = (si, pid, pid', pth) \in Sig$ where $si \in SigTerm^A$, $pid, pid' \in PId$, $pth \in PathId$, and let $sender \in Var_{PId}$, $active_{si} \in Var_{Boolean}$, $bt \in Term_{Boolean}(Var)$ $x \in Var_{ss}$

$$\frac{P \xrightarrow{input(st)\ via\ pth[bt]}_{B} P' \quad (\sigma,\sigma') \xrightarrow{deliver(sig,pth)}_{I} (\sigma'',\sigma''')}{ms \vdash \langle \pi::P_{\varepsilon,\rho},(\sigma,\sigma'),\Theta\rangle \xrightarrow{\tau}_{PI} \langle \pi::P'_{\varepsilon,\rho'},(\sigma'',\sigma'''),\Theta\rangle} \quad \begin{array}{l} \|now\|_{\varepsilon,\rho} = time \\ \|bt\|_{ms,\varepsilon,\rho'} = true \\ si = \|op'(t_1,\ldots,t_m)\|_{ms,\varepsilon,\rho} \end{array}$$

where $\rho' = \rho\begin{bmatrix} \varepsilon(sender) \mapsto From(sig) \\ \varepsilon(x) \mapsto si \\ \varepsilon(active_{si}) \mapsto false \end{bmatrix}$

A transition inferred from this rule models the behaviour when the TTCN tester performs a receive event guarded by a Boolean expression. The constraint is coded in the signal term of the basic process.

5.6.3.9 Inference rule Proc 9

A process instance performs an internal event when a signal from a specified path is delivered by the input port process, provided an optional Boolean term evaluates to true.

Let $sig = (si, pid, pid', pth) \in Sig_\perp$ where $si \in SigTerm^A$ or $si = \perp_{st}$, $pid, pid' \in PId$, $pth \in PathId$ and $bt \in Term_{Boolean}(Var)$

$$\frac{P \xrightarrow{otherwise\ pth\ [bt]}_B P' \quad (\sigma, \sigma') \xrightarrow{deliver(sig, pth)}_I (\sigma'', \sigma''')}{ms \vdash \langle \pi :: P_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi :: P'_{\varepsilon,\rho}, (\sigma'', \sigma'''), \Theta \rangle} \quad \begin{array}{l} \|now\|_{\varepsilon,\rho} = time \\ \|bt\|_{ms,\varepsilon,\rho} = true \end{array}$$

A transition inferred from this rule models the behaviour when a TTCN tester performs an otherwise event guarded by a Boolean expression.

5.6.3.10 Inference rule Proc 10

A process instance performs an internal event when a time-out signal is requested.

Let $st = op'(t_1, \ldots, t_m) \in SigTerm_{ss}(Var)$ with $op' : s_1, \ldots, s_m \to ss \in SigOp$, $s_i \in Sort, i = 1, \ldots, m$, $ss \in SigSort$, $t_i \in Term_{s_i}(Var), i = 1, \ldots, m$, $bt \in Term_{Boolean}(Var)$. Let $sig \in Sig$ and $sender \in Var_{PId}$, $active_{si} \in Var_{Boolean}$

$$\frac{P \xrightarrow{input(st)\ [bt]}_B P' \quad (\sigma, \sigma') \xrightarrow{timeout(sig)}_I (\sigma'', \sigma''')}{ms \vdash \langle \pi :: P_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi :: P'_{\varepsilon,\rho'}, (\sigma'', \sigma'''), \Theta \rangle} \quad C$$

$$C \equiv \begin{array}{l} \|now\|_{\varepsilon,\rho} = time \\ \|bt\|_{ms,\varepsilon,\rho} = true \\ Intr(sig) = \|op'(t_1, \ldots, t_m)\|_{ms,\varepsilon,\rho} \end{array}$$

where $\rho' = \rho \begin{bmatrix} \varepsilon(sender) \mapsto From(sig) \\ \varepsilon(active_{si}) \mapsto false \end{bmatrix}$

A transition inferred from this rule models the behaviour when a TTCN tester performs a time-out event guarded by a Boolean expression.

5.6.3.11 Inference rule Proc 11

A process instance can perform an internal event when the basic process performs a save event and the input port process preserves the signal.

Let $ss \in SigSort$

$$\frac{P \xrightarrow{save(ss)}_B P' \quad (\sigma, \sigma') \xrightarrow{preserve(ss)}_I (\sigma'', \sigma''')}{ms \vdash \langle \pi :: P_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi :: P'_{\varepsilon,\rho}, (\sigma'', \sigma'''), \Theta \rangle} \quad \|now\|_{\varepsilon,\rho} = time$$

A transition inferred from this rule models the behaviour when an SDL process instance performs a save action.

5.6.3.12 Inference rule Proc 12

A process instance performs an internal event when the basic process performs an assignment event.

Let $x \in Var_s$, $t \in Term_s(Var)$ for some $s \in Sort$

$$\frac{P \xrightarrow{x := t}_B P'}{ms \vdash \langle \pi :: P_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi :: P'_{\varepsilon,\rho'}, (\sigma, \sigma'), \Theta \rangle} \quad \|now\|_{\varepsilon,\rho} = time$$

where $\rho' = \rho\left[\varepsilon(x) \mapsto \|t\|_{ms,\varepsilon,\rho}\right]$

A transition inferred from this rule models the behaviour when an SDL process instance performs an assignment action or a TTCN tester performs an assignment (from an assignment list).

5.6.3.13 Inference rule Proc 13

A process instance may perform an internal event when a Boolean term evaluates to *true*.

Let $bt \in Term_{Boolean}(Var)$

$$\frac{P \xrightarrow{[bt]}_B P'}{ms \vdash \langle \pi :: P_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi :: P'_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \rangle} \quad \begin{array}{l} \|now\|_{\varepsilon,\rho} = time \\ \|bt\|_{ms,\varepsilon,\rho} = true \end{array}$$

A transition inferred from this rule models the evaluation of a decision construct in SDL and the evaluation of a qualifier in TTCN except in the case of qualifiers used as guards in send, receive, otherwise, and time-out events.

5.6.3.14 Inference rule Proc 14

A process instance can perform an internal event and become the undefined process instance, denoted $\perp_{pi}$ when the basic process performs a *stop* event.

$$\frac{P \xrightarrow{stop}_B nil}{ms \vdash \langle \pi :: P_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \perp_{pi}} \quad \|now\|_{\varepsilon,\rho} = time$$

A transition inferred from this rule models the behaviour when an SDL process instance performs a stop action or the behaviour when a TTCN tester stops execution after assignment of a final verdict.

5.6.3.15 Inference rule Proc 15

A process instance can perform a create event when the basic process performs a create event and there are less than the declared maximum number of existing process instances with basic process X in the module storage *ms*.

Let $D(X) = (Q, (init, max), ((), (y_1, \dots, y_m), ((z_1, t_1), \dots, (z_n, t_n))))$ be the declaration of basic process X. Let $t_i' \in Term(Var), i = 1, \dots, m$ and $a_i \in A, i = 1, \dots, m$ where $a_i = \|t_i'\|_{ms,\varepsilon,\rho}$ and $y_i \in Var_s \Rightarrow t_i' \in Term_s(Var)$, $i = 1, \dots, m$ for $s \in Sort$

$$\frac{P \xrightarrow{create\ X(t_1', \dots, t_m')}_B P'}{ms \vdash \langle \pi :: P_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \rangle \xrightarrow{create\ X(a_1, \dots, a_m)}_{PI} \langle \pi :: P'_{\varepsilon,\rho'}, (\sigma, \sigma'), \Theta \rangle} \quad \begin{array}{l} \|now\|_{\varepsilon,\rho} = time \\ \#pri_X < max \end{array}$$

where $\rho' = \rho\left[\varepsilon(offspring) \mapsto unique!^A()\right]$ with $unique!^A()$ an operation generating a unique unused *pid* value and $pri_X = \left\{pid \in BPPID_X \middle| ms(pid) = p \Rightarrow p \neq \bot_{pi}\right\}$.

A transition inferred from this rule models the behaviour when an SDL process instance successfully creates another SDL process instance.

5.6.3.16 Inference rule Proc 16

A process instance can perform a create event when the basic process performs a create event and the declared maximum number of process instances with basic process X exists in the module storage *ms*.

Let $D(X) = (Q, (init, max), ((), (y_1, \ldots, y_m), ((z_1, t_1), \ldots, (z_n, t_n))))$ be the declaration of basic process X. Let $t_i' \in Term(Var), i = 1, \ldots, m$ and $a_i \in A, i = 1, \ldots, m$ with $a_i = \|t_i'\|_{ms, \varepsilon, \rho}$ and $y_i \in Var_s \Rightarrow t_i' \in Term_s(Var), i = 1, \ldots, m$ for $s \in Sort$

$$\frac{P \xrightarrow{create\ X(t_1', \ldots, t_m')}_B P'}{ms \vdash \langle \pi :: P_{\varepsilon, \rho}, (\sigma, \sigma'), \Theta \rangle \xrightarrow{create\ X(a_1, \ldots, a_m)}_{PI} \langle \pi :: P'_{\varepsilon, \rho'}, (\sigma, \sigma'), \Theta \rangle} \quad \begin{array}{l} \|now\|_{\varepsilon, \rho} = time \\ \#pri_X = max \end{array}$$

where $\rho' = \rho\left[\varepsilon(offspring) \mapsto Null^A()\right]$ with $Null^A()$ an operation generating a specific *pid* value and $pri_X = \left\{pid \in BPPID_X \middle| ms(pid) = p \Rightarrow p \neq \bot_{pi}\right\}$.

A transition inferred from this rule models the behaviour when an SDL process instance performs an unsuccessful attempt to create another SDL process instance.

5.6.3.17 Inference rule Proc 17

A process instance performs an internal event when the basic process performs a call event.

Let $D(X) = (Q, (0, \infty), ((x_1, \ldots, x_k), (y_1, \ldots, y_m), ((z_1, t_1), \ldots, (z_n, t_n))))$ be the declaration of procedure X. Let $x_i' \in Var, i = 1, \ldots, k$ and $t_i' \in Term(Var), i = 1, \ldots, m$ with $x_i \in Var_s \Rightarrow x_i' \in Var_s, i = 1, \ldots, k$ and $y_i \in Var_s \Rightarrow t_i \in Term_s(Var), i = 1, \ldots, m$ for $s \in Sort$

$$\frac{P \xrightarrow{call\ X(x_1', \ldots, x_k')(t_1', \ldots, t_m')}_B P'}{ms \vdash \langle \pi :: P_{\varepsilon, \rho}, (\sigma, \sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi :: P'_{\varepsilon, \rho} :: Q_{\varepsilon', \rho'}, (\sigma, \sigma'), \Theta \rangle} \quad \|now\|_{\varepsilon, \rho} = time$$

where

$$\varepsilon' = \varepsilon \begin{bmatrix} x_i \mapsto \varepsilon(x_i'), i = 1, \ldots, k \\ y_j \mapsto l_j', j = 1, \ldots, m \\ z_h \mapsto l_h, h = 1, \ldots, n \end{bmatrix}$$

$$\rho' = \rho \begin{bmatrix} l_j' \mapsto \|t_j'\|_{ms, \varepsilon, \rho}, j = 1, \ldots, m \\ l_h \mapsto \|t_h\|_{ms, \varepsilon, \rho}, h = 1, \ldots, n \end{bmatrix}$$

This transition causes the insertion of the called basic process as the new first element of the sequence of basic processes. The environment ε' for the called procedure maps call-by-reference parameters, call-by-value parameters, and local variables to locations. The storage ρ' maps locations assigned to call-by-value parameters and local variables to values.

A transition inferred from this rule models the behaviour when an SDL process instance calls a procedure.

5.6.3.18 Inference rule Proc 18

A process instance can perform an internal event when the basic process performs a return event.

$$\frac{P \xrightarrow{return}_B P'}{ms \vdash \langle \pi :: Q_{\varepsilon,\rho} :: P_{\varepsilon',\rho'}, (\sigma, \sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi :: Q_{\varepsilon,\rho'}, (\sigma, \sigma'), \Theta \rangle} \quad \|now\|_{\varepsilon,\rho} = time$$

The storage ρ' is preserved such that value assignments to call-by-reference parameters are preserved.

A transition inferred from this rule models the behaviour when an SDL process instance returns from a procedure.

5.6.3.19 Inference rule Proc 19

A process instance can send an implicit signal via a specified route when the basic process performs an output-via event. Sending an implicit signal is an observable event.

Let $st \in SigTerm(Var)$, $r \in Routes$, $sig \in Sig$ with $sig = (\|st\|_{ms,\varepsilon,\rho}, \rho(\varepsilon(self)), \bot_{pid}, \bot_{pth})$ and $self \in Var_{PId}$

$$\frac{P \xrightarrow{output(st)\ via\ r}_B P'}{ms \vdash \langle \pi :: P_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \rangle \xrightarrow{output(sig)\ via\ r}_{PI} \langle \pi :: P'_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \rangle} \quad \|now\|_{\varepsilon,\rho} = time$$

A transition inferred from this rule models the behaviour when an SDL process instance performs an output-via action or when a TTCN tester performs a send event.

5.6.3.20 Inference rule Proc 20

A process instance can send an explicit signal to a specified process instance when the basic process performs an output-to event. Sending an explicit signal is an observable event.

Let $st \in SigTerm(Var)$, $p \in Term_{PId}(Var)$, $sig \in Sig$ with $sig = (\|st\|_{ms,\varepsilon,\rho}, \rho(\varepsilon(self)), \|p\|_{ms,\varepsilon,\rho}, \bot_{pth})$ and $self \in Var_{PId}$

$$\frac{P \xrightarrow{output(st)\ to\ (p)}_B P'}{ms \vdash \langle \pi :: P_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \rangle \xrightarrow{output(sig)}_{PI} \langle \pi :: P'_{\varepsilon,\rho}, (\sigma, \sigma'), \Theta \rangle} \quad \|now\|_{\varepsilon,\rho} = time$$

A transition inferred from this rule models the behaviour when an SDL process instance performs an output-to action.

5.6.3.21 Inference rule Proc 21

A process instance can send an explicit signal to a specified process instance via a specific route when the basic process performs an output-to-via event. Sending an explicit signal is an observable event.

Let $st \in SigTerm(Var)$, $p \in Term_{PId}(Var)$, $r \in Routes$, $sig \in Sig$ with $sig = (\|st\|_{ms,\varepsilon,\rho}, \rho(\varepsilon(self)), \|p\|_{ms,\varepsilon,\rho}, \bot_{pth})$ and $self \in Var_{PId}$

$$\frac{P \xrightarrow{output(st)\ to\ (p)\ via\ r}_{B} P'}{ms \vdash \left\langle \pi::P_{\varepsilon,\rho}, (\sigma,\sigma'), \Theta \right\rangle \xrightarrow{output(sig)\ via\ r}_{PI} \left\langle \pi::P'_{\varepsilon,\rho}, (\sigma,\sigma'), \Theta \right\rangle} \quad \|now\|_{\varepsilon,\rho} = time$$

A transition inferred from this rule models the behaviour when an SDL process instance performs an output-to-via action. In CCITT Recommendation Z.100 [1] no requirement exists on the existence of a receiving process instance at time of sending.

5.7 Module process

A module consists of a number of process instances which perform events in an interleaved order and communicate asynchronously. A module is used to model the behaviour of a block in SDL or an abstract tester. In the case of TTCN there is always only one process instance, the TTCN tester, and there exists only one module, the abstract tester. The interleaving of events is used to model the concurrent behaviour of process instances in the module, and asynchronous communication is due to the fact that process instances do not consume received signals instantaneously.

5.7.1 States of a module process

The set of states of a module process is defined over the set of module storages as defined in subclause 5.6.3.

5.7.2 Events of a module process

The set of events a module may perform is defined by the set:

$$\varepsilon_M = Sig_{input} \uplus Sig_{output} \uplus \{\tau\}$$

where

$$Sig_{input} = (Sig_\perp \times PathId)$$
$$Sig_{output} = (Sig_\perp \times PathId)$$

An event $(sig, pth) \in Sig_{input}$ models that the signal *sig* may be sent to a module via the path *pth* and is denoted *input(sig) via pth*. An event $(sig, pth) \in Sig_{output}$ models that the signal *sig* may be sent from a module via the path *pth* and the event is denoted *output(sig) via pth*. The event τ denotes an event that is not observable in the environment of a module.

5.7.3 Semantics of a module process

The semantics of a module is given by a labelled transition system.

$$M = (ModStor, \varepsilon_M, \longrightarrow_M)$$

where *ModStor* is the set of states, ε_M is the set of events and $\longrightarrow_M$ is the transition relation associated with the set of inference rules given below. $ms \in ModStor$ shall be used to denote the state of a module, and each inference rule is of one of the following two forms:

(a) $$\frac{ms \vdash p \xrightarrow{e}_{PI} p'}{ms \xrightarrow{e'}_M ms'} \quad C$$

stating that $ms \xrightarrow{e'}_M ms'$ is a transition in M provided that condition *C* holds and $ms \vdash p \xrightarrow{e}_{PI} p'$ is a transition of process instance *p*, derivable from the module storage $ms \in ModStor_m$ of a module *m*. When this transition is performed, the module changes its state from *ms* to *ms'* while performing the event *e'*.

The second form of rule is used to model communication between two process instances residing inside the same module. Such rules are of the form:

(b) $$\frac{ms \vdash p \xrightarrow{e}_{PI} p' \quad ms' \vdash q \xrightarrow{e'}_{PI} q'}{ms \xrightarrow{e''}_M ms''} \quad C$$

where $ms, ms', ms'' \in ModStor_m$ are module storages of a module *m*, related such that $ms' = ms[pid_p \mapsto p']$ and $ms'' = ms'[pid_q \mapsto q']$ where pid_p and pid_q are process instance identifiers of *p* and *q* respectively. This rule states that $ms \xrightarrow{e''}_M ms''$ is a transition in M,

provided condition C holds, $ms \vdash p \xrightarrow{e}_{PI} p'$ and $ms' \vdash q \xrightarrow{e}_{PI} q'$ are transitions of process instances derivable from the module storages ms, ms' respectively. When these transitions are performed, the module m performs the event e'' changing its module storage from ms to ms''. ms' is an intermediate state of module m while performing the event e''. However, an observer of module process m does not notify this intermediate state, because the event e'' is considered to be atomic.

NOTE: An analogy to the world of micro-processors may be of help to understand the intuition behind the second form of rule. Assume that *load(ad, reg)* is an assembler operation in the micro-processor M, which loads the contents of the memory at address *ad* into the register *reg* of M. The load operation is defined as the two micro-code operations *enable(ad)*, which sets up the address bus, and the micro-code operation *read(reg, ad)*, which copies the contents of memory address *ad* into register *reg*. Before M has started to perform the assembly operation *load(ad, reg)*, it is in state *m*, and when M has completed the *load(ad, reg)* operation it is in state *m″*. Obviously, when M has performed the micro-code operation *enable(ad)* but before it has performed the micro-code operation *read(reg, ad)*, it is in a state *m′* which is not *m* and which is not *m″*. However, since it is not possible to interrupt M while performing the assembly operation *load*, the state *m′* is not possible to observe as a user of M. This could be modelled with the rule:

$$\frac{m \vdash p \xrightarrow{enable(ad)} p' \quad m' \vdash q \xrightarrow{read(reg,ad)} q'}{m \xrightarrow{load(ad,reg)} m''}$$

where p, p' are the states of the address bus, q, q' are the states of the register *reg*, and m, m' and m'' are states of M.

5.7.3.1 Inference rule M 1

A module performs an internal transition when communication of an explicit signal between two process instances (or from and to one process instance) within the module takes place. In order for the communication to be performed a route capable of conveying the signal has to exist. The transitions of the involved process instances are performed in different but related module storages.

Let $ms, ms', ms'' \in ModStor_m$ where $m \in ModId$, $X, Y \in BPId_m$, $pid_1 \in BPPId_X$ such that $ms(pid_1) = p_1$, and $pid_2 \in BPPId_Y$ such that $ms(pid_2) = p_2$ and $sig, sig' \in Sig$ such that $sig' \in Inst(sig)$

$$\frac{ms \vdash p_1 \xrightarrow{output(sig)}_{PI} p_1' \quad ms' \vdash p_2 \xrightarrow{receive(sig')}_{PI} p_2'}{ms \xrightarrow{\tau}_M ms''} \quad Intr(sig) \in r_m(X, Y)$$

where $ms' = ms[pid_1 \mapsto p_1']$ and $ms'' = ms'[pid_2 \mapsto p_2']$.

NOTE: If the communication involves only a single process instance $(X = Y)$ then $p_2 = p_1'$.

A transition inferred from this rule models the behaviour of an SDL block when a process instance inside the block sends a signal instance using output or output-to to another process instance inside the same block.

5.7.3.2 Inference rule M 2

A module performs an internal event when a process instance in the module performs an output of an explicit signal via a specified route but the specified process instance has ceased to exist. The result of the transition is that the signal is discarded.

Let $ms, ms' \in ModStor_m$, where $m \in ModId$, $X, Y \in BPId_m$, $pid \in BPPId_Y$ such that $ms(pid) = p$, and $(X, Y) \in Routes_m$. Let $sig \in Sig$ with $To(sig) = pid'$

$$\frac{ms \vdash p \xrightarrow{output(sig)\ via\ (X,Y)}_{PI} p'}{ms \xrightarrow{\tau}_{M} ms'} \quad \begin{array}{l} pid' \in BPPId_Y \\ ms(pid') = \bot_{pi} \end{array}$$

where $ms' = ms[pid \mapsto p']$.

A transition inferred from this rule models the behaviour of an SDL block when a signal instance is discarded inside the block.

5.7.3.3 Inference rule M 3

A module performs an internal event when a process instance performs an output of an explicit signal for which no receiving process instance exists.

Let $ms, ms' \in ModStor_m$, where $m \in ModId$, $ms(pid) = p$ and let $sig \in Sig$ with $To(sig) = pid'$

$$\frac{ms \vdash p \xrightarrow{output(sig)}_{PI} p'}{ms \xrightarrow{\tau}_{M} ms'} \quad To(sig) = pid' \Rightarrow ms(pid') = \bot_{pi}$$

where $ms' = ms[pid \mapsto p']$.

A transition inferred from this rule models the behaviour of an SDL block when a signal instance is discarded inside the block due to the non-existence of the specified receiving process instance.

5.7.3.4 Inference rule M 4

A module performs an internal event when two process instances communicate via a specified route.

Let $ms, ms', ms'' \in ModStor_m$ where $m \in ModId$, $X, Y \in BPId_m$, $pid_1 \in BPPId_X$, $pid_2 \in BPPId_Y$ such that $ms(pid_1) = p_1$ and $ms(pid_2) = p_2$, and $sig, sig' \in Sig$ such that $sig' \in Inst(sig)$

$$\frac{ms \vdash p_1 \xrightarrow{output(sig)\ via(X,Y)}_{PI} p_1' \quad ms' \vdash p_2 \xrightarrow{receive(sig')}_{PI} p_2'}{ms \xrightarrow{\tau}_{M} ms''} \quad Intr(sig) \in r_m(X,Y)$$

where $ms' = ms[pid_1 \mapsto p_1']$ and $ms'' = ms'[pid_2 \mapsto p_2']$.

A transition inferred from this rule models the behaviour of an SDL block when a process instance inside the block sends a signal instance using output-via or output-to-via to another process instance inside the block.

5.7.3.5 Inference rule M 5

A module performs an internal event when a process instance performs an output of an implicit signal via a specified route in the module and there exists no process instance of the specified basic process that can receive the signal.

Let $ms, ms' \in ModStor_m$, where $m \in ModId$, $X, Y \in BPId_m$, $pid \in BPPId_X$ such that $ms(pid) = p$ and let $sig \in Sig$ with $To(sig) = \bot_{pi}$

$$\frac{ms \vdash p \xrightarrow{output(sig)\ via\ (X,Y)}_{PI} p'}{ms \xrightarrow{\tau}_{M} ms'} \quad \begin{array}{l} Intr(sig) \in r_m(X,Y) \\ \forall pid' \in BPPId_Y : ms(pid') = q \Rightarrow q = \bot_{pi} \end{array}$$

where $ms' = ms[pid \mapsto p']$.

A transition inferred from this rule models the behaviour of an SDL block when an implicit signal is discarded for which no receiving process instance exists within the SDL block .

5.7.3.6 Inference rule M 6

A module performs an internal event when a process instance sends an explicit signal to a process instance in the same module via a specific route, but the route does not lead to the specified process instance. The result is that the signal is discarded.

Let $ms, ms' \in ModStor_m$, where $m \in ModId$, $X, Y \in BPId_m$, $pid \in BPPId_X$ such that $ms(pid) = p$ and let $sig \in Sig$ with $To(sig) = pid'$

$$\frac{ms \vdash p \xrightarrow{output(sig)\ via\ (X,Y)}_{PI} p'}{ms \xrightarrow{\tau}_{M} ms'} \quad \begin{array}{l} Intr(sig) \in r_m(X,Y) \\ pid' \notin BPPId_Y \\ pid' \in ModPId_m \end{array}$$

where $ms' = ms[pid \mapsto p']$.

A transition inferred from this rule models the behaviour of an SDL block when an explicit signal instance is discarded inside the block due to sending the signal via a specific signal route not leading to the specified process instance.

5.7.3.7 Inference rule M 7

A module performs an output event via a specific path when a process instance sends an explicit signal to a process instance not within the module. Another condition for this event to take place is that the signal can be conveyed from the process instance to an associated path capable of conveying the signal.

Let $ms, ms' \in ModStor_m$, where $m \in ModId$, $X \in BPId_m$, $pid \in BPPId_X$ such that $ms(pid) = p$, $pth \in PathId$ and $sig \in Sig$

$$\frac{ms \vdash p \xrightarrow{output(sig)}_{PI} p'}{ms \xrightarrow{output(sig)\ via\ pth}_{M} ms'} \quad \begin{array}{l} Intr(sig) \in r_m(X,pth) \\ To(sig) \notin ModPId_m \end{array}$$

where $ms' = ms[pid \mapsto p']$.

NOTE: There may be several paths capable of conveying signal *sig*, *i.e.* $\#\{pth \mid Intr(sig) \in r_m(x,pth)\} > 1$. In this case, the rule states that the path is arbitrarily chosen.

A transition inferred from this rule models the behaviour of an SDL block when a process instance inside the block sends a signal instance out of the block using output or output-to.

5.7.3.8 Inference rule M 8

A module performs an output via a specific path when a process instance within the module performs an output via the path. The condition for the event to take place is that the path specified can convey the signal to be sent.

Let $ms, ms' \in ModStor_m$, where $m \in ModId$, $X \in BPId_m$, $pid \in BPPId_X$ such that $ms(pid) = p$, $pth \in PathId$ such that $(X, pth) \in Routes$ and let $sig \in Sig$

$$\frac{ms \vdash p \xrightarrow{output(sig)\ via\ (X, pth)}_{PI} p'}{ms \xrightarrow{output(sig)\ via\ pth}_{M} ms'} \quad Intr(sig) \in r_m(X, pth)$$

where $ms' = ms[pid \mapsto p']$.

A transition inferred from this rule models the behaviour of an SDL block when a process instance inside the block sends a signal instance out of the block using output-via or output-to-via. In the case of TTCN, the transition models the behaviour of an abstract tester when a TTCN tester performs a send event.

