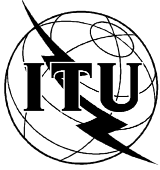

INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.500

(05/97)

SERIES Z: PROGRAMMING LANGUAGES

Methods for validation and testing

Framework on formal methods in conformance testing

ITU-T Recommendation Z.500

(Previously CCITT Recommendation)

ITU-T Z-SERIES RECOMMENDATIONS

PROGRAMMING LANGUAGES

FORMAL DESCRIPTION TECHNIQUES (FDT)	Z.100–Z.199
Specification and Description Language (SDL)	Z.100–Z.109
Application of Formal Description Techniques	Z.110–Z.119
Message Sequence Chart	Z.120–Z.129
PROGRAMMING LANGUAGES	Z.200–Z.299
CHILL: The ITU-T high level language	Z.200–Z.209
MAN-MACHINE LANGUAGE	Z.300–Z.499
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.399
QUALITY OF TELECOMMUNICATION SOFTWARE	Z.400–Z.499
METHODS FOR VALIDATION AND TESTING	**Z.500–Z.599**

For further details, please refer to ITU-T List of Recommendations.

ITU-T RECOMMENDATION Z.500

FRAMEWORK ON FORMAL METHODS IN CONFORMANCE TESTING

Source

ITU-T Recommendation Z.500 was prepared by ITU-T Study Group 10 (1997-2000) and was approved under the WTSC Resolution No. 1 procedure on the 6th of May 1997.

FOREWORD

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the ITU. The ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

INTELLECTUAL PROPERTY RIGHTS

The ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. The ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, the ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

© ITU 1998

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the ITU.

CONTENTS

		Page
1	Scope	1
2	Normative references	1
2.1	Conformance testing	1
2.2	Formal description techniques	2
3	Definitions	2
3.1	Terms from other related standards	2
3.2	Terms defined in this Recommendation	3
4	Abbreviations	4
5	Mathematical concepts and notation conventions	5
5.1	Sets	5
5.2	Logic	5
5.3	Relations	6
5.4	Functions	6
6	The meaning of conformance	6
6.1	Introduction	6
6.2	Specifications	6
6.3	Implementations	7
6.4	Conformance of an implementation to a formal specification	8
7	Testing concepts	10
7.1	Introduction	10
7.2	Test architecture	10
7.3	Formal model of the test architecture	11
7.4	Test execution	12
8	Conformance testing	14
8.1	Introduction	14
8.2	Definition of conformance testing	14
8.3	Test generation	14
8.4	Test suite size reduction	15
8.5	Fault coverage	16
8.6	Test suite cost	16
9	Compliance	17
9.1	Introduction	17
9.2	Compliance with clause 6: The meaning of conformance	17
9.3	Compliance with clause 7: Testing concepts	18
9.4	Compliance with clause 8: Conformance testing	18
Annex A		18
A.1	Specifications	18
A.2	Implementation options and instantiated specifications	23
A.3	Implementations and models of implementations	28
A.4	Conformance by implementation relations	30
A.5	Conformance by requirements	33
A.6	Test architecture	34
A.7	Specifications of tests	35
A.8	References	42

Introduction

Many protocol and service specifications are nowadays described in formal notations called Formal Description Techniques (FDTs). Examples of standardized FDTs are SDL, LOTOS, Estelle and ASN.1. There also exists a formal notation for test suite specifications: TTCN. The use of FDTs has the following advantages:

– they describe the formats and behaviours in an unambiguous way;

– they give a basis for rigorous validation including conformance testing.

Conformance to a communication protocol or service standard is considered to be a prerequisite for the correct interoperability of open systems. Conformance testing, i.e. the assessment by means of testing whether a product conforms to its specification, is an important issue in product development, because it increases the confidence in correct interoperability.

This Recommendation "Framework on Formal Methods in Conformance Testing" (FMCT) defines the meaning of conformance if formal methods are used for the specification of a communication protocol or service. It is also meant to guide computer aided test generation.

This Recommendation defines a framework for the use of formal methods in conformance testing. It is intended for implementers, testers, and specifiers involved in conformance testing to guide in defining conformance and the testing process of an implementation with respect to a specification that is given as a formal description.

FRAMEWORK ON FORMAL METHODS IN CONFORMANCE TESTING

(Geneva, 1997)

1 Scope

This Recommendation is applicable where a formal specification of a communication protocol or service exists, from which a conformance test suite shall be developed. It can guide the manual process as well as the development of tools for computer aided test case generation.

This Recommendation defines a framework and does not prescribe any particular test case generation method, nor does it prescribe any specific conformance relation between a formal specification and an implementation. It is supplementary to the joint ITU-T/ISO standard "Conformance Testing Methodology and Framework" (CTMF) [ISO/IEC 9646], which is applicable to a wide range of products and specifications, including specifications in natural language. FMCT interprets conformance testing concepts in a formal context.

2 Normative references

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; all users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published.

2.1 Conformance testing

NOTE – The following set of references will be referred to as CTMF is this Recommendation.

– ITU-T Recommendation X.290 (1995) (equivalent to ISO/IEC 9646-1:1994), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – General concepts*.

– ITU-T Recommendation X.291 (1995) (equivalent to ISO/IEC 9646-2:1994), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – Abstract test suite specifications*.

– CCITT Recommendation X.292 (1992) (equivalent to ISO/IEC 9646-3:1992), *OSI conformance testing methodology and framework for protocol Recommendations for CCITT applications – The Tree and Tabular Combined Notation (TTCN)*.

– ITU-T Recommendation X.293 (1995) (equivalent to ISO/IEC 9646-4:1994), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – Test realization*.

– ITU-T Recommendation X.294 (1995) (equivalent to ISO/IEC 9646-5:1994), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – Requirements on test laboratories and clients for the conformance assessment process*.

– ITU-T Recommendation X.295 (1995) (equivalent to ISO/IEC 9646-6:1994), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – Protocol profile test specification*.

– ITU-T Recommendation X.296 (1995) (equivalent to ISO/IEC 9646-7:1995), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – Implementation conformance statements*.

2.2 Formal description techniques

– ITU-T Recommendation Z.100 (1993), *CCITT Specification and Description Language (SDL)*.

– ITU-T Recommendation Z.120 (1996), *Message Sequence Charts (MSC)*.

– ISO/IEC 8807:1989, *Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*.

– ISO/IEC 9074:1989, *Information processing systems – Open Systems Interconnection – Estelle: A formal description technique based on an extended state transition model*.

3 Definitions

3.1 Terms from other related standards

NOTE – Although the following definitions are given in ITU-T X.290 and ISO/IEC 9646-1, they are repeated here because the meanings of these terms are important for the formal interpretations in this Recommendation.

3.1.1 abstract test method: The definition of how an IUT is to be tested, given at an appropriate level of abstraction to make the description independent of any particular realization of a Means of Testing, but with enough detail to enable tests to be specified for this test method.

3.1.2 conformance test suite: A complete set of test cases, possibly combined into nested test groups, that is selected to perform dynamic conformance testing for one or more protocols.

3.1.3 conformance testing: Testing the extent to which an IUT is a conforming implementation.

3.1.4 conforming implementation: An IUT which satisfies both static and dynamic conformance requirements, consistent with the capabilities stated in the ICS(s).

3.1.5 dynamic conformance requirement: All those requirements (and options) which determine what observable behaviour is permitted by the relevant specification(s) in instances of communication.

3.1.6 fail (verdict): A test verdict given when the observed test outcome either demonstrates non-conformance with respect to (at least one of) the conformance requirements on which the test purpose of the test case is focused, or contains at least one invalid test event, with respect to the relevant specification(s).

3.1.7 implementation conformance statement (ICS): A statement made by the supplier of an implementation or system claimed to conform to a given specification, stating which capabilities have been implemented. The ICS can take several forms: protocol ICS, profile ICS, protocol specific ICS, and information object ICS.

3.1.8 implementation under test (IUT): An implementation of one or more protocols in an adjacent user/provider relationship, being that part of the real open system which is to be studied by testing.

3.1.9 implementation extra information for testing (IXIT): A statement made by the supplier or implementor of an IUT which contains or references all of the information (in addition to that given in the ICS) related to the IUT and its testing environment, which will enable the testing laboratory to run an appropriate test suite against the IUT. An IXIT can take several forms: protocol IXIT, profile IXIT, profile specific IXIT, and information object IXIT, TMP implementation statement.

3.1.10 means of testing (MOT): The combination of equipment and procedures that can perform the derivation, selection, parameterization and execution of test cases, in conformance with a reference standardized ATS, and can produce a conformance log.

3.1.11 parameterized abstract test suite: A selected abstract test suite in which all relevant parameters have been supplied with values in accordance with the appropriate ICS(s) and IXIT(s).

3.1.12 pass (verdict): A test verdict given when the observed test outcome gives evidence of conformance to the conformance requirement(s) on which the test purpose of the test case is focused, and when no invalid test event has been detected.

3.1.13 point of control and observation (PCO): A point within a testing environment where the occurrence of test events is to be controlled and observed, as defined in an Abstract Test Method.

3.1.14 reference standardized abstract test suite (ATS): The standardized ATS for which a Means of Testing is realized.

3.1.15 static conformance requirement: One of the requirements that specify the limitations on the combinations of implemented capabilities permitted in a real open system which is claimed to conform to the relevant specification(s).

3.1.16 test purpose: A prose description of a well defined objective of testing, focusing on a single conformance requirement or a set of related conformance requirements as specified in the appropriate specification.

3.1.17 test verdict: A statement of "pass", "fail", or "inconclusive", as specified in an abstract test case, concerning conformance of an IUT with respect to that test case when it is executed.

3.2 Terms defined in this Recommendation

3.2.1 compatibility: See 6.4.3.

3.2.2 complete (test suite): See 8.2.

3.2.3 conformance: See 6.4.

3.2.4 conformance testing: See 8.2.

3.2.5 dynamic conformance: See 6.4.2 and 6.4.3.

3.2.6 exhaustive (test suite): See 8.2.

3.2.7 fault model: See 8.4.1.

3.2.8 fault coverage: See 8.5.

3.2.9 formal specification: See 6.2.

3.2.10 implementation: See 6.3.

3.2.11 implementation access point: See 7.2.

3.2.12 implementation access point, (formal model of): See 7.3.

3.2.13 implementation option: See 6.2.

3.2.14 implementation relation: See 6.4.2.

3.2.15 implementation under test: See 3.1.

3.2.16 implementation under test, (formal model of): See 7.3.

3.2.17 instantiated specification: See 6.2.

3.2.18 interaction point: See 7.3.

3.2.19 looser specification: See 8.4.2.

3.2.20 model of implementation: See 6.3.

3.2.21 mutant: See 8.4.1.

3.2.22 observation: See 7.4.1.

3.2.23 **parameterized specification**: See 6.2.

3.2.24 **point of control and observation**: See 3.1.

3.2.25 **point of control and observation, (formal model of)**: See 7.3.

3.2.26 **satisfaction relation**: See 6.4.3.

3.2.27 **sound (test suite)**: See 8.2.

3.2.28 **specification**: See 6.2.

3.2.29 **static conformance**: See 6.4.1.

3.2.30 **test architecture**: See 7.2.

3.2.31 **test case, (formal model of)**: See 7.3.

3.2.32 **test case execution**: See 7.4.1.

3.2.33 **test context**: See 7.2.

3.2.34 **test context, (formal model of)**: See 7.3.

3.2.35 **test generation**: See 8.3.

3.2.36 **test purpose**: See 3.1 and 7.4.2.

3.2.37 **test purpose, (formal model of)**: See 7.4.2.

3.2.38 **test suite cost**: See 8.6.

3.2.39 **test suite, (formal model of)**: See 7.3.

3.2.40 **test suite reduction**: See 8.4.

3.2.41 **tester**: See 7.2.

3.2.42 **tester, (formal model of)**: See 7.3.

3.2.43 **weaker implementation relation**: See 8.4.2.

4 Abbreviations

This Recommendation uses the following abbreviations:

CTMF Conformance Testing Methodology and Framework

FDT Formal Description Technique

FMCT Formal Methods in Conformance Testing

IAP Implementation Access Point

ICS Implementation Conformance Statement

IUT Implementation Under Test

IXIT Implementation extra Information for Testing

LOTOS Language of Temporal Ordering Specifications

LTS Labelled Transition System

MSC Message Sequence Chart

PCO Point of Control and Observation

SDL Specification and Description Language

TTCN Tree and Tabular Combined Notation

5 Mathematical concepts and notation conventions

5.1 Sets

Any subset of a set is denoted with upper case letters (for example, A, B, C) or a series of upper case letters (for example, *SPECS, IMPLS, TESTS*). Elements of a set are denoted with lower case letters (for example, a, b, c).

The following operations on sets are used in this Recommendation:

$\{a,b,c,...\}$	The set containing elements a, b, c, ...; the order in which elements appear in a set is not important.
$\emptyset$	The empty set, i.e. the set with no elements.
$a \in A$	a is an element of the set A.
$\{a \in A \mid P(a)\}$	The set containing all elements of A which satisfy predicate P. Sometimes $\{a \mid P(a)\}$ is used when the set A can be deduced from the context.
$A \subseteq B$	A is a subset of B, i.e. all elements of A are also elements of B.
$A = B$	The set A is equal to the set B, i.e. A is a subset of B and B is a subset of A.
$A \subset B$	A is a proper subset of B, i.e. A is a subset of B and A is not equal to B.
$A \cap B$	The intersection of A and B, i.e. the set containing all elements that are both in A and B.
$\bigcap_{i \in I} A_i$	Generalized intersection, i.e. the intersection of all sets A_i: $A_1 \cap A_2\ ...\ \cap A_n$, where n is a natural number.
$A \cup B$	The union of A and B, i.e. the set containing all elements that are either in A or B (or in both of them).
$\bigcup_{i \in I} A_i$	Generalized union, i.e. the union of all sets A_i: $A_1 \cup A_2 \dots \cup A_n$, where n is a natural number.
$A \times B$	The Cartesian product of A and B denoting the set of all ordered pairs (a, b) such that $a \in A$ and $b \in B$.
$A_1 \times A_2 \times ... \times A_n$	The generalized Cartesian product of A_1, A_2, ..., A_n denoting the set of all ordered pairs (a_1, a_2, …, a_n) such that $a_1 \in A_1$, $a_2 \in A_2$, ..., $a_n \in A_n$.
$A - B$	The set difference of A and B, i.e. the set containing all elements of A that are not in B.
Powerset(A)	The powerset of A, i.e. the set containing all subsets of A.
$R_{\geq 0}$	The set of the positive real numbers, including zero

The symbol '/' superimposed on a (set) operator is used to denote the negation of the operator, e.g. $a \notin A$ (a is not element of the set A), $A \not\subset B$ (A is not a proper subset of B), etc.