5.7.3.9 Inference rule M 9

A module may receive a signal from a path if there is process instance capable of receiving the signal and there exists a route from the path to the process instance.

Let $ms, ms' \in ModStor_m$, where $m \in ModId$, $X \in BPId_m$, $pid \in BPPId_X$ such that $ms(pid) = p$, let $sig \in Sig_\perp$ and $pth \in PathId$

$$\frac{ms \vdash p \xrightarrow{receive(sig)}_{PI} p'}{ms \xrightarrow{input(sig)\ via\ pth}_{M} ms'} \quad Intr(sig) \in r_m(pth, X)$$

where $ms' = ms[pid \mapsto p']$.

NOTE: The signal *sig* may be an undefined signal, which corresponds to a TTCN "unforeseen message".

In the case of SDL, a transition inferred from this rule models the behaviour of an SDL block when a process instance inside the block receives a signal instance from the environment of the block. In the case of TTCN, the transition models the behaviour of an abstract tester when a TTCN tester receives a message.

5.7.3.10 Inference rule M 10

When a process instance in a module performs an internal event so does the module.

Let $ms, ms' \in ModStor_m$, where $m \in ModId$, $X \in BPId_m$, $pid \in BPPId_X$ such that $ms(pid) = p$

$$\frac{ms \vdash p \xrightarrow{\tau}_{PI} p'}{ms \xrightarrow{\tau}_{M} ms'}$$

where $ms' = ms[pid \mapsto p']$.

A transition inferred from this rule models the behaviour of an SDL block or an abstract tester when an internal event is performed by a process instance or a TTCN tester.

5.7.3.11 Inference rule M 11

A process instance in a module may create a new process instance defined over a basic process which is defined in the module. When the create action is performed the formal parameters of the created process instance are substituted by the actual parameters. Furthermore, local variables are created and initialised. The condition for the create to be performed is that the number of existing process instances over the same basic process in the module is less than the maximal number that may exist simultaneously.

Let $ms, ms', ms'' \in ModStor_m$, where $m \in ModId$, $X, Y \in BPId_m$, $pid \in BPPId_Y$ such that $ms(pid) = p$. If the declaration D of a basic process identified by X is assumed to be $D(X) = (Q, (init, max), (), (y_1, \ldots, y_m), ((z_1, t_1), \ldots, (z_n, t_n)))$

$$\frac{ms \vdash p \xrightarrow{create\ X(a_1, \ldots, a_m)}_{PI} p'}{ms \xrightarrow{\tau}_M ms''} \quad \# pri_X < max$$

where $ms'' = ms\left[ms'_{sto}(pid)\,(ms'_{env}(pid)(offspring)) \mapsto \left(Q_{\varepsilon', \rho'}, \langle\rangle, \langle\rangle\right)\right]$, $ms' = ms[pid \mapsto p']$ and $pri_X = \left\{pid' \in BPPId_X \mid ms(pid') = q \Rightarrow q \neq \perp_{pi}\right\}$.

The intuition with the expression $ms'' = ms\left[ms'_{sto}(pid)\,(ms'_{env}(pid)(offspring)) \mapsto \left(Q_{\varepsilon', \rho'}, \langle\rangle, \langle\rangle\right)\right]$ is that the process instance identified by the value of *offspring* is replaced by the newly created process instance $\left(Q_{\varepsilon', \rho'}, \langle\rangle, \langle\rangle\right) \in Proc$. The storage ρ' and environment ε' mappings associated with the newly created process instance $\left(Q_{\varepsilon', \rho'}, \langle\rangle, \langle\rangle\right)$ are defined as

$$\varepsilon' = \varepsilon_\perp \begin{bmatrix} self \mapsto l_1 \\ parent \mapsto l_2 \\ offspring \mapsto l_3 \\ sender \mapsto l_4 \\ now \mapsto l_5 \\ active_{si} \mapsto l_{si}, si \in SigTerm^A \\ y_i \mapsto l'_i, i = 1, \ldots, n \\ z_j \mapsto l''_j, j = 1, \ldots, m \end{bmatrix}$$

$$\rho' = \rho_\perp \begin{bmatrix} l_1 \mapsto ms'_{sto}(pid)(ms'_{env}(pid)(offspring)) \\ l_2 \mapsto ms'_{sto}(pid)(ms'_{env}(pid)(self)) \\ l_5 \mapsto time \\ l_{si} \mapsto false, si \in SigTerm^A \\ l'_i \mapsto a_i, i = 1, \ldots, n \\ l''_j \mapsto \|t_j\|, j = 1, \ldots, m \end{bmatrix}$$

where $\varepsilon_\perp$ denotes the undefined environment and $\rho_\perp$ denotes the undefined storage mapping.

A transition inferred from this rule models the behaviour of an SDL block when a process instance inside the block performs a create action, and the create action is successful.

5.7.3.12 Inference rule M 12

A module performs an internal event when a process instance within the module performs a create action but fails to create a new process instance because the number of currently existing process instances of the specified basic process equals the maximal number that may exist simultaneously.

Let $ms, ms' \in ModStor_m$, where $m \in ModId$, $X, Y \in BPId_m$, $pid \in BPPId_Y$ such that $ms(pid) = p$. If the declaration D of a basic process identified by X is assumed to be $D(X) = (Q, (init, max), (), (y_1, \ldots, y_m), ((z_1, t_1), \ldots, (z_n, t_n)))$

$$\frac{ms \vdash p \xrightarrow{create\ X(a_1, \ldots, a_m)}_{PI} p'}{ms \xrightarrow{\tau}_M ms'} \quad \# pri_X = max$$

where $ms' = ms[pid \mapsto p']$ and $pri_X = \{pid' \in BPPId_X \mid ms(pid') = q \Rightarrow q \neq \bot_{pi}\}$.

A transition inferred from this rule models the behaviour of an SDL block when a process instance inside the block performs a create action and the create action is unsuccessful.

5.8 Path process

A path process is a means to inter-connect two module processes or a module process with the environment. Two path processes, one for each directi on of communication, models the behaviour of a bi-directional SDL channel or a TTCN PCO. A uni-directional SDL channel is represented by a single path process.

5.8.1 States of a path process

A state of a path process is an element of the set:

$$PathProc = \{(pth, \theta) \mid pth \in PathId, \theta \in Sig_\bot^*\}$$

The intuition is that θ contains the signals that are currently conveyed by the path process identified by *pth*. The reason that a path process may convey an undefined signal is that TTCN PCOs may convey unforeseen messages.

5.8.2 Events of a path process

The set of events a path process can perform is defined by:

$$\varepsilon_{Path} = Sig_{input} \uplus Sig_{output}$$

where

$$Sig_{input} = Sig_{output} = Sig_\bot$$

An event $sig \in Sig_{input}$ is denoted by *input(sig)*. *input(sig)* models the reception of a signal from a module process or the environment. An event $sig \in Sig_{output}$ is denoted by *output(sig)*. *output(sig)* models the sending of a signal to a module process or the environment.

5.8.3 Semantics of a path process

The semantics of a path process is given by a labelled transition system:

$$Path = (PathProc, \varepsilon_{Path}, \longrightarrow_{Path})$$

where $\longrightarrow_{Path}$ is the transition relation associated with the set of inference rules given below. Each inference rule is of the form $\frac{}{t}$ C stating that t is a transition in Path provided that condition C holds.

Throughout the rest of this subclause let $pth \in PathId$ and $\theta \in Sig_\bot^*$.

5.8.3.1 Inference rule Path 1

A path process can always receive a signal.

Let $sig = (si, pid, pid', \bot_{pth})$ with $si \in SigTerm^A$, $pid, pid' \in PId$, and $sig' = (si, pid, pid', pth)$

$$\frac{}{(pth,\theta) \xrightarrow{input(sig)}_{Path} (pth, sig'::\theta)} \quad (Spec \in Spec_{SDL}) \Rightarrow (Intr(sig) \neq \perp_{st})$$

The side-condition of the inference rule states that in case of an SDL specification the received signal *sig* cannot be an undefined signal.

The result of the receive event is that the signal is appended as the last element to the sequence of signals currently conveyed by the path process and that the path process identifier is made part of the signal.

A transition inferred from this rule models the behaviour when an SDL channel receives a signal instance or when a TTCN PCO receives an ASP or PDU.

5.8.3.2 Inference rule Path 2

A path process can always emit a signal provided that the sequence of conveyed signals is not empty.

Let $sig = (si, pid, pid', pth)$ with $si \in SigTerm^A$ and $pid, pid' \in PId$

$$\frac{}{(pth, \theta::sig) \xrightarrow{output(sig)}_{Path} (pth, \theta)}$$

A transition inferred from this rule models the behaviour when an SDL channel delivers a signal instance or when a TTCN PCO delivers an ASP or PDU.

5.9 System

The system level is the top level of the CSR and is used to model an SDL system and a TTCN test system.

5.9.1 States of system

A state of a system is an element of the set:

$$Sys = Pa_{pth_1} \times \ldots \times Pa_{pth_k} \times M_{m_1} \times \ldots \times M_{m_n}$$

where $Pa_{pth_i} \in PathProc$ for $pth_i \in PathId, i = 1, \ldots, k$, and $M_{m_j} \in ModStor$ for $m_j \in ModId, j = 1, \ldots, n$. That is, a state of a system is a tuple of the states of the path processes and the states of the module processes in the system. Each state of a path process is indexed with its path identifier and each state of a module process is indexed with its module identifier. In the case of TTCN, there is always only one module.

The following notation is introduced for the description of a system state:

$s[pth \mapsto Pa, m \mapsto M]$ denotes the system state which is equal to s except for the state of the path indexed by pth which has been replaced by the state Pa and the state of the module indexed by m which has been replaced by M. Either of $pth \mapsto Pa$ or $m \mapsto M$ may be omitted.

5.9.2 Events of a system

The set of events a system may perform is defined by:

$$\varepsilon_S = Sig_{input} \uplus Sig_{output} \uplus \{\tau\}$$

where

$$Sig_{input} = Sig_{\perp}$$

$Sig_{output} = Sig$

An event $sig \in Sig_{input}$ models the reception of a signal in the system from the environment. This event is denoted $input(sig)$. An event $sig \in Sig_{output}$ models the sending of a signal from the system to the environment and is denoted $output(sig)$. Events not visible in the environment are denoted by τ.

5.9.3 Semantics of a system

The semantics of a system is given by a labelled transition system:

$$S = \left(Sys, \varepsilon_S, \longrightarrow_S, s_0\right)$$

where Sys is the set of states of a system, ε_S is the set of events and $\longrightarrow_S$ is the transition relation associated with the set of inference rules given below, and s_0 is the initial system state. Each rule is of the form:

$$\frac{Pa_{pth} \xrightarrow{e_p}_{Path} Pa'_{pth} \quad M_m \xrightarrow{e_m}_{M} M'_m}{s \xrightarrow{e_s}_{S} s'} \quad C$$

where either of the premises may be omitted. The rule states that $s \xrightarrow{e_s}_{S} s'$ is a transition of the system if $Pa_{pth} \xrightarrow{e_p}_{Path} Pa'_{pth}$ is a transition of the path process identified by $pth \in PathId$, and $M_m \xrightarrow{e_m}_{M} M'_m$ is a transition of the module process identified by $m \in ModId$ and the condition C holds.

5.9.3.1 Inference rule S 1

A system performs an internal event whenever a module process within the system performs an internal event.

Let $M_m, M'_m \in ModStor$ where $m \in ModId$ and $s, s' \in Sys$

$$\frac{M_m \xrightarrow{\tau}_{M} M'_m}{s \xrightarrow{\tau}_{S} s'}$$

where $s' = s\left[m \mapsto M'_m\right]$.

A transition inferred from this rule models the behaviour of an SDL system when a block within the system performs an internal event, or a TTCN system when the abstract tester performs an internal event.

5.9.3.2 Inference rule S 2

A system performs an internal event when a signal is sent from a module process to a path process.

Let $Pa_{pth}, Pa'_{pth} \in PathProc$ where $pth \in PathId$, $M_m, M'_m \in ModStor$ where $m \in ModId$, $s, s' \in Sys$ and let $sig \in Sig$

$$\frac{Pa_{pth} \xrightarrow{input(sig)}_{Path} Pa'_{pth} \quad M_m \xrightarrow{output(sig)\ via\ pth}_{M} M'_m}{s \xrightarrow{\tau}_{S} s'}$$

where $s' = s\left[pth \mapsto Pa'_{pth}, m \mapsto M'_m\right]$.

A transition inferred from this rule models the behaviour of an SDL system when a signal instance is delivered from a block to a channel, or when a message in a TTCN system is submitted to a PCO from the abstract tester.

5.9.3.3 Inference rule S 3

A system performs an internal event when a signal is delivered from a path process to a module process.

Let $Pa_{pth}, Pa'_{pth} \in PathProc$ where $pth \in PathId$, $M_m, M'_m \in ModStor$ where $m \in ModId$, $s, s' \in Sys$ and let $sig \in Sig_{\perp}$

$$\frac{Pa_{pth} \xrightarrow{output(sig)}_{Path} Pa'_{pth} \quad M_m \xrightarrow{input(sig)\ via\ pth}_{M} M'_m}{s \xrightarrow{\tau}_{S} s'}$$

where $s' = s[pth \mapsto Pa'_{pth}, m \mapsto M'_m]$.

A transition inferred from this rule models the behaviour of an SDL system when a signal instance is delivered to a block from a channel, or when a message in a TTCN system is delivered to the abstract tester from a PCO.

5.9.3.4 Inference rule S 4

A system performs an internal event discarding a signal when an explicit signal cannot be received by any module process in the system and the specified receiving process instance is neither a process instance in the environment.

Let $Pa_{pth}, Pa'_{pth} \in PathProc$ where $pth \in PathId$, $s, s' \in Sys$, and $sig \in Sig$

$$\frac{Pa_{pth} \xrightarrow{output(sig)}_{Path} Pa'_{pth}}{s \xrightarrow{\tau}_{S} s'} \quad \begin{array}{l} To(sig) \notin ModPId_{env} \\ \forall m \in ModId: M_m \xrightarrow{input(sig)\ via\ pth}\!\!\!\!\!\!/\;\;_{M} \end{array}$$

where $s' = s[pth \mapsto Pa'_{pth}]$.

The condition $\forall m \in ModId: M_m \xrightarrow{input(sig)\ via\ pth}\!\!\!\!\!\!/\;\;_{M}$ states that there is no module process in the system that may receive signal *sig*. The result of the event is that signal *sig* is discarded.

A transition inferred from this rule models the behaviour of an SDL system when a signal instance is discarded inside the system.

NOTE: The discarding of signals for which no receiving process exists reflects the semantics of SDL in CCITT Recommendation Z.100 [1].

5.9.3.5 Inference rule S 5

A system performs an input action when a path process receives a signal from the environment. The condition for the event to occur is that the signal is sent from a process instance in the environment and that there exists a route leading from an environment process to the path process.

Let $Pa_{pth}, Pa'_{pth} \in PathProc$ where $pth \in PathId$, $s, s' \in Sys$ and let $sig \in Sig_{\perp}$

$$\frac{Pa_{pth} \xrightarrow{input(sig)}_{Path} Pa'_{pth}}{s \xrightarrow{input(sig)}_{S} s'} \quad \begin{array}{l} From(sig) \in ModPId_{env} \\ (FROMENV, pth) \in Routes \end{array}$$

where $s' = s[pth \mapsto Pa'_{pth}]$.

NOTE: An undefined signal $Intr(sig) = \bot_{st}$ may be sent from the environment.

A transition inferred from this rule models the behaviour of an SDL system when signal instance is sent to the system from the environment, or when in a TTCN system a (unforeseen) message is sent to the TTCN system from the environment.

5.9.3.6 Inference rule S 6

A system performs an output action when a signal is delivered from a path process to the environment. The condition for this event to occur is that the specified path is connected to the environment.

Let $Pa_{pth}, Pa'_{pth} \in PathProc$ where $pth \in PathId$, $s, s' \in Sys$ and let $sig \in Sig$

$$\frac{Pa_{pth} \xrightarrow{output(sig)}_{Path} Pa'_{pth}}{s \xrightarrow{output(sig)}_{S} s'} \quad (pth, TOENV) \in Routes$$

where $s' = s[pth \mapsto Pa'_{pth}]$.

A transition inferred from this rule models the behaviour of an SDL system when a signal instance is delivered to the environment from a channel, or when a message in a TTCN system is delivered to the environment from a PCO.

5.9.4 Derivation of the initial system state

In order to derive the behaviour of a system, the initial system state shall be established. The initial system state is a state where the initial number of process instances for each basic process exist, the sequence of timers of each timer process is empty, the signal queues of each input port process are empty, all variables are assigned their initial value and the sequence of signals of each path process is empty.

Annex A: Introduction to the CSR

In this annex basic information on the modelling technique of the CSR is provided. The intention is to explain, informally, how the rules that make up the model are to be interpreted and how the compositional structure of the model supports reasoning about the behaviour of SDL systems and TTCN test cases. This means that only the first Clause of the annex is self-contained. However, the explanation of the derivation of transitions and the use of the CSR to compare behaviour of specifications to some extent relies on information provided in the CSR definition (Clause 5).

The annex is organised as follows. Clause A.1 presents the basic modelling concept *Labelled Transition Systems*. In the following Clauses examples are used to illustrate how the execution of an SDL specification or TTCN test case is modelled in the CSR and how the compositional structure of the model supports comparison of system behaviour given different interpretations of behaviour, i.e. observable behaviour or internal behaviour. Before reading Clauses A.2 and A.3 it is recommended to read the CSR definition. Furthermore, the presentation is to some extent relying on knowledge of the conceptual models of SDL and TTCN defined in Clause 4.

A.1 Labelled Transition Systems

The concept of L*abelled Transition Systems (LTS)* is the basis on which the CSR is defined. A LTS consists of a set of states, a set of actions, a transition relation and an initial state. This can be written as:

$$LTS = (S, A, T, s_0)$$

where S is a set of *states*, A is a set of *actions* (or events), T is a *transition relation*, and s_0 is the *initial state*. The transition relation defines which pairs of states are related by an action. Thus the transition relation is defined as:

$$T \subseteq S \times A \times S.$$

If s_1 and s_2 denote states in S and a is an action in A then $(s_1, a, s_2) \in T$ means that if in state s_1 an action a is performed the system evolve to state s_2. A tuple (s_1, a, s_2) is termed *transition*. The terms *pre-* and *post-state* are used later when referring to the initial and terminal state of a transition. An alternative notation to denote (s_1, a, s_2) is $s_1 \xrightarrow{a} s_2$. The latter notation is used in the CSR. The notation $s_1 \xrightarrow{a}$ means that there exists at least one post-state such that if in pre-state s_1 action a is performed the system can transform to a post-state.

An LTS can be used to model the behaviour of systems when the set of states, actions and elements of the transition relation are defined. The basic assumption is that every state of a system is represented as a state in the set of states of the LTS. From this assumption and the definition of the transition relation, it follows that only a single transition is applied even if more than one transition are possible. The behaviour of a system is defined as the set of sequences of actions starting from an initial state.

EXAMPLE 1: Let a system be represented as the LTS (S,A,T,s_1) where $S = \{s_1,\ldots,s_5\}$, $A = \{a,b\}$, $T = \{(s_1,a,s_2),(s_1,a,s_3),(s_1,b,s_4),(s_3,a,s_5)\}$, and s_1 is the initial state. The behaviour of the system can be represented graphically as shown in figure A.1. □

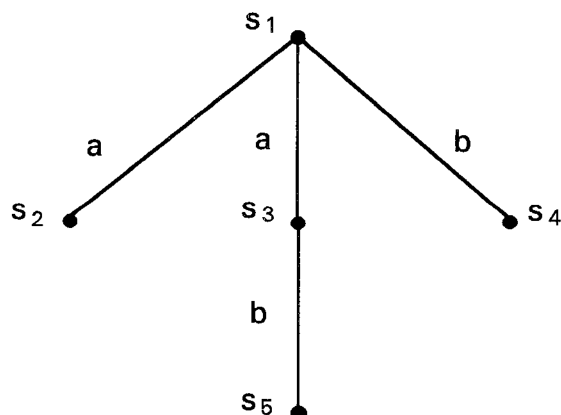

Figure A.1: Graphical representation of the complete behaviour

In this system there are two resulting states possible when in state s_1 the action *a* is performed. Such a system is said to be non-deterministic, as the resulting state is not uniquely determined by information of the current state and the performed action.

In the CSR model the term process is used to refer to the entity whose behaviour is modelled in terms of an LTS. The presentation often does not make an explicit distinction between the process and the state of an LTS representing the current process state.

When the behaviour of systems are defined by LTSs usually the states are structured values.

EXAMPLE 2: In the CSR there is a process for the handling of active timers. A state of this process is defined as an ordered sequence of timer descriptors. Each timer description is a structure containing a timer identifier, a start time value, and an expiration time value. □

The set of actions consists of actions that represents all events that may cause the system to perform a transition. The CSR also reflects the observability of events. The internal event, denoted τ, models when a change of state may occur without the environment being able to observe whether this change of state has taken place or not.

EXAMPLE 3: If a TTCN tester or SDL process instance are considered to be black boxes, then the start timer operation or set timer action are internal events not observable in the environment as they do not require any action in the environment to be performed. □

As the CSR is a model covering all basic SDL systems and TTCN test cases the definition of the state set, actions set, and transition relation of the transition systems are implicitly defined. For instance, the state set of the timer process is every sequence of timer description instances that are possible for any specific SDL or TTCN specification. Even if the CSR model is restricted to a single TTCN test case (or SDL system) it may not be possible to define the sets explicitly, e.g. by enumerating all states, as some may still be infinite. Also the set of actions of the timer process is defined implicitly in the sense that the actions are defined using parameters.

EXAMPLE 4: The action *expired(si)* modelling the expiration of a timer represents the set (or class) of actions which can be derived by substitution of an actual timer identifier for the parameter *si* in a particular SDL or TTCN specification. □

As for the set of states and actions the set of transitions is also defined implicitly. The set of transitions for each LTS in the CSR is defined in terms of *inference rules*. For each class of actions an inference rule is defined that specifies the conditions which have to be fulfilled by the pre-state and the action. Furthermore, the inference rule specifies the properties of the post-state. The inference rules are defined in accordance with the following schema:

$$\frac{p_1 \xrightarrow{a_1} p_1' \quad \cdots \quad p_n \xrightarrow{a_n} p_n'}{q \xrightarrow{b} q'} \quad C$$

This inference rule specifies the conditions which are to be fulfilled in order to enable the transition from state q to q' by action b. The conditions are the list of transitions $p_1 \xrightarrow{a_1} p_1', \cdots, p_n \xrightarrow{a_n} p_n'$ which shall all be possible and the predicate C that shall hold. The conditions are denoted the *premises* of the inference rule. Then the inference rule can be read as: "If action a_1 may be performed changing state p_1 into p_1' and ... and action a_n may be performed changing state p_n into p_n' and the predicate C is fulfilled then action b may be performed changing state q into q'". When the premises of an inference rule are fulfilled this enables the derived transition. However, it does not mean that an enabled transition is actually executed. In a state there may be more inference rules whose premises are all satisfied and among those only a single transition is executed. The choice which transition to execute is made either internally by the process itself or externally due to an interaction with another process or the environment.

It should also be noted that the shown inference rule schema is the most general form. In the CSR there are inference rules defined with no premises or associated predicate. In such cases the transition may always be performed.

EXAMPLE 5: In the CSR the following inference rule models the behaviour of the process handling active timers when a time-out occurs.

$$\frac{}{\Theta::(t_1,t_2,si) \xrightarrow{expired(si)}_T \Theta} \quad e(\Theta::(t_1,t_2,si))$$

This inference rule states that there are no requirements on other transitions to be enabled in order to perform the action *expired(si)*. There is, however, the predicate e on the pre-state $\Theta::(t_1,t_2,si)$ that requires that the pre-state is such that the timer identified by si is actually to expire. If predicate e is fulfilled for the pre-state the action *expired(si)* can be performed changing the state of the timer process into the post-state Θ. From the inference rule also the relationship between the pre- and post-state can be derived. In this rule the states are related such that the post-state can be derived from the pre-state by removing the timer descriptor of the first timer from the list of active timers. □

EXAMPLE 6: The assignment action of an SDL process instance or a TTCN test case defined on the level of a process instance is modelled by the inference rule

$$\frac{P \xrightarrow{x := t}_B P'}{ms \vdash \langle \pi::P_{\varepsilon,\rho}, (\sigma,\sigma'), \Theta \rangle \xrightarrow{\tau}_{PI} \langle \pi::P'_{\varepsilon,\rho'}, (\sigma,\sigma'), \Theta \rangle} \quad \|now\|_{\varepsilon,\rho} = time$$

For the internal transition τ to be enabled at the process instance level the premise transition of the basic process must be possible and the predicate must be satisfied. In this case it means that the internal transition may be applied only if the value of the local time variable *now* is consistent with the global time and the pre-state of the basic process is such that it may perform the assignment action "x := t". The inference rule also indicates how the pre-state changes when the transition is performed. For the assignment action the state changes include the post-state P' of the basic process and the new value to be associated to the variable x. To be more specific, it is the value of the variable substituted for the parameter x that is updated. Remember that the event $x := t$ is a generic action representing any assignment action. For further details on the other elements of the state of the process instance please refer to the CSR definition Clause (Clause 5). □

A.2 Use of inference rules

This Clause explains, on a detailed technical level, how the derivation of transitions is performed in the CSR. Therefore some knowledge of the CSR is required, particularly knowledge of the structure and the entities is of importance (subclause 5.1). The following description is not a definition of the

theory of transition relation derivation, rather it is a step-by-step explanation on how the theory is applied.

Alternatively, this Clause can be very well read in parallel with Clause 5. As an informal introduction to the use of inference rules this Clause can link theory and application. In order to make the presentation more feasible an example is used to explain the basic concepts.

EXAMPLE: This example shows how to derive a sequence of two transitions from a given state using the inference rules of the CSR. More specifically the transition sequence models a possible behaviour of a TTCN tester that receives a message which is subsequently consumed. Then the part of the test case of concern may look like:

U ? IConReq (counter := 1)

with a constraint reference denoted iconReq.

It should be noted that the behaviour description is simplified as for instance possible intermediate internal transitions due to update of the local time are not taken into account. Furthermore, not the complete set of variable mappings are present in the state descriptions.

The starting state of the TTCN tester is modelled as the following state of a process instance:

$$\left\langle \begin{bmatrix} DC ::= input(iconReq)\ via\ U\ [true];DC' \\ \rho(\varepsilon(self))=piid, \rho(\varepsilon(now))=time, \\ \rho(\varepsilon(counter))=\bot, \rho(\varepsilon(verdict))=preP \end{bmatrix}, (\langle\rangle,\langle\rangle), \langle\rangle \right\rangle$$

From left to right the state of the basic process is made up from a state of a basic process and its environment and storage $([DC,\varepsilon,\rho])$, a state of an input process $(\langle\rangle,\langle\rangle)$, and a state of a timer process $(\langle\rangle)$. Particularly:

- the basic process defines a state where an input of a message *iconReq* (connect request) from a PCO named *U* is expected;
- the environment and storage maps variable *self* to the value *piid*, *now* to *time*, the value of variable *counter* is undefined, and variable *verdict* is assigned value *preP;*
- the input process has an empty sequence of received signals and an empty sequence of saved signals; and, finally,
- the timer process is in a state with no active timers represented by the empty sequence.