5.2 Logic

The following logic notations are used in this Recommendation:

$\neg p$	Not p, i.e. the negation of p
$p \wedge q$	p and q, the conjunction of p and q
$p \vee q$	p or q, the disjunction of p and q
$p \Rightarrow q$	p implies q, also read as: not p or q

$p \Leftrightarrow q$	p is equivalent with q, i.e. $(p \Rightarrow q) \wedge (q \Rightarrow p)$
$\forall\ a \in A$	For all elements a of set A
$\exists\ a \in A$	There exists an element a in set A

5.3 Relations

Relations are denoted as lower case abbreviation, that are underlined (for example $\underline{\text{rel}}$).

Let A and B be sets, then a binary relation $\underline{\text{rel}}$ between A and B is a subset of their Cartesian product: $\underline{\text{rel}} \subseteq A \times B$. The element $a \in A$ is related to $b \in B$ if $(a,b) \in \underline{\text{rel}}$ (or alternatively $a\ \underline{\text{rel}}\ b$). Analogously, an n-ary relation is a subset of $A_1 \times A_2 \times ... \times A_n$.

The *domain* $\underline{\text{dom(rel)}}$ of relation $\underline{\text{rel}} \subseteq A \times B$ is defined as the set containing all elements in A that are related to some $b \in B$, i.e.:

$$\underline{\text{dom(rel)}} = \{a \in A \mid \exists\ b \in B : (a,b) \in \underline{\text{rel}}\}$$

A (binary) relation $\underline{\text{rel}}$ *on* A is a subset of $A \times A$.

5.4 Functions

Function names are denoted as lower case abbreviation, that are underlined (e.g. $\underline{\text{func}}$).

A *partial function* $\underline{\text{func}}$ is a relation between two sets A and B with the property that for each $a \in A$ there exists at most one $b \in B$ such that $\langle a,b \rangle \in \underline{\text{func}}$, i.e.:

$$\forall a \in A : \forall b_1,b_2 \in B : (\langle a,b_1 \rangle \in \underline{\text{func}} \wedge \langle a,b_2 \rangle \in \underline{\text{func}}) \Rightarrow b_1 = b_2$$

When a function is introduced, its signature is given in the following style:

$$\underline{\text{func}} : A \rightarrow B$$

A (*total*) *function* $\underline{\text{func}} : A \rightarrow B$ is a partial function satisfying $\underline{\text{dom}}(\underline{\text{func}}) = A$.

6 The meaning of conformance

6.1 Introduction

Conformance involves defining whether an implementation is a valid implementation of a given specification with respect to a certain correctness notion. In order to formalize the concept of conformance, implementations will be modelled by formal objects called models. Conformance can be characterized by relations between models of implementations and specifications, or by satisfaction of specified requirements by models of implementations, or by both. This clause defines conformance, which includes definitions of specifications, implementations, models of implementations, implementation relations and conformance requirements.

6.2 Specifications

A (*formal*) *specification* prescribes the behaviour of a system using an FDT. If the FDT allows the use of parameters, a specification with formal parameters is referred to as a *parameterized specification*. If the formal parameters are instantiated with actual values or if there are no formal parameters, the specification is an instantiated specification.

The set of instantiated specifications is denoted by *SPECS*. A parameterized specification s is considered as a function from its parameter space D_s (the set of all possible actual values for the formal parameters) to *SPECS*:

$$s : D_s \rightarrow SPECS$$

Specifications often contain descriptions of features that are optional to support in a product that implements the standard. The freedom to support certain features or not to support them is named implementation options. Specifications usually contain such options. The implementation options are represented by the formal parameters of a specification, i.e. the specification is parameterized over its implementation options.

A specification that contains *implementation options* defines a set of instantiated specifications; one instantiated specification for each choice between the options. A specification shall clearly indicate which combinations of support of options are allowed.

Information on implementation options is required to be included in the protocol/profile ICS proforma. The ICS describes a specific choice of the implementation options in the ICS proforma. A specification parameterized over its ICS proforma is instantiated by an ICS. The ICS shall contain sufficient information to provide actual values for the implementation options.

The term *specification* is used in the remainder of this Recommendation to denote an *instantiated specification*.

NOTE – CTMF requires the protocol/profile ICS to be published as an annex to the protocol/profile standard.

Example

Suppose a specification allows to implement different levels of support of features. The levels are numbered from 0 up to 3. The standard, parameterized specification can then be represented as: *s*(*level:* 0...3). The Protocol ICS may contain the information that *level* = 2 is claimed to be implemented. This means that the instantiated specification of the implementation under test is, in fact, *s*(2).

6.3 Implementations

An implementation consists of a combination of hardware and/or software. The implementations have physical connectors or programming interfaces for communication with their environment or end-users. The set of implementations is denoted as *IMPS*.

A specification is a formal object while an implementation is a physical one. In order to formalize the concept of conformance, these different kinds of objects have to be related. Implementations cannot be subject to formal reasoning as they are not formal objects. Therefore, it is not possible to define directly a formal relation between implementations and specifications.

In the remainder of this Recommendation, it is assumed that any implementation IUT $\in$ *IMPS* can be modelled by an element m_{IUT} in a formalism *MODS* (e.g. labelled transition systems, finite state machines). This assumption is referred to as the *test assumption*. The activity of testing consists of extracting information from IUT by testing it, such that from this information the model m_{IUT} can be constructed in sufficient detail to decide conformance about it.

NOTE 1 – Only the possibility to construct a model is assumed, not that the model is *a priori* known.

NOTE 2 – The formalism *MODS* used to model the behaviour of an implementation, may be the same as the formalism *SPECS* that is used for the specification.

An implementation may have more than one model for a specific choice of *MODS*. The test assumption implies that it is not possible to distinguish between these models by testing. Therefore, it is sufficient to model an implementation IUT by a single element m_{IUT} of the set of possible models *MODS*.

NOTE 3 – It is assumed that the IUT can be modelled with sufficient precision such that the model can represent the IUT with respect to the properties that are prescribed by the specification.

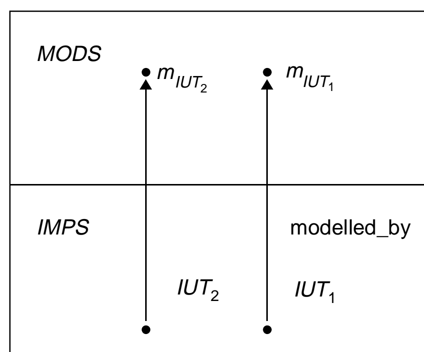

Figure 1/Z.500 – The relation between elements of *IMPS* and *MODS*

6.4 Conformance of an implementation to a formal specification

Conformance between an implementation and a specification exists when the implementation is correct with respect to the specification. Conformance is defined in two parts:

- static conformance; and
- dynamic conformance.

An implementation conforms to a specification if, and only if, it both statically and dynamically conforms.

6.4.1 Static conformance

Static conformance involves the correct instantiation of a parameterized specification. An implementation under test IUT with corresponding implementation conformance statement ICS_{IUT} conforms statically to a parameterized specification $s : D_s \rightarrow SPECS$ if ICS_{IUT} is contained in the domain of s, i.e. $ICS_{IUT} \in D_s$; which means that $s(ICS_{IUT})$ is defined. Checking static conformance corresponds to type checking of the actual parameter ICS_{IUT} with respect to the 'type' D_s.

Static conformance means that ICS_{IUT} is accepted by the parameterized specification, hence that the specific combination of implementation options described in ICS_{IUT} is allowed.

The allowable combinations and ranges of implementation options in the ICS_{IUT}, i.e. the specification of the set D_s, can be described by *static conformance requirements*. A static conformance requirement is a requirement that specifies the limitations on the ranges and the combinations of implemented options and capabilities permitted in an implementation by the specification (refer to CTMF, Part 1, 3.4.4).

NOTE – If static conformance requirements extend beyond the normal type checking of actual parameter values in the FDT employed, these requirements may be expressed in the behaviour part of the formal description (see Annex A).

6.4.2 Dynamic conformance

Dynamic conformance involves the permitted observable behaviour of an implementation in instances of communication as described by the specification. Dynamic conformance between an implementation and a specification is formally characterized by a relation between the model of the implementation and the specification. This relation is called an *implementation relation*. It will be denoted as $\underline{\text{imp}}$, where $\underline{\text{imp}}$ has the signature:

$$\underline{\text{imp}} \subseteq MODS \times SPECS$$

An implementation IUT conforms dynamically to an instantiated specification s with respect to relation $\underline{\text{imp}}$, if $m_{IUT}\ \underline{\text{imp}}\ s$. In this case m_{IUT} is a conforming model of s with respect to $\underline{\text{imp}}$. As such implementation relation $\underline{\text{imp}}$ expresses correctness between specification s and implementation IUT.

An instantiated specification can have several conforming implementations. For a specification $s \in SPECS$ an implementation relation $\underline{\text{imp}}$ the set M_s denotes the set of all conforming models in $MODS$, and is given by:

$$M_s = \{ m \in MODS \mid m\ \underline{\text{imp}}\ s \}$$

NOTE – For defining conformance, it is a prerequisite that the following objects exist: a specification $s \in SPECS$, an implementation IUT $\in IMPS$, documentation of choice between options ICS_{IUT}, and an implementation relation. The implementation relation is not universal; for different application areas, different implementation relations may be used. Example of implementation relation for Estelle, LOTOS and SDL can be found in the example below and further elaboration is found in Annex A.

Figure 2 illustrates how an instantiated specification $s \in SPECS$ determines a set of conforming implementations I_s. The set I_s denotes the set of implementations that can be modelled by models in M_s. Therefore, the set I_s is the set of implementations that implement the specification s correctly.

Assuming that s is a protocol specification, elements of I_s will be able to communicate using that protocol. If s is the specification of a complete stack of protocols (a profile) and a distributed application, elements of I_s will be able to interoperate.

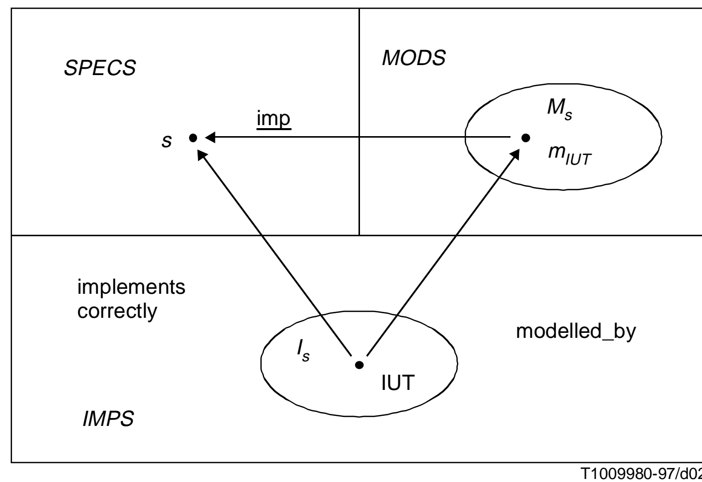

Figure 2/Z.500 – Relations between *IMPS*, *MODS*, and *SPECS*

Example

Examples of implementation relations which are often used, if both *MODS* and *SPECS* are chosen to be Estelle or SDL, are *trace equivalence* and *trace preorder*. If trace equivalence is required, the set of execution traces of the implementation shall be equal to the set of traces of the specification. In case of trace preorder, the first set shall be included in the second. (This means that only part of the specified behaviour has to be implemented.)

Examples of implementation relations which are often used for LOTOS are *failure equivalence* and *failure preorder*. If failure preorder is required, the set of traces of the implementation shall be included in the set of traces of the specification, and the implementation shall not produce deadlocks that are not specified.

6.4.3 Dynamic conformance requirements

In clause 6 conformance has been defined in an abstract way by means of an instantiated specification together with an implementation relation between the sets *MODS* and *SPECS*. On the other hand, in CTMF, the definition of dynamic conformance is based on the concept of a collection of dynamic conformance requirements. Both approaches can be used to specify conforming observable behaviour. They can be used separately or in combination. Conformance by means of an instantiated specification with an implementation relation was defined in 6.4.2. This subclause shows that the use of dynamic conformance requirements is a possible refinement of this definition, and that both approaches define the same concept with the same expressive power.

A dynamic conformance requirement is a requirement that specifies what observable behaviour is permitted in instances of behaviour (refer to CTMF, Part 1, 3.4.3). It is a property that needs to be satisfied by (the model of) the implementation in order for the implementation to conform.

Dynamic conformance requirements are expressed in a requirements language. *REQS* denotes the set of all requirements that can be expressed in that particular language. In the requirement approach, an instantiated specification $s \in SPECS$ is expressed as a set of requirements $R_s \subseteq REQS$, that is $SPECS = Powerset(REQS)$. An element of $r \in R_s$ represents a single dynamic conformance requirement. In general, the set R_s can be infinite.

Dynamic conformance between an implementation and a specification in the requirement approach is formally characterized by a relation between the model of the implementation and the specification. This relation is called a *satisfaction relation*. It will be denoted as sat, where sat has the signature:

$$\underline{\text{sat}} \subseteq MODS \times REQS$$

An implementation IUT conforms dynamically to specification $R_s \subseteq REQS$ if the model m_{IUT} of IUT satisfies all conformance requirements in R_s. The set M_{R_s} of models of conforming implementations in the requirements approach is given by:

$$M_{R_s} = \{m \in MODS \mid \forall r \in R_s : m \ \underline{\text{sat}}\ r\}$$

For a particular conformance requirement $r_i \in R_s$, the set M_{r_i} denotes the set of all models in *MODS* satisfying requirement r_i, i.e. $M_{r_i} = \{m \in MODS \mid m \ \underline{\text{sat}}\ r_i\}$. Figure 3 shows how intersection of the sets M_{r_i} determines the set M_{R_s} of conforming implementations.

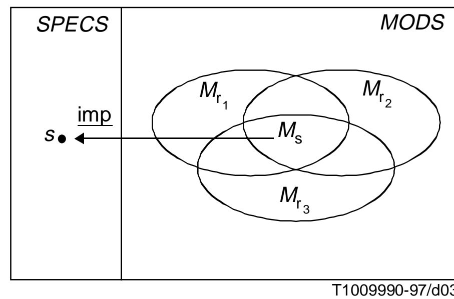

Figure 3/Z.500 – Conformance requirements and conforming implementations

6.4.4 Combining specifications

Different formal specifications, not necessarily expressed in the same formal language, can be combined to define one single set of models of conforming implementations. If the specifications s_1, s_2, s_3, .. s_n with implementation relations $\underline{\text{imp}}_1$, $\underline{\text{imp}}_2$, $\underline{\text{imp}}_3$, ... $\underline{\text{imp}}_n$, defined over the same class of models of implementations *MODS*, define the set of conforming implementations M_{s_1}, M_{s_2}, M_{s_n}, respectively, then the set of conforming implementations defined by s_1, s_2, s_3, .. s_n is $M_{s_1} \cap M_{s_2} \cap .. \cap M_{s_n}$.

The specifications s_1, s_2, s_3, .. s_n *are consistent if this intersection is not empty*. Consistency implies that the collection of specifications is implementable. It is a requirement for combining specifications.

A particular case of combination of specifications is to combine an instantiated behaviour specification *s* (with implementation relation imp) with a requirement specification *R* (with satisfaction relation sat). The requirement in *R* can specify additional requirements of conforming implementations, or they can serve as an alternative specification for exactly the same set of conforming implementations. In the latter case, *s* (with imp) and *R* (with sat) are said to be *compatible*:

$$\forall\ m \in MODS : m\ \underline{\text{imp}}\ s \Leftrightarrow \forall\ r \in R : m\ \underline{\text{sat}}\ r$$

7 Testing concepts

7.1 Introduction

Testing is a means to extract knowledge about a system by experimenting with it. The experimentation is carried out by executing test cases. The execution of a test case leads to an observation from which it can be concluded whether a system has a certain property or not. This clause defines the basic concepts employed to describe the execution of test cases – no reference is made to a notion of conformance or to a property being tested. These basic concepts include the environment in which the test cases are executed, the formal modelling of this environment, test cases and test suites, the execution of test cases, the observations that can be made during test execution, the interpretation of observations, and the test purpose related to a test case and to a test suite.

7.2 Test architecture

The *test architecture* is a description of the environment in which the IUT is tested. It describes the relevant aspects of how the IUT is embedded in other systems during the testing process, and how the IUT communicates via these embedding systems and with the tester. Figure 4 gives an abstract view of the test architecture.

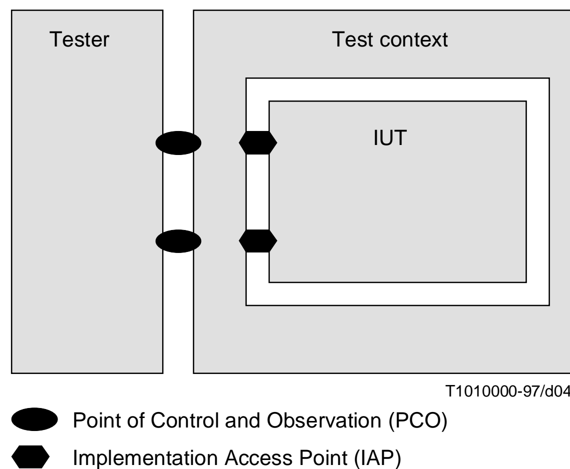

Figure 4/Z.500 – Test Architecture

A *test architecture* consists of:

– a tester;

– an Implementation Under Test (IUT) (see 3.1);

– a test context;

– Points of Control and Observation (PCO) (see 3.1);

– Implementation Access Points (IAP).

The *tester* is the implementation of a test suite (see 7.3). It carries out the experiments by executing the test cases and observing the results. The tester communicates with the test context via the PCOs, and indirectly with the IUT via the test context. The tester can be sub-structured in different components (e.g. in a Lower Tester and an Upper Tester, CTMF).

The *test context* is the system in which the IUT is embedded, and via which the IUT communicates with the tester. It relates events that occur at PCOs in the communication between the test context and the tester, to events that occur at the IAP's in the communication between the test context and the IUT.

An *Implementation Access Point* (IAP) is an interaction point in the test architecture where the IUT interacts with its environment, i.e. the test context, and via the test context (indirectly) with the tester.

NOTE – In certain cases, IAPs and PCOs may coincide.

7.3 Formal model of the test architecture

In order to be able to reason about the testing process in a formal setting, the entities of the test architecture must be formally modelled.

Test suite – The formal description of the tester is given by a *test suite*. A test suite specifies the entire set of experiments which are to be executed by the tester. The tester is the implementation of the test suite.

The experiments that constitute a test suite are called *test cases*. Each test case specifies the behaviour of the tester in a separate experiment that tests an aspect of the IUT, and that leads to an observation and a verdict (see 7.4.1).

The formal language in which a test case is expressed is called the *test notation*. The test notation is denoted by *TESTS*. It follows that a test suite, being a set of test cases, is an element of *Powerset*(*TESTS*).

It is assumed that specifications of test cases and test suites are correctly implemented in the tester according to the semantics of the test notation. This assumption is reasonable, since such specifications are far less complex than system specifications.

Implementation under test – Following the test assumption (see 6.3), the IUT is formally expressed by its model $m_{IUT} \in MODS$.

Test context – The formal model of a test context is a transformation of behaviour as it is observed at the IAPs to behaviour as it is observed modelled at the PCOs. It is described as a function *C* on *MODS*:

$$C : MODS \rightarrow MODS$$

It follows that the behaviour of the IUT as observed at the PCOs is formally expressed as $C(m_{IUT})$.

Point of control and observation and implementation access point – The PCOs and IAPs are formally modelled as interaction points in *MODS*. The properties of these interaction points depend on how interactions between entities are defined in the formalism *MODS* that has been chosen. Care must be taken that the modelling of interactions in *MODS* faithfully models the interactions as they occur at the PCOs and IAPs in the test architecture.

7.4 Test execution

Test execution of a test suite $T \subseteq TESTS$ consists of running the tester that implements *T* in combination with the IUT and the test context for each test case $t \in T$. The run of one test case is called *test case execution*. The objective of test case execution is to experiment with the IUT in order to investigate whether or not the IUT responds correctly to a certain behaviour.

7.4.1 Test case execution

Test case execution is the running of the implementation of one test case $t \in TESTS$ in combination with the IUT and the test context. During the run an *observation* is made. It can include a log of occurring interactions, a (preliminary) verdict, or anything which is considered important for determining the result of the test case execution. When a test case execution leads to an observation – $\sigma \in OBS$ – the result is defined by a verdict assignment $\underline{\text{verd}}_t$ which may depend on the test case $t \in T$:

$$\underline{\text{verd}}_t : OBS \rightarrow \{\text{pass, inconclusive, fail}\}$$

An implementation under test IUT *passes* a (correctly) implemented test case *t* (in a particular given test context) if, and only if, the test execution of the IUT with *t* leads to an observation σ for which the verdict pass is assigned:

$$\text{IUT } \underline{\text{passes}}\ t \Leftrightarrow \underline{\text{verd}}_t\ (\sigma) = \text{pass}$$

Example

The following table exemplifies observations. Each row in the table contains results of measurements. The sequence of measurements comprises the observation of one test run.

The measurement of an event is denoted as the type of the observed event at (@) the location of the observed interaction.

Time	Event	Parameters
0003	Connect @ PCO-1	sender = "1223" address = "4545" level = 3
0005	Accept @ PCO-1	sender = "4545" address = "1223" level = 2
0012	Disconnect @ PCO-1	sender = "1223" address = "4545" reason = 5

7.4.2 Formal model of test case execution

For the interpretation of the result of a test case execution, a model of test case execution must be defined by a function. The function exec calculates the observations for models of IUT's (m_{IUT}) contained in a model of a test context (C):

$$\underline{\text{exec}}: TESTS \times MODS \rightarrow OBS$$

The expression $\underline{\text{exec}}(t,C(m_{IUT}))$ models the observation that is made by the test case t of the IUT modelled as m_{IUT} in the test context C.

If exec correctly models test execution, i.e. test execution leads to an observation σ if, and only if, $\underline{\text{exec}}(t,C(m_{IUT})) = \sigma$, then it can be concluded from test execution of an IUT with a test case t that:

$$IUT\ \underline{\text{passes}}\ t \Leftrightarrow \underline{\text{verd}}_t\ (\underline{\text{exec}}(t,C(m_{IUT}))) = \text{pass}$$

The subset of $MODS$ for which $\underline{verd}_t\ (\underline{exec}(t,C(m))) = \text{pass}$, is called the *formal test purpose* P_t :

$$P_t = \{m \in MODS \mid \underline{\text{verd}}_t\ (\underline{\text{exec}}(t,C(m))) = \text{pass}\}$$

Hence, the objective of testing an IUT with test case t is to conclude whether the model of the IUT is a member of its formal test purpose P_t, i.e. IUT passes t if, and only if, m_{IUT} is an element of P_t.

A prose description may accompany the formal test purpose of a test case. Such a prose description of the objective of testing is called the (*informal* or *natural language*) *test purpose* (see 3.1).

A formal test purpose as a subset of *MODS* can be specified using the same methods as the ones used to specify sets of conforming implementations, i.e. either by an instantiated behaviour specification together with an implementation relation, or by a requirement or a set of requirements with a satisfaction relation. Such a formal expression denoting a formal test purpose, may also itself be referred to as a formal test purpose.

Example

A Message Sequence Chart (MSC) describes a sequence of behaviour. Together with the implementation relation *inverse trace preorder* (all sequences of behaviour of the MSC shall be contained in the implementation) and MSC specify a formal test purpose, namely the set of models of implementations that contain the behaviour specified by the MSC.

NOTE – The formal test purpose P_t is related to the test case t. It is not prescribed which of them follows from the other; given a test case t its formal test purpose P_t can be calculated, or given a formal test purpose p (e.g. given as a formal requirement) a test case t_p can be developed such that $P_{t_p} = p$. Such a test case may not exist.

7.4.3 Test suite execution

The process of executing a test suite consists of executing each test case which belongs to the test suite consecutively. Each single test case execution is assigned a verdict, as stated in 7.4.1. To be executable, a test suite needs to be finite. However, when modelling test suite execution with exec, finiteness is not required.

An implementation under test IUT passes a test suite $T \subseteq TESTS$ if, and only if, it passes all the test cases in the test suite:

$$\text{IUT}\ \underline{\text{passes}}\ T \Leftrightarrow \forall t \in T : \text{IUT}\ \underline{\text{passes}}\ t$$

If IUT passes a test suite, then it follows that the model of IUT is contained in all the formal test purposes of the test cases of the test suite:

$$IUT\ \underline{\text{passes}}\ T \Leftrightarrow m_{IUT} \in \bigcap_{t \in T} P_t$$

The set $\bigcap_{t \in T} P_t$ is the formal test purpose of T. It is denoted by P_T. Like for formal test purposes of test cases an (informal or natural language) test purpose may accompany the formal test purpose to give a prose description of the objective of testing with T.

8 Conformance testing

8.1 Introduction

This clause employs the testing concepts defined in clause 7 to test the property of conformance defined in clause 6. This clause includes definitions of conformance testing, test generation from the formal specification, test suite coverage and ways to limit the size of the test suite.

8.2 Definition of conformance testing

Conformance testing is the assessment by means of testing, whether an implementation conforms to its specification. Conformance testing is aimed at collecting information for building a model m_{IUT} of the behaviour of implementation IUT. This model is used to decide whether m_{IUT} is an element of the set of models of conforming implementations, i.e. whether $m_{IUT} \in M_s$.

In general it is not possible to obtain certainty concerning conformance between implementations and specifications due, for example, to non-determinism in the implementation, restrictions in the observability and controllability of the implementation and the practical limitation of only being able to execute a finite number of tests. A test suite T can have the following properties depending on the relation between P_T and M_s.

Exhaustive: Test suite T is exhaustive if the set P_T of all models that pass test suite T is a subset of the set of conforming models M_s: $P_T \subseteq M_s$. This means that all passing implementations are compliant.

Sound: Test suite T is sound if the set of conforming models M_s is a subset of the set P_T of models that pass implementation IUT: $M_s \subseteq P_T$. That means that all implementations that do not pass are not compliant.

Complete: Test suite T is complete if it is both sound and exhaustive, that is, the set of conforming models equals the set of models that pass the implementation: $P_T = M_s$.

If test suite T is neither sound nor exhaustive, then nothing concerning conformance can be concluded by means of testing.

Ideally a test suite is complete. In general it is not possible to construct a finite exhaustive test suite.

NOTE – Exhaustive test suites are not likely to be encountered in practice. Exhaustiveness is a useful concept for theoretical argumentation about (infinite) model of test suites, and as a means of comparison for realistic test suites (see 8.5: Coverage).

8.3 Test generation

In the test generation process a test suite (i.e. a set of test cases) is generated from a formal specification. The use of an FDT for the specification is a prerequisite for automatic test generation.

Test generation is defined as a function gen that provides a test suite for an instantiated specification, given an implementation relation imp and a test context C:

$$\underline{\text{gen}}^{C,\underline{\text{imp}}} : SPECS \rightarrow Powerset(TESTS)$$

The generated test suite is expressed in a test notation (see 7.3: test notation).

The generated test suite is required to be sound (see 8.2). Therefore, a generated test suite has to fulfill the following property:

$$\forall m \in MODS, m \ \underline{\text{imp}} \ s \Rightarrow m \in P_T$$

where:

$$T = \underline{\text{gen}}^{C,\underline{\text{imp}}}(s)$$

Test generation, as defined above, works on instantiated specifications: $\underline{\text{gen}}^{C,\underline{\text{imp}}} : SPECS \rightarrow Powerset(TESTS)$. A test suite for a specification $s : D_s \rightarrow SPECS$ and ICS is obtained by $\underline{\text{gen}}^{C,\underline{\text{imp}}}(s(ICS))$.