How to derive the next event that may be performed by the process instance in the state described is now to be explained. Intuitively it is expected that after reception of a connect request in its input process, the process instance may consume the connect request and then proceed to its next state denoted by *DC'*.

In terms of the CSR model, the following two transitions should be possible for the process instance:

- Transition 1 (receive):

$$pre-state: \left\langle \begin{bmatrix} DC::=input(iconReq)\ via\ U\ [true];DC' \\ r(e\ (self))=piid, r(e\ (now))=time, \\ r(e\ (counter))=\perp, r(e\ (verdict))=preP \end{bmatrix}, (\langle\rangle, \langle\rangle), \langle\rangle \right\rangle$$

$$event: \quad receive(\langle iconReq, piid', piid, U\rangle)$$

$$post-state: \left\langle \begin{bmatrix} DC::=input(iconReq)\ via\ U\ [true];DC' \\ r(e\ (self))=piid, r(e\ (now))=time, \\ r(e\ (counter))=\perp, r(e\ (verdict))=preP \end{bmatrix}, (\langle iconReq, piid', piid, U\rangle, \langle\rangle), \langle\rangle \right\rangle$$

- Transition 2 (input):

$$pre-state: \left\langle \begin{bmatrix} DC::=input(iconReq)\ via\ U\ [true];DC' \\ \rho(\varepsilon\ (self))=piid, \rho(\varepsilon\ (now))=time, \\ \rho(\varepsilon\ (counter))=\perp, \rho(\varepsilon\ (verdict))=preP \end{bmatrix}, (\langle iconReq, piid', piid, U\rangle, \langle\rangle), \langle\rangle \right\rangle$$

$$event: \quad \tau$$

$$post-state: \left\langle \begin{bmatrix} DC'::=counter:=1;\ DC1 \\ \rho(\varepsilon\ (self))=piid, \rho(\varepsilon\ (sender))=piid', \\ \rho(\varepsilon\ (now))=time, \rho(\varepsilon(active_{iconReq}))=false, \\ \rho(\varepsilon\ (counter))=\perp, \rho(\varepsilon\ (verdict))=preP, \\ \rho(\varepsilon\ (x))=iconReq \end{bmatrix}, (\langle\rangle, \langle\rangle), \langle\rangle \right\rangle$$

The following inference rules are needed to derive these two transitions and cited from Clause 5.

To establish that the receive event is possible in the initial state it shall be verified that the input process may perform a receive action. To derive that this is possible inference rule **I 1** of the input process transition system is used.

(I 1)
$$\frac{}{(\sigma, \sigma') \xrightarrow{receive(sig)}_{I} (sig::\sigma, \sigma')}$$

Inference rule **I 1** states that an input process may always perform a *receive* event with parameter *sig*. The result of performing the receive event is that the received signal *sig* is added to the sequence of received signals. In this inference rule, the pre-state of an input process is represented as a tuple of two sequences of signals: σ and σ', and the post-state is given by the sequences of signals $sig::\sigma$ and σ' reflecting the state change caused by the receive action.

Like every inference rule defined in Clause 5, the pre- and post-states are given in a format using parameters. This means that pre- and post-states are represented in a generic form which, when the inference rule has to be evaluated, shall be instantiated by actual states. For the rule considered (σ, σ') is only a place holder. This property applies to events also. For example, in an instance of a receive event a specific signal is substituted for the parameter *sig*.

In order to establish that in the pre-state of transition 1 the receive event *iconReq* from the PCO named *U* may be performed it shall be derived that

$$pre-state: \quad (\langle\rangle, \langle\rangle)$$
$$event: \quad receive(\langle iconReq, piid', piid, U\rangle)$$
$$post-state: \quad (\langle iconReq, piid', piid, U\rangle, \langle\rangle)$$

is a transition of an input process. This can be done by using the following substitutions in **I 1**: the empty signal list $\langle\rangle$ is substituted for σ and σ' and $\langle iconReq, piid', piid, U\rangle$ is substituted for *sig*. By definition the concatenation of a sequence with an empty sequence results in the same sequence:

$$\langle iconReq, piid', piid, U\rangle :: \langle\rangle = \langle\langle iconReq, piid', piid, U\rangle\rangle.$$

Based on these substitutions the above transition can be inferred from **I 1**.

To further elaborate on the application of inference rules to verify that the two transitions are possible, inference rule **I 3** is considered next.

$$\text{(I 3)} \quad \frac{}{(\sigma::sig::\sigma', \sigma'') \xrightarrow{deliver(sig, p)}_{I} (\sigma::\sigma'::\sigma'', \langle\rangle)} \quad \begin{array}{l} Path(sig) = p \\ \forall sig' \in \sigma': Path(sig') \neq p \end{array}$$

There is an obvious difference compared to **I 1**: the use of a side-condition. A side-condition restricts the applicability of an inference rule.

Inference rule **I 3** is to be interpreted as follows: an input process can deliver a message received from a specific PCO. However, it is not any message received on PCO that can be delivered. The side-condition states that only the first signal from the sequence of stored signals from the specified PCO can be delivered. The first part of the side-condition $Path(sig) = p$ assures that a signal *sig* is delivered which actually has been received from the specified PCO. This condition, however, would not prevent an input process to deliver a signal which although received from the specified PCO, is not the first signal in the sequence of stored signals for that PCO. For example, in the sequence of received signals:

$$\langle\langle s, piid, piid', p\rangle, \langle s', piid_1, piid_1', q\rangle, \langle s'', piid'', piid''', p\rangle\rangle$$

two signals are received from PCO p and the third signal (counted from right to left) might be delivered. However, the second condition $\forall sig' \in \sigma': Path(sig') \neq p$ prevents that the third signal is delivered as there is a signal to the right in the sequence for which the condition does not hold.

Then for **I 3** it should be validated that:

$$\begin{array}{ll} pre-state: & (\langle iconReq, piid', piid, U\rangle, \langle\rangle) \\ event: & deliver(\langle iconReq, piid', piid, U\rangle, U) \\ post-state: & (\langle\rangle, \langle\rangle) \end{array}$$

is a transition of an input process. In order to verify this transition a substitution shall be performed based on the inference rule **I 3**. For the state parameters σ, σ' and σ'' the empty sequence $\langle\rangle$ is substituted, for the signal parameter *sig* the signal instance $\langle iconReq, piid', piid, U\rangle$ is substituted and for the path identifier the PCO identifier *U* is substituted. With these substitutions an instance of the inference rule I 3 is given as:

$$\frac{}{(\langle\rangle::\langle iconReq, piid', piid, U\rangle::\langle\rangle, \langle\rangle) \xrightarrow{deliver(\langle iconReq, piid', piid, U\rangle, U)}_{I} (\langle\rangle::\langle\rangle::\langle\rangle, \langle\rangle)}$$

where the side-conditions are trivially fulfilled:

$Path(\langle iconReq, piid', piid, U\rangle) = U$ and
$\forall sig' \in \langle\rangle: Path(sig') \neq p$ because there is no signal in the empty sequence.

Now it has been proved that the transition is possible based on the inference rule **I 3** with the above substitutions.

Up to now only inference rules with no premises have been considered. However most of the inference rules in Clause 5 do have premises, such as the following two inference rules **Proc 6** and **Proc 8**. Both inference rules, and those just discussed, are used in proving that the sequence of the receive and input action is possible.

$$\textbf{(Proc 6)}\quad \frac{(\sigma,\sigma') \xrightarrow{receive(sig)}_{I} (\sigma'',\sigma''')}{ms \vdash \langle \pi::P_{\varepsilon,\rho}, (\sigma,\sigma'), \Theta\rangle \xrightarrow{receive(sig)}_{PI} \langle \pi::P_{\varepsilon,\rho}, (\sigma'',\sigma'''), \Theta\rangle} \quad \begin{array}{l} \|now\|_{\varepsilon,\rho} = time \\ To(sig) = \rho(\varepsilon(self)) \end{array}$$

$$\textbf{(Proc 8)}\quad \frac{P \xrightarrow{input(st)\ via\ pth\ [bt]}_{B} P' \quad (\sigma,\sigma') \xrightarrow{deliver(sig,pth)}_{I} (\sigma'',\sigma''')}{ms \vdash \langle \pi::P_{\varepsilon,\rho}, (\sigma,\sigma'), \Theta\rangle \xrightarrow{\tau}_{PI} \langle \pi::P'_{\varepsilon,\rho'}, (\sigma'',\sigma'''), \Theta\rangle} \quad \begin{array}{l} \|now\|_{\varepsilon,\rho} = time \\ \|bt\|_{ms,\varepsilon,\rho} = true \\ \|st\|_{ms,\varepsilon,\rho} = si \end{array}$$

where

$$\rho' = \rho\begin{bmatrix} \varepsilon(sender) \mapsto From(sig) \\ \varepsilon(x) \mapsto si \\ \varepsilon(active_{si}) \mapsto false \end{bmatrix}$$

The basic principle of substitution which was elaborated in the previous paragraphs remains valid for the rest of this Clause. Thus, in order to apply a specific inference rule, a substitution requires definition. The new ingredient is that usually more than one step is necessary to establish the desired result. In the following this is exemplified.

To verify the receive action, the inference rule **Proc 6** is used with the following substitutions based on the pre-state of Transition 1:

$$\begin{bmatrix} DC::=input(iconReq)\ via\ U\ [true];DC' \\ \rho(\varepsilon(self))=piid, \rho(\varepsilon(now))=time, \\ \rho(\varepsilon(counter))=\bot, \rho(\varepsilon(verdict))=preP \end{bmatrix} \text{ for } \pi::P_{\varepsilon,\rho},$$

$(\langle\rangle,\langle\rangle)$ for (σ,σ'),

$\langle\rangle$ for Θ, and

$\langle iconReq, piid', piid, U\rangle$ for sig

Then inference rule **Proc 6** gives the following instance:

$$\frac{(\langle\rangle,\langle\rangle) \xrightarrow{receive(\langle iconReq,piid',piid,U\rangle)}_{I} (\langle\langle iconReq,piid',piid,U\rangle\rangle,\langle\rangle)}{ms \vdash \left\langle \begin{bmatrix} DC::=input... \\ \rho(\varepsilon(self))=piid, \\ \rho(\varepsilon(now))=time, \\ \rho(\varepsilon(counter))=\bot, \\ \rho(\varepsilon(verdict))=preP \end{bmatrix}, (\langle\rangle,\langle\rangle), \langle\rangle \right\rangle \xrightarrow{receive(\langle iconReq,piid',piid,U\rangle)}_{PI} \left\langle \begin{bmatrix} DC::=input... \\ \rho(\varepsilon(self))=piid, \\ \rho(\varepsilon(now))=time, \\ \rho(\varepsilon(counter))=\bot, \\ \rho(\varepsilon(verdict))=preP \end{bmatrix}, (\langle iconReq,piid',piid,U\rangle,\langle\rangle), \langle\rangle \right\rangle}$$

provided that it can be proven that the input process can perform the receive event and the side-conditions are fulfilled.

The proof is straightforward as it has already been verified that the input process is able to receive signal $\langle iconReq, piid', piid, U\rangle$ and thus, the input process can perform the receive event. To check the validity of the side-conditions is not as straightforward.

Side-condition $\|now\|_{\varepsilon,\rho} = time$ is to assure that the local clock of the process instance is consistent with the global time. As stated previously, the events that may be performed to ensure this

condition are not considered in this example. However, the value of the variable *now* is the value of the global clock and thus this predicate is fulfilled. The second side-condition, $To(sig) = \rho(\varepsilon(self))$, guarantees that the process instance consumes only signals sent to that process instance. Every process instance is unambiguously determined by its process instance identifier stored in variable *self* and every signal carries a destination address in term of a process instance identifier. If signal parameter *sig* is substituted by $\langle iconReq, piid', piid, U\rangle$ and variables *now* and *self* are interpreted with respect to the actual environment and storage then it can be deduced that:

$$\|now\|_{\varepsilon,\rho} = \rho(\varepsilon(now)) = time$$
$$To(sig) = piid = \rho(\varepsilon(self))$$

and thus the side-condition holds.

To summarise, it has been proven that Transition 1 (the receive action) is a possible transition of the process instance by proving that the input process can perform the receive event and by verifying that the side-conditions hold.

It shall now be proven that Transition 2 is a possible subsequent transition of the process instance. The procedure is the same as for Transition 1. In order to apply inference rule **Proc 8** the following substitutions are applied based on the pre-state of Transition 2:

$$\begin{bmatrix} DC::= input(iconReq)\ via\ U\ [true];DC' \\ \rho(\varepsilon(self))=piid, \rho(\varepsilon(now))=time, \\ \rho(\varepsilon(counter))=\perp, \rho(\varepsilon(verdict))=preP \end{bmatrix} \text{ for } \pi::P_{\varepsilon,\rho}, \text{ and thus } DC, DC' \text{ for } P, P' \text{ respectively,}$$

$(\langle iconReq, piid', piid, U\rangle, \langle\rangle)$ for (σ, σ'),

$\langle\rangle$ for Θ and

$\langle iconReq, piid', piid, U\rangle$ for *sig*

When these substitutions are performed then rule **Proc 8** becomes:

$$\begin{array}{c} DC \xrightarrow{input(st)\ via\ pth\ [bt]}_{B} DC' \quad (\langle iconReq, piid', piid, U\rangle, \langle\rangle) \xrightarrow{deliver(\langle iconReq, piid', piid, U\rangle, pth)}_{I} (\langle\rangle, \langle\rangle) \\ \hline ms \vdash \left\langle \begin{bmatrix} DC::= input\ldots \\ \rho(\varepsilon(self))=piid, \\ \rho(\varepsilon(now))=time, \\ \rho(\varepsilon(counter))=\perp, \\ \rho(\varepsilon(verdict))=preP \end{bmatrix}, (\langle iconReq, piid', piid, U\rangle, \langle\rangle), \langle\rangle \right\rangle \xrightarrow{\tau}_{PI} \left\langle \begin{bmatrix} DC' \\ \rho(\varepsilon(self))=piid, \\ \rho(\varepsilon(now))=time, \\ \rho(\varepsilon(counter))=\perp, \\ \rho(\varepsilon(verdict))=preP, \\ \rho(\varepsilon(sender))=piid', \\ \rho(\varepsilon(x))=si, \\ \rho(\varepsilon(active_{si}))=false \end{bmatrix}, (\langle\rangle, \langle\rangle), \langle\rangle \right\rangle \end{array}$$

However, it can be seen that this substitution is not all that is necessary to replace all parameters by actual terms or values. For example no substitution for path identifier *pth* has been defined. The missing substitutions can easily be derived, *U* for *pth*, *iconReq* for signal parameter *st*, and, finally, *true* for Boolean term *bt*. Then the completely instantiated transition rule is:

$$\begin{array}{c} DC \xrightarrow{input(iconReq)\ via\ U\ [true]}_{B} DC' \quad (\langle iconReq, piid', piid, U\rangle, \langle\rangle) \xrightarrow{deliver(\langle iconReq, piid', piid, U\rangle, U)}_{I} (\langle\rangle, \langle\rangle) \\ \hline ms \vdash \left\langle \begin{bmatrix} DC::= input\ldots \\ \rho(\varepsilon(self))=piid, \\ \rho(\varepsilon(now))=time, \\ \rho(\varepsilon(counter))=\perp, \\ \rho(\varepsilon(verdict))=preP \end{bmatrix}, (\langle iconReq, piid', piid, U\rangle, \langle\rangle), \langle\rangle \right\rangle \xrightarrow{\tau}_{PI} \left\langle \begin{bmatrix} DC' \\ \rho(\varepsilon(self))=piid, \\ \rho(\varepsilon(now))=time, \\ \rho(\varepsilon(counter))=\perp, \\ \rho(\varepsilon(verdict))=preP, \\ \rho(\varepsilon(sender))=piid', \\ \rho(\varepsilon(x))=si, \\ \rho(\varepsilon(active_{si}))=false \end{bmatrix}, (\langle\rangle, \langle\rangle), \langle\rangle \right\rangle \end{array}$$

Applying inference rule **I 3** to the second transition of the premise it is proven that the input process can perform the deliver event. This proof has previously been outlined in detail. What still needs to

be proved is that the basic process is capable to perform the specified input event and that the side-condition of inference rule **Proc 8** is fulfilled.

To prove the first transition of the premise the inference rule **B 1** of the transition system for basic processes is used:

(B 1) $$\frac{}{e;P \xrightarrow{e}_B P}$$

The rule states that a basic process can always performs the first event of a sequence of events. Thus from the substitution of pre-state and event of **B 1**:

$$\begin{array}{ll} pre-state: & input(iconReq)\ via\ U\ [true];DC' \\ event: & input(iconReq)\ via\ U\ [true] \\ post-state: & DC' \end{array}$$

it can be concluded that it is a transition of the basic process. To finalise the proof of Transition 2 the side-condition is to be checked:

$$\|now\|_{\varepsilon,\rho} = \rho(\varepsilon(now)) = time$$

$\|true\|_{ms,\varepsilon,\rho'} = \|true\|_{ms,\varepsilon,\rho} = true$, and

$$\|iconReq\|_{ms,\varepsilon,\rho} = iconReq$$

NOTE: Inference rule **Proc 8** specifies an update to the environment and storage of the process instance due to the transition. Also, the value of variables *sender*, *x*, and $active_{si}$ are updated with the values *piid'*, *si*, and *false*.

A.3 Usage of the CSR

This Clause illustrates how the CSR can be used to compare two specifications of a system one in SDL and the other one in TTCN.

This Clause is structured as follows: The next subclause introduces the example that will be used to illustrate the usage of the CSR. The Clause continues to present an SDL process graph and a TTCN test case for this sample specification along with their representations in the CSR. The main part is devoted to a discussion of a notion of behaviour of a (part of) system and how to relate SDL system behaviour (or to be more precise SDL process behaviour) and TTCN test case behaviour.

A.3.1 The INRES protocol

The example given is based on a small part of the INRES protocol Hogrefe [13]. More specifically, the specifications are for parts of the initiator process. The SDL specification is taken from "OSI formal specification case study: the INRES protocol and service" [13]. The TTCN specification is made for this ETR. The usage of TTCN as a system specification language does not correspond with it's intended use as a test specification notation. This has only been done to illustrate the capabilities of the CSR to compare two descriptions of the same object. A TTCN specification of the initiator process is given to allow for a comparison of the two descriptions with respect to the defined behaviour.

The INRES protocol gives two users, called Initiator and Responder, the ability to exchange Protocol Data Units (PDU). The INRES protocol is a connection-oriented protocol.

The connection establishment procedure is activated when the Initiator receives a connect request (ICONreq) from its Initiator-user. The Initiator sends a connect request PDU (CR PDU) to the Responder. If within a predefined duration of time no connect confirmation PDU (CC PDU) is received, the sending of a CR PDU is repeated. If four attempts are unsuccessful to set-up a connection to the Responder or a disconnect request is received the connection establishment

procedure is terminated. In this case the Initiator sends a disconnect indication (IDISind) to the Initiator-user.

A.3.2 An SDL process and a TTCN test case for the INRES protocol

An SDL process graph for the connection establishment procedure is shown in figure A.2. For the purpose of this Clause it is sufficient to concentrate on the behaviour, thus the precise definition of e.g. signal symbols is not considered.

A TTCN test case for the connection establishment procedure is given in table A.2. The test purpose could be stated as whether the Responder IUT is capable to accept a connect request within a given time limit. Again, the emphasis is on test case behaviour, thus all constraints, timer, and variable declarations are omitted. In table A.2 a transformed version of the TTCN test case is given which results from applying the transformation for a TTCN test case as described in Annex B, subclauses B.3.3 and B.3.4. In terms of the CSR model what is shown in table A.2 is a basic process.

A similar transformation can be applied to the SDL process graph. The transformation rules are defined in Annex B, subclause B.6.2.

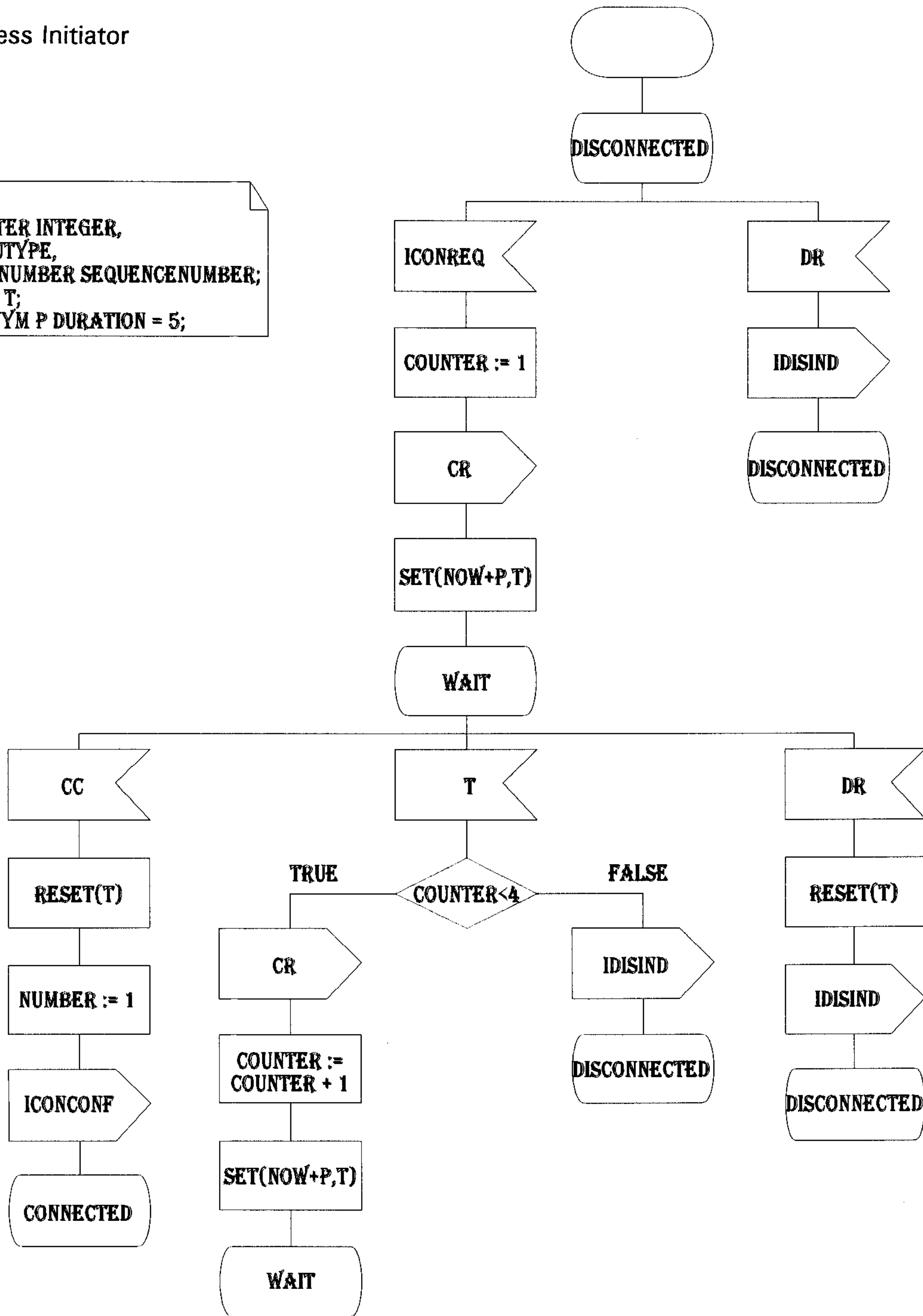

Figure A.2: Part of the Initiator process of the INRES connection establishment procedure

Table A.1: TTCN test case for the INRES connection establishment procedure

Test Case Dynamic Behaviour

Test Case Name: Initiator

Group: Initiator_Group

Purpose:

Default:

Comments: This is not a "real" test case for the INRES connection establishment procedure. Rather, it is a description of the procedure in TTCN. This description is further used in later subclauses of this annex when reasoning about behaviour.

Nr	Label	Behaviour Description	Constr. Ref	Verdict	C
1		U ? ICONreq (counter := 1)	iconReq		
2		L ! CR START T, t	cr		
3	L1	L ?CC (number :=1) CANCEL T	cc		
4		U ! ICONconf	iconConf	Pass	
5		?timeout T			
6		[counter < 4]			
7		L ! CR (counter := counter + 1) START T,t	cr		
8		-> L1			
9		[counter >= 4]			
10		U ! IDISind	idisInd	Fail	
11		L ? DR CANCEL T	dr		
12		U ! IDISind	idisInd	Fail	
13		L ? DR	dr		
		U ! IDISind	idisInd	Fail	

Detailed Comments:

Table A.2: Transformed TTCN test case

DC ::= input(iconReq) via U [true] ; counter := 1 ; DC1 ⊕ input(dr) via L [true] ; DC2

DC1 ::= output(cr) via L ; set(t, T) ; DC11

DC11 ::= input(cc) via L [true] ; number := 1 ; reset(T) ; DC111

⊕ input(T) [true] ; DC112

⊕ input(dr) via L [true] ; reset(T) ; DC113

DC111 ::= output(iconConf) via U ; verdict := pass ; stop ; nil

DC112 ::= [counter < 4] ; DC1121

⊕ [counter >= 4] ; DC1122

DC113 ::= output(idisInd) via U ; verdict := fail ; stop ; nil

DC 1121 ::= output(cr) via L ; counter := counter + 1 ; set(t, T) ; DC11

DC1122 ::= output(idisInd) via U ; verdict := fail ; stop ; nil

DC2 ::= output(idisInd) via U ; verdict := fail ; stop ; nil

A.3.3 Evaluation of the behaviour of a process instance

For the basic process shown in table A.2 the sequences of events the process is able to perform are easily derived. The only thing to be done is to apply the defined inference rules for basic processes (subclause 5.3). How to apply these rules has been demonstrated in the previous subclause. Step-by-step the behaviour of the basic process can be determined. Figure A.3 is a graphical representation of the behaviour as a tree.

In this example the nodes of the tree are labelled by basic processes states (see also table A.2) and the edges are labelled by events. A tree is a general structure that can be used for the representation of the behaviour of any other entity, e.g. process instances, modules, etc. How the behaviour of a basic process evolves can be determined by following the edges from the root towards the leaves.

Referring to figure A.3, one leaf (the second from the left) is labelled with a basic process different from *nil*, namely *DC11*. The meaning is that the leaf is to be substituted by the sub-tree starting at the node labelled *DC11*. This way it is possible to represent infinite behaviour.