Instantiating the parameterized specification $s : D_s \to SPECS$ with the ICS can be postponed until after test generation. This means that test generation works on parameterized specifications, generating a test suite parameterized over the ICS:

$$\underline{\text{pgen}}^{C,\underline{\text{imp}}} : (D \to SPECS) \to (D \to Powerset(TESTS))$$

where D is the domain of all possible implementation statements.

A test suite for a specification $s : D_s \to SPECS$ and implementation conformance statement ICS is obtained by:

$$(\underline{\text{pgen}}^{C,\underline{\text{imp}}} : (s : D_s \to SPECS))(ICS)$$

It is required that the generated parameterized test suites are sound for each possible instantiation with an ICS.

8.4 Test suite size reduction

Each test suite used for conformance testing shall be sound (see 8.3), which means that each test execution resulting in a fail-verdict indeed indicates an error in the implementation under test (see 8.2). Exhaustiveness of test suites is not required, which means that not necessarily all errors in an implementation under test will be detected (see 8.2). Usually the number of test cases that would be required to test exhaustively, is very large, or even infinite, implying that exhaustive testing is not feasible in practice. To reduce the size of a test suite with respect to the size of an ideal, exhaustive test suite, in order to make test suite execution feasible in practice, different test-suite size reduction strategies are identified.

8.4.1 Fault model

The test-suite size reduction strategies can be presented in terms of a *fault model*. A fault model F is a set of models of non-conforming implementations. It is expressed as a subset of $MODS - M_s$:

$$F \subseteq MODS - M_s$$

A fault model can be described as a modification (mutant) of the specification $s \in SPECS$. Let $\Delta_s \in SPECS$ contain a modification with respect to the original specification s, then the fault model described by Δ_s is $(M_{\Delta_s} - M_s)$.

The largest fault model is the set of all non-conforming implementations $MODS - M_s$.

Example

Suppose both the set of specifications $SPECS$ and the set of models $MODS$ are equal to the set of Input/Output Finite State Machines. For a particular specification s two types of mutants may be recognized:

a) mutants with transfer faults, i.e. the final state of tested transitions may be different from the final state specified by specification s;

b) mutants with output fault, i.e. the observed output after a specific input may not be the one specified by the specification.

8.4.2 Test-suite size limiting strategies

The test-suite size reduction strategies guarantee to obtain a sound test suite, either starting from a sound, but probably too large test suite T, or starting from a specification s, an implementation relation $\underline{\text{imp}}$, and a test generation function $\underline{\text{gen}}^{C,\underline{\text{imp}}}$ that generates sound, but probably too large test suites $\underline{\text{gen}}^{C,\underline{\text{imp}}}(s)$. The following strategies are identified:

1) Take any subset T' of the test suite T. The fault model for which T' tests is the complement of its formal test purpose: $F = MODS - P_{T'}$.

2) Loosen the specification s to s' such that $M_s \subset M_{s'}$ (i.e. put less requirements on the implementation), and generate a sound test suite for this looser specification: $\underline{\text{gen}}^{C,\underline{\text{imp}}}(s')$ is a sound test suite for s with respect to $\underline{\text{imp}}$.

A special case for a requirement specification $R \subseteq REQS$ is the looser specification $R' \subset R$ consisting of a selection of dynamic conformance requirements.

3) Weaken the implementation relation $\underline{\text{imp}}$ to $\underline{\text{imp}}'$ such that $\{m \in MODS \mid m \ \underline{\text{imp}} \ s\} \subset \{m \in MODS \mid m \ \underline{\text{imp}}' \ s\}$ (i.e. allow more implementations as being correct), and generate a sound test suite for this weaker implementation relation: $\underline{\text{gen}}^{C,\underline{\text{imp}}'}(s)$is a sound suite for s with respect to $\underline{\text{imp}}'$.

4) Make a stronger test assumption (see 6.3), i.e. instead of assuming $m_{IUT} \in MODS$, assume $m_{IUT} \in MODS'$ with $MODS' \subset MODS$.

5) Specify explicitly a fault model $F \subset MODS - M_s$, e.g. by considering mutants Δ_s of s, and generate a sound test suite that at least detects all non-conforming implementations in F, i.e. $P_T \cap F = \emptyset$.

Different test-suite size reduction strategies of this (not necessarily exhaustive) list can be combined to reduce the size of a test suite.

Example

If we have a state-machine specification ranging over the integers, hence with (theoretically) infinite state space, $MODS = SPECS$, and if the implementation relation requires that all sequences of behaviour (*traces*) of the specification are exhibited by the implementation, then the following sequence of test-suite size limiting strategies can be applied sequentially:

- make the stronger test assumption that the number of states in any implementation is constrained to a particular number n;
- consider the weaker implementation relation: all traces of the specification up to length l must be exhibited by the implementation relation;
- consider the looser specification s' such that s' contains only a finite subset of the traces of s.

Now any test generation function that generates sound test suites for the weaker implementation relation, generates, when applied to s', a sound test suite for the original specification with respect to the original implementation relation.

NOTE – Note the difference between a mutant Δ_s and a looser specification s' of s. A mutant describes the expected faults for which tests are generate (the fault model is a subset of the set of implementations specified by Δ_s). A looser specification specifies implementations that are *not* tested (the fault model is a subset of the *complement* of the implementations specified by s').

8.5 Fault coverage

Fault coverage is a normalized measure of the extent that a test suite approximates exhaustiveness with respect to a fault model F. This means that it expresses a quantification of the quality of a test suite in terms of its error-detecting capabilities. A coverage measure can be used to compare test suites; a high coverage expresses a high quality.

Fault coverage is expressed as a function with the following signature:

$$\underline{\text{cov}}_F : Powerset(TESTS) \rightarrow [0,1]$$

with the requirement that the coverage must increase if more erroneous implementations in F are detected [P_T is the formal test purpose of test suite T (see 7.4.1); $F - P_T$ is the subset of F of all models of implementations that fail with T]:

$$F - P_{T_1} \subseteq F - P_{T_2} \Rightarrow \underline{\text{cov}}_F(T_1) \leq \underline{\text{cov}}_F(T_2)$$

If F is omitted, it is assumed to be the set of all erroneous implementations: $\underline{\text{cov}}(T) = \underline{\text{cov}}_{MODS - M_s}(T)$.

8.6 Test suite cost

The costs that are made to generate, maintain, implement, and execute a test suite are expressed as the $\underline{\text{cost}}$ of that test suite. This means that $\underline{\text{cost}}$ expresses a quantification of the efforts in terms of money, time, etc. needed for a test suite. A cost measure can be used to compare test suites; a low cost expresses low efforts.

Cost can be expressed as a function with the following signature:

$$\underline{\text{cost}} : Powerset(TESTS) \rightarrow R_{\geq 0}$$

with the requirement that cost increases with the size of a test suite:

$$T_1 \subseteq T_2 \;\Rightarrow\; \underline{\text{cost}}(T_1) \leq \underline{\text{cost}}(T_2)$$

The cost usually depends on the number and the length of the test cases in the test suite.

9 Compliance

9.1 Introduction

The term *compliance* refers to meeting the requirements specified in this Recommendation. The term is used to distinguish between *compliance* with this Recommendation and *conformance* of an implementation under test to a specification (refer to CTMF).

This Recommendation defines a framework for the use of formal methods in conformance testing. It is intended for implementers, testers, and specifiers involved in conformance testing to guide in defining conformance and the testing process of an implementation with respect to a specification that is given as a formal description.

The framework in this Recommendation is presented at a high level of abstraction, e.g. it abstracts from specific test generation algorithms, even from a specific formal description technique. The framework defines terminology, abstract concepts, and minimal requirements on, and relations between these concepts. Hence, use of the framework requires instantiating these concepts with specific choices for the formal description technique (*SPECS*), for the set of models (*MODS*), for the implementation relation (imp), for test generation algorithms, etc., while demonstrating the requirements on, and the relations between them.

This clause explicitly gives the requirements for such an instantiation to comply with this Recommendation. Moreover, it identifies assumptions underlying the conformance testing process, whose validation is outside the scope of the conformance testing process itself. The requirements for each of the previous clauses are given in separate subclauses of this clause. Annex A provides possible instantiations for formal description techniques.

9.2 Compliance with clause 6: The meaning of conformance

To comply with clause 6, the parties involved in the conformance testing process shall identify and agree on the following:

1) A parameterized specification, referred to as $s : D_s \rightarrow SPECS$ in clause 6 – The parameter space D_s identifies all possibilities for implementation options of the specification. *SPECS* is a formalism of instantiated specifications. The parameterized specification is assumed to be correct and validated, and serves as the reference point for the conformance testing process.

2) An implementation under test IUT, together with its corresponding implementation conformance statement, referred to as ICS_{IUT} in clause 6 – The IUT with corresponding ICS_{IUT} shall conform statically to $s : D_s \rightarrow SPECS$, i.e. $ICS_{IUT} \in D_s$.

3) A modelling formalism, referred to as *MODS* in clause 6, such that any possible implementation under consideration can be assumed to be modelled by an element of *MODS*.

4) An implementation relation, referred to as imp in clause 6, such that $\underline{\text{imp}} \subseteq MODS \times SPECS$, or, if $SPECS = Powerset(REQS)$, with *REQS* a logical or property language, a satisfaction relation, referred to as sat in clause 6, such that $\underline{\text{sat}} \subseteq MODS \times REQS$.

It is understood that the IUT conforms to $s : D_s \rightarrow SPECS$ if, and only if, $ICS_{IUT} \in D_s$ and the model of the IUT, m_{IUT}, is imp-related to $s(ICS_{IUT})$, i.e. $m_{IUT}\ \underline{\text{imp}}\ s(ICS_{IUT})$.

If more than one specification is used to define the set of models of implementations, then the specifications shall be consistent (see 6.4.4). The IUT conforms to the collection of specifications if, and only if, it conforms to each of the specifications.

9.3 Compliance with clause 7: Testing concepts

To comply with clause 7, the parties involved in the conformance testing process shall identify and agree on the following:

1) a tester, an implementation under test IUT, a test context, and their mutual interfaces referred to as Implementation Access Points IAPs (between IUT and test context), Points of Control and Observation PCOs (between IUT and tester);

2) a test suite, referred to by T in clause 7, such that $T \subseteq TESTS$, where $TESTS$ is a test notation;

3) a function $C : MODS \rightarrow MODS$ modelling the test context;

4) for each $t \in T$: a verdict assignment $\underline{\text{verd}}_t : OBS \rightarrow \{\text{pass, fail}\}$, where OBS is a set of possible observations made during test execution;

5) a test execution modelling function $\underline{\text{exec}} : TESTS \times MODS \rightarrow OBS$, that is assumed to correctly model the test execution of a test case with an IUT contained in test context.

It is understood that the test execution of T on IUT passes – IUT passes T – if, and only if, $m_{IUT} \in P_T$ (see 7.4.2).

9.4 Compliance with clause 8: Conformance testing

To comply with clause 8, the parties involved in the conformance testing process shall identify and agree on the following:

1) a test generation function $\underline{\text{gen}}^{C,\underline{\text{imp}}} : SPECS \rightarrow Powerset(TESTS)$, such that the generated test suites are sound for the applicable domain;

2) zero, one, or more test-suite size reduction strategies (see 8.4.2);

3) optionally, a coverage function $\underline{cov}_F : Powerset(TESTS) \rightarrow [0,1]$ with respect to a fault model $F \subseteq MODS$. The function $\underline{\text{cov}}_F$ shall be demonstrated to be a coverage function (see 8.5);

4) optionally, a cost function $\underline{\text{cost}} : Powerset(TESTS) \rightarrow R_{\geq 0}$: the function cost shall be demonstrated to be a cost function (see 8.6).

It is understood that a generated test suite, when executed in compliance with 9.3, gives an indication about the conformance of an IUT in compliance with 9.2. If $\neg$ (IUT passes T), the IUT is not conforming; but if IUT passes T, then only the absence of an indication of non-conformance can be concluded.

Annex A

This Annex is concerned with an interpretation of terminology in the FDTs Estelle, LOTOS and SDL. It does not define new terms which are not already in the main part of this Recommendation. It gives a language specific interpretation to those terms for which it is useful to consider them within the specific context of one of the FDTs.

NOTE – This annex only gives examples to support the general definitions and is not meant to give a recommended style of specification for testability. It is descriptive in nature rather than prescriptive, and informative rather than normative.

A.1 Specifications

A.1.1 Introduction

Specifications are behaviour descriptions in one of the standardized FDTs: Estelle, LOTOS or SDL. The set $SPECS$ denotes the set of all instantiated specifications written in one particular FDT.

A common example is used to illustrate most of the terminology in this Recommendation for the FDTs Estelle, LOTOS and SDL. This enables comparison between the different FDTs and their interrelation with the terminology in this Recommendation.

The common example describes the behaviour of an initiator-entity in a connection oriented protocol, i.e. first a connection is built before data can be sent. In the following, an informal description of the protocol is given. This informal description is not meant to give a precise and complete description of the protocol behaviour. It describes the protocol behaviour to an extent that is sufficient for instantiation as a common example in the different FDTs.

Common example

The common example describes an initiator entity in a connection oriented protocol. The architecture is given in Figure A.1.

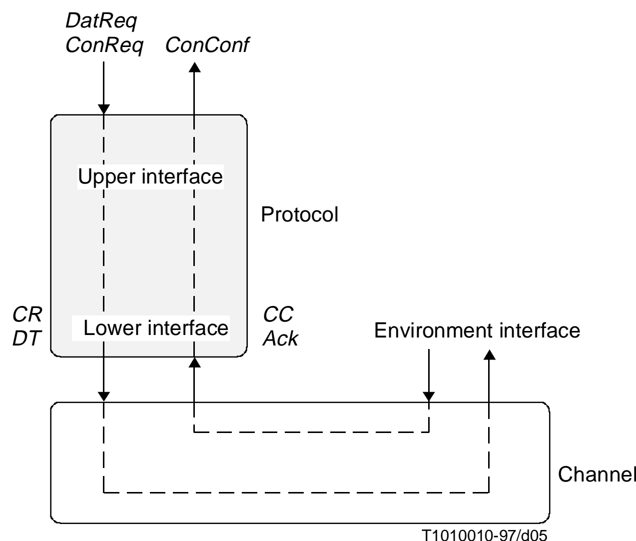

Figure A.1/Z.500 – Common example architecture

Two phases can be distinguished: the *connection phase* (this phase deals with the connection set-up) and the *data phase* (in this phase, data is sent by the initiator). The following constraints apply to the connection phase:

– In the initial state, the initiator waits for an upper level connect request (*ConReq*) to occur.