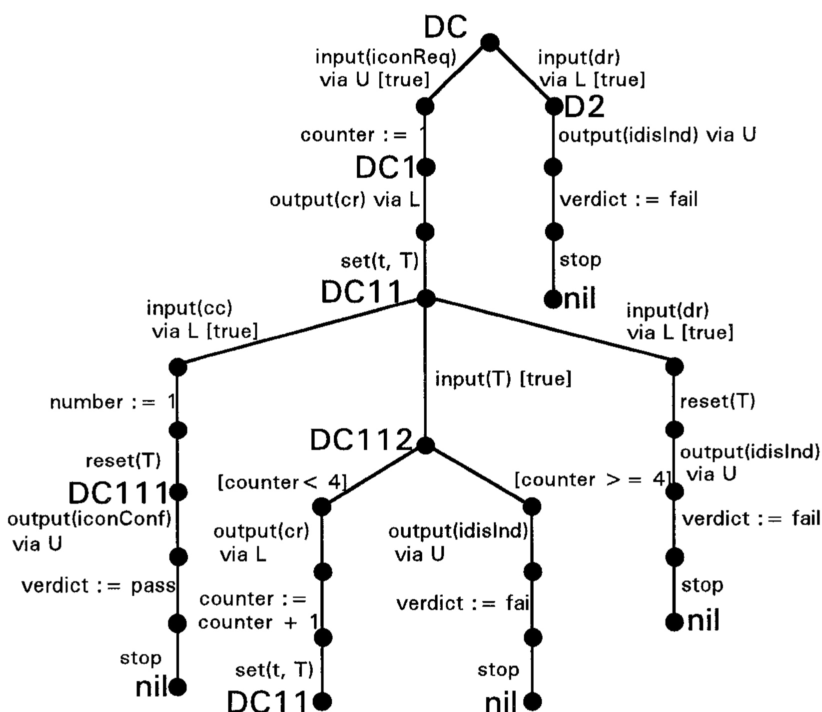

Figure A.3: Behaviour tree for basic process

Equally well, the behaviour of a basic process representing the SDL process graph can be generated.

Proceeding towards the next level in the CSR model, the process instance level, the possible behaviour of a process instance can also be graphically represented as a tree. Figure A.4 shows two trees: part a) of the figure depicts the behaviour of a process instance resulting from the transformation of the TTCN test case (table A.1 and figure A.3) and part b) shows the behaviour of a process instance resulting from the transformation of the SDL process (figure A.1). Certain details of a process instance and in the behaviour of process instance are not shown. Particularly the nodes are not labelled with a state of a process instance. This information has been left out in order to simplify the graphical representation. Furthermore, no transitions are shown which are concerned with the update of the local time or the update of the sequence of running timers. Again this is done to simplify the presentation.

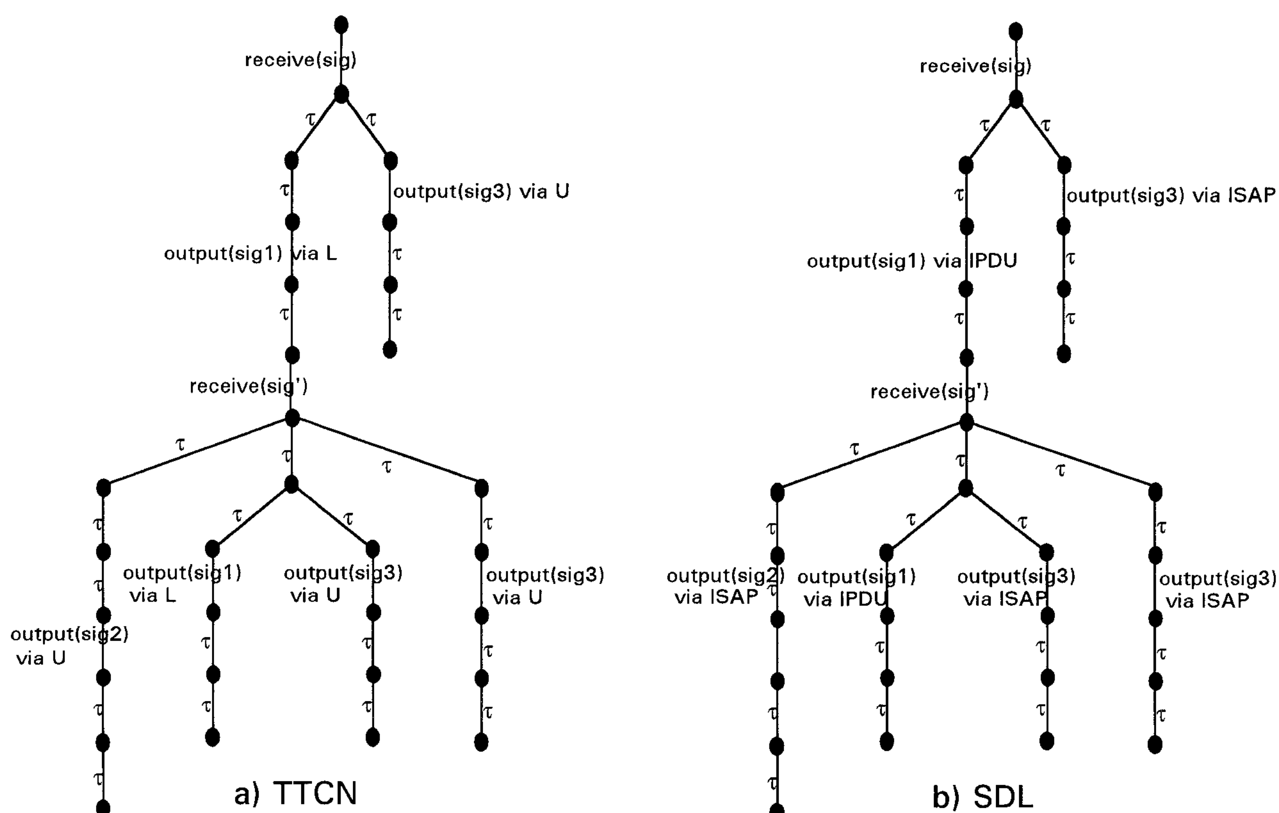

Figure A.4: Behaviour of a process instance a) TTCN b) SDL

Comparing figures A.3 and A.4 a) it is seen that the trees are different. For the following two reasons the tree shown in figure A.4 a) has changed compared to figure A.3.

1) Before a process instance can consume a signal, it shall have received a signal in its input process. This behaviour is reflected only at the process instance level.

2) The labels of edges have changed. Some events of a basic process, e.g. incrementing the *counter* variable, have been made internal events in figure A.4 a) and other events, e.g. the sending of a message, has been renamed, or, to be more precise, the identity of the message sent has changed: for instance in the output event *output(cr) via L* the data term *cr* has been substituted by a signal $sig1 = \langle cr, piid, \perp_{pid}, \perp_{pth} \rangle$.

The mapping of events to internal events at hierarchical higher levels in the CSR model is referred to as *abstraction* from behaviour. It can be motivated by assuming that an observer looks at a process instance as a black box. Whether an internal events corresponds to an update of a variable or the start of a timer cannot be distinguished by the observer. On the other hand, if a process instance interacts with its environment, e.g. by sending a signal, this can very well be observed.

Before comparing the TTCN test case behaviour and the SDL behaviour shown in figure A.4 it should be noted that the behaviour of the SDL process instance is only partially represented in the graph. More specifically, the infinite behaviour of the SDL process due to the return to the initial state *disconnected* (figure A.1) is not part of the graphical representation.

Comparing the TTCN and SDL graphs of figure A.4 it is obvious that they are structurally equivalent, What can be investigated is whether the process instances are also behaviourally equivalent.

If all internal actions are removed from the sequences of events derived form the trees then the complete sequences of observable events for the TTCN test case and the SDL process are:

a1) receive(sig) ; output(sig3) via U

a2) receive(sig) ; output(sig1) via L ; receive(sig') ; output(sig2) via U
a3) receive(sig) ; output(sig1) via L ; receive(sig') ; output(sig1) via L
a4) receive(sig) ; output(sig1) via L ; receive(sig') ; output(sig3) via U
a5) receive(sig) ; output(sig1) via L ; receive(sig') ; output(sig3) via U

b1) receive(sig) ; output(sig3) via ISAP
b2) receive(sig) ; output(sig1) via IPDU ; receive(sig') ; output(sig2) via ISAP
b3) receive(sig) ; output(sig1) via IPDU ; receive(sig') ; output(sig1) via IPDU
b4) receive(sig) ; output(sig1) via IPDU ; receive(sig') ; output(sig3) via ISAP
b5) receive(sig) ; output(sig1) via IPDU ; receive(sig') ; output(sig3) via ISAP

A strong argument in favour of a positive answer to the above question is that the sets of sequences of these process instances (a1 ... b5) are equal if the differences in denoting output ports, e.g. L and U versus ISAP and IPDU, are neglected.

This example illustrates how the CSR can be used to compare behaviour of different systems and how the model reflects abstraction of behaviour for different levels of observation.

Annex B: Definition of the transformations

This annex provides information on the relation between TTCN and SDL constructs and the domains of the CSR. However, this annex does not define a complete mapping from TTCN and SDL into the CSR.

B.1 The identification of the range

B.1.1 Introduction

In the description of the CSR a number of entities have been identified and used without further specification of their contents. The contents of these entities is dependent on a specific instance of an SDL specification or a TTCN test suite. In the following subclause an overview of these entities from the CSR is provided. The transformations of an SDL system and a TTCN test suite provide the contents of these domains.

B.1.2 The range

The set of path identifiers: *PathId.*
The set of module identifiers: *ModId.*
The set of basic process identifiers: *BPId.*
The set of process instance identifiers: *PId.*
For each module identifier: $m \in ModId$:
the set of basic process identifiers for that module identifier: $BPId_m$.
For each process identifier: $X \in BPId$:
the set of process instance identifiers for that basic process identifier: $BPPId_X$.
For each module identifier: $m \in ModId$:
the set of process instance identifiers for the basic process identifiers of the processes in the module m.
identified by the module identifier: $ModPID_m$.
For each basic process identifier: $X \in BPId$:
a process expression $P \in BProc$
For each basic process identifier: $X \in BPId$:
a declaration $D(X)$,
where: $D : BPId \rightarrow BProc \times N \times N_\infty \times Var^* \times Var^* \times (Var \times Term(Var))^*$
For each module identifier: $m \in ModId$:
the routes inside the module identified by the module identifier: $Routes_m$
For each module identifier: $m \in ModId$:
the route function r_m that identifies the set of signals that a route in the module identified by the module identifier may convey.
The signature $\Sigma = \langle Sort, OP \rangle$

The variables of each sort s: $s \in Sort, Var_s$.

B.2 Transformation of SDL

In this Clause the transformation from a basic SDL specification into the CSR is sketched. The transformation description is based on the abstract grammar AS.1 of SDL CCITT Recommendation Z.100 [1]. The concept of service which has become part of basic SDL in 1992 is not considered in the mapping. Neither are channels with no delay considered. The following assertions apply to the AS.1 grammar

- Implicit transitions have been replaced by explicit transitions;
- Join/label constructs are eliminated;
- Shorthand's have been substituted.

The range of the transformation is based on Clause B.1. The following structured domain is defined as the range of the mapping.

Table B.1: A refined range definition

Range	:: *ModId PathId Moddecl*-**set** *Routes SigmaAlg*
Moddecl	:: $BPid_m$ *Route BpDecl*
Routes	:: (*RouteId RouteId*)-**set**
SigmaAlg	:: *Sort Operators*
Route	:: *RouteId RouteId* $\rightarrow 2^{SigTerm^A \cup \{\perp_{st}\}}$
RouteId	= (*BPid* $\cup$ *PathId*)
BpDecl	:: *Identifier* $\rightarrow$ *BpExpr Inst Par* (*Var Term*(*Var*))* *BpMap*
BpExpr	= *PrefixExpr* | *ChoiceExpr* | *PrioChoiceExpr* | *BpNm* | **NIL**
PrefixExpr	:: *BpEvent BpExpr*
ChoiceExpr	:: *PrefixExpr BpExpr*
PrioChoiceExpr	:: *PrefixExpr BpExpr*
BpNm	:: *Token*
BpMap	:: *BpNm* $\rightarrow$ *BpExpr*
Par	:: *Refparml Valparml*
Refparml	:: *Var**
Varparml	:: *Var**
Var	:: *Term*
ModId	:: *Identifier*-**set**
PathId	:: *Identifier*-**set**
BPid	:: *Identifier*-**set**
Identifier	:: *Token*

NOTE: The domain of the basic process has been refined to allow finite representation of a process graph. The mapping *BpMap* relates identifiers of process expression with the basic process expression.

Transformation from a basic SDL system definition based on the AS.1 definition:

def trf-system(sys) =
    pre: (**mk**-*System-definition*(snm,blset,chset,sigs,dt,syntset) = sys)
        (modidset = {id | for all bldef $\in$ blset,
            id = **mk**-*Identifier*(**s**-*Block-name*(bldef))}),
        (pathidset = {pthid,pthid' | for all c $\in$ chset, (pthid,pthid) = trf-chanid(c)}),
        (moddscrset = trf-blockdefset(blset)),
        (routes = trf-routes(blset,chset)),
        (sigalg = trf-datasigs(blset,sigs,dt,syntset))
    post: **mk**-*Range*(modidset,pathidset,modscrset,routes,sigalg)
type: trf-system: *System-definition* $\rightarrow$ *Range*

The transformation of an SDL system results in a CSR representation *Range* of the object of the SDL specification.

def trf-chanid(cd) =
    pre: (**mk**-*Channel-definition*(chnm,,cp,**nil**) = cd)
    post: **mk**-*Identifier*(chnm) nil
    pre: (**mk**-*Channel-definition*(chnm,,cp,cpn) = cd)
    post: **mk**-*Identifier*(unique(chnm)) **mk**-*Identifier*(unique(chnm))
type: trf-chanid: *Channel-definition* → *Identifier* [*Identifier*]

The generation of path identifiers are derived from the channels. In case of bi-directional channels two unique path identifiers are derived.

def trf-blockdefset(blset) =
    pre: blset
    post: {moddcl | For all bldef ∈ blset, moddcl = trf-blockdef(bldef)}
type: trf-blockdefset: *Block-definition*-**set** → *Moddecl*-**set**

The transformation of the set of block definitions.

def trf-blockdef(bldef) =
    pre: (**mk**-*Block-definition*(
        blnm,prdefset,sigdefset,chrouset,sigrset,dtset,syntdefset) = bldef)
    post: **mk**-*Moddecl*(
        {id, id' | for all prd ∈ prdefset, id = **s**-*Process-name*(prd),
            for all prod ∈ s-Procedure-definition-set(prd),
                id' ∈ gen-bpnameset(prod)},
        trf-sigchanrouteset(chrouset,sigrset),
        trf-prdefset(prdefset))
type: trf-blockdef: *Block-definition* → *Moddecl*

Transform a block definition into a module declaration.

def gen-bpnameset(prod) =
    pre: (**mk**-*Procedure-definition*(prdnm,,{},,,,) = prod)
    post: {**mk**-*Identifier*(prdnm)}
    pre: (**mk**-*Procedure-definition*(prdnm,,prddef,,,,) = prod)
    post: {**mk**-*Identifier*(prdnm)} $\cup \bigcup_{p \in prddef}$ gen-bpnameset(p)
type: gen-bpnameset: *Procedure-definition* → *Identifier*-**set**

Generate the set of basic process identifiers from a procedure definition and every embedded procedure definition.

def trf-routes(blset,chset) = (...)
type: *Block-definition*-**set** *Channel-definition*-**set** → *Routes*

Transform a set of block definitions and a set of channel definitions into the set of routes. This contains the pairs of connected processes and pairs of connected paths and processes in the system.

def trf-datasigs(sigs,dt,syntset) = (...)
type: *Block-definition*-**set** *Data-type-definition* *Syntype-definition*-**set** → *SigmaAlg*

Transform the data type definition of the SDL system to a many sorted algebra. The information is derived from the partial data type definitions at the system level and the definitions in the set of blocks.

def trf-prdefset(prset) =
    pre: (prset ={})

post: **mk**-*BpDecl*([])
pre: prdef ∈ prset
post: **mk**-*BpDecl*(trf-prdefset(prset\{prdef}) + [trf-prdef(prdef)]
type: trf-prdefset: *Process-definition*-**set** → *BpDecl*

Transform a set of process definitions into a set of basic process declarations.

def trf-prdef(pd) =
pre: (**mk**-*Process-definition*(
pnm,inst,parml,prodset,sigset,dts,sts,srds,srrcs,vars,views,tms,pg) = pd),
(bpid = **mk**-*Identifier*(pnm)),
(bpe,bpm = trf-processgraph(pg)(bpid))
post: bpid ↦
bpe,
trf-inst(inst),
trf-parameters(parml),
trf-localvars(vars),
bpm
type: trf-prdef: *Process-definition* → (*Identifier* → *BpExpr Inst Par* (*Var Term*(*Var*))*)

Transform a process definition into a basic process declaration and a basic process map *bpm*.

def trf-sigchanrouteset(chrcset,sigrset) =
pre: sigrset = {}
post: **mk**-*Route*([])
pre: sigr ∈ sigrset
post:(trf-sigchanrouteset(chrcset,sigrset\{sigr}) + trf-sigroute(chrcset,sigr))
type: trf-sigchanrouteset: *Channel-to-route-connection*-**set** *Signal-route-definition*-**set** → *Route*

Transform a signal route definition set to a route function which identifies the signals that may be conveyed between two processes or a process and a path.

def trf-sigroute(chrc,sigroute) =
pre: (**mk**-*Signal-route-definition*(rnm,sp1,nil) = sigroute),
(**mk**-*Signal-route-path*(orig,dest,sigset) = sp1),
(origid = gen-proc-or-path-id(orig,rnm,chrc)),
(destid = gen-proc-or-path-id(dest,rnm,chrc))
post: [origid destid ↦ gen-sigterm(sigset)]
pre: (**mk**-*Signal-route-definition*(rnm,sp1,sp2) = sigroute),
(**mk**-*Signal-route-path*(orig1,dest1,sigset1) = sp1),
(origid1 = gen-proc-or-path-id(orig1,rnm,chrc)),
(destid1 = gen-proc-or-path-id(dest1,rnm,chrc)),
(**mk**-*Signal-route-path*(orig2,dest2,sigset1) = sp2),
(origid2 = gen-proc-or-path-id(orig2,rnm,chrc)),
(destid2 = gen-proc-or-path-id(dest2,rnm,chrc)),
post: [origid1 destid1 ↦ gen-sigterm(sigset1),
origid2 destid2 ↦ gen-sigterm(sigset2)]
type: trf-sigroute: *Channel-to-route-connection*-**set** *Signal-route-definition* →
[*Identifier Identifier* → *SigTerm*^{A}]

Transform a signal route definition into one or two maps from a pair of process identifiers or a process identifier and a path identifier into a set of interpretations of signal terms.

Process graph transformation

def trf-processgraph(pg)(bpid) =
pre: (**mk**-*Process-graph*(psn,stset) = pg),
(**mk**-*Transition*(tr) = psn)
post: trf-transition(tr,stset,bpid)([])
type: trf-processgraph: *Process-graph Identifier* → *BpExpr BpMap*

Transform a process graph to a basic process expression. The second parameter is the basic process identifier of the process or procedure being transformed.

def trf-transition(tr,stset,bpid)(bpm) =
pre: (**mk**-*Transition*(grnl,tord) = tr),
(act = **hd**(grnl)),
(**not**(**is**-*Output-node*(act))),
(bpe,bpm' = trf-transition(**mk**-*Transition*(**tl**(grnl),tord),stset,bpid)(bpm))
post: **mk**-*PrefixExpr*(trf-action(act),bpe),bpm'
pre: (**mk**-*Transition*(grnl,tord) = tr),
(act = **hd**(grnl)),
(**is**-*Output-node*(act)),
post: trf-output(act,**mk**-*Transition*(**tl**(grnl),tord),stset,bpid)(bpe)
pre: (**mk**-*Transition*(< >,tord) = tr)
(**mk**-*Terminator*(tn) = tord))
post: trf-terminator(tn,stset,bpid)(bpm)
pre: (**mk**-*Transition*(< >,tord) = tr)
(**mk**-*Decision-node*(,,) = tord))
post: trf-decision(tord,stset,bpid)(bpm)
type: trf-transition: *Transition State-node*-**set** *Identifier BpMap* → *BpExpr BpMap*

Transform a transition to a basic process expression. Note that due to that, the output statement may cause a choice in the basic process to be generated this action is handled separately.

def trf-action(gn) =
pre: (**mk**-*Task-node*(tn) = gn)
post: trf-task(tn)
pre: (**mk**-*Create-request-node*(cn) = gn)
post: trf-create(cn)
pre: (**mk**-*Call-node*(cn) = gn)
post: trf-call(cn)
pre: (**mk**-*Set-node*(sn) = gn)
post: trf-set(sn)
pre: (**mk**-*Reset*(rn) = gn)
post: trf-reset(rn)
type: trf-action: *Graph-node* → *BpEvent*

Transform a graph node to a basic process event.

def trf-task(tn) =
pre: (**mk**-*Task-node*(as) = tn)
mk-*Assignment-statement*(varid,expr) = as)
post: (**mk**-*BpEvent*("varid := expr"))
type: trf-task: *Task-node* → *BpEvent*

Transform a task node. In this mapping only complete SDL specifications are considered so the use of informal text in a task construct is not considered.

def trf-output(on,tr,stset,bpid)(bpm) =
pre: (**mk**-*Output-node*(sigid,exprl,**nil**,{}) = on),
(st = **mk**-*Term*(sigid,exprl)),
(bpe,bpm' = trf-transition(tr,stset,bpid)(bpm))
(outev = **mk**-*BpEvent*("**output**(st)"))
post: **mk**-*PrefixExpr*(outev,bpe) bpm'
pre: (**mk**-*Output-node*(sigid,exprl,dest,{}) = on),
(st = **mk**-*Term*(sigid,exprl)),
(bpe,bpm' = trf-transition(tr,stset,bpid)(bpm)),
(**mk**-*Expression*(expr) = dest),
(outev = **mk**-*BpEvent*("**output**(st) **to** dest"))

post: **mk**-*PrefixExpr*(outev,bpe) bpm'
pre: (**mk**-*Output-node*(sigid,exprl,dest,{}) = on),
(st = **mk**-*Term*(sigid,exprl)),
(bpe,bpm' = trf-transition(tr,stset,bpid)(bpm)),
(**mk**-*Process-identifier*(expr) = dest),
(outev = **mk**-*BpEvent*("**output**(st) **via** (bpid,dest)")
post: **mk**-*PrefixExpr*(outev,bpe) bpm'
pre: (**mk**-*Output-node*(sigid,exprl,**nil**,viaset) = on),
(st = **mk**-*Term*(sigid,exprl)),
(rset = derive-routes(viaset)),
(bpe,bpm' = trf-transition(tr,stset,bpid)(bpm)),
(pfel = trf-outputvia(rset,st,bpe))
post: merge-pfix(pfel) bpm'
pre: (**mk**-*Output-node*(sigid,exprl,dest,viaset) = on),
(st = **mk**-*Term*(sigid,exprl)),
(bpe,bpm' = trf-transition(tr,stset,bpid)(bpm)),
(**mk**-*Expression*(expr) = dest),
(rset = derive-routes(viaset)),
(pfel = trf-outputtovia(rset,dest,st,bpe))
post: merge-pfix(pfel) bpm'
pre: (**mk**-*Output-node*(sigid,exprl,dest,{}) = on),
(st = **mk**-*Term*(sigid,exprl)),
(bpe,bpm' = trf-transition(tr,stset,bpid)(bpm)),
(**mk**-*Process-identifier*(expr) = dest),
(outev = **mk**-*BpEvent*("**output**(st) **via** (bpid,dest)")
post: **mk**-*PrefixExpr*(outev,bpe) bpm'
type: trf-output: *Output-node Transition State-node*-**set** *Identifier BpMap* →
BpExpr BpMap

Transform an output action with the following transition into a basic process expression. This way of handling the output action is due to as there may be more routes specified in the via construct the receiver of the signal may not be statically determined.

def derive-routes(viaset) = (...)
type: *Direct-via* → (*RouteId RouteId*)-**set**

Derive from a via specification in an output action the route by which the signal may be conveyed. Note that for two process instances connected by more signal routes there is only a single element generated in the route descriptor set. For signal routes in the via specification connected to the environment in such a way that more channels may convey the signal from the block more elements may be generated in the route set.

def trf-outputvia(rset,st,bpe) =
pre: (rset = {}),
post: < >
pre: (r ∈ rset),
(oev = **mk**-*BpEvent*("**output**(st) **via** r")),
(pfe = **mk**-*PrefixExpr*(oev,bpe))
post: (pfe :: trf-outputvia(rset\{r},st,bpe))
type: trf-outputvia: (*RouteId RouteId*)-**set** *SignalTerm BpExpr* → *PrefixExpr**

Transform an output action with a via construct into a list of basic process output via events. This list is to be structured into a list of alternative output events.

def trf-outputtovia(rset,p,st,bpe) =
pre: rset = {}
post: < >
pre: (r ∈ rset),
(oev = **mk**-*BpEvent*("**output**(st) **to** p **via** r")),
(pfe = **mk**-*PrefixExpr*(oev,bpe)),

post: (pfe :: trf-outputtovia(rset\{r},p,st,bpe))
type: trf-outputtovia: *(RouteId RouteId)*-**set** *Expression SignalTerm BpExpr* → *PrefixExpr**

Transform an output action to a specific process instance with a via construct. As the destination is known only at execution time every route which may be derived from the via specification forms an alternative event even though it may at execution time turn out that the route does connect to the specified process instance. The result is a list of basic process prefix expressions which is to be structured as alternatives.

def trf-create(cn) =
pre: mk-*Create-request-node*(procid,exprl) = cn)
post: mk-*BpEvent*("**create** procid(exprl)")
type: trf-create: *Create-request-node* → *BpEvent*

Transform a create action into a basic process create event.

def trf-call(cn) =
pre: (**mk**-*Call-node*(prid,exprl),
(rpl,vpl = conv-prodexprlist(exprl))
post: mk-*BpEvent*("**call** prid(rpl)(vpl)")
type: trf-call: *Call-node* → *BpEvent*

Transform a procedure call node into a basic process call event. The parameters of the procedure call are distinguished whether they are call by reference parameters or call-by-value parameters.

def trf-set(sn) =
pre: (**mk**-*Set-node*(timeexpr,tid,exprl) = sn),
(timid = conv-timeridentifier(tid,exprl))
post: mk-*BpEvent*("**set**(timeexpr,timid)")
type: trf-set(sn): *Set-node* → *BpEvent*

Transform a set timer action to a basic process set timer event.

def trf-reset(rn) =
pre: (**mk**-*Reset-node*(tid,exprl) = rn),
(timid = conv-timeridentifier(tid,exprl))
post: mk-*BpEvent*("**reset**(timid)")
type: trf-reset: *Reset-node* → *BpEvent**

Transform a reset action to a basic process reset timer event.

def trf-terminator(tn,stset,bpid)(bpm) =
pre: (**mk**-*Nextstate-node*(snm) = tn)
post: trf-nextstate(snm,stset,bpid)(bpm)
pre: (**mk**-*Stop*() = tn)
post: **mk**-*PrefixExpr*(**mk**-*BpEvent*("**stop**"), **NIL**) bpm
pre: (**mk**-*Return*() = tn)
post: **mk**-*PrefixExpr*(**mk**-*BpEvent*("**return**"), **NIL**) bpm
type: trf-terminator: *Terminator State-node*-**set** *Identifier BpMap* → *BpExpr BpMap*

Transform a terminator action.

def trf-nextstate(snm,stset,bpid)(bpm) =
pre: (stset = {}),
(bpnm = **mk**-*BpNm*(snm))
post: bpnm bpm
pre: (snode ∈ stset),
(**mk**-*State-node*(snm,,,) = snode),
(bpnm = **mk**-*BpNm*(snm))

post: bpnm (bpm + [bpnm ↦ trf-statenode(snode,stset\{snode})(bpm)])
pre: (snode ∈ stset),
(snode(snm',,,) = snode),
(snm ≠ snm'),
(bpnm = **mk**-*BpNm*(snm))
post: bpnm bpm
type: trf-nextstate: *State-name State-node*-**set** *Identifier BpMap* → *BpExpr BpMap*

Transform a nextstate node.

def trf-statenode(sn,stset,bpid)(bpm) =
pre: (**mk**-*State-node*(snm,saveset,inpset,spset) = sn)),
(sapfel = trf-saveset(snm,saveset)),
(sppfel stset',bpm' = trf-spontaneousset(spset,stset,bpid)(bpm)),
(ippfel, stset'', bpm'' = trf-inputset(inpset,stset',bpid)(bpm')),
(bpe = merge-pfix(sapfel::sppfel :: ippfel))
post: bpe bpm''
type: trf-statenode: *State-node State-node*-**set** *Identifier BpMap* → *BpExpr BpMap*

Transform a state with associated input nodes, save nodes and spontaneous transitions.

def trf-saveset(snm,saveset) =
pre: (saveset = {},
post: < >
pre: (savesig ∈ saveset),
(pfe = trf-save(snm,savesig)
post: pfe :: trf-saveset(snm,saveset\{savesig}))
type: trf-saveset: *State-name Save*-**set** → *PrefixExpr**

Transform a non-empty set of save constructs into a list of basic process prefix expressions.

def trf-save(snm,sigid) =
pre: (bpe = **mk**-*BpExpr*(snm),
(sev = **mk**-*BpEvent*("**save**(sigid)")
post: (**mk**-*PrefixExpr*(sev,bpe))
type: trf-save: *State-name Signal-identifier* → *PrefixExpr*

Transform a save statement to a basic process prefix expression.

def trf-spontaneousset(spset,stset,bpid)(bpm) =
pre: spset = {}
post: (< >, stset, bpm)
pre: (sptr ∈ spset),
(**mk**-*Spontaneous-transition*(tr) = sptr),
(bpev = **mk**-*BpEvent*("sender := self")),
(bpe,bpm' = trf-transition(tr,stset,bpid)(bpm)),
(pfe = **mk**-*PrefixExpr*(bpev,bpe)),
(stset' = stset \ **dom**(bpm)),
(pfel,stset'',bpm'' = trf-spontaneousset(spset\{sptr},stset')(bpm')),
(pfel' = pfe :: pfel)
post: (pfel', stset'', bpm'')
type: trf-spontaneousset: *Spontaneous-transition*-**set** *State-node*-**set** *Identifier BpMap* →
*PrefixExpr** *State-node*-**set** *BpMap*

Transform a set of spontaneous transitions into a list of basic process prefix expressions. Note that in the CSR the assignment of the basic process instance identifier value to the variable "sender" reflects the execution of a spontaneous input action.

def trf-inputset(inpset,stset,bpid)(bpm) =
pre: (inpset = {})

```
        post: (< >, stset, bpm)
        pre: (inode ∈ inpset),
                (pfe,stset',bpm' = trf-input(inode,stset,bpid,bpm)),
                (pfel,stset'',bpm'' = trf-inputset(inpset\{inode},stset')(bpm)),
                (pfel' = pfe :: pfel)
        post: (pfel', stset'', bpm'')
type: trf-inputset: Input-node-set State-node-set Identifier BpMap →
        PrefixExpr* State-node-set BpMap
```

Transform a set of input nodes to a list of basic process prefix expressions.

```
def trf-input(in,stset,bpid)(bpm) =
        pre: (mk-Input-node(sigid,varl,tr) = in),
                (st = conv-sigvarl(sigid,varl)),
                (inpev = mk-BpEvent("input(st)")),
                (bpe,bpm' = trf-transition(tr,stset,bpid)(bpm)),
                (pfe = mk-PrefixExpr(inpev,bpe)),
                (stset' = stset\dom(bpm'))
        post: (pfe, stset', bpm')
type: trf-input: Input-node State-node-set Identifier BpMap →
        PrefixExpr State-node-set BpMap
```

Transform an input node with an associated transition to a basic process expression,

```
def trf-decision(dn,stset,bpid)(bpm) =
        pre: (mk-Decision-node(quest,answset,elseansw) = dn),
                (cond = derive-elsecondition(quest,answset)),
                (btev = mk-BpEvent("[cond]")),
                (mk-Transition(tr) = elseansw),
                (bpe,bpm' = trf-transition(tr,stset,bpid)(bpm)),
                (pfe = mk-PrefixExpr(btev,bpe)),
                (pfel,bpm'' = trf-answset(quest,answset,bpid)(bpm')
        post: merge-pfix(pfe :: pfel) bpm''
        pre: (mk-Decision-node(quest,answset,nil) = dn),
                ((pfel,bpm' = trf-answset(quest,answset,bpid)(bpm'))
        post: merge-pfix(pfe :: pfel) bpm'
type: trf-decision: Decision-node State-node-set Identifier BpMap → BpExpr BpMap
```

Transform a decision construct into alternatives each with a Boolean term as the initial event.

```
def derive-elsecondition(quest,answset) = (...)
type: derive-elsecondition: Decision-question Decision-answer-set → Term
```

Derive a Boolean term which represents the negation of the conditions of all other decision answers. This term represents the condition of the "else" transition.

```
def trf-answset(quest,answset,stset,bpid)(bpm) =
        pre: (answset = {})
        post: (< >, bpm)
        pre: (answ ∈ answset),
                (mk-Decision-answer(rc,tr) = answ),
                (cnd = derive-condition(quest,rc)),
                (btev = mk-BpEvent("[cnd]")),
                (bpe,bpm' = trf-transition(tr,stset,bpid)(bpm)),
                (pfe = mk-PrefixExpr(btev,bpe)),
                (pfel, bpm'' = trf-answset(quest,answset\{answ},stset,bpid)(bpm'))
        post: (pfe :: pfel) bpm''
type: trf-answset: Decision-question Decision-answer-set State-node-set
        Identifier BpMap →PrefixExpr* BpMap
```

Transform a set of answers of a decision construct into a list of basic process prefix expressions. These are to be structured as alternatives.

def derive-condition(quest,rng) = (...)
type: derive-condition: *Decision-question Range-condition* $\rightarrow$ *Term*

Derive a Boolean term from a decision question and a range condition in a decision answer.

def trf-inst(inst) = (...)
type: trf-inst: *Number-of-instances* $\rightarrow N \times N_\infty$

Transform the number of instances information from a process declaration.

def trf-parameters(parml) =
type: trf-parameters: *Process-formal-parameter** $\rightarrow$ *Par*

Transform a list of formal parameters of a process declaration into a list of call-by-reference parameters and a list of call-by-value parameters.

def trf-localvars(vars) = ...
type: trf-localvars: *Variable-definition*-**set** $\rightarrow (Var \times Term(Var))^*$

Transform a set of local variable declarations and initial value assignments.

B.2.1 Generator functions

def gen-proc-or-path-id(orig,rnm,chrc) = (...)
type: gen-proc-or-path-id: (*Process-identifier* | ENVIRONMENT)
Signal-route-name Channel-to-route-connection-**set** $\rightarrow$ *Identifier*

Construct an identifier which is either a process identifier or a path identifier based on the specified process identifier. If the environment is specified the result is a path identifier derived from the channel to route connection set and the name of the signal route.

def gen-sigterm(sigset) = (...)
type: gen-sigterm: *Signal-identifier*-**set** $\rightarrow 2^{SigTerm^A \cup \{\perp_{st}\}}$

Generate the powerset of interpretations of the signal identifier set.

B.2.2 Conversion functions

def conv-processidentifier(pid) = (...)
type: conv-processidentifier: *Process-identifier* $\rightarrow$ *Route*

Convert a receiving process identifier into a route descriptor. Note that in order for this route description to be derived the process identifier of the sending process instance must be available.

def conv-expressionlist(exprl) = (...)
type: conv-expressionlist: [*Expression*]* $\rightarrow$ *Term**

Convert a list of expressions into a list of terms.

def conv-prodexprlist(prodid,exprl) = (...)
type: conv-prodexprlist: *Procedure-identifier* [*Expression*]* $\rightarrow$ *Var* Term(Var)**

Convert a procedure call parameter list into a call-by-reference parameters (Var*) and a list of call-by-value parameters (Term(Var)*). In order to perform this separation of the parameters the procedure declaration is needed, so the procedure identifier is needed in order to retrieve the basic process declaration from which the sort of the parameters in the parameter list can be derived.

def conv-timeridentifier(tid,exprl) = (...)
type: conv-timeridentifier: *Timer-identifier Expression** → *SigTerm(Var)*

Convert a timer identifier with an optional list of expressions into a signal term. Timers are handled as signals in the CSR.

def conv-signalident(sigid) = (...)
type: conv-signalident: *Signal-identifier* → *SigSort*

Convert a signal identifier of a save construct into a signal sort in the CSR.

def conv-sigvarl(sigid,varl) = (...)
type: conv-sigvarl: *Signal-identifier* [*Variable-identifier*]* → *SigTerm*

Convert a signal identifier and associated list of variables into a signal term.

B.2.3 Auxiliary functions

def merge-pfix(pfl) =
  pre: (**len**(pfl) = 1),
  post: **hd**(pfe)
  pre: (pfe = **hd**(pfl))
  post: **mk**-*ChoiceExpr*(pfe,merge-pfix(**tl**(pfl))
type: merge-pfix: *PrefixExpr*+ → *BpExpr*

Construct a structure of basic process choice expressions from a list of basic process prefix expressions.

B.3 Transformation of TTCN

The transformation of TTCN defines a set-up of all entities in the CSR. With respect to the conceptual model for TTCN (subclause 4.2) the TTCN tester is transformed to a *process instance*, PCOs are related to *paths*, abstract tester is related to a module and the TTCN system is related to *system*.

The three processes of the TTCN tester, called Tree, Timer, and Input, are transformed to a basic process, Timer process, and Input process. The TTCN tester maintains sets of parameters, variables and constants. For all these parameters, variables and constants, locations are allocated in the storage environment of the process instance in the CSR, which models the TTCN tester, and an initial assignment of values is performed.

The transformation of a TTCN test suite is defined in subclauses B.3.3 and B.3.4 which deal with the transformation of the static part of a TTCN test suite (data, variables, PCOs, etc.) and the dynamic part (the TTCN test cases). Subclause B.3.1 defines the assumptions made on the input domain of the transformations, i.e. a subset of TTCN with the same expressive power as full TTCN. Subclause B.3.2 is devoted to the transformation of data type definitions (including ASN.1).

B.3.1 Assumptions on TTCN

TTCN allows several ways to specify properties for the contents of an element in a PCO queue.

The contents of the element, i.e. the type of the element and the value of certain fields of the element is of importance for testing in several ways.

For checking the type, the type of the expected message can be indicated. For checking the value of certain fields, a constraint for the type can be indicated.

As a third and last possibility TTCN allows the use of qualifiers which use references to fields of a message. Using constraints with actual parameters equal to the expected values provides the same expressive power and indicates a more direct relation to the message.

- **Assumption:** references to fields of a message are not used in a qualifier.
- **Assumption:** references to fields of a message are not used as the left hand side of an assignment.

References to fields are only used in the right hand side of an assignment on the behaviour line to update variables which are used for historical information. The use of references to fields in the left hand side of an assignment has the same function as the use of actual parameters for the constraints, where the parameter indicates the field of the message.

The references to fields in the qualifiers are not necessary, because one can parameterise the constraint with a specific value or define different constraints for the different qualifiers (where the values for the fields are provided in the constraint).

The use of the references to fields in assignments is necessary unless for each of the possible values for the field different constraints are made. In that case the value of the field is represented in the execution path. This is, however, impossible if the number of possible values is infinite.

B.3.1.1 Assumptions on referencing parts of messages

There are two reasons to reference to a certain part of a message. One reason is to restrict the value of this part of the message and another is to store the value for reuse. The restriction on messages is useful with respect to the sending and receiving of messages, whereas the storing for reuse of parts of messages is only necessary when receiving messages. To restrict parts of messages to certain values, TTCN has the constraints (with actual parameters for those cases where the restrictions are determined dynamically). To store the value received for reuse, TTCN offers the possibility to reference a part of a message within the right hand side of an assignment.

TTCN is less restrictive with respect to the places where references to fields of a message may occur.

- **Assumption:** Only the constraints on the right hand side of an assignment are used to reference a part of a message.

Due to this assumption, the order of evaluation of a event line in terms of its components can be the same for both sending and receiving events.

```
PCO ! PDU      [Q]   (A:=v1,B:=v2)      Constraint
```

If no reference to PDU in the qualifier and the left hand side of the assignments the order:

```
3              1     2                  3
```

can become

```
2              1     3                  2
```

and:

```
PCO ? PDU      [Q]   (A:=v1,B:=v2)      Constraint
```

If no reference to PDU in the qualifier and the left hand side of the assignments the order:

```
1              2     3                  1
```

can become:

```
2              1     3                  2
```

Conclusions: a restriction on the possible values for a part of a message are made by a constraint (possibly by usage of a specific actual value for a formal parameter). For the storage of the value of

a part of a received message the right hand side of an assignment on the behaviour line may contain references to named fields of the PDU. A reference to a part of a message should be to a named field by using its name.

B.3.1.2 Assumptions on constraints

In the following it will be shown that due to the fact that if one set of alternatives deals completely with the reception of a message from a PCO queue some of the constructs which are defined for constraints become redundant. (As they are also difficult to describe in a formal manner these constructs shall be eliminated from the constructs that are mapped to abstract data types).

The type of a message identifies the set of all possible values.

A constraint for a type of a message identifies a subset of the set of all possible messages.

The set of all possible messages can be constructed by varying the values for the parts of the message. To identify a subset of the set of all possible messages the set of allowed values for the parts of a message should be restricted.

A type of a message is built from subtypes, which are types themselves. Restricting the set of possible messages by using a constraint is performed by restricting the set of allowed values for the subtypes of the type of the message. A constraint is defined by supplying constraints for each of the subtypes of the type.

Possible restrictions for a type are:

- no restriction;
- subset of specific values;
- one specific value.

Depending on the type different ways to indicate a subset of specific values may exist. Some types may have special values to indicate a restriction.

Sometimes the type to be restricted could be adapted to the needs for restrictions (without changing the set of possible values).

Due to the semantics of the behaviour part of TTCN, the restriction of a type to a subset of specific values could also be expressed by a set of alternatives of constraints with exactly one of the specific values. Because of the possible infiniteness of the set of possible values or the great number of possible values this becomes impractical.

For those parts of a message for which the value is to be used in calculations (and not only to pass to an acknowledging message) probably only a finite set of possible values will exist.

- **Assumption**: a constraint that allows several (but a finite number of) values, are specified as several constraints which allow each of these values one by one.

The use of actual parameters to indicate these values enhances the readability of the test, because the test then stresses what is of importance with respect to the message.

- **Assumption**: for the different constructs, the constraints on the set of values are indicated as follows:

  - constraints for SEQUENCE type by restrictions on subtypes;
  - constraints for SET type by restrictions on subtypes;
  - constraints for CHOICE type by restrictions on subtypes and restriction on the number of alternatives;
  - constraints for SEQUENCE OF type by restrictions on the number of elements and by restrictions on the subtype;

- constraints for SET OF type by restrictions on the number of elements and by restrictions on the subtype;
- constraints for OPTIONAL types by restrictions on the type or the special value OMIT, indicating the absence of the type or the special value ? indicating the presence of any value of type. If the presence of a specific value is necessary, but the absence is also allowed this is expressed using two constraints. One where the absence is mandatory and one where the specific value is present. (indicating a specific value implies the presence of the type);
- constraints for any type by restriction on the type by indicating a set of allowed values of the type or the special value "?" to indicate no restriction (* for optional types).

TTCN offers a lot of constructs which can be used in constraints. Their semantics are mostly based on examples and the application of these constructs in combination is not investigated, therefore only a subset will be considered. For the other constructs transformation rules will be provided.

The use of actual parameters is only a special way of indicating specific values. From this it can be concluded that the use of specific values, the OMIT, the ? and the * are sufficient to express all possible restrictions on messages.

- **Assumption:** in constraints only specific value, valuelist, Omit, AnyOne and AnyOrOmit are used.

In the following pages transformations for the constructs for constraints mentioned in ISO 9646 [2], Part 3 are provided.

First some general transformations for parts of test cases that are no longer considered due to the assumptions made.

- PDU.fieldN refers to a field of a PDU;
- ValueN is used for any value of the correct type;
- VariableN is used for any variable of the correct type;
- T: indicates that the following is a type definition;
- C: indicates that the following is a constraint definition.

1) The use of assignments to indicate values of certain fields:

```
+preamble
     PCO ! PDU       {PDU.field1 := value1}
          + rest1
```

can be transformed into:

```
+preamble
     PCO ! PDU       Constraint(value1)
          + rest1
```

2) The use of the qualifier to check the values of certain fields:

```
+preamble
     PCO ? PDU       [PDU.field1 = value1]
          + rest1
     PCO ? PDU       [PDU.field1 = value2]
          + rest2
     PCO ?OTHERWISE                         FAIL
```

can be changed to:

```
+preamble
     PCO ? PDU       Constraint(value1)
          + rest1
```

```
      PCO ? PDU        Constraint(value2)
            + rest2
      PCO ?OTHERWISE                              FAIL
```

3) If the qualifier is used to deal with an infinite number of values:

```
+preamble
      PCO ? PDU        [PDU.field1 > value1]
            + rest1
      PCO ? PDU        [PDU.field1 <= value2]
            + rest2
      PCO ?OTHERWISE                              FAIL
```

can be changed to:

```
+preamble
      PCO ? PDU        {variable1 := PDU.field1}
           [variable1 > value1]
                 + rest1
           [variable1 <= value2]
                 + rest2
      PCO ?OTHERWISE                              FAIL
```

4) Transformation if valuelist is used:

```
T: a : type                T:  a : type
C: a : (value1, value2)    C1: a : value1
                           C2: a : value2

+preamble                  +preamble
      ?T    C                    ?T     C1
            +rest                       +rest
                                 ?T     C2
                                        +rest
```

The constraint C allows two values for field a. The constraints C1 and C2 allow only one of the values in each constraint.

5) Transformation if range is used:

if finite then in the same way as 4.
else in the same way as 3.

6) Transformation if complement is used:

```
T: a : type ...                         T:  a : type ...
C: a : complement(value1, value2)       C1: a : ( value1, value2)
                                        C2: a : ?

+preamble                               +preamble
      ?T    C                                 ?T     C1     (FAIL)
            +rest                                    +rest
                                              ?T     C2     (PASS)
                                                     +rest
```

The constraint C allows all possible values not equal to value1 or value2. The test case does not deal with a message which contains these values. In the transformed version C1 deals with the values value1 and value2 and C2 deals with all possible values. However, because of the priority for the alternatives in a case where the alternative with C2 shall be executed this can only be if C1 did not match and therefore the forbidden values are not part of the message.

7) Transformation if ifpresent is used:

```
T: a : type OPTIONAL ...          T:  a : type OPTIONAL
C: a : value1 IF_PRESENT ...      C1: a : value1
                                  C2: a : *

+preamble                         +preamble
     ?T    C                           ?T     C1          (PASS)
           +rest                              +rest
                                       ?T     C2          (PASS)
                                              +rest
```

For this transformation the same explanation applies as for transformation 6.

8) Transformation if permutation is used:

```
T: a : SEQUENCE OF type               T: a : SET OF type
C: a : { perm(value1, value2) }       C1: a : {value1, value2}
```

This transformation uses the semantics already provided by ASN.1. The semantics of a SET (OF) type includes that any permutation of the fields is allowed.

9) Transformation if length is used (note that this means that it is finite):

```
T: a : stringtype                     T: a : SEQUENCE OF type
C: a : "some*thing"[13]               C1: a : "some????thing"
```

The use of the length indication means that this is information which is available at specification time. Therefore it is only an abbreviation.

10) Transformation if subset is used:

```
T: a : SET OF type              T: a : SET {b : type OPTIONAL,
                                            c : type OPTIONAL,
                                            d : type OPTIONAL }

C: a : subset( {value1,         C1: a : SET{b : value1 IF_PRESENT,
         value2,                            c : value2 IF_PRESENT,
         value3})                           d : value3 IF_PRESENT}
```

This transformation is very straightforward, but note that the use of the ifpresent construct also results in new transformations. Note that the indication of tags have intentionally been left out.

11) Transformation if superset is used:

```
T: a : SET OF type              T: a : SET {b : type,
                                            c : type,
                                            d : type OPTIONAL}

C: a : superset({ value1,       C1 : a : SET{b : value1,
        value2})                             c : value2,
                                             d : * }
```

Notice that an upper limit for the number of elements of the set has to be introduced. Note that the tags have been left out.

12) Transformation if wildcards are used:

```
T: a : stringtype               T: a : SEQUENCE {b : type,
                                                 c : type,
```

```
                                      d : type }

C: a : "f?o" ...           C1 : a : SEQUENCE { b : "f",
                                      c : ?,
                                      d : "o" }
```

This transformation shows that a further refinement of the type can transform wildcards inside values into wildcards instead of values.

B.3.2 Transformation of TTCN data types and values

This subclause provides the details on how to transform a TTCN type definition to an algebraic specification. In the following a set of base data types is identified for which it is assumed that algebraic specifications exist. The main part of this subclause deals with the transformation of ASN.1 data type definitions to algebraic specifications. At the end the transformation of TTCN structured data types, ASP and PDU data types is described which resembles most of the transformation of ASN.1.

B.3.2.1 Base data types

TTCN identifies the following predefined data types INTEGER, BOOLEAN, BITSTRING, HEXSTRING, OCTETSTRING and the CharacterString predefined types NumericString, PrintableString, TelexString, VideoString, VisibleString, IA5String, GraphicString and GeneralString. The CharacterString types only differ in their representation and can be reduced to either OCTETSTRING or HEXSTRING.

BOOLEAN, INTEGER, BITSTRING, OCTETSTRING, and HEXSTRING are considered base data types.

TTCN identifies the following predefined operators, some of them applicable to values of any data type defined in a TTCN test suite.

Arithmetic operators: the arithmetic operators are +, -, *, /, and MOD and represent the operations addition, subtraction, multiplication, division and modulo. Operands shall be of INTEGER type (or a derivative of INTEGER type) and the result is also of INTEGER type.

Relational operators: the following relational operators are defined by TTCN: = (equality), < (less than), > (greater than), < > (not equal), > = (greater than or equal), and < = (less than or equal). Operands = and < > can be applied to operands of any compatible type. The others have INTEGER type operands. The result of applying these operators is a Boolean value.

Boolean operators: the Boolean operators are AND, OR, and NOT which stands for conjunction, disjunction and negation. Operands are of type Boolean. The operators result in Boolean value.

Algebraic specifications for these base data types can be found in e.g.. "Fundamentals of algebraic specifications 1" [7], and even in standards and recommendations, e.g. ISO 8807 [8] and CCITT Recommendation Z.100 [1]. It is assumed that algebraic specifications of these data types are available. They are implicitly used in the following.

B.3.2.2 TTCN data types

In TTCN, data types can be defined using either ASN.1 or TTCN defined constructs.

B.3.2.2.1 ASN.1 data type definitions

In a TTCN test suite data type definitions are either explicit or implicit. Explicit by use of the ASN.1 notation, particularly the *Type* and *TypeAssignment* grammar rules from ISO 8824 [9], and implicit by reference to an ASN.1 definition in an ISO Standard or CCITT Recommendation. In the following the transformation of explicit ASN.1 type definitions to algebraic specification is considered. It is assumed that implicit ASN.1 type definitions have been resolved which means that the definitions have been substituted for the references.

In order to get an abstract model the transformation does not deal with coding details, e.g. *basic encoding rules (BER)*, see ISO 8825 [10] or *tagging* and the *ordering of elements in a set*. The definition of the transformation is based mainly on research results reported in "Combining ASN.1 support with the LOTOS language" [11] and "System documentation for the one 2 one translator and user guide" [12].

Optional parameters and fields: as a default parameters of ASPs or fields of PDUs are optional and thus can be omitted in instances of an ASP or a PDU. To deal with absent optional parameters or fields a specific value, denoted by the term ABSENT, is added to every sort in all types. ABSENT can be used instead of any value for an ASP parameter or a PDU field.

Matching values and matching mechanism: constraints denote values for ASPs and PDUs to be sent or to be received. In a constraint for an incoming ASP or PDU, TTCN allows to substitute special symbols (- (omit), ? (any), * (any or omit)) for values. During the matching process of a constraint against the received values these symbols indicate the application of a specific matching mechanism.

The handling of these matching values and the matching mechanism is similar to the handling of optional parameters in ASPs or PDUs. Specific values are added to the sort of every type denoted by the terms OMIT, ANY, and ANYOROMIT.

In an algebraic specification for each of the specific values (including ABSENT) an operation (or constant) is introduced as follows:

```
<ABSENT : -> _sort_>;
<OMIT : -> _sort_>;
<ANY : -> _sort_>;
<ANYOROMIT : -> _sort_>;
```

where ABSENT, etc. is the name of the operation and _sort_ is the sort of the type. The semantics of these operations with respect to equality is defined by the following equations. The general structure of an equation is a triple <V, Eq, e> with V a set of variables, Eq a set of equations which define conditions to be fulfilled in order to apply equation e. Each equation consists of two terms, referred to as left-hand side and a right-hand side. An equation states that both terms denote the same value in an algebra and thus may be exchanged for each other in any context.

```
<{}, {}, <eq(ABSENT, ABSENT), true>>
<{}, {}, <eq(OMIT, OMIT), true>>
<{}, {}, <eq(ANY, ANY), true>>
<{}, {}, <eq(ANYOROMIT, ANYOROMIT), true>>

<{}, {}, <eq(ABSENT, OMIT), true>>

<{x : _sort_}, {<ne(x, ABSENT), true>}, <eq(x, OMIT), false>>
<{x : _sort_}, {<ne(x, ABSENT), true>}, <eq(x, ANY), true>>
<{x : _sort_}, {}, <eq(x, ANYOROMIT), true>>

<{x, x' : _sort_}, {}, <ne(x, x'), not(eq(x, x'))>>
```

The first four equations state that each of the specific values ABSENT, etc. is equal to itself. The fifth equation relates ABSENT in received ASPs or PDUs to OMIT in constraints. The next three equations are related to the matching mechanism as defined by TTCN:

- no values matches the OMIT with the exception of ABSENT used to indicate an absent ASP parameter or PDU field;
- if any value is expected then there should be one different from ABSENT;
- if any or no value is expected then any value matches.

It is assumed that ABSENT is only used for ASPs and PDUs and not in constraints. OMIT, ANY, and ANYOROMIT are to be used only in constraints.

Transformation schema: to guide the transformation this Clause discusses the general schema used throughout the rest of this section to present the transformation from ASN.1 type definitions to algebraic specifications.

A general form of an ASN.1 type definition is:

```
_name_ ::= _type_
```

name is the identifier of the type. _name_ is unique for the test suite. _type_ is either a primitive ASN.1 type or a structured ASN.1 type. Component types of a structured ASN.1 type should have unique names, denoted e.g. by _element1_, _element2_, ... The transformation of such an ASN.1 type definition results in a type specification whose general form is as follows:

```
_name_ = < {_name_},
      { <_name_ : _type_ -> _name_>,
      ...
      <eq : _name_, _name_ -> Bool>,
      <ne : _name_, _name_ -> Bool> },
      { ... } >
```

Every type specification has a unique name associated and consists of three parts: a set of sorts, a set of operations, and a set of equations.