– An upper level connect request (*ConReq*) is followed by a lower level connect request (*CR*).

– After a lower level connect request (*CR*) a lower level connect confirm (*CC*) is received from the receiving entity.

– A lower level connect confirm (*CC*) is followed by an upper level connect confirm (*ConConf*).

– At any time the initiator entity can receive a disconnect request (*DisReq*) at the upper level. A disconnect request brings the protocol back to the initial state.

The data phase is entered after a connection has been set up. In the data phase, data can be sent from initiator to receiver. The following constraints hold for the data phase:

– Data phase can only begin after an upper level connect confirm (*ConConf*) has been sent.

– An upper level data request (*DatReq*) is followed by a lower level data (*DT*).

– After a lower level data (*DT*), a lower level acknowledgment (*AK*) can occur.

– Only after the reception of a lower level acknowledgement (*AK*) a new upper level data request (*DatReq*) can occur.

– During the data phase, the initiator entity can receive a disconnect request (*DisReq*) at the upper level, which brings the protocol back to the initial state.

The common example also incorporates a description of a test architecture, in order to illustrate the concepts of 7.2. The upper interface of the initiator entity can be directly accessed by the tester, while the lower interface can be accessed only through a *medium*. This medium is assumed to transmit only correctly or loose messages. It cannot create, duplicate or reorder messages.

A.1.2 Specifications in Estelle

An Estelle description of the common example described in A.1.1 can be found in Figure A.2. Note that non-determinism is used in order to model the fact that the channel can lose messages.

```
specification Example;
  default individual queue;
  timescale seconds;

  channel ISAP(User, Protocol);
    by User    : ConReq;DatReq;Dis;
    by Protocol: ConConf;

  channel MSAP(Protocol, Channel_S);
    by Protocol : CR;DT;
    by Channel_S: CC;Ack;

  module Protocol_M
    ip U: ISAP(Protocol);
       L: MSAP(Protocol)
  end;

  body Protocol_B for Protocol_M

  state disconnected, wait, connected, sending;

  initialize to disconnected begin end;

  trans

  from disconnected to wait
    when U.ConReq
      begin output L.CR end;

  from wait to connected
    when L.CC
      begin output U.ConConf end;


  from connected to sending
    when U.DatReq
      begin output L.DT end;

  from sending to connected
    when L.Ack

      begin end;

  from wait, connected, sending to disconnected
    when U.Dis
      begin end;

  end;
```

```
module Channel_M
   ip E: MSAP(Protocol)
      L: MSAP(Channel_S);
end;

body Channel_B for Channel_M

state empty;

initialize to empty begin end;

trans

when L.CR
   begin output E.CR end;
   begin end;

when E.CC
   begin output L.CC end;
   begin end;

when L.DT
   begin output E.DT end;
   begin end;

when E.Ack
   begin output L.Ack end;
   begin end;

end;

modvar Protocol_I: Protocol_M;
       Channel_I: Channel_M;


initialize
   begin
      init Protocol_I with Protocol_B;
      init Channel_I with Channel_B;
      connect Protocol_I.L to
   Channel_I.L
   end;

end.
```

Figure A.2/Z.500 – Estelle specification of the common example

In order to define conformance, implementation relations will be defined in A.4.2. However, it is not convenient to define meaningful implementation relations based on the Estelle syntactic description. The usual way to define them is to express first the semantics of the Estelle specification in some kind of more fundamental (mathematical) model, and then to define implementation relations based on such a mathematical model.

The semantics of the Estelle specification is expressed in some kind of labelled transition systems in which input actions and output actions are distinguished. We call such systems Input-Output State Machines (IOSM).

Definition 1 (IOSM): an input-output state machine (IOSM) is a 4-tuple $M = \langle S, L, T, s_0 \rangle$ where:

– S is a finite non-empty set of states;

– L is a finite non-empty set of interactions;

– $T \subseteq S \times ((\{?,!\} \times L) \cup \{\tau\}) \times S$ is the transition relation. Each element from T is a transition, from an origin state to a destination state. This transition is associated either to an observable action (input *?a* or output *!a*), or to the internal action τ.

– s_0 is the initial state of the IOSM.

The set *SPECS* is chosen to be the set of all IOSM. Figure A.3 shows the IOSM model of the Estelle specification of the common example. Note that the IOSM model does not allow to indicate at which interaction point a given message occurs. This is why messages have to be renamed to indicate different interaction points.

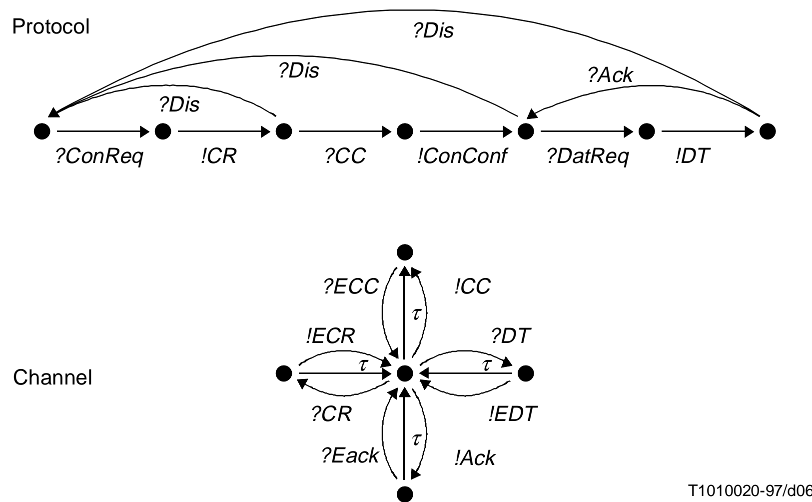

Figure A.3/Z.500 – IOSM model of the Estelle specification from Figure A.2

NOTE – There is no well-accepted established work yet on this subject. The presentation above is based on the work described in [Phalippou 92] and [Phalippou 94], which is one of the currently available works in this field. Other mathematical models could be used to describe the semantics of the Estelle specification in a way suitable for a definition of implementation relations.

We can mention:

– Mealy machines [Hopcroft 79] on which are based most classical test generation methods for input-output systems (W, UIO, DS, etc.);

– a derived model of Mealy machines called PNFSM (partially-specified and non-deterministic extensions of Mealy machines) [Luo 93].

A.1.3 Specifications in LOTOS

A LOTOS specification is a language construct satisfying the syntax described in [ISO 8807]. The purpose of a LOTOS specification is to describe system behaviour. System specifications at different abstraction levels can be obtained by abstraction from irrelevant internal system details.

The following restrictions are inherent to LOTOS specifications:

– LOTOS assumes that communication is *synchronous*, i.e. entities participate synchronously in interactions;

– communication is *atomic*, i.e. either communication between participating entities is successful for all entities, or for none of them;

– LOTOS is only able to describe temporal timing aspects; absolute time aspects cannot be described.

A LOTOS specification of the common example described in A.1.1 is given in Figure A.4.

```
(* specification : Initiator (BASIC LOTOS)

   This specification describes the Initiator Protocol *)

specification Initiator_Protocol[ConReq, ConConf, DatReq, CR, CC
                                 DT,Ack, Dis]                    : exit
behaviour
   Initiator [ConReq, ConConf, DatReq, CR, CC, DT, Ack, Dis]
where
   process Initiator [ConReq, ConConf, DatReq, CR, CC, DT, Ack, Dis] : exit :=
           Connection_Phase [ConReq, CR, CC, ConConf, Dis ] >>
           Data_Phase[ConReq, ConConf, DatReq, CR, CC, DT, Ack, Dis ]
   endproc (* Initiator *)

   process Connection_Phase [ConReq, CR, CC, ConConf, Dis] : exit :=
           ConReq ; CR; (CC ; ConConf ; exit
                         []
                         Dis; Connection_Phase [ConReq, CR, CC, ConConf, Dis])
   endproc (* Connection_Phase *)

   process Data_Phase [ConReq, ConConf, DatReq, CR, CC,
                       DT, Ack, Dis]                   : exit :=
           DatReq ; DT ; (Ack ; Data_Phase[ConReq, ConConf, DatReq, CR, CC,
                                           DT, Ack, Dis]
                          [] Dis; Initiator [ConReq, ConConf, DatReq, CR,
                                             CC, DT, Ack, Dis])
           []
           Dis; Initiator [ConReq, ConConf, DatReq, CR, CC, DT, Ack, Dis]
   endproc (* Initiator *)
endspec (* Initiator_Protocol *)
```

Figure A.4/Z.500 – LOTOS specification of the initiator protocol

The semantics of LOTOS specifications is given as labelled transition systems (LTSs). Labelled transition systems are represented by directed graphs where edges are labelled.

Definition 2 (labelled transition system): A labelled transition system is a 4-tuple $TS = \langle S, L, T, s_0 \rangle$ such that:

– S is a (countable) non-empty set of states;

– L is a (countable) set of observable actions;

– $T \subseteq S \times (L \cup \{\tau\}) \times S$ is the transition relation, where the special label $\tau \notin L$ represents a non-observable (or internal) action;

– $s_0 \in S$ is the *initial state*.

No distinction is made between input actions and output actions since communication is synchronous, and therefore the notion of inputs and outputs does not exist. Asynchronous communication is modelled in LOTOS by explicitly modelling the intermediate medium between communicating entities (see A.6.3).

The formalism of labelled transition systems is very basic to behaviour description formalisms. A lot of other formalisms (e.g. FSMs, EFSMs, IOSMs) can be expressed in terms of labelled transition systems.

For specifications written in LOTOS, the set *SPECS* is instantiated by the set of all labelled transition systems *LTS* that can represent LOTOS specifications. In Figure A.5 the labelled transition system corresponding to the common example described in A.1.1 is depicted. The initial state is indicated by a grey dot.

A.1.4 Specifications in SDL

A specification in SDL of the common example described in A.1.1 is given in Figures A.11, A.12, A.13, A.14, and A.15.

Dynamic conformance is concerned only with the external behaviour of a system. An SDL specification does not explicitly express the external behaviour of a system. So, to test for dynamic conformance a representation of the observable behaviour derived from the SDL specification is needed. A suitable representation for the external behaviour for an SDL specification is by an Asynchronous Communication Tree (ACT) or a Labelled Transition System (LTS). For a formal definition of ACTs, confer to [Hogrefe 88].

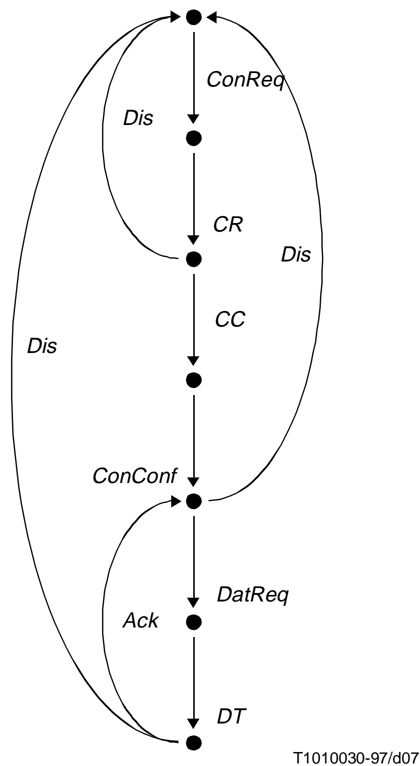

Figure A.5/Z.500 – LTS model of LOTOS specification

Then, when SDL is used for specification of systems the set *SPECS* is the set of all ACTs or LTSs which can be derived from any SDL specification. For the example, a small part of an ACT representation of the observable behaviour is illustrated in Figure A.6. The ACT representation associates to each state information on signals in the different channels. In Figure A.6 only channels that convey a signal in the current state are mentioned. If all channels of the system are empty, this is denoted by $Q_i = \langle\, \rangle$. The state information is however not needed when test cases are defined from the ACT representation.

The term SDL specification in this context only refers to *instantiated specifications*. Generic SDL specifications represent a number of different instantiated specifications, i.e. a generic SDL specification represents a set of representations in the set *SPECS*.

A.2 Implementation options and instantiated specifications

A.2.1 Introduction

The ICS Proforma (IPf) of a formal protocol specification is described as its formal parameters, and the ICS of an implementation as actual parameters. Ideally, the parameterization of the formal specification should be done in such a way that the Static Conformance Review amounts to type-checking in the FDT. This would give a well-studied semantics to these concepts.

There are two aspects to implementation options:

– Variables, that are used, for example, to denote options (e.g. answers to Yes/No questions).

– Restrictions on the possible values of these variables.

The first aspect is easily modelled in any FDT that allows specifications to be parameterized by values. As to the second aspect, these restrictions can be checked by a predicate (Boolean function/expression) using the parameters. However, it is difficult to model the restrictions in a way that would reduce the Static Conformance Review to type-checking.

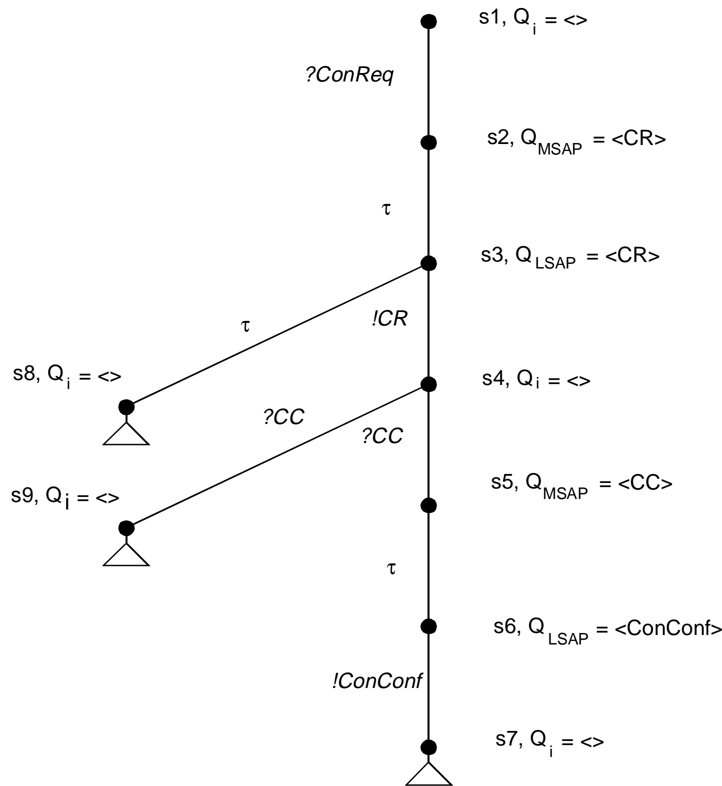

Figure A.6/Z.500 – An ACT representation of part of the observable behaviour

The next subclause gives (limited) examples which, for the moment, support the following conclusion:

> The IPf of a formal protocol specification can be described as its formal parameters and the ICS of an implementation as actual parameters, such that the Static Conformance Review is described by a predicate in the formal specification.