All identifiers to denote the type, the sort, and the operations are inherited from the ASN.1 type definitions.

The specification referred to by _name_ is a partial type specification which exactly specifies one sort, e.g. _name_. However, this partial type definition is part of a type specification which is comprised of all data type definitions. References to sorts, e.g. the Boolean sort, and operations not part of the type are resolved in that predefined type specification.

The use of _name_ as an identifier of a type, a sort, and an operation becomes obvious from the context in which _name_ is used.

It is assumed that all type specifications additionally comprise all operations and equations introduced previously with the necessary changes to identifiers, e.g. change of sort identifiers.

ASN.1 primitive types: the ASN.1 Boolean type, integer type, real type, bitstring type, and octetstring type are transformed to the corresponding algebraic specifications of the base data types (subclause B.3.2.1).

Enumerated type: an enumerated type identifies a set of distinct integer values.

```
_name_ ::= ENUMERATED {
      _element1_ ( _number1_ ),
      ...
      _elementn_ ( _numbern_ ) }
```

```
_name_ = < {_name_},
      { <_element1_ : -> _name_>,
      ...
      <_elementn_ : -> _name_>,
      ... (* all operations for specific values *),
      <eq : _name_, _name_ -> Bool>,
      <ne : _name_, _name_ -> Bool> },
      { <{}, {}, <eq(_element1_, _element1_), true>>,
      <{}, {}, <eq(_element1_, _element2_), false>>,
      ...
      <{}, {}, <eq(_element1_, _elementn_), false>>,
```

```
<{}, {}, <eq(_element2_, _element1_), false>>,
<{}, {}, <eq(_element2_, _element2_), true>>,
...
<{}, {}, <eq(_elementn_, _elementn_), true>>,
... (* all equations for specific values *),
<{x, x' : _name_}, {}, <ne(x,x'), not(eq(x, x'))>> } >
```

- The elements of the enumeration type are referred to by their names in the type specification, and are denoted by the operations *_elementi_*.
- All operations for specific values and all equations for specific values refer to the operations and equations for ABSENT, ...

Null type: A type definition of a NULL types is given by:

```
_name_ ::= NULL
```

```
_name_ = < {_name_},
      { <_name_ : -> _name_ >,
      ... (* all operations for specific values *),
      <eq : _name_, _name_ -> Bool>,
      <ne : _name_, _name_ -> Bool> },
      { ... (* all equations for specific values *),
      <{}, {}, <eq(_name_, _name_), true>>,
      <{x, x' : _name_}, {} , <ne(x, x'), not(eq(x, x'))>> } >
```

ASN.1 constructor types: it is assumed that every element of an ASN.1 constructor type is explicitly named. Furthermore, it is assumed that for all referenced component types algebraic specifications exist.

Sequence types: a sequence type is defined as follows:

```
_name_ ::= SEQUENCE {
      _element1_ _type1_,
      ...
      _elementn_ _typen_ }
```

```
_name_ = < {_name_},
      { <_name_ : _type1_, ..., _typen_ -> _name_>,
      <_element1_ : _name_ -> _type1_>,
      ...
      <_elementn_ : _name_ -> _typen_>,
      ... (* all operations for specific values *),
      <eq : _name_, _name_ -> Bool>,
      <ne : _name_, _name_ -> Bool> },
      { <{x1 : _type1_, ..., xn : _typen_}, {},
            <_element1_(_name_(x1, ..., xn)), x1>>,
      ...
      <{x1 : _type1_, ..., xn : _typen_}, {},
            <_elementn_(_name_(x1, ..., xn)), xn>>,
      ... (* all equations for specific values *)
      <{x1, x1' : _type1_, ..., xn, xn' : _typen_}, {}
            <eq(_name_(x1, ..., xn), _name_(x1', ..., xn')),
            and(eq(xn, xn'), and(...eq(xn, xn')...)>>,
      <{x, y : _name_}, {}, <ne(x, y)), not(eq(x, y))>> } >
```

- The operations *_elementi_* provide access to the component types of the sequence type.

Sequence-of type: a sequence-of type is defined as follows:

```
_name_ ::= SEQUENCE OF _type_


_name_ = < {_name_},
      { <empty : -> _name_>,
      <+ : _type_, _name_ -> _name_>,
      <length : _name_ -> Integer>,
      <get : _name_, Integer -> _type_>,
      ... (* all operations for specific values *),
      <eq : _name_, _name_ -> Bool>,
      <ne : _name_, _name_ -> Bool> },
      { <{}, {}, <length(absent), error>>,
      <{}, {}, <length(empty), 0>>,
      <{t: _type_, s : _name_}, {}, <length(+(t, s)), plus(1, length(s))>>,
      <{i : Integer, s : _name_}, {<i <= 0, true>, <i > length(s), true>},
            <get(y, i), error>>,
      <s : _name_, t: _type_}, {<i = 1, true>},
            <get(+(t, s), i), t>>,
      <{i : Integer, s : _name_, t : _type_},
            {<1 < i, true>, <i <= length(s), true>},
            <get(+(t, s), i), get(s, i -1)>>,
      ... (* all equations for specific values *),
      <{}, {}, <eq(empty, empty), true>>,
      <{t : _type_, s : _name_}, {}, <eq(empty, +(t, s)), false>>,
      <{t : _type_, s : _name_}, {}, <eq(+(t, s), empty), false>>,
      <{t, t' : _type_, s, s' : _name_}, {<eq(t, t'), true>},
            <eq(+(t, s), +(t', s')), eq(s, s')>>,
      <{t, t' : _type_, s, s' : _name_}, {<ne(t, t'), true>},
            <eq(+(t, s), +(t', s')), false>>,
      <{s, s' : _name_}, {}, <ne(s, s'), not(eq(s, s'))>> } >
```

- Operation + creates instances of a sequence-of type.
- Operation *length* returns the number of elements in the sequence-of.
- Operation *extract* returns the ith element from the sequence-of.
- Operation *get* is only defined for integers *1* to *length(s)*. The use of get with an integer less than *0* or greater than *length(s)* results in an *error* indication.

Set type: a set type is defined as follows:

```
_name_ ::= SET {
      _element1_ _type1_,
      ...
      _elementn_ _typen_ }


<name> = < {_name_},
      { <_name_ : _type1_, ..., _typen_ -> _name_>,
      <_element1_ : _name_ -> _type1_>,
      ...
      <_elementn_ : _name_ -> _typen_>,
      ... (* all operations for specific values *),
      <eq : _name_, _name_ -> Bool>,
      <ne : _name_, _name_ -> Bool> },
      { <{x1 : _type1_, ..., xn : _typen_}, {},
            <_element1_(_name_(x1, ..., xn)), x1>>,
      ...
      <{x1 : _type1_, ..., xn : _typen_}, {},
```

```
        <_elementn_(_name_(x1, ..., xn)), xn>>,
    ... (* all equations for specific values *),
    <{x1, x1' : _type1_, ..., xn, xn' : _typen_,}, {}
        <eq(_name_(x1, ..., xn), _name_(x1', ...,  xn')),
        and(eq(xn, xn'), and(...eq(xn, xn')...)>>,
    <{x, y : _name_}, {}, <ne(x, y)), not(eq(x, y))>> } >
```

Set-of type: a set-of type is defined as follows:

```
_name_ ::= SET OF _type_
```

```
<name> = < {_name_},
    { <empty : -> _name_>,
    <+ : _type_, _name_ -> _name_>,
    <length : _name_ -> Integer>,
    <get : _name_, Integer -> _type_>,
    ... (* all operations for specific values *),
    <eq : _name_, _name_ -> Bool>,
    <ne : _name_, _name_ -> Bool> },
    { <{}, {}, <length(absent), error>>,
    <{}, {}, <length(empty), 0>>,
    <{t: _type_, s : _name_}, {}, <length(+(t, s)), plus(1, length(s))>>,
    <{i : Integer, s : _name_}, {<i <= 0, true>, <i > length(s), true>},
        <get(y, i), error>>,
    <s : _name_, t: _type_}, {<i = 1, true>},
        <get(+(t, s), i), t>>,
    <{i : Integer, s : _name_, t : _type_},
        {<1 < i, true>, <i <= length(s), true>},
        <get(+(t, s), i), get(s, i -1)>>,
    ... (* all equations for specific values *),
    <{}, {}, <eq(empty, empty), true>>,
    <{t : _type_, s : _name_}, {}, <eq(empty, +(t, s)), false>>,
    <{t : _type_, s : _name_}, {}, <eq(+(t, s), empty), false>>,
    <{t, t' : _type_, s, s' : _name_}, {<eq(t, t'), true>},
        <eq(+(t, s), +(t', s')), eq(s, s')>>,
    <{t, t' : _type_, s, s' : _name_}, {<ne(t, t'), true>},
        <eq(+(t, s), +(t', s')), false>>,
    <{s, s' : _name_}, {}, <ne(s, s'), not(eq(s, s'))>> } >
```

Choice type: a choice type is defined as follows:

```
_name_ ::= CHOICE {
    _element1_ _type1_,
    ...
    _elementn_ _typen_ }
```

```
_name_ = < {_name_},
    { <is_element1_ : _name_ ->Bool>,
    ...
    <is_elementn_ : _name_ ->Bool>,
    <_element1_ : _type1_ -> _name_>,
    ...
    <_elementn_ : _typen_ -> _name_>,
    <conv_element1_ : _name_ ->  _type1_>,
    ...
    <conv_elementn_ : _name_ ->  _typen_>,
```

```
... (* all operations for specific values *),
<eq : _name_, _name_ -> Bool>,
<ne : _name_, _name_ -> Bool> },
{ <{x1 : _type1_}, {}, <is_element1_(_name_(x1)), true>>,
<{x1 : _type1_}, {}, <is_element2_(_name_(x1)), false>>,
...
<{x1 : _type1_}, {}, <is_elementn_(_name_(x1)), false>>,
...
<{xn : _typen_}, {}, <is_element1_(_name_(xn)), false>>,
<{xn : _typen_}, {}, <is_element2_(_name_(xn)), false>>,
...
<{xn : _typen_}, {}, <is_element1_(_name_(xn)), true>>,
<{x1 : _type1_}, {}, <conv_element1_(_name_(x1)), x1>>,
<{x2 : _type2_}, {}, <conv_element1_(_name_(x2)), error>>,
...
<{xn : _typen_}, {}, <conv_element1_(_name_(xn)), error>>,
...
<{xn : _typen_}, {}, <conv_elementn_(_name_(xn)), xn>>,
... (* all equations for specific values *),
<{ x, x' : _name_}, {},
      <eq(x, x'), or(eq(conv_element1_(x), conv_element1_(x')),
      or(eq(conv_element1_(x), conv_element2_(x')),
      ...
      or(eq(conv_element1_(x), conv_elementn_(x')),
      ...
      or(eq(conv_elementn_(x), conv_elementn_(x'))...)>>,
<{x, x' : _name_}, {}, <ne(x, x'), not(eq(x, x'))>> }>
```

- Operation *is_element_* determines the type of the chosen alternative.
- Operation *conv_elementi_* converts the chosen alternative to its type.

Selection type: a selection type is defined as follows:

```
_name_name_ ::= CHOICE {
      _element1_ _type1_,
      ...
      _elementn_ _typen_ }

_name_ ::= _elementi_ < _name_name_


_name_ = < {_name_},
      { <_name_ : _typei_ -> _name_>,
      ... (* all is_elementi_ and conv_elementi_ operations *)
      ... (* all operations for specific values *),
      <eq : _name_, _name_ -> Bool>,
      <ne : _name_, _name_ -> Bool> },
      { ... (* all is_elementi_ and conv_elementi_ equations *),
      ... (* all equations for specific values *),
      <{x, x' : _typei_}, {}, <eq(_name_(x), _name_(x')), eq(x, x')>>,
      <{x, x' : _name_}, {}, <ne(x, x'), not(eq(x, x'))>> }>
```

Object identifier type: an object identifier type is defined as follows:

name ::= **OBJECT IDENTIFIER**

Values of the type are essentially sequences of integers. An object identifier type is represented thus as a SEQUENCE OF INTEGER type.

name ::= SEQUENCE OF INTEGER

Character string types: ASN.1 identifies the following character string types: NumericString, PrintableString, TeletexString, VideotexString, VisibleString, IA5String, GraphicString, and GeneralString. In addition, the different denotations of time (generalised and universal time) are based on the VisibleString type.

As stated before character string types can be reduced to the OCTETSTRING or HEXSTRING type. It is assumed that these reductions have already been performed.

Object descriptor: in ASN.1 an object descriptor type consists of text. An object descriptor thus may be defined as follows:

name ::= _characterstring_

As the type definition reduces to a character string the transformation for character string types applies.

ASN.1 external types: the external type is not used in TTCN.

B.3.2.2.2 Test suite type definitions

Type definitions are provided by using tables. This subclause uses these predefined tables. The transformation schemata of the previous section apply.

Simple type definitions using tables: TTCN defines that a simple type is defined by a name for the type, its base type, and certain type restrictions. Restrictions are defined to be either a list of distinct values, a range of INTEGER values, or a length range. Lists of distinct values can be specified for all pre-defined or test suite defined data types.

Table B.2: Simple type definitions

Simple Type Definitions		
Type Name	**Type Definition**	**Comments**
... _name_ ...	... _type_ ...	... _freetext_ ...
Detailed Comment: _freetext_		

freetext is any text which serves the purpose of a comment, however the transformation does not deal with comments. The transformation results in an algebraic specification named _name_, and renaming of sort _type_ by _name_ throughout the type specification.

```
_type_ = < {_type_},
      { <_op1_ : ... -> _type_>,
      ...,
      ... (* all operations for specific values *),
      <eq : _type_ , _type_ -> Bool>,
      <ne : _type_, _type_ -> Bool> }
      { ... (* all equations for specific values *),
      <{x, x' : _type_}, {}, <ne(x, x'), not(eq(x, x'))>> } >
```

```
_name_ = < {_name_},
      { <_op1_ : ... -> _name_>,
      ...,
      ... (* all operations for specific values *),
      <eq : _name_ , _name_ -> Bool>,
      <ne : _name_, _name_ -> Bool> }
      { ...
      ... (* all equations for specific values *)
      <{x, x' : _name_}, {}, <ne(x, x'), not(eq(x, x'))>> } >
```

Test suite type definitions using ASN.1: TTCN allows to specify data types in ASN.1. The definition of an ASN.1 type is provided in a tabular form as shown below. References to types within the type definition have to be defined locally in the same table, in other tables, or by reference in the ASN.1 type reference table.

Table B.3: ASN.1 type definition

ASN.1 Type Definition
Type Name: _name_ **Comments:** _freetext_
Type Definition
... _type_ ...
Detailed Comment: _freetext_

It is assumed that all references to other tables have been resolved such that all referenced types are locally defined. The transformation can be performed as described in subclause B.3.2.2.1.

Structured type definitions: a structured type definition may be used as subtype in ASPs, PDUs, and other structured type definitions. The information to be provided with the definition are the name of the structured type and its elements. For each element its name and type is stated (along with an additional attribute restricting the length of string types). All elements are considered to be optional.

The definition of a structured type is given according to the following table format.

Table B.4: Structured type definition

Structured Type Definition		
Type Name: _name_ **Comments:** _freetext_		
Element Name	**Type Definition**	**Comments**
... _elementi_ ...	... _typei_ ...	... _freetexti_ ...
Detailed Comments: _freetext_		

Due to the strong similarities of structured type definition and ASN.1 sequence types the transformation procedure follows the same rules as stated for an ASN.1 sequence type (subclause B.3.2.2.1).

ASP type definitions: *Abstract Service Primitives (ASP)* define events a tester may send or receive during test execution. The definition of an ASP type is to be provided in a table shown below. For each element its name and type is stated (along with an additional attribute restricting the length of string types). All elements are considered to be optional.

Table B.5: ASP type definition

ASP Type Definition		
ASP Name: _name_ **PCO Type:** _pco_ **Comments:** _freetext_		
Parameter Name	**Parameter Type**	**Comment**
... _elementi_ ...	... _typei_ ...	... _freetexti_ ...
Detailed Comments: _freetext_		

The indication of the PCO where the ASPs are expected to occur is of no relevance for the transformation of the ASP type definition to an algebraic specification. It is assumed that the macro symbol is not used.

```
_name_ = < {_name_},
      { <_name_ : _type1_, ..., _typen_ -> _name_>,
      <_element1_ : _name_ -> _type1_>,
      ...
      <_elementn_ : _name_ -> _typen_>,
      ... (* all operations for specific values *),
      <eq : _name_, _name_ -> Bool>,
      <ne : _name_, _name_ -> Bool> },
      { <{x1 : _type1_, ..., xn : _typen_}, {},
            <_element1_(_name_(x1, ..., xn)), x1>>,
```

```
...
<{x1 : _type1_, ..., xn : _typen_}, {},
     <_elementn_(_name_(x1, ..., xn)), xn>>,
... (* all equations for specific values *),
<{x1, x1' : _type1_, ..., xn, xn' : _typen_,}, {}
     <eq(_name_(x1, ..., xn), _name_(x1', ..., xn')),
     and(eq(xn, xn'), and(...eq(xn, xn')...)>>,
<{x, y : _name_}, {}, <neq(x, y)), not(eq(x, y))>> } >
```

TTCN allows to specify ASPs in ASN.1. The definition of an ASN.1 type is provided in a tabular form as shown below. References to types within the type definition have to be defined locally in the same table, in other tables, or by reference in the ASN.1 type reference table.

Table B.6: ASN.1 ASP type definition

ASN.1 ASP Type Definition
ASP Name: _name_ **PCO Type:** _pco_ **Comments:** _freetext_
Type Definition
... _type_ ...
Detailed Comment: _freetext_

It is assumed that all references to other tables have been resolved such that all referenced types are locally defined. The transformation can be performed as described in subclause B.3.2.2.1.

PDU type definitions: *Protocol Data Units (PDU)* define events a tester may send or receive during test execution and are similar to an ASP definitions. Thus, the same transformation procedure apply which is outlined below.

Table B.7: PDU type definition

PDU Type Definition		
PDU Name: _name_ **PCO Type:** _pco_ **Comments:** _freetext_		
Field Name	**Field Type**	**Comments**
... _elementi_ ...	... _typei_ ...	... _freetext_ ...
Detailed Comments: _freetext_		

The indication of the PCO where this PDU is expected to occur is of no relevance for the transformation of the PDU type definition to an algebraic specification. It is assumed that the macro symbol is not used.

```
_name_ = < {_name_},
    { <_name_ : _type1_, ..., _typen_ -> _name_>,
    <_element1_ : _name_ -> _type1_>,
    ...
    <_elementn_ : _name_ -> _typen_>,
    ... (* all operations for specific values *),
    <eq : _name_, _name_ -> Bool>,
    <ne : _name_, _name_ -> Bool> },
    { <{x1 : _type1_, ..., xn : _typen_}, {},
        <_element1_(_name_(x1, ..., xn)), x1>>,
    ...
    <{x1 : _type1_, ..., xn : _typen_}, {},
        <_elementn_(_name_(x1, ..., xn)), xn>>,
    ... (* all equations for specific values *),
    <{x1, x1' : _type1_, ..., xn, xn' : _typen_,}, {}
        <eq(_name_(x1, ..., xn), _name_(x1', ..., xn')),
        and(eq(xn, xn'), and(...eq(xn, xn')...)>>,
    <{x, y : _name_}, {}, <neq(x, y)), not(eq(x, y))>> } >
```

TTCN allows to specify PDUs in ASN.1. The definition of an ASN.1 PDU type is provided in a tabular form as shown below. References to types within the type definition have to be defined locally in the same table, in other tables, or by reference in the ASN.1 type reference table.

Table B.8: ASN.1 PDU type definition

ASN.1 PDU Type Definition
PDU Name: _name_ **PCO Type:** _pco_ **Comments:** _freetext_
Type Definition
... _type_ ...
Detailed Comment: _freetext_

It is assumed that all references to other tables have been resolved such that all referenced types are locally defined. The transformation can be performed as described in subclause B.3.2.2.1.

B.3.2.3 An example - The Verdict type

EXAMPLE: TTCN distinguishes two type of verdicts: preliminary and final verdicts which are denoted *(P)*, *(F)*, *(I)*, *P*, *F*, *I* respectively. A particular variable *(R)* is introduced to hold the current verdict while executing a TTCN test case. Initially *R* is assigned the value *none*. Whenever a verdict is coded in the *Verdict* column of a *Dynamic Behaviour Description* table the value of *R* is updated according to the following rules: if the verdict coded is a preliminary verdict the result of evaluating the coded verdict against *R* is determined according to the following table:

Table B.9: Calculation of the variable R

Current value of R	Entry in verdict column		
	(P)	**(I)**	**(F)**
none	P	I	F
P	P	I	F
I	I	I	F
F	F	F	F

If the verdict coded is a final verdict assignment the result of evaluating the coded verdict against *R* is determined according to the following table:

Table B.10: Calculation of the final verdict R

Current value of R	Entry in verdict column			
	P	**I**	**F**	**R**
none	P	I	F	error
P	P	I	F	P
I	error	I	F	I
F	error	error	F	F

The transformation to an algebraic specification of the Verdict type has to identify the sort and the operations to represent and calculate values and provide a representation of the above introduced rules to calculate the value of *R*. A possible solution is indicated in the following algebraic specification. The sort is denoted *Verdict*. For every possible verdict *((P), (F), (I), P, F, I,* and *none)* a constant is introduced which are denoted *preP, preF, preI, P, F, I* and *none*. The operation *c* has two arguments of sort *Verdict* and returns a term of sort *Verdict*. The semantics of operation *c* as determined by tables B.9 and B.10 and is defined in the equations of the specification.

```
Verdict = < {Verdict},
    { <none : -> Verdict>, <preP : -> Verdict>, <preI : -> Verdict>,
    <preF : -> Verdict>, <P : -> Verdict>, <I : -> Verdict>, <F -> Verdict>,
    <error : -> Verdict> }, <c : Verdict, Verdict -> Verdict>,
    { <{}, {}, <c(none, preP), preP>>, <{}, {}, <c(none, preI), preI>>,
    <{}, {}, <c(none, preF), preF>>, <{}, {}, <c(preP, preP), preP>>,
    <{}, {}, <c(preP, preI), preI>>, <{}, {}, <c(preP, preF), PreF>>,
    <{}, {}, <c(preI, preP), preI>>, <{}, {}, <c(preI, preI), preI>>,
    <{}, {}, <c(preI, preF), preF>>, <{}, {}, <c(preF, preP), preF>>,
    <{}, {}, <c(preF, preI), preF>>, <{}, {}, <c(preF, preF), preF>>,
    <{}, {}, <c(none, P), P>>, <{}, {}, <c(none, I), I>>,
    <{}, {}, <c(none, F), F>>, <{}, {}, <c(none, none), error>>,
    <{}, {}, <c(preP, P), P>>, <{}, {}, <c(preP, I), I>>,
    <{}, {}, <c(preP, F), F>>, <{}, {}, <c(preP, none), P>>,
    <{}, {}, <c(preI, P), error>>, <{}, {}, <c(preI, I), I>>,
    <{}, {}, <c(preI, F), F>>, <{}, {}, <c(preI, none), I>>,
    <{}, {}, <c(preF, P), error>>, <{}, {}, <c(preF, I), I>>,
    <{}, {}, <c(preF, F), F>>, <{}, {}, <c(preF, none), F>> }
```

B.3.3 The range for TTCN

This subclause provides the details to derive the range (Clause B.1) of the transformation from a TTCN test suite. In a first step all identifier sets shall be set up. In a second step the storage environment for test suite parameters, test suite constants, test suite variables, and test case variables shall be defined.

For TTCN test suite data types it is assumed that these data types have been transformed as defined in the previous section. The signature $\Sigma = \langle Sort \cup SigSort, OP \cup SigOP \rangle$ (subclause 5.2) is derived from the algebraic specifications of data types as follows:

- the set *Sort* is comprised of all sorts of those algebraic specifications which result from the transformation of a TTCN simple data type, ASN.1 data type, or TTCN structured type and all basic data types;
- the set *SigSort* is comprised of the sorts of all algebraic specifications corresponding to an ASP or a PDU data types;
- the set *OP* is comprised of all operations of all algebraic specifications which generate instances of sorts from *Sort;*
- the set *SigOP* is comprised of all operations which generate instances of sorts from *SigSort* corresponding to ASPs, PDUs or timers (see also the transformation of timer declarations below).

B.3.3.1 Sets of identifiers

For TTCN only one module exists. As a default this module is identified by *mod*. The set of all module identifiers, denoted by *ModId*, thus consists of the single element *mod*.

$$ModId = \{ mod \}$$

PCOs are declared by use of the following table proforma:

Table B.11: PCO declaration

PCO Declaration			
PCO Name	**PCO Type**	**Role**	**Comments**
$_pco_1_$ ... $_pco_i_$	$_pcotype_1_$ ... $_pcotype_i_$	$_pcorole_1_$ ... $_pcorole_i_$	$_freetext_1_$ ... $_freetext_i_$
Detailed Comments: _freetext_			

The information necessary to fill the set *PathId* is derived from the entries in column PCO Name. Every PCO in the TTCN test suite is associated with two paths in the CSR which are referred to by the path identifiers $_pco_{j1}_$ and $_pco_{j2}_$, for $1 \leq j \leq i$. $_pco_{j1}_$ denotes the path which conveys signals from the environment to the abstract tester, and $_pco_{j2}_$ denotes the path which conveys signal from the abstract tester to the environment. The set *PathId* thus is given by:

$$PathId = \{ _pco_{11}_, _pco_{12}_, ..., _pco_{i1}_. _pco_{i2}_ \}, i \geq 1.$$

The identification of the *PCO type* becomes of importance if the set of signals to be conveyed by a route has to be determined. TTCN associates with every ASP and (possibly not all) PDUs a *PCO type*. Thus if it is to deduce the ASPs or PDUs conveyed by a route this information can be obtained by joining the *PCO Declaration* table with all *ASP* and *PDU Declaration* tables over the *PCO type* (see also the definition of r_m below).

The transformation to the CSR is only concerned with one test case which is uniquely identified throughout the TTCN test suite by its test case identifier, denoted by e.g. *tcid*. The set $BPId_{mod}$ of basic process identifiers within module *mod* is given by:

$$BPId_{mod} = \{ tcid \}$$

Due to the fact that only one module exists the set of all basic process identifiers consists of those identifiers defined for module *mod*.

$$BPId = BPId_{mod} = \{ tcid \}$$

A *pid-value* is associated to every process instance. The set of all pid-values for process *tcid* is denoted by:

$$BPPId_{tcid} = \{ pid_{tcid} \}$$

It is a set which consists of exactly one *pid-value* because at any time there exists at most one process instance in the TTCN system. The set of all pid-values thus reduces to:

$$PId = ModPid_{mod} = BPPId_{tcid} = \{ pid_{tcid} \}$$

The set Routes binds the process identifier *tcid* to all paths pco_j, $1 \leq j \leq i$. This enables *tcid* to send to and to receive signals from all paths.

$$Routes = \{ (tcid, pco_1), ..., (tcid, pco_i) \}$$

In TTCN it is recommended to use the following table proformas for the declaration of test suite parameters, test suite constants, test suite variables, and test case variables. For all parameters, constants, and variables sets of identifiers are derived from the corresponding entries in the Parameter Name, Constant Name, and Variable Name columns. These sets are denoted by *TSP, TSC, TSV*, and *TCV* for test suite parameters, constants, variables and test case variables. The sort of every parameter, constant or variable is derivable from the entry in the Type column. Variables, parameters, and constants of the same sort are comprised in the same set. For example,

$$Var_{_type_} = \{ _tsp_{k}_, _tsp_{l}_, tcv_{m}_, _tsv_{n}_ \}$$

is the set of variables of sort *_type_* if $_tsp_{k}_$, $_tsp_{l}_$, $_tcv_{m}_$, $_tsv_{n}_$ are defined to be of type _type_.

Table B.12: Test suite parameter declaration

Test Suite Parameter Declaration			
Parameter Name	**Type**	**PICS/PIXIT Ref**	**Comments**
$_tsp_1_$	$_type_1_$	$_freetext_1_$	$_freetext_1_$
...	...	...	...
$_tsp_j_$	$_type_j_$	$_freetext_j_$	$_freetext_j_$
Detailed Comments: _freetext_			

is transformed to

$$TSP = \{ _tsp_1_, ..., _tsp_j_ \}, j \geq 1$$

Table B.13: Test suite constant declaration

Test Suite Constant Declaration			
Constant Name	**Type**	**Value**	**Comments**
$_tsc_{1}_$	$_type_{1}_$	$_c_{1}_$	$_freetext_{1}_$
...	...	...	...
$_tsc_{k}_$	$_type_{k}_$	$_c_{k}_$	$_freetext_{k}_$
Detailed Comments: _freetext_			

is transformed to

$$TSC = \{ _tsc_{1}_, ..., _tsc_{k}_ \}, k \geq 1$$

Table B.14: Test suite variable declaration

Test Suite Variable Declaration			
Variable Name	**Type**	**Value**	**Comments**
$_tsv_{1}_$	$_type_{1}_$	$_v_{1}_$	$_freetext_{1}_$
...	...	...	...
$_tsv_{m}_$	$_type_{m}_$	$_v_{m}_$	$_freetext_{m}_$
Detailed Comments: _freetext_			

is transformed to

$$TSV = \{ _tsv_{1}_, ..., _tsv_{m}_ \}, m \geq 1$$

Table B.15: Test case variable declaration

Test Case Variable Declaration			
Variable Name	**Type**	**Value**	**Comments**
$_tcv_{1}_$	$_type_{1}_$	$_v'_{1}_$	$_freetext_{1}_$
...	...	...	...
$_tcv_{n}_$	$_type_{n}_$	$_v'_{n}_$	$_freetext_{n}_$
Detailed Comments: _freetext_			

is transformed to

$$TCV = \{ _tcv_{1}_, ..., _tcv_{n}_ \}, n \geq 1$$

The set *Var* of all variables is the union of the sets *TSP*, *TSC*, *TSV*, *TCV* and a set Var_{Aux} of auxiliary variables, e.g. *SendObject* and *ReceivedObject* and the variable *R* which holds the last verdict assignment ISO 9646 [2], Part 3, Annex B. Particularly, Var_{Aux} contains variables which serve as formal parameters to constraint declarations.

$$Var = TSP \cup TSC \cup TSV \cup TCV \cup Var_{Aux}$$

The sets of data terms and signal terms over variable set *Var* with respect to signature Σ' can be derived as outlined in subclause 5.2 (Data and signals). The notation as introduced in subclause 5.2 is adopted in the following to denote terms, set of terms, etc..

For each ASP and PDU declaration in a TTCN test suite it can be assumed that there is a set $SigTerm_{ss}(Var)$ in the CSR with *ss* denoting the sort of the ASP or PDU. For every ASP sort, denoted by $asp_1, ..., asp_p, p \geq 1$, and PDU sort, denoted by $pdu_1, ..., pdu_q, q \geq 1$, a set of constraints exists, denoted by $SigTerm_{aspi}, 1 \leq i \leq p$ and $SigTerm_{pdui}, 1 \leq i \leq q$. Let $SigTerm^A{}_{aspi}$, $1 \leq i \leq p$ and $SigTerm^A{}_{pdui}$, $1 \leq i \leq q$, denote the interpretation of signal terms from the sets $SigTerm_{aspi}, 1 \leq i \leq p$ and $SigTerm_{pdui}, 1 \leq i \leq q$.

The information about which PCO is capable to convey which ASPs or PDUs is stated with the declaration of the ASP or the PDU type. The set $r_m(tcid, pco_j)$ denotes all signal interpretation which can be carried by the route $(tcid, pco_j)$.

$r_m(tcid, pco_{j1}) = \{$
$\{ SigTerm^A{}_{aspk} \mid asp_k$ *is defined over _pcotype_ and* pco_j *is of PCO type _pcotype_* $\}$,
$\{ SigTerm^A{}_{pdul} \mid pdu_l$ *is defined over _pcotype_ and* pco_j *is of PCO type _pcotype_* $\}$,
$\{ \perp \} \}$
$r_m(tcid, pco_{j2}) = \{$
$\{ SigTerm^A{}_{aspk} \mid asp_k$ *is defined over _pcotype_ and* pco_j *is of PCO type _pcotype_* $\}$,
$\{ SigTerm^A{}_{pdul} \mid pdu_l$ *is defined over _pcotype_ and* pco_j *is of PCO type _pcotype_* $\}$,
$\}$
for all pco_j, $1 \leq j \leq i$, asp_k, $1 \leq k \leq p$, and pdu_l, $1 \leq l \leq q$.

The use of $\perp$ in the definition of $r_m(tcid, pco_{j1})$ is due to the fact that an abstract tester may receive an unforeseen signal.

B.3.3.2 Transformation of values, expressions, timers, and constraints

For literal values the transformation starts out with determining the type of the value. From this information the corresponding algebraic specification, identified by the type, can be deduced. The transformation of the value in a term is done by constructing a term, which should represent the value, from the constants and constructor operations of the algebraic specification.

If the value is given for a structured type then it is assumed that the values assigned to the elements of the structured type have already been transformed to a term of appropriate sort. To construct the term for the structured type then the constructor operation defined for that type has only to be applied. This also applies for ASN.1 values.

In a recursive descend all operators and values of an expression are transformed to their corresponding representation as terms in the CSR. This can be accomplished as follows:

- the result type of the expression can be deduced either from the parameter or field of an ASP or a PDU, if used in the context of a constraint declaration, or from the left hand side of an assignment, if used in an assignment. This information is used to determine the corresponding algebraic specification to search for the operations to be used for the transformation of operators and sub-expression on this level;

- from the operations found the type of the sub-expressions is determined. If a sub-expression reduces to a value then the transformation for values can be applied. If a sub-expression is composed of other expressions, then this procedure can be re-iterated for each of the sub-expressions.

In the CSR a timer is modelled as a signal. For each timer declaration in a TTCN test suite (see table proforma below) a signal sort and a signal declaration is assumed to be in *SigSort* and *SigOP*.

Table B.16: Timer declaration

Timer Declaration			
Timer Name	**Duration**	**Unit**	**Comments**
$_tid_{1}_$ ... $_tid_{o}_$	$_dur_{1}_$ ... $_dur_{o}_$	$_unit_{1}_$ ... $_unit_{o}_$	$_freetext_{1}_$ ... $_freetext_{o}_$
Detailed Comments: _freetext_			

The timer names are assumed to define a sort in *SigSort*

$$SigSort \supset \{ _tid_{1}_, ..., _tid_{o}_ \}$$

and an operation in *SigOP*

$$SigOP \supset \{ <_tid_{1}_ : -> _tid_{1}_>, ..., <_tid_{o}_ : -> _tid_{o}_> \}.$$

Constraints define values for ASPs and PDUs to be sent and received. Constraints are declared by using the following table proforma:

Table B.17: Constraint declaration

Constraint Declaration		
Constraint Name: $_consid_ (_par_{1}_ : _type_{1}_, ..., _par_{m}_ : _type_{m}_)$ **Type:** _type_ **Derivation Path:** _path_ **Comments:** _freetext_		
Parameter or Field Name	**Parameter or Field Value**	**Comment**
... $_parorfield_{i}_$ ...	... $_value_{i}_$ ...	... $_freetext_{i}_$ ...
Detailed Comment: _freetext_		

In the CSR ASP and PDU constraints are signal terms. In general constraint declarations are signal terms and not signal ground terms due to the parameterisation of constraint declarations. In the following the details on how to transform a constraint declaration to a signal term are provided.

Constraints use various combinations to specify values for parameters or fields of an ASP or a PDU, e.g. literal values, data object references, expressions, special matching mechanisms, or references to other constraints. Without loss of generality it can be assumed that all values for parameters and fields are given as data terms or signal terms of the appropriate sort (dependent on the type of the parameter or field). For example:

- a literal value of type *Integer* is denoted by a term of sort *Integer;*
- data object references, i.e. test suite parameters or constants, are of specific sort and are to be found in set *Var*. Every element of *Var* is a term of specific sort (subclause 5.2);
- an expression is a term representing a particular value;
- special matching mechanisms are mapped to terms (subclause B.3.2). Note that only value list, omit, any, or any or omit are considered;

- references to constraints are substituted by the signal terms representing the referenced constraints.

In order to generate a term of sort _type_ the *generator* or *constructor* operation, as defined in the algebraic specification for type _type_, is applied to the terms specified for the parameters and fields. This application of a generator or constructor operation results in a signal term.

B.3.3.3 Storage environment

The storage environment is determined by the partial functions ε and ρ. Their initialisation is given below:

$$\varepsilon' = \varepsilon_{\perp}\begin{bmatrix} self \mapsto l_1 \\ parent \mapsto l_2 \\ offspring \mapsto l_3 \\ sender \mapsto l_4 \\ now \mapsto l_5 \\ v \mapsto l_v, \quad v \in Var_{Aux} \\ active_{si} \mapsto l_{si}, \quad si \in SigTerm^A \\ tsp_i \mapsto l'_i, \quad i = 1,\dots,j \\ tsc_i \mapsto l''_i, \quad i = 1,\dots,k \\ tsv_i \mapsto l'''_i, \quad i = 1,\dots,m \\ tcs_i \mapsto l''''_i, \quad i = 1,\dots,n \end{bmatrix}$$

ε maps all elements of set *Var* to storage locations.

$$\rho = \rho_{\perp}\begin{bmatrix} l_1 \mapsto tcid \\ l_2 \mapsto \perp \\ l_5 \mapsto time \\ l_v \mapsto \perp, \quad v \in Var_{Aux} \\ l_{si} \mapsto false, \quad si \in SigTerm^A \\ l'_i \mapsto v_i, \quad i = 1,\dots,j \\ l''_i \mapsto c_i, \quad i = 1,\dots,k \\ l'''_i \mapsto \perp \vee v'_i, \quad i = 1,\dots,m \\ l''''_i \mapsto \perp \vee v''_i, \quad i = 1,\dots,n \end{bmatrix}$$

Initially, the variables $v \in Var_{Aux}$ (storage locations l_v for all $v \in Var_{Aux}$) are assigned the undefined value ($\perp$). The values (values v_i, $1 \leq i \leq j$) assigned to test suite parameters (storage location l'_i, $1 \leq i \leq j$) are deduced from the PICS and PIXIT and are assumed to be available at initialisation time. The values (values c_i, $1 \leq i \leq j$) assigned to test suite constants (storage location l''_i, $1 \leq i \leq k$), are deduced from the TTCN test suite. The test suite variables and test case variables (storage location l'''_i, $1 \leq i \leq m$, and l''''_i, $1 \leq i \leq n$) are assigned either the initial value defined in the TTCN test suite (values v'_i, $1 \leq i \leq m$, and v''_i, $1 \leq i \leq n$ or assigned the undefined value ($\perp$).

B.3.4 Transformation of TTCN behaviour descriptions

The operational semantics of TTCN ISO 9646 [2], Part 3, Annex B are defined with respect to an evaluation model, called A*bstract Evaluation Tree (AET)*. An AET is a tree-like representation of a test case with all defaults attached and repeats substituted by goto's and tree attachments.

This subclause aims at a description of the transformation of AETs to basic processes. In the following a definition of an AET is provided (subclause B.3.4.1). Emphasis is put on the tree structure and, what is essential with respect to the definition of the operational semantics of TTCN, the ordering of TTCN statements which is reflected in the ordering of edges from a node. As part of the description of the transformation the subset of the BPA used is identified (subclause B.3.4.2). The transformation is defined recursively over the structure of an AET and is defined in subclause B.3.4.3.

B.3.4.1 Abstract Evaluation Trees (AET)

An AET is a tree like representation of a TTCN test case. A node of an AET denotes a level of indentation in a dynamic behaviour description of a test case where the level of the node in the AET equals the level of indentation. Edges are labelled with TTCN statements. All labels of edges from a node corresponds to a set of alternatives. The set of AETs is defined as follows:

Definition: the set AET of *Abstract Evaluation Trees* is the set of rooted trees such that $t \in$ AET implies:

1) every edge is labelled by a TTCN statement;
2) the level of a node equals the level of indentation in the dynamic behaviour description, where the level of a node is defined as:
   2.1) the level of the root of an AET is zero;
   2.2) the level of a node, which is not the root, is the level of its predecessor plus one;
3) the left-to-right order of labels of edges from a node n on level m corresponds to the top-to-bottom order of the set of alternatives on the level of indentation m.

A $t \in$ AET is denoted by $t = (a_1\ t_1, ..., a_m\ t_m)$.

B.3.4.2 Identification of the subset of the Basic Process Algebra (BPA)

The transformation ranges over the set $\varepsilon_{TTCN} \cup \varepsilon_{Common}$ of TTCN events and common events and the operators*;* and $\oplus$ (see subclauses 5.3.2 and 5.3.3).

B.3.4.3 Definition of the transformation of AETs

This section introduces a function *trans* mapping a $t \in$ AET to a basic process. *trans* is recursively defined over the structure of an AET.

A TTCN test case may attach test steps, defaults, or local trees. Attached tree may be expanded as described in ISO 9646 [2], Part 3. In general, however, it is not possible in advance to expand all attached trees due to recursive attachments. Therefore, it is assumed that the procedure as defined in Annex B, subclause B.4.4 is applied to handle attachments. If during evaluation of the behaviour of a TTCN test case on a certain level a tree attachment is encountered then the expansion and the transformation (as defined hereafter) is performed.

The transformation *trans* of an AET $t = (a_1\ t_1, ..., a_m\ t_m)$ to a basic process is defined as

$$trans(t) = \#a_1\# \text{ ";" } trans(t_1) \text{ "}\oplus\text{" } ... \text{ "}\oplus\text{" } \#a_m\# \text{ ";" } trans(t_m)$$

with #...# as described below.

Referring to the TTCN grammar (see ISO 9646 [2], Part 3, Annex A) an a_i, $1 \leq i \leq m$, stands for a behaviour line which may have one of the following forms:

Table B.18: TTCN statement lines

Test Case Dynamic Behaviour					
Test Case Name: tcid Group: group Purpose: freetext Default: Comments:					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**C.**
1		pco ! aspname [boolexp] (ass_1, ..., ass_m) top_1, ... top_n	cref	ver	
2		pco ? aspname [boolexp] (ass_1, ..., ass_m) top_1, ... top_n	cref	ver	
3		pco ? OTHERWISE [boolexp] (ass_1, ..., ass_m) top_1, ... top_n		fail	
4		? TIMEOUT tid [boolexp] (ass_1, ..., ass_m) top_1, ... top_n		ver	
5		[boolexp] (ass_1, ..., ass_m) top_1, ... top_n		ver	
6		(ass_1, ..., ass_m) top_1, ... top_n		ver	
7		top_1, ... top_n		ver	
8		-> label			
Detailed Comments: freetext					

#...# is defined as follows:

(1) #pco ! aspname [boolexp] (ass_1, ..., ass_m) top_1, ... top_n cref ver# =
[bt] ";" output (s) via p ";" #(ass_1, ..., ass_m) top_1, ... top_n cref ver#
where

[bt] is a data term of sort Boolean which result from the transformation of Boolean expression *boolexp*.

s is a signal ground term which results from the transformation of *cref* according to the following procedure:

- if *cref* is a constraint reference without parameters then *s* is the signal ground term which results from the transformation of the referenced constraint declaration as described in subclause B.3.3;
- if *cref* is a constraint reference with parameters which are variables then *s* is the signal term which result from *cref* by substitution of actual for formal parameters in the constraint declaration and transformation of the constraint declaration as described in subclause B.3.3;
- if *cref* is a constraint reference with parameters which are variables and constraint references then *s* is the signal term which result from *cref* by substitution of actual for formal parameters related to variables in the constraint declaration, by processing the constraint references which are passed as an actual parameter due to these rules, and transformation of the constraint declaration as described in subclause B.3.3.

  Even when actual parameters are passed in a constraint reference, *s* equals some signal ground term *s'* due to the fact that the variables are bound to values.

p is the path identifier derived from PCO *pco* (subclause B.3.3).

and

#(ass_1, ..., ass_m) top_1, ... top_n cref ver# as defined in (6).

(2) #pco ? aspname [boolexp] (ass_1, ..., ass_m) top_1, ... top_n cref ver# =
input(s) via p [bt] ";" #(ass_1, ..., ass_m) top_1, ... top_n ver#

where

s is a signal ground term which results from the transformation of cref according to the following procedure:

- if *cref* is a constraint reference without parameters then *s* is the signal ground term which results from the transformation of the referenced constraint declaration as described in subclause B.3.3;
- if *cref* is a constraint reference with parameters which are variables then *s* is the signal term which result from *cref* by substitution of actual for formal parameters in the constraint declaration and transformation of the constraint declaration as described in subclause B.3.3;
- if *cref* is a constraint reference with parameters which are variables and constraint references then *s* is the signal term which result from *cref* by substitution of actual for formal parameters related to variables in the constraint declaration, by processing the constraint references which are passed as an actual parameter due to these rules, and transformation of the constraint declaration as described in subclause B.3.3.
  Even when actual parameters are passed in a constraint reference, *s* equals some signal ground term *s'* due to the fact that the variables are bound to values.

p is the path identifier derived from PCO *pco* (subclause B.3.3).

[bt] is a data term of sort Boolean which result from the transformation of Boolean expression *boolexp*.

and

#(ass_1, ..., ass_m) top_1, ... top_n ver# as defined in (6).

(3) #pco ? OTHERWISE [boolexp] (ass_1, ..., ass_m) top_1, ... top_n fail# =
otherwise p [bt] ";" #(ass_1, ..., ass_m) top_1, ..., top_n fail#

where

p is the path identifier derived from PCO *pco* (Section B.3.3).

[bt] is a data term of sort Boolean which result from the transformation of Boolean expression *boolexp*.

and

#(ass_1, ..., ass_m) top_1, ... top_n ver# as defined in (6).

(4) #? TIMEOUT tid [boolexp] (ass_1, ..., ass_m) top_1, ... top_n ver# =
input(s) [bt] ";" #(ass_1, ..., ass_m) top_1, ... top_n ver#

where

s the signal term representing the timer *tid* (Section B.3.3).

[bt] is a data term of sort Boolean which result from the transformation of Boolean expression *boolexp*.

and

#(ass_1, ..., ass_m) top_1, ... top_n ver# as defined in (6).

(5) #[boolexp] (ass_1, ..., ass_m) top_1, ... top_n ver# =
[bt] ";" #(ass_1, ..., ass_m) top_1, ... top_n ver#

where

[bt] is a data term of sort Boolean which result from the transformation of Boolean expression *boolexp*,

and

#(ass_1, ..., ass_m) top_1, ... top_n ver# as defined in (6).

(6) #(ass_1, ..., ass_m) top_1, ... top_n ver# =
ass_1 ";" ... ";" ass_m ";" #top_1, ... top_n ver#

where

ass_i = x := t for ass_i = x := exp
for all i, $1 \leq i \leq n$ with
$x \in Var$.
t a data term of the same sort as variable x which results from the transformation of expression exp.

and $\#top_1, \ldots top_n$ ver5# as defined in (7).

(7) $\#top_1, \ldots top_n$ ver# =
top_1 ";" ... ";" top_n ";" R := c(R, ver)
where

$$top_i = \begin{cases} set(t,s) & \text{if } top_i = \text{"Start tid (tv)"} \\ reset(s) & \text{if } top_i = \text{"Cancel tid"} \\ read(x,s) & \text{if } top_i = \text{"Read tid (x)"} \end{cases}$$

for all i, $1 \leq i \leq n$ with
s the signal term representing the timer *tid* (subclause B.3.3).
t a term of sort *Time* denoting the absolute point in time when the timer is to expire. *tv* is either the default assignment made in the timer declaration or a value explicitly set. *t* is computed by adding the interpretations of terms *tv'* and *now* where *tv'* is a data term of sort Time which results from the transformation of expression *tv*.
x a variable of sort *Time*.

and
c as defined in subclause B.3.2 and *ver* the transformation of *ver* (Section B.3.2).

(8) #-> label# = P, where P is a basic process name. P identifies uniquely a node in the AET which is ancestor to the parent node of -> *label* (refer also to Clause B.2 "Transformation of SDL").

Some constructs of a TTCN statements are optional. Therefore, the transformation rules are to be interpreted as applying only to those constructs which actually occurs in a TTCN statement.

Boolean expressions serve the purpose of a guard. In the transformation this has been made explicit. Boolean expressions are considered to be part of a send, receive, OTHERWISE, TIMEOUT event or assignment list.

The execution of a test case terminates if a final verdict is assigned. In this case the transformation adds a *";" stop ; nil* after update to variable *R*.

Comments: ISO 9646 [ ], Part 3, Annex B defines that if an alternative has been selected for execution the components are executed in sequence. This motivates the use of the sequence ;-operator in the transformation of TTCN statements.

ISO 9646 [ ], Part 3, Annex B defines that "processing of a set of alternatives is instantaneous, i.e. the status of any events cannot change during the matching process" (cited from ISO 9646 [2], Part 3). The semantics defined for the ⊕-operator (subclause 5.6) performs the evaluation instantaneous. To be more precise evaluation of a set of alternatives and execution of an event coincide.

Annex C: Examples

C.1 Introduction

In the following Clauses some examples for the transformations from SDL and TTCN to the CSR are presented. The examples are provided in order to show as much as possible different aspects of the transformation and therefore are not useful as an SDL specification and/or a TTCN testcase. It is very difficult to find a real specification and/or test which uses all the different events and still remains manageable.

The examples are divided in a data part and a behaviour part. The data part will be transformed to a Σ-algebra and the behaviour part is transformed to a basic process with possible references to terms over the Σ-algebra.

C.2 SDL to CSR

C.2.1 A simple SDL example

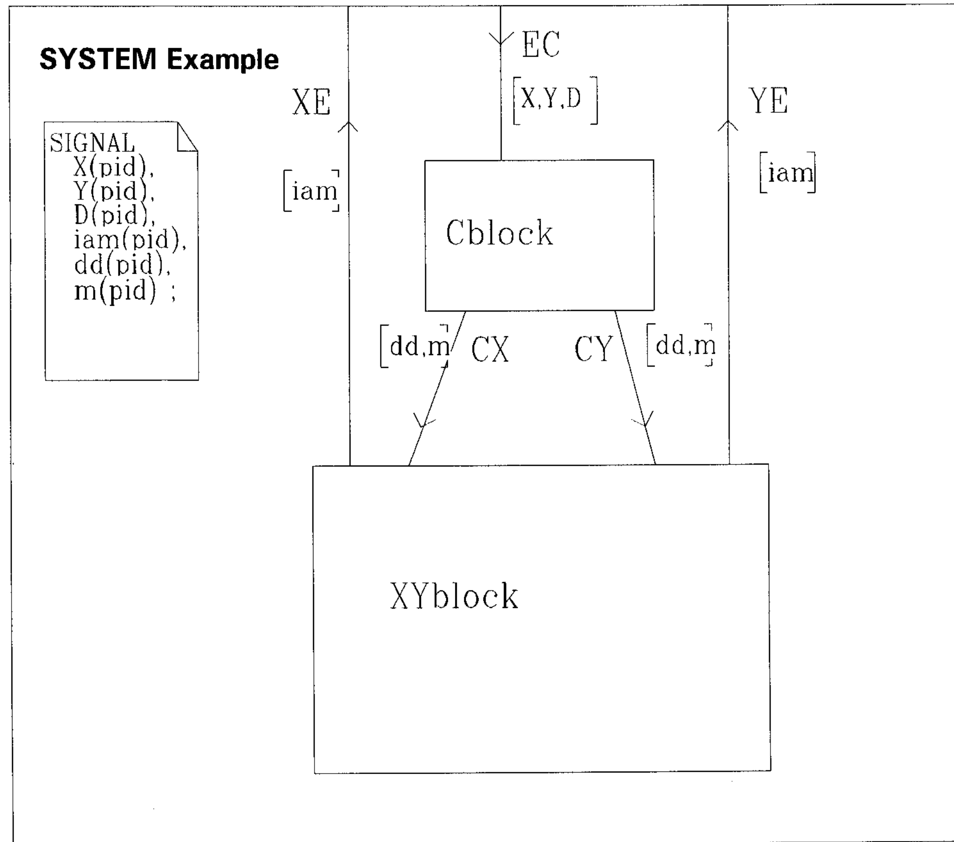

Figure C.1: An SDL system

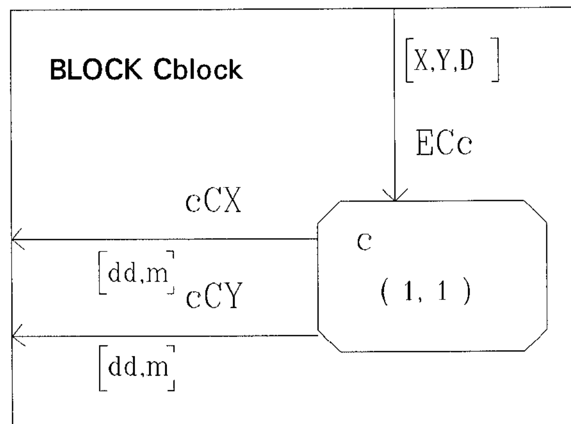

Figure C.2: The Cblock

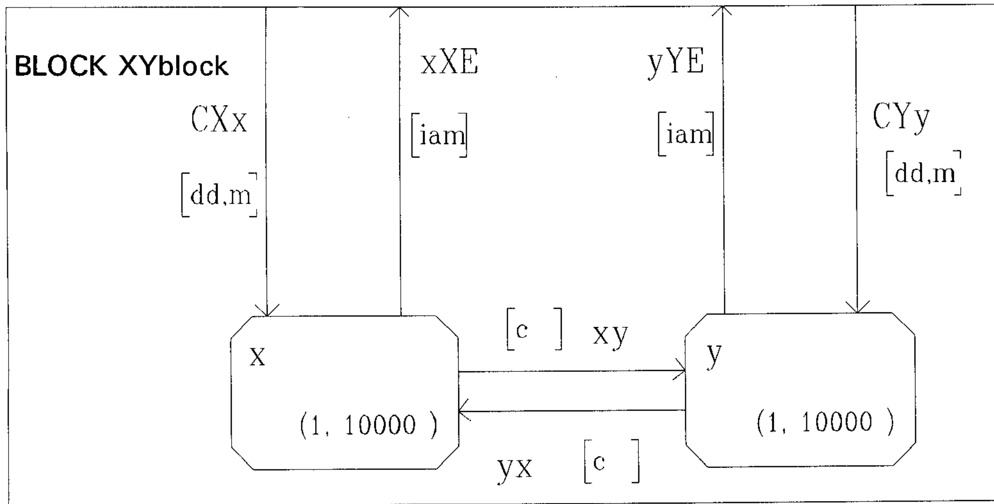

Figure C.3: The XYblock

PROCESS c
DCL x,y,z pid ;
BOTH
X(x)
Y(y)
D(z)
ONLYY
ONLYX
dd TO z
BOTH
ONLYX
X(x)
D(z)
Y
p(x,y)
dd TO z
BOTH
ONLYX
ONLYY
Y(y)
D(z)
X
p(x,y)
dd TO z
BOTH
ONLYY