As an example, a simplified version of the Transport Protocol is treated. In this version only the requirements on supported classes are taken into account. The requirements are taken from Table D.5.2 of the Transport Protocol PICS Proforma [ISO 8703] (restricted to classes 0 until 4).

Each row of this table identifies a single option, the value of which can be either Yes or No (Support column). The O in the status field means that the capability is optional. The implementor can fill in either Yes or No. The O.1 in C0 and C2 means that both are optional, but at least one of them should be supported by the implementor (so at least one of them should have the answer Yes). Status C0:O in row C1 means that C1 is optional if C0 is supported, otherwise C1 is prohibited.

Table A.1/Z.500 – Supported classes of the Transport Protocol

Index	Class	References	Status	Support
C0	Class 0	14	O.1	Yes No
C1	Class 1	14	C0:O	Yes No
C2	Class 2	14	O.1	Yes No
C3	Class 3	14	C2:O	Yes No
C4	Class 4	14	C2:O	Yes No

To summarize, the following restrictions are imposed by this table:

- at least Class 0 or Class 2 must be implemented;
- Class 1 requires Class 0; and
- both Class 3 and Class 4 require Class 2.

The next subclauses contain specifications in the FDTs Estelle, LOTOS and SDL, which only show the main ingredients as far as modelling the PICS Proforma is concerned.

A.2.2 Implementation options and instantiated specifications in Estelle

Estelle does not provide built-in constructs for modelling options, but the ICS and PICS proforma can be modelled using standard language constructs. This can be done in several ways. The example below illustrates one way of modelling implementation option and instantiated specifications in Estelle.

The ICS proforma is modelled by a record with Boolean fields which represent the *index* fields of the ICS proforma. The constraints on these fields (i.e. the *status* fields of the ICS proforma) are checked by a function called `Static_conformance_review`. The parameterized specification being modelled by a parameterized Estelle module, then the instantiated specification is created by assigning a value to this parameter in the `initialize` transition.

```
specification Transport_Protocol;

type PICS_Proforma =
      record
        C0 : boolean; (* ref. 14 *)
        C1 : boolean; (* ref. 14 *)
        C2 : boolean; (* ref. 14 *)
        C3 : boolean; (* ref. 14 *)
        C4 : boolean; (* ref. 14 *)
      end;

var PICS : PICS_Proforma;

function Static_Conformance_Review (P : PICS_Proforma) : boolean;
         (* check static conformance requirements: *)
         (* (C0 or C2) and (C1 => C0) and ((C3 or C4) => C2) *)
         begin
           Static_Conformance_Review := ((P.C0 or P.C2) and
                                        (not(P.C1) or P.C0) and
                                        (not(P.C3 or P.C4) or P.C2));
         end;

procedure Get_PICS_Value (var P : PICS_Proforma);
primitive; (* get PICS value *)

(* Channel, Module header, Module body and Module variable definitions *)

initialize
   begin
      Get_PICS_Value(PICS);
      if (Static_Conformance_Review(PICS)) then
      begin
       (* instantiate appropriate modules *)
        ...
      end
   end;
end.
```

A.2.3 Implementation options and instantiated specifications in LOTOS

A LOTOS specification can be explicitly parameterized. The data types ("sorts") used in the parameter list are defined globally. In the example, the only data type used is bool which is taken from the standard library of predefined types. Future enhancements to LOTOS may enable the description of the static conformance requirements in such a way that the Static Conformance Review amounts to type-checking.

```
SPECIFICATION Transport_Protocol [t,n] (c0,c1,c2,c3,c4 : bool) : NOEXIT

LIBRARY Boolean
ENDLIB

BEHAVIOUR ...

ENDSPEC (* Transport_Protocol *)
```

Parameters may be used in Boolean expressions in so-called guards to select/restrict further behaviour.

```
SPECIFICATION Transport_Protocol [t,n] (c0,c1,c2,c3,c4 : bool) : NOEXIT

LIBRARY Boolean
ENDLIB

BEHAVIOUR TPEntity[t,n](c0,c1,c2,c3,c4)

WHERE

   PROCESS TPEntity [t,n] (c0,c1,c2,c3,c4 : bool) : NOEXIT :=
           ...
           [c4] -> Splitting[t,n]
           []
           [not(c4)] -> NoSplitting[t,n]
           ...
   ENDPROC (* TPEntity *)

ENDSPEC (* Transport_Protocol *)
```

The next example illustrates the use of static conformance requirements.

```
SPECIFICATION Transport_Protocol [t,n] (c0,c1,c2,c3,c4 : bool) : NOEXIT

LIBRARY Boolean
ENDLIB

BEHAVIOUR [ClassesConform(c0,c1,c2,c3,c4)] -> TPEntity[t,n](c0,c1,c2,c3,c4)

WHERE

   TYPE StaticConformance IS Boolean
   OPNS ClassesConform : bool, bool, bool, bool, bool -> bool
   EQNS FORALL c0, c1, c2, c3, c4 : bool
        OFSORT bool
        ClassesConform(c0,c1,c2,c3,c4) = (c0 or c2) and
                                         ((c3 or c4) implies c2) and
                                         (c1 implies c0)
   ENDTYPE (* StaticConformance *)

   PROCESS TPEntity [t,n] (c0,c1,c2,c3,c4 : bool) : NOEXIT :=
           ...
   ENDPROC (* TPEntity *)

ENDSPEC (* Transport_Protocol *)
```

NOTE – A draw-back of this way of specifying the static conformance requirements is that if the actual parameters do not satisfy ClassesConform, then the behaviour specified is STOP. This means that an implementation that does nothing at all would be conforming! Something similar may also hold for the examples in the other FDTs. One solution to this problem is to use one of the enhancements to LOTOS, *viz.* the module concept. Next an alternative is given, which is certainly not recommended, since it yields very obscure specifications. It is only shown as an example that the above problem may be circumvented. A data type is used whose values are all possible combinations of values that are statically conforming. This solution is also possible in the other FDTs.

```
SPECIFICATION Transport_Protocol [t,n] (classes : ValidClasses) : NOEXIT

LIBRARY Boolean
ENDLIB

TYPE  ValidClasses
SORTS ValidClasses
OPNS  only0, only01, only2, only23, only24, only234,
      only02, only023, only024, only0234,
      only012, only0123, only0124, only01234:        -> ValidClasses
ENDTYPE (* ValidClasses *)

BEHAVIOUR

  LET c0:bool = has0(classes),
      c1:bool = has1(classes),
      c2:bool = has2(classes),
      c3:bool = has3(classes),
      c4:bool = has4(classes)
  IN TPEntity[t,n](c0,c1,c2,c3,c4)
```

```
  WHERE
  TYPE SupportedClasses IS ValidClasses, Boolean
  OPNS has0, has1, has2, has3, has4 : ValidClasses -> bool
  EQNS OFSORT bool
       has0(only0)  = true
       has0(only01) = true
       ...
  ENDTYPE (* SupportedClasses *)

  ...

ENDSPEC (* Transport_Protocol *)
```

A.2.4 Implementation options and instantiated specifications in SDL

In the FMCT framework, specifications can be parameterized to allow for different *implementation options*. For a specific implementation, the selected options are defined in the ICS document. When these parameter values are applied to the parameterized specification, the result is an *instantiated specification*.

In SDL generic system specifications are used to specify parameterized systems. When the parameters have been applied, the instantiated specification is denoted a *specific system specification*.

In the following example, parameters are indicated by declaring them `EXTERNAL`. In order to check the ICS on static conformance, a special channel is introduced: the error channel. If the specification is parameterized by a not statically conforming ICS, the boolean value `FALSE` is sent on this channel, otherwise nothing is sent on this channel.

```
SYSTEM TransportProtocol

SYNONYM C0 Boolean = EXTERNAL;
SYNONYM C1 Boolean = EXTERNAL;
SYNONYM C2 Boolean = EXTERNAL;
SYNONYM C3 Boolean = EXTERNAL;
SYNONYM C4 Boolean = EXTERNAL;

CHANNEL ErrorChannel
  /* Channel to convey FALSE if the static conf. review fails */
  FROM StatConfReview TO ENV
  WITH Boolean;
ENDCHANNEL;

BLOCK StatConfReview;
  /* This block contains only one process which performs
     the static conformance review */

  SIGNALROUTE ErrorRoute
  FROM StaticReviewer TO ENV
  WITH Boolean;

  CONNECT ErrorChannel AND ErrorRoute;

  PROCESS StaticReviewer (1, 1);
          DCL correct Boolean;

          START;
          TASK correct := (C0 OR C2) AND ((C3 OR C4) => C2) AND (C1 => C0);
          /* if correct = true, then the PICS conform */
          DECISION correct;
                   (TRUE) : STOP;          /* conforming: do nothing    */
                   (FALSE): OUTPUT FALSE; /* not conf: send FALSE via
                                             channel ErrorChannel        */
          ENDDECISION;
  ENDPROCESS;
ENDBLOCK;

BLOCK
  ...
ENDBLOCK;

...

ENDSYSTEM;
```

The example illustrates how the ICS can be encoded in the SDL specification in terms of parameterization of the SDL specification.

System specifications may be defined to allow for different options to be implemented. There may be dependencies between the implementation options. The implementation options and their possible dependencies are defined in the ICS Proforma. In [ISO 9646], the actual options implemented in an implementation are specified in the Implementation Conformance Statement. The example illustrates how the ICS can be encoded in the SDL specification in terms of parameterization of the SDL specification.

A.3 Implementations and models of implementations

A.3.1 Introduction

An implementation is a physical, non-formal object. For all FDTs the set *IMPS* denotes the set of all implementations. Because implementations are non-formal objects mathematical reasoning over implementations is not possible. A model is a formal abstraction of the implementation that is suitable for mathematical reasoning over implementations.

Testing aims at building a model of an implementation based on the observations that can be made from experiments performed on the implementation. The kind of observations that can be made from the implementation after experimentation, determines the abstraction level of the model that is constructed. In general, the more can be observed from the implementation, the more detailed a model can be constructed of that implementation.

In practice, only a limited number of experiments are performed on the implementation. Due to practical feasibility, it is not possible to exactly model every aspect of the implementation. The model obtained by means of testing should only reflect the behaviour of the implementation for the experiments that were performed.

A.3.2 Implementations and models of implementations in Estelle

Implementations are physical objects, the set of which is denoted by *IMPS*. However, for test study at a theoretical level, implementations need to be modelled by mathematical objects. For modelling implementations, we have similar choices as in the case of specifications. One possibility would be to describe implementations in Estelle. However, for the purpose of testing (with the aim of defining implementation relations) it is more convenient to represent implementations by IOSMs. Therefore, the set *MODS* is chosen to be the set of all IOSMs *IOSM*.

Examples of some implementations of the protocol of Figure A.3 are given in Figure A.7. We have limited ourselves to implementations which are "close" to the specification, because when developing products, it is unlikely to produce something that is very far from what is expected. However, as will be explained in the next subclause, some of these implementations are conformant and some others are not conformant.

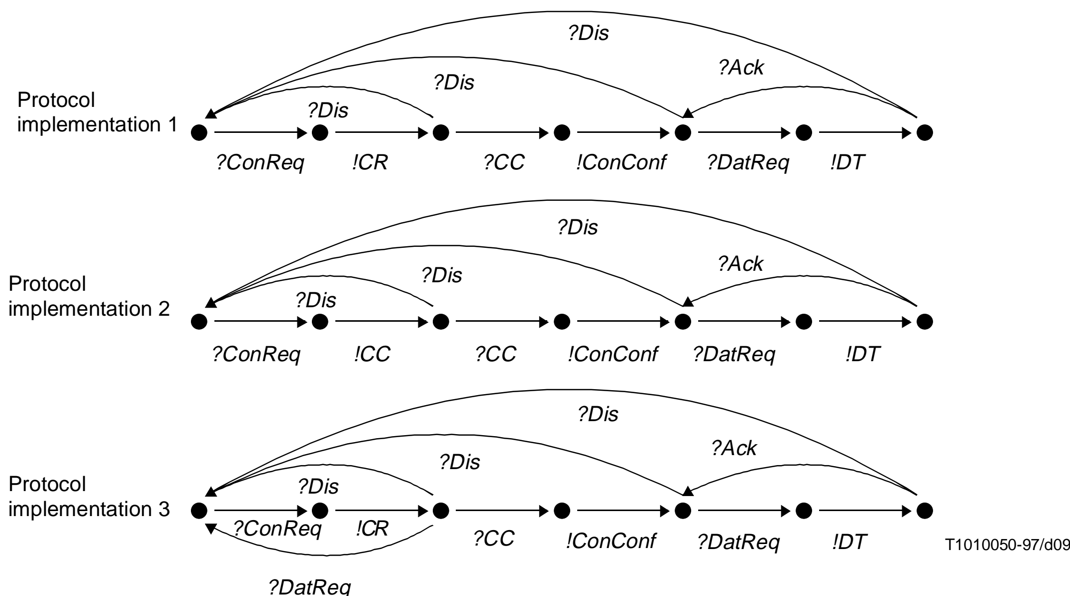

Figure A.7/Z.500 – IOSM models of implementations

A.3.3 Implementations and models of implementations in LOTOS

A prerequisite for building a model of an implementation of which it is claimed to implement a system described by a LOTOS specification is the existence of a suitable formalism to express the model (e.g. the formalism must be able to express all relevant details that is tested for). The formalism must also be chosen in such a way that it is easy to establish whether the implementation satisfies the LOTOS specification [that is, whether the model of the implementation is related to the LOTOS specification by the implementation relation (see A.4.3)].

Many formalisms can be used to describe models of implementations (e.g. transition systems). However, for implementations of systems described by LOTOS specifications, it is common practice to choose the model formalism *MODS* equal to the set of labelled transition system *LTS*.

Figure A.8 depicts some models of implementations of the system described by the specification of Figure A.5.

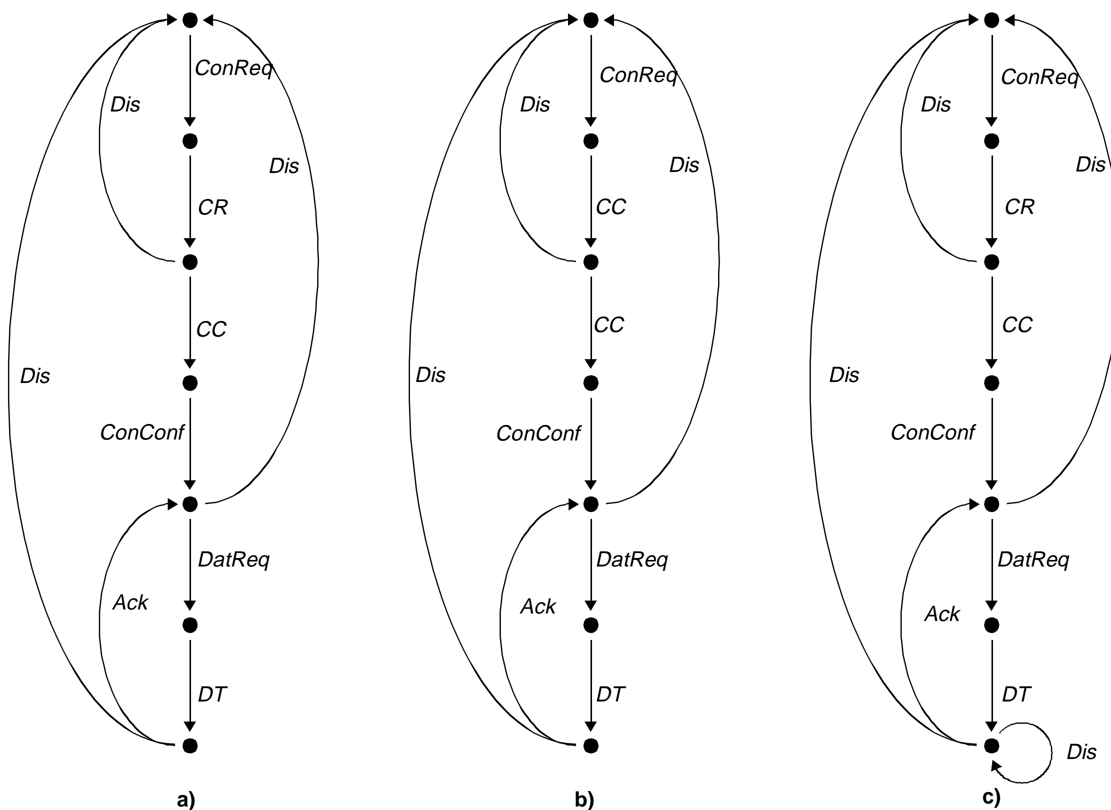

Figure A.8/Z.500 – LTS models of implementations

A.3.4 Implementations and models of implementations in SDL

Implementations denote executable or physical systems that implement systems specified in SDL. The set *IMPS* contains all such implementations.

Formal methods in the conformance testing process can be applied only when the elements of the process are elements with a formal semantics. Then the implementations of the set *IMPS* cannot be used for this purpose. However, it is assumed that for every implementation in *IMPS* there exists a formal model that represents the properties of the implementation. This assumption is denoted a *test assumption* and the acceptance of the assumption is basic to the whole formal approach.

For each implementation in *IMPS* there exists at least one model in the set of models *MODS*. If more models exist for a particular implementation, these are similar to the extent that they cannot be distinguished by test. The set *MODS* must also consist of models that can be basis for definition of formalized relations to the set specifications *SPECS*. It may even be that the formalisms used in the two sets are the same, e.g. ACT or LTS.

In order to illustrate the concepts of the formal framework, the following implementations of the protocol block of the example specification are assumed. The implementations are denoted I_1, I_2, and I_3:

- I_1 is an implementation that has a model that is identical with the model of the specification. This means that the block description of the protocol shown in Figures A.12 and A.13 can be seen as a model also for this implementation.
- I_2 implements the protocol as specified in Figure A.13 as well, except for the behaviour that can be expressed in SDL as shown in Figure A.9. This is, the implementation sends a signal *DT* instead of a *CR* when a connect request is received.
- I_3 similarly implements the protocol as defined in the SDL specification. In addition, in state *connected*, it may receive a new connect request and initiate again the connection phase as shown in Figure A.9.

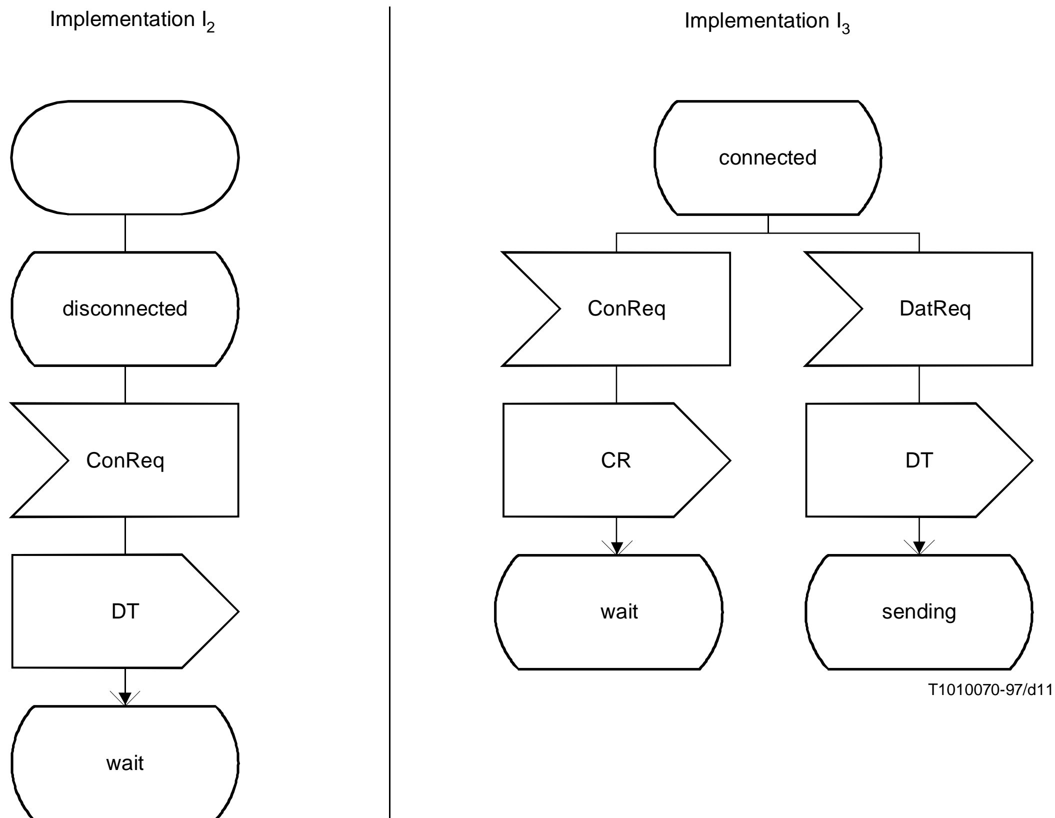

Figure A.9/Z.500 – The changed behaviour of implementations I_2 and I_3

A.4 Conformance by implementation relations

A.4.1 Introduction

An implementation conforms to its specification if the model of the implementation is related to the specification by some implementation relation imp. As such, the relation imp $\subseteq$ *MODS* $\times$ *SPECS* constitutes the notion of correctness of an implementation with respect to the specification.

A.4.2 Conformance by implementation relations in Estelle

As stated above, the definition of the implementation relations is based on the IOSM model rather than on the Estelle syntactic description. Well-accepted implementation relations for input-output automata include trace equivalence or inclusions, or a variant of trace equivalence which is adapted for incompletely specified automata, named quasi-equivalence (defined on PNFSM). When quasi-equivalence is adapted to the IOSM model, this relation becomes the following one [the outputs allowed after a given trace σ in IOSM S are denoted by $O(\sigma, S)$]:

$$R_5(I,S) \text{ iff } (\forall\ \sigma \in Tr(S))\ (\sigma \in Tr(I) \Rightarrow (O(\sigma,I) = O(\sigma,S)))$$

We chose to take this $R_5(I,S)$ relation as an example of imp relation. According to this implementation relation:

- the implementation 1 of Figure A.7 is conformant, which is not surprising, since this implementation is equal to the specification;
- the implementation 2 of Figure A.7 is not conformant: there is a bug in the program of the protocol, and a CC PDU is sent instead of a CR PDU when a *ConReq* is received by the protocol entity;
- the implementation 3 of Figure A.7 is conformant. This may be surprising, since this implementation has a strange behaviour: if a *DatReq* is received by the protocol entity before the link has been established, then the connection is broken (this has the same effect as a disconnection request). The conformance can be explained in the following way: this behaviour belongs to the non-specified part of the protocol, and the chosen implementation relation says that the implementation is free to do anything in the unspecified parts of the specification. However, a different implementation relation may be chosen with a different interpretation of non-specified parts.

A.4.3 Conformance by implementation relations in LOTOS

In case specifications are described in the language LOTOS, an implementation relation is a relation between elements of the set *LTS* (the set chosen to instantiate *MODS* with) and elements of the set *LTS* (the set chosen to instantiate *SPECS* with). Numerous proposals for implementation relations catching an intuitive understanding of correctness have been done. These proposals will not be discussed here. The use of implementation relations in LOTOS will be illustrated by considering one particular example of such relation.

A well-accepted implementation for LOTOS specifications is the *conformance* relation, denoted by conf. The conf-relation captures the idea that an implementation *I* is a correct implementation for specification *S* if, and only if, implementation *I* does not contain deadlocks that were not specified in *S*.

A definition of the conf relation is given below, where *Tr*(*S*) denotes the set of traces of *S*, *L* denotes the universe of observable actions, *a* denotes an observable action and $\overset{\sigma}{\Rightarrow}$ denotes the sequence σ of observable actions:

$$
\begin{aligned}
I \;\underline{\text{conf}}\; S \;=\text{def}\quad & \forall \sigma \in Tr(S), \forall A \subseteq L: \\
& \text{if } \exists I' : I \overset{\sigma}{\Rightarrow} I' \text{ and } \forall a \in A : I' \overset{a}{\not\Rightarrow} \\
& \text{then } \exists S' : S \overset{\sigma}{\Rightarrow} S' \text{ and } \forall a \in A : S' \overset{a}{\not\Rightarrow}
\end{aligned}
$$

If the conf relation is chosen as the implementation relation (that is as the notion of correctness), then it can be checked which of the models given in Figure A.8 are correct implementations of specification model A.5.

- Model **a)** in Figure A.8 is a correct model (of the implementation) of the specification under implementation conf. In particular, the model equals the specification in all its behavioural aspects. Therefore, any implementation satisfying this model is a correct implementation of the specification depicted in Figure A.5 under implementation relation conf.
- Any implementation satisfying model **b)** of Figure A.8 does not conform to specification of Figure A.5. The reason is that the specification is able to perform the sequence of actions *ConReq* · *CR*, while the implementation is not. Because the conf-relation does not allow this to be a correct model of an implementation, the implementation itself is non-conformant.
- Any implementation satisfying model **c)** of Figure A.8 does conform to specification of Figure A.5. Although the implementation can perform the sequence:

  $$ConReq \cdot CR \cdot CC \cdot ConConf \cdot DatReq \cdot DT \cdot Dis \cdot Dis$$

  and the specification cannot, no deadlocks are introduced in the implementation that are not specified. Hence, under implementation relation conf, it can be concluded that all implementations with model **c)** of Figure A.8 are correct implementations of specification.

A.4.4 Conformance by implementation relations in SDL

An *implementation relation* defines a criterion for conformance of an implementation. An implementation is a conforming implementation when the pair of models of the implementation and the SDL specification is a member of the implementation relation.

Several of the implementation relations defined are based on the existence of similar sequences of events in the implementation and the specification. Sequences of observable events can be derived both from ACT and LTS representations of an SDL specification. A sequence of observable events is denoted a *trace* and a *trace set* $Tr(S)$ denotes the set of all possible traces of a specification S. A trace of the ACT shown in Figure A.6 is $\langle ConReq{\cdot}CR{\cdot}CC{\cdot}ConConf\rangle$.

Only a very limited class of SDL specifications specify a behaviour that is expressed by a finite set of traces. The reason is the unbounded queue property of SDL channels and input queues of processes. Even for simple systems specified in SDL, it should be possible to send any number of signals conveyed by the channel from the environment to the system. E.g. in the example specification, it should be possible to send any number of *ConReq* signals to the system. Hence, in practice, exhaustive test of the observable behaviour of an SDL system is not possible. Still the trace models are useful in the definition of a formal requirement for conformance of the dynamic behaviour of an implementation.

A relation that is often used as an implementation relation is the trace inclusion relation $\leq_{tr}$ (trace pre-order). Two models S_1 and S_2 satisfy the relation $S_1 \leq_{tr} S_2$ if, and only if, the trace set of S_1 is a subset of the trace set of S_2.

The trace inclusion relation may be used as an implementation relation in two ways dependent on how the requirements of the specification are interpreted. If the specification is assumed to specify the *maximal allowed behaviour* of an implementation, the trace set of a conforming implementation is a subset of the trace set of the specification. For a conforming implementation I and a specification S this is denoted by $I \leq_{tr} S$ or $(I,S) \in \leq_{tr}$.

The other interpretation of requirements specified by an SDL specification is that it defines the *minimal required behaviour* of an implementation. In this case, the trace set of a specification S is a subset of a conforming implementation I, $I \geq_{tr} S$.