Figure C.4: The process c

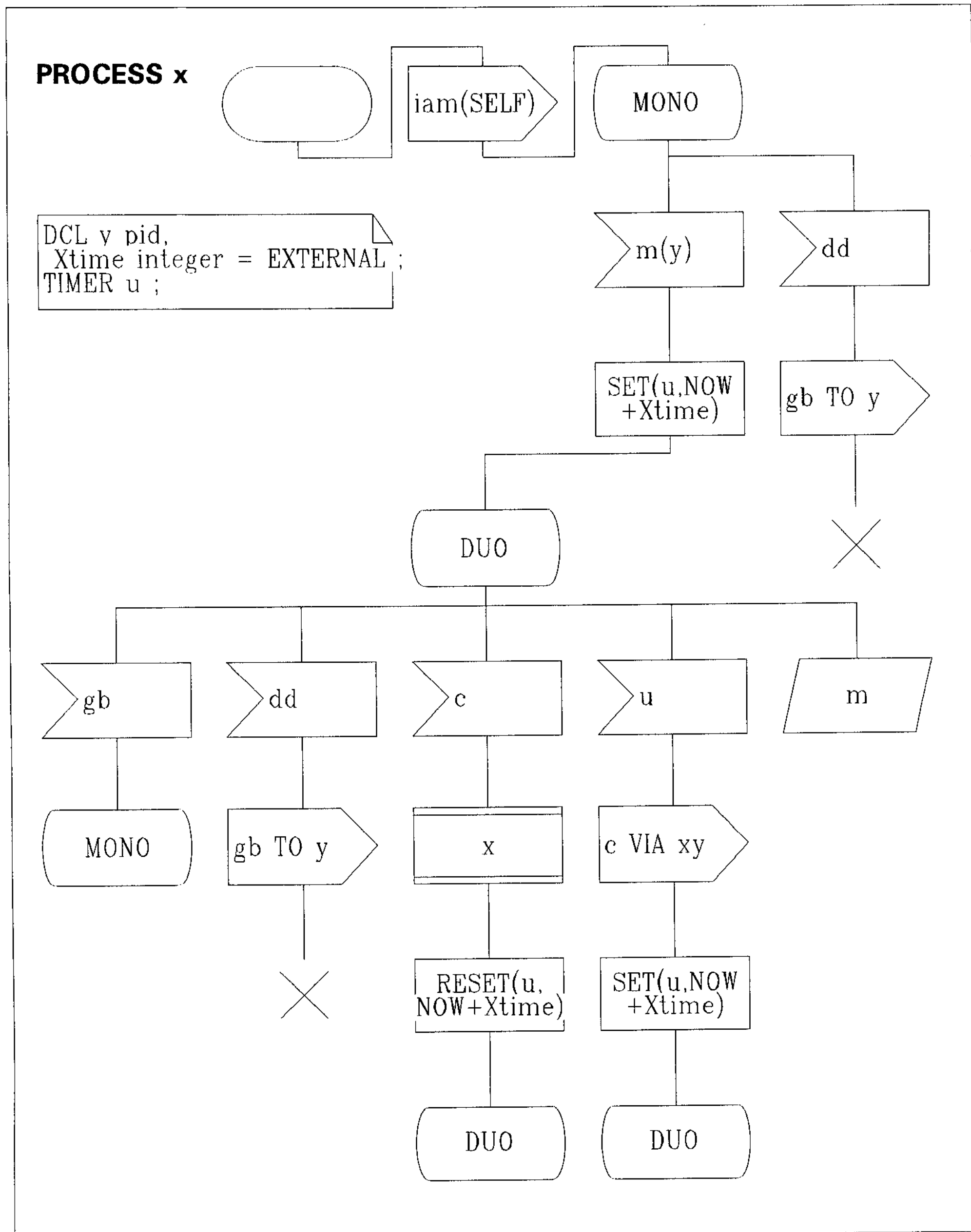

Figure C.5: The process x

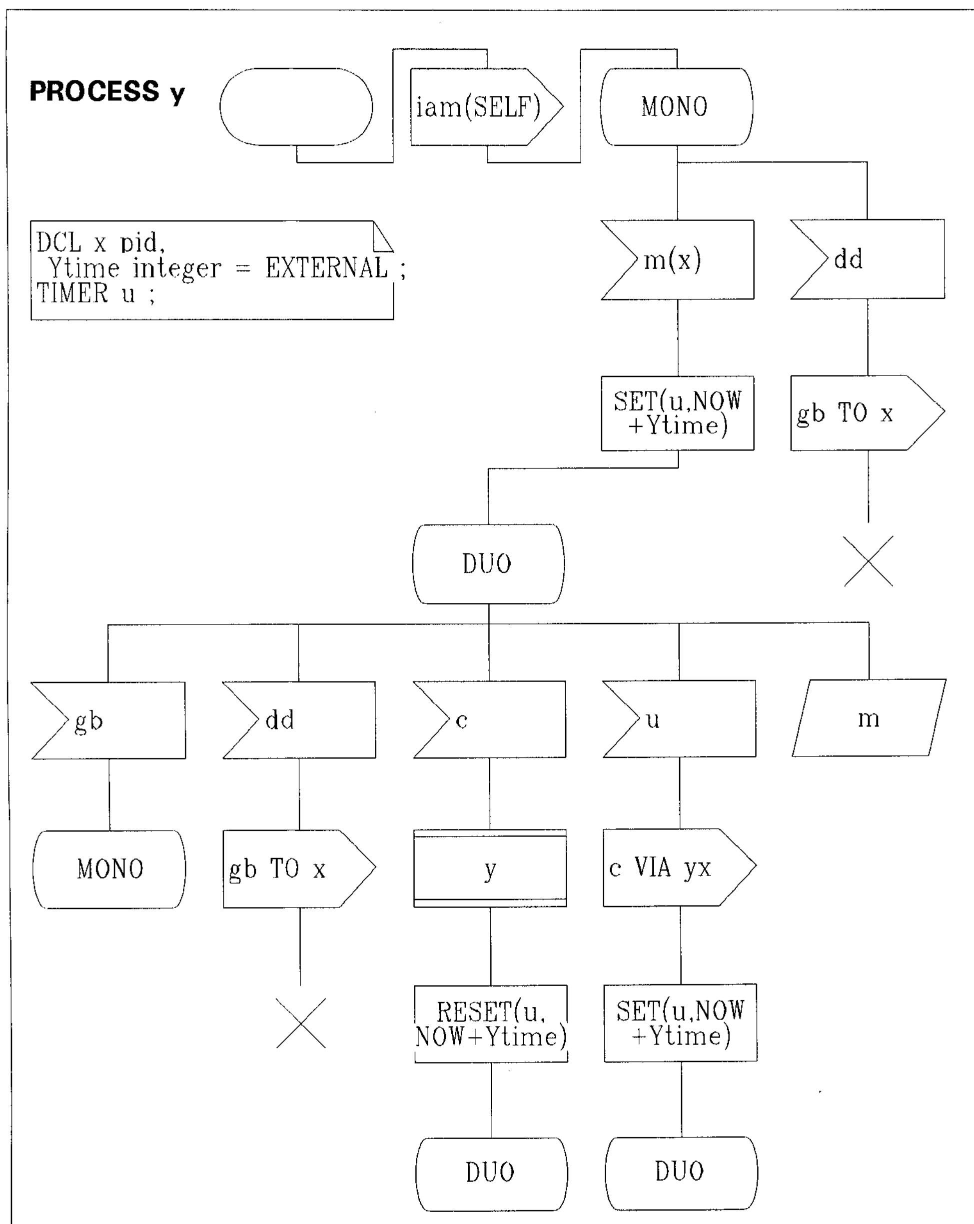

Figure C.6: The process y

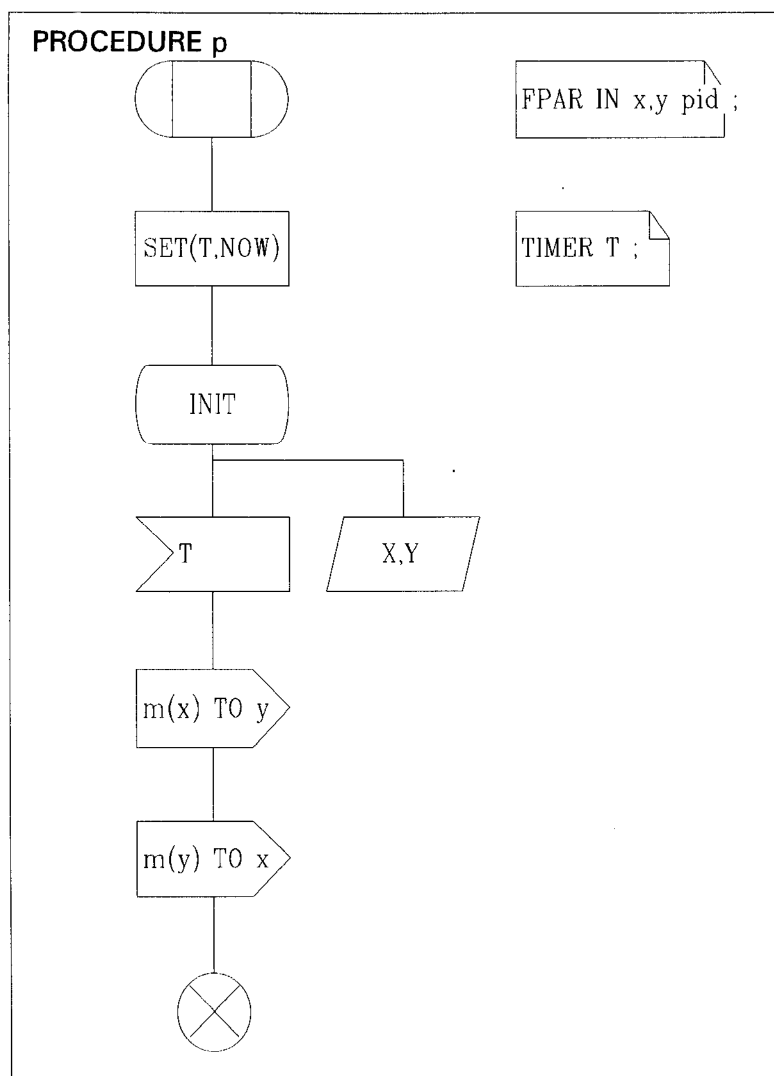

Figure C.7: The procedure p

C.2.2 Its CSR representation

As indicated in Annex B, Clause B.1, the CSR representation consists of several entities. In order to represent an SDL system in the CSR it is sufficient to provide the contents/ values of these entities.

PathId := { EC, XE, YE, CX, CY }
ModId := { Cblock, XYblock }
BPId := { c, x, y }
PId := { 1, 2, .. , 10001, 10002, .. ,20001 }
$BPId_{Cblock}$:= { c }
$BPId_{XYblock}$:= { x, y }
$BPPId_c$:= { 1 }
$BPPId_x$:= { 2, .. , 10001 }
$BPPId_y$:= { 10002, .. , 20001 }
$ModPId_{Cblock}$:= { 1 }
$ModPId_{XYblock}$:= { 2, .. , 20001 }
P_c := P_{both} ; nil, where

P_{both} := (input(X(x)) ; P_{onlyy}) + (input(Y(y)) ; P_{onlyx}) + input(D(z)) ; output(dd) to z ; P_{both}

P_{onlyx} := (input(X(x) ; call(p(x,y)) ; P_{both}) + (input(D(z) ; output(dd) to z ; P_{onlyx}) + save(Y) ; P_{onlyx}

P_{onlyy} := (input(Y(y) ; call(p(x,y)) ; P_{both}) + (input(D(z) ; output(dd) to z ; P_{onlyy}) + save(X) ; P_{onlyy}

P_x := output(iam(SELF)) ; P_{monox} ; nil, where

P_{monox} := (input(m(y)) ; set(u,now+Xtime) ; P_{duox}) + (input(dd) ; output(gb) to y ; stop)
P_{duox} := (input(gb) ; P_{monox}) + (input(dd) ; output(gb) to y ; stop) +
(input(c) ; create(x) ; reset(u,now+Xtime) ; P_{duox}) +
(input(u) ; output(c) via xy ; set(u,now+Xtime) ; P_{duox}) + save(m) ; P_{duox}
P_y := output(iam(SELF)) ; P_{monoy} ; nil, where
P_{monoy} := (input(m(x)) ; set(u,now+Ytime) ; P_{duoy}) + (input(dd) ; output(gb) to x ; stop)
P_{duoy} := (input(gb) ; P_{monoy}) + (input(dd) ; output(gb) to x ; stop) +
(input(c) ; create(y) ; reset(u,now+Ytime) ; P_{duoy}) +
(input(u) ; output(c) via yx ; set(u,now+Ytime) ; P_{duoy}) + save(m) ; P_{duoy}
P_p := set(T,now) ; P_{init} ; nil, where
P_{init} := (input(T) ; output(m(x)) ; output(m(y)) to x ; return) + save(X,Y) ; P_{init}

D(c) := (P_c, 1, 1, ( ), ( ), ((x, $\perp_t$), (y, $\perp_t$), (z, $\perp_t$)))
D(x) := (P_x, 1, 10000, ( ), ( ), ((y, $\perp_t$), (Xtime, $\perp_t$)))
D(y) := (P_y, 1, 10000, ( ), ( ), ((x,$\perp_t$),(Ytime,$\perp_t$)))
D(p) := (P_p, 0, ∞, ( ), ((x,$\perp_t$),(y,$\perp_t$), ( )))

$Routes_{Cblock}$:= { (EC, c), (c, CX), (c, CY) }
$Routes_{XYblock}$:= { (CX, x), (CY, y), (x, XE), (y, YE), (x, y), (y, x) }
Routes : { (EC, c), (c, CX), (c, CY), (CX, x), (CY, y), (x, XE), (y, YE), (x, y), (y, x) }

r_{Cblock}((EC, c)) := { X, Y, D }
r_{Cblock}((c, CX)) := { dd, m }
r_{Cblock}((c, CY)) := { dd, m }
$r_{XYblock}$((CX, x)) := { dd, m }
$r_{XYblock}$((CY, y)) := { dd, m }
$r_{XYblock}$((x, XE)) := { iam }
$r_{XYblock}$((y, YE)) := { iam }
$r_{XYblock}$((x, y)) := { c }
$r_{XYblock}$((y, x)) := { c }

The signature $\Sigma = \langle Sort, OP \rangle$
The variables of each sort s:$s \in Sort: Var_s$

C.3 TTCN to CSR

C.3.1 A simple TTCN example

The TTCN syntax used here is based on the TTCN.MP syntax rules, however a number of the keywords have been eliminated. The use or non use of these keywords does not effect the transformation to the CSR.

$Suite Example

Simple Type Definitions		
Type Name	**Type Definition**	**Comments**
PIDtype	INTEGER	Alternative name
Detailed Comment:		

Test Suite Parameter Declaration			
Parameter Name	**Type**	**PICS/PIXIT Ref**	**Comments**
Xtime Ytime	INTEGER INTEGER		
Detailed Comments:			

Test Case Variable Declaration			
Variable Name	**Type**	**Value**	**Comments**
count x y	INTEGER PIDtype PIDtype	0	
Detailed Comments:			

PCO Declaration			
PCO Name	**PCO Type**	**Role**	**Comments**
pcoC pcoX pcoY	PCOtype1 PCOtype2 PCOtype2	UT LT LT	
Detailed Comments:			

Timer Declaration			
Timer Name	**Duration**	**Unit**	**Comments**
timer	500*(Xtime+Ytime)	sec	
Detailed Comments:			

ASN.1 ASP Type Definition
ASP Name: iamtype **PCO Type:** PCOtype2 **Comments:**
Type Definition
iamtype ::= **SEQUENCE**{ field PIDtype } PIDtype ::= INTEGER
Detailed Comment:

ASN.1 ASP Type Definition
ASP Name: Dtype **PCO Type:** PCOtype1 **Comments:**
Type Definition
Dtype ::= **SEQUENCE**{ field PIDtype } PIDtype ::= INTEGER
Detailed Comment:

ASN.1 ASP Type Definition
ASP Name: Xtype
PCO Type: PCOtype1
Comments:
Type Definition
Xtype ::= **SEQUENCE**{ field PIDtype } PIDtype ::= INTEGER
Detailed Comment:

ASN.1 ASP Type Definition
ASP Name: Ytype
PCO Type: PCOtype1
Comments:
Type Definition
Ytype ::= **SEQUENCE**{ field PIDtype } PIDtype ::= INTEGER
Detailed Comment:

Constraint Declaration		
Constraint Name: coniam		
Type: iamtype		
Derivation Path:		
Comments:		
Parameter or Field Name	**Parameter or Field Value**	**Comment**
field	?	
Detailed Comment:		

Constraint Declaration		
Constraint Name: conD(fpar:INTEGER)		
Type: Dtype		
Derivation Path:		
Comments:		
Parameter or Field Name	**Parameter or Field Value**	**Comment**
field	fpar	
Detailed Comment:		

Constraint Declaration		
Constraint Name: conX(fpar:INTEGER) **Type:** Xtype **Derivation Path:** **Comments:**		
Parameter or Field Name	**Parameter or Field Value**	**Comment**
field	fpar	
Detailed Comment:		

Constraint Declaration		
Constraint Name: conY(fpar:INTEGER) **Type:** Ytype **Derivation Path:** **Comments:**		
Parameter or Field Name	**Parameter or Field Value**	**Comment**
field	fpar	
Detailed Comment:		

Test Case Dynamic Behaviour

Test Case Name: testcase

Group: Example/

Purpose: This testcase checks if the system is capable to produce at least one thousand process instances within a specified amount of time.

Default:

Comments:

Nr	Label	Behaviour Description	Constraints Ref	Verdict	C
1		pcoX?iamtype (x:=iamtype.field)	coniam		
2		pcoY?iamtype (y:=iamtype.field)	coniam	**(PASS)**	
3		pcoC!Xtype	conX(x)		
4		pcoC!Ytype	conY(y)		
5		(count:=2) START timer			
6		+newcomers(pcoX,pcoY)			
7		+post(count)			
8		[TRUE]		**PASS**	
9		?TIMEOUT timer		**I**	
10		pcoY?OTHERWISE		**FAIL**	
11		pcoX?OTHERWISE		**FAIL**	
12		**$Tree** post(fpar:INTEGER))			
13		[fpar>0]			
14		pcoC!Dtype (fpar:=fpar-1)	conD(fpar)		
15		+post(fpar)			
16		[fpar=0]		**(PASS)**	
17		[fpar<0]		**I**	
18		**$Tree** newcomers(fpco1,fpco2:PCOtype)			
19		[count<1000]			
20		fpco1?iamtype (count:=count+1)	coniam		
21		+newcomers(pco1,pco2)			
22		fpco2?iamtype (count:=count+1)	coniam		
23		+newcomers(pco1,pco2)			
24		fpco1?**OTHERWISE**		**FAIL**	
25		fpco2?**OTHERWISE**		**FAIL**	
26		[count=1000]		**(PASS)**	
27		[count>1000]		**(PASS)**	

Detailed Comments:

C.3.2 Its CSR representation

As indicated in Annex B, Clause B.1 the CSR representation consists of several entities. In order to represent a TTCN test suite in the CSR it is sufficient to provide the contents/ values of these entities.

PathId := { pcoC, pcoX, pcoY }
ModId := { Example } /* Free of choice for TTCN */
BPId := $BPId_{Example}$:= { testcase }
PId := $BPPId_{testcase}$:= $ModPId_{Example}$:= { 1 }
$P_{testcase}$:=
(input(iamtype **$Constraint** coniam) via pcoX [TRUE] ; (x:=iamtype.field) ;
(input(iamtype **$Constraint** coniam) via pcoY [TRUE] ; (y:=iamtype.field) ; R:=preP ;
output(Xtype **$Constraint** conX(x)) via pcoC ;
output(Ytype **$Conxtraint** conY(y)) via pcoC ;
(count:=2) ; set(timerduration, timer) ;

```
            (P_newcomers(pcoX,pcoY) ;
               P_post(count) ;
                 [TRUE] ; R:=P; stop ; nil
          ⊕ input(timer) [TRUE] ; R:=I ; stop ; nil )
   ⊕ otherwise pcoY [TRUE]; R:=F ; stop ; nil )
⊕ otherwise pcoX [TRUE]; R:=FAIL ; stop ; nil )

P_post(fpar:INTEGER)) :=
( [fpar>0] ;
   output(Dtype $Constraint conD(fpar)) via pcoC ; (fpar:=fpar-1) ;
      P_post(fpar)
⊕ [fpar=0] ; R:=preP
⊕ [fpar<0] ; R:=I ; stop ; nil )

P_newcomers(fpco1,fpco2:PCOtype) :=
( [count<1000] ;
   ( input(iamtype $Constraint coniam) via fpco1 [TRUE] ; (count:=count+1) ;
      P_newcomers(fpco1,fpco2)
   ⊕ input(iamtype $Constraint coniam) via fpco2 [TRUE] ; (count:=count+1) ;
      P_newcomers(fpco1,fpco2)
   ⊕ otherwise fpco1 ; R:=F ; stop ; nil
   ⊕ otherwise fpco2 ; R:=F ; stop ; nil )
⊕ [count=1000] ; R:=preP)
⊕ [count>1000] ; R:=preP )
```

D(testcase) := ($P_{testcase}$, 1, 1, (Xtime, Ytime), ( ), ((count,0),(x,$\perp_t$),(y,$\perp_t$),(R, none)))

$Routes_{Example}$:= { (testcase, pcoC), (pcoX, testcase), (pcoY, testcase) }

Routes := { (testcase, pcoC), (pcoX, testcase), (pcoY, testcase) }

$r_{Example}$((testcase, pcoC)) := $\{s \mid s \in SigTerm\} \cup \{\perp_{st}\}$

$r_{Example}$((pcoX, testcase)) := $\{s \mid s \in SigTerm\} \cup \{\perp_{st}\}$

$r_{Example}$((pcoY, testcase)) := $\{s \mid s \in SigTerm\} \cup \{\perp_{st}\}$

The signature $\Sigma = \langle Sort, OP \rangle$

The variables of each sort s: $s \in Sort, Var_s$

Abstract data type, 24
Abstract service primitive, 18; 19; 56
Algebra, 22; *25*; 26
Basic process, 21; 23; 28; 38; 41; 43; 44; 45; 46; 47; 51; 53; 54; 59
    declaration, *29*; 45; 46; 54
Basic process algebra, 21; *28*; 29; 30
    operators, *30*
Basic SDL, 16
Cardinality, 52
Carrier set, 25
Clock
    global, 27
Concatenation, *22*
Discrete time, 19
Disjoint union, *23*
Environment, 16; 24; 25; *26*; 28; 38; 39; 40; 46; 54; 57; 58; 59
    undefined, *39*; 54
Error element, 25
Events
    basic process, *29*
    input port process, *36*
    module, *49*
    path process, *55*
    process instance, *39*
    system, *56*
    timer process, *33*
Identifier
    basic process, *23*
    module process, *23*; 56
    path identifier, 23
    path process, 23; 47; 55; 56
    process identifier, *23*
    process instance, 23; 24; 26; 27; 38; 39; 47
Inference rules
    general form, *22*
Input port process, 21; 36; 37; 38; 41; 43; 44; 59
Input queue, *36*
Input stream, *18*; 19
Interpretation
    data term, *26*
    operation, 26
    revealed variable, *26*
    signal ground term, 26
    signal term, *26*; 27
Labelled transition system, 22; 32; 36; 40; 49; 55; 57
Module process, 22; 23; 28; 39; *49*; 55; 56; 57; 58
Module storage, 26; *39*; 40; 41; 45; 46
Operation, *24*
Ordering
    strict, *27*
    total, *27*
Path process, 22; 23; 44; 49; 52; 53; *55*; 56; 57; 58; 59
    undefined, 41
Procedure
    declaration, *29*; 46
Process instance, 21; 23; 26; *38*; 49; 50; 51; 52; 53; 54; 58; 59
    undefined, *38*; 45; 50; 51
Process instance identifier, 49
Protocol data unit, 18; 19; 56
Routes, *28*; 39; 47; 50; 51; 52; 53; 58
    function, *28*
Saved signals queue, *36*; 38
SDL active timer, 33
SDL block, *16*; 19; 22; 23; 49; 50; 51; 52; 53; 54; 55
SDL channel, 16; 19; 22; 23; 55; 56
SDL input port, 16; 36; 37; 38; 41
SDL procedure, 38
SDL process graph, 23; 29; 30
SDL process instance, *16*; 22; 25; 38; 42; 43; 44; 45; 46; 47; 48
SDL signal identifier, 25
SDL signal instance, *16*
SDL signal route, *16*; 28
SDL specification, 22
SDL system, *16*; 19; 22; 56; 57; 58; 59
SDL timer, 41
Semantics
    basic process, *32*
    input port process, 36
    module process, 49
    path process, *55*
    process instance, *40*
    system, *57*
    timer process, *33*
Sequence, 27
    empty, *22*
    finite, *22*
    first element, *22*
    last element, *22*
Set of alternatives, 18; 19
Side-condition, *22*; 32; 36; 41; 49; 50; 55; 57
Signal, 24; 26; 27; 39
    explicit, *27*; 39; 47; 50; 51; 52; 58
    implicit, *27*; 39; 47; 51
    instantiation, *27*
    saved, 36
    undefined, *26*; 28; 44; 53; 55; 59
Signal operation, *25*
Signal sort, *25*
Signature, *24*; 25
Snapshot, 18
Sort, *24*
State
    basic process, *29*
    input port process, *36*
    module process, 40
    path process, *55*
    process instance, *38*
    system, *56*
    timer process, *33*
Storage, 25; *26*; 38; 39; 40; 46; 47; 54

undefined, *39*; 54
Storage environment, *26*; 38
System, 22; *56*
initial state, 59
Term
data, *24*; 25; 26
data ground, *24*; 25
signal, *25*; 26; 43
signal ground, *25*; 27; 33
undefined, 26
Time, 27; 39; 41
Timer, *27*; 41; 42
Timer process, 21; 33; 38; 41; 59
Transition, 22
TTCN abstract evaluation tree, *18*; 23
TTCN abstract tester, *19*; 22; 23; 28; 49; 53
TTCN constraint, 25
TTCN input queue, *36*; 37; 38
TTCN PCO, 18; 19; 22; 23; 28; 55; 56
TTCN running timer, 33
TTCN system, *19*; 22; 56; 57; 58; 59
TTCN test case, 31
TTCN test suite, 22
TTCN tester, *17*; 22; 36; 37; 38; 42; 43; 44; 45; 47; 49
TTCN timer, 41
TTCN unforeseen message, 26
Union, *23*
Variable, 24; 38
revealed, 25; 26; 39

History

Document history	
June 1993	First Edition
February 1996	Converted into Adobe Acrobat Portable Document Format (PDF)