The maximal allowed behaviour implementation relation implies a very weak requirement on a conforming implementation. An implementation that cannot perform any external action is a conforming implementation of any specification, as the empty set is a subset of any trace set. It is not possible to test for the minimal required behaviour implementation relation due to the infinite trace sets of most SDL specifications.

In Table A.2 it is shown which implementations conform with respect to the specified example (here denoted by S) and the two implementation relations.

Table A.2/Z.500 – Overview of the conformance of I_1, I_2, and I_3 with respect to different implementation relations

(Impl, Spec) satisfies	$\leq_{tr}$	$\geq_{tr}$
(I_1,S)	true	true
(I_2,S)	false	false
(I_3,S)	false	true

As the trace set of implementation I_1 is identical to that of the specification, both the maximal allowed behaviour and minimal required behaviour relation are satisfied for I_1. Implementation I_2 does not satisfy $\leq_{tr}$ as the trace $\langle ConReq{\cdot}DT\rangle$ is not a member of the trace set of the specification. Similarly, the implementation relation $\geq_{tr}$ is not satisfied either, as the trace set of I_2 does not include the trace $\langle ConReq{\cdot}CR\rangle$. For implementation I_3 the implementation relation $\leq_{tr}$ is not satisfied as the implementation can, for example, perform the trace $\langle ConReq{\cdot}CR{\cdot}CC{\cdot}ConConf{\cdot}ConReq{\cdot}CR\rangle$, that is not a trace of the specification. As I_3 can perform every trace of the specification it conforms according to the implementation relation $\geq_{tr}$.

The model of an implementation is not known in advance for conformance testing. It can only be approximated by performing experiments on the implementation and observing responses. Non-deterministic system specifications make it impossible to ensure that an implementation model is a complete description of the possible behaviour.

In the example specification it may not be possible to determine if an implementation can perform a specific trace. This is the case for trace ⟨*ConReq*·*CR*⟩. The unreliable channel may always discard the signal *CR* such that it never occurs as an observable event in the environment. However, it is not possible to derive from a number of experiments in which the signal *CR* has not been observed whether the trace has been implemented. So the trace preorder implementation relations may provide a sound basis as a conformance criterion only if an additional assumption is made on the specification and/or implementation. For instance, it may be assumed that information is provided on how non-determinism of the specification is resolved in the implementation.

Another approach is to define implementation relations that take into account limitations of real conformance testing of systems specified in SDL. The relations **asco** and **aconf** are two such implementation relations proposed in [TrVe 92] for implementation relations that can be used for SDL systems that communicate with the environment via a single channel.

To satisfy these implementation relations only a subset of the trace set is considered. This subset includes only traces that are minimal with respect to adding signals to a channel and reordering of external events due to delays on channels. For instance, the trace ⟨*ConReq*·*CR*⟩ is a minimal trace that the implementation is to be tested for. Examples of traces for which this trace is minimal are ⟨*ConReq*·*CR*·*ConReq*⟩ where an input signal is appended and ⟨*ConReq*·*ConReq*·*CR*⟩ where the input signal *ConReq* is shifted in front of output signal *CR*.

The **asco** relation is satisfied if for each minimal trace with the last action being an output signal, the implementation can perform only a subset of the actions possible according to the specification. For **aconf** this must be true also for all prefix traces of the minimal traces. How the three implementations are distinguished by the two relations is shown in Table A.3.

Table A.3/Z.500 – Conformance of I_1, I_2, and I_3 with respect to the implementation relations asco and aconf

(Impl, Spec) satisfies	asco	aconf
(I_1,S)	true	true
(I_2,S)	false	false
(I_3,S)	true	true

For implementation I_1 both of the implementation relations are satisfied as for every trace the same set of events may be observed for the specification and the implementation. For implementation I_2 the minimal trace ⟨*ConReq*·*CR*⟩ where the set of events after ⟨*ConReq*⟩ for the implementation is {*deadlock*,*DT*} while the specification event set is {*deadlock*,*CR*}. As the implementation event set is not a subset of the specification, I_2 is a non-conforming implementation for both relations. Finally, as the implementation is tested only for minimal traces of the specification, the kind of additional behaviour as in implementation I_3 is not considered and I_3 is a conforming implementation.

A.5 Conformance by requirements

Conformance between an implementation and a specification can alternatively be characterized by means of requirements. For the denotation of requirements, a requirement language *REQS* is used. An implementation conforms to its specification if the properties in languages *REQS* which hold for the specification are satisfied by the implementation.

For instance, the language *REQS* can be instantiated by logical languages, e.g. Hennessy-Milner Logic, CTL. Checking whether a model of an implementation satisfies a certain requirement is called model checking. Model checking can only be performed as the model of an implementation is explicitly available.

As another example, we introduce below a simple property language. This property language can be used to express properties on labelled transition systems.

For $\sigma \in L^*$ and $A \subseteq L$ the set *REQS* is instantiated by $\text{REQS}_{\sigma} = \{$ **after** σ **from** $A \mid \sigma \in L^*$ and $A \subseteq L$ }. A labelled transition system *T* satisfies property **after** σ **from** *A* if $\exists T' : T \overset{\sigma}{\Rightarrow} T'$ and $\forall a \in A : T' \overset{a}{\Rightarrow}$.

In Figure A.8 models **a)** and **c)** satisfy property **after** *ConReq* **from** {*CR*}, but model **b)** does not satisfy this property.

A.6 Test architecture

A.6.1 Introduction

The test architecture is a description of the environment in which the IUT is tested. It is necessary to model the environment of the IUT because the behaviour of the IUT may not be directly observable by the tester due to possible limitations in observability imposed by, e.g. an intermediate communication medium between tester and IUT.

Each FDT has facilities for modelling the environment of an IUT.

A.6.2 Test architecture in Estelle

Test architecture is described in Estelle through the following constructs:

– *module* constructs are used to describe the components (tester, IUT, test context);

– *body* constructs describe the behaviour of the various components;

– *interaction points* represent the PCOs and the IAPs.

Since Estelle modules are linked by channels which have an FIFO queue semantics, these modelling choices are compatible with the PCO semantics as described by ISO 9646 (and which corresponds to the TTCN PCO semantics).

This approach provides an explicit modelling of the test context as a component by its own. Some attempts have been made to represent the test context in a way closer to what is described in the main body of this Recommendation, i.e. as a transformation function on the *MODS* set (in our case: on the set of IOSM): see [Phalippou 92] for instance.

A.6.3 Test architecture in LOTOS

LOTOS does not provide *special* constructs for modelling the behaviour of system environments. Instead, the environment is specified just like any other process. No distinction is made between modelling environments and modelling systems within environments.

Since communication in LOTOS is synchronous, asynchronous communication is modelled by explicitly modelling the intermediate communication medium (e.g. FIFO queues). In the specification below, a basic LOTOS specification communicating through a queue is given. In this example, all inputs for the system occurring at gate `a` must travel through a reliable queue. Communication between the queue and the system is hidden. Note that the queue is not an FIFO queue.

```
specification QueueCommunication[a, b, x, y] : noexit
behaviour
  hide ia in System [ia, b, x, y] |[ia]| Queue [ia, a]
where
  process System [a, b, x, y] : noexit:=
  a ;  x ;  stop [] b ;  (x ;  stop [] y ;  stop )
  endproc (* System *)

  process Queue[ia, a] : noexit:=
  a ; (ia ; stop ||| Queue [ia, a])
  endproc (* Queue *)
endspec (* QueueCommunication *)
```

In LOTOS, the tester, the IUT and the test context are modelled by LOTOS processes. PCOs and IAPs are modelled by gates (i.e. constructs in LOTOS to model interaction points).

A.6.4 Test architecture in SDL

The only requirement an SDL specification puts on the environment of a system is that it obeys the constraints given by the system specification. Then, in order to model the properties of an environment in which an implementation is to be tested, the test context must be specified as part of the system specification. In the common example the unreliable medium (*Channel*) can be seen as such a test context for the protocol.

The entities of the test architecture are defined in an SDL specification using the same constructs as for the original system specification. The tester, the test context and IUT can be specified using the *block* construct. The behaviour of the entities are defined by the *processes* of the blocks. Communication between the test architecture entities (blocks) is modelled by channels. As channels in SDL are FIFO-queues, they can be used to model PCOs in accordance with [ISO 9646]. Channels are used also to specify IAPs.

Referring to the common example, the entities of the test architecture can be identified as follows in the system diagram Figure A.13. The block *Protocol_M* specifies the IUT and the properties of the test context are specified by the block *Medium_M*. The PCOs of the test architecture are the channels *ISAP* and *LSAP* while the IAPs are the channels *ISAP* and *MSAP*. In this example, both the IUT and test context are specified by a single block, in general a set of blocks may be used to specify either of these entities.

A.7 Specifications of tests

A.7.1 Introduction

The tests are the procedures which are used to check conformance of the implementations with respect to the specification. Test systems can be seen as distributed components which interact with the implementations. Since they are just particular kind of distributed systems, tests can be described using the standard FDTs Estelle, LOTOS and SDL. This is studied in A.7.2, A.7.3 and A.7.4.

However, the conformance testing framework of [ISO 9646] makes use of some particular concepts (such as test architecture, PCOs, test suite structure, verdicts) which are not built-in constructs of the standard FDTs Estelle, LOTOS and SDL. On the contrary, TTCN language has been specifically designed to formally describe test cases. It is studied in A.7.5.

A.7.2 Specifications of tests in Estelle

Estelle has been designed as a specification language, not as a test description language. Therefore, contrary to TTCN, Estelle has no built-in constructs which represent the test concepts which appear in the main body of this Recommendation (test case, test suite, test event, verdicts). Of course Estelle can be seen as a programming language, and it is possible to describe in Estelle the behaviour part of the test cases (which, in an input-output framework, correspond to outputs and inputs of messages). Moreover, the above mentioned test concepts can be modelled in Estelle by making some particular choices (e.g. use a variable to code the verdict, use one body to represent the behaviour of one test case, etc.). But these are only particular tricky modelling choices. As a consequence, Estelle will not be used in this Annex as a test description language. When specifications are described in Estelle, TTCN will be used as the test description language. Since both languages use FIFO queue input-output communication mechanism, there is no problem to relate them for test execution (see A.8).

A.7.3 Specifications of tests in LOTOS

LOTOS itself can be used for the specification of tests [Bri 88, Tre 92]. A test case can be modelled by a LOTOS process where each state is labelled by one of the elements {**pass**, **fail**, **inconclusive**}. An example of such a test is given in Figure A.10.

It is not easy to model verdicts in LOTOS. One possibility is to use the special label σ (successful termination) for passing a verdict.

Similar to Estelle and SDL, LOTOS is not designed to describe test cases. The language TTCN has been designed for the denotation of tests. However, TTCN expressions represent test cases that are asynchronous by nature. Because LOTOS communication is synchronous, this can be a problem in case tests are described by TTCN expressions.

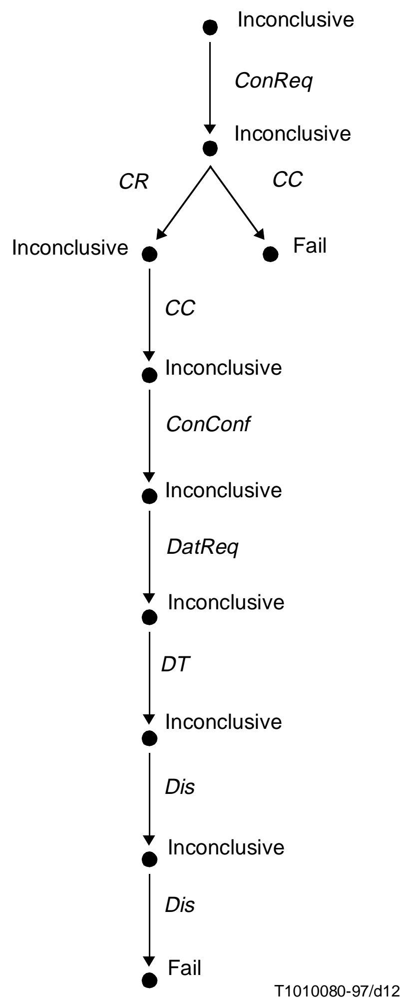

Figure A.10/Z.500 – Test case representation as labelled transition system

A.7.4 Specifications of tests in SDL

SDL is a general purpose specification technique for modelling communication systems. It may be used also for specification of test cases for conformance testing. However, it is not designed specifically for this application area. So to use SDL for test case specification as defined in [ISO 9646], the concepts of this Recommendation must be defined in terms of SDL constructs.

Most of the [ISO 9646] concepts can be modelled in SDL directly. The concepts of PCOs and message exchanges can be modelled by channels and signal instances. A test case may be modelled as an SDL procedure and a test suite as a process. Then the order of test case execution is defined in the process definition. The verdict assignments can be modelled by variables associated to each test case. The verdict assignments that result from test case executions may be passed to the environment via signals. This approach is defined in [R1072]. As ASN.1 may be used in combination with SDL for data definitions [Z105] rules for data in [ISO 9646] can be used directly.

A few concepts defined in the test methodology of [ISO 9646] cannot be specified in SDL. This is the case for real time requirements and handling of unforeseen messages. In SDL real time requirements cannot be modelled as the SDL semantics defines only a discrete model for time and nothing is stated on the time taken to perform a transition. In an SDL specification meaning is given only to declared signals. Then, handling of unforeseen messages that in TTCN is covered by the *otherwise* construct can be modelled only through explicit declaration of signals to represent such messages.

A.7.5 Specification of tests in TTCN

The standardized language Tree and Tabular Combined Notation (TTCN), has been designed for the denotation and specification of conformance test cases which can be expressed abstractly in terms of control and observation of protocol data units and abstract service primitives. TTCN is provided in two forms:

– a graphical form suitable for human readability;

– a machine-processable form suitable for transmission of TTCN descriptions between machines.

The test cases described by TTCN expressions denote *abstract* test cases, which are compiled into executable test cases to be run on the physical test machine.

TTCN is a high level language for specifying the test cases. However, just as in the case of specifications (see A.1), it is easier to define the formal things on a more basic model. Since TTCN has an input-output semantics, we use *IOSM* as a semantic model of TTCN test cases.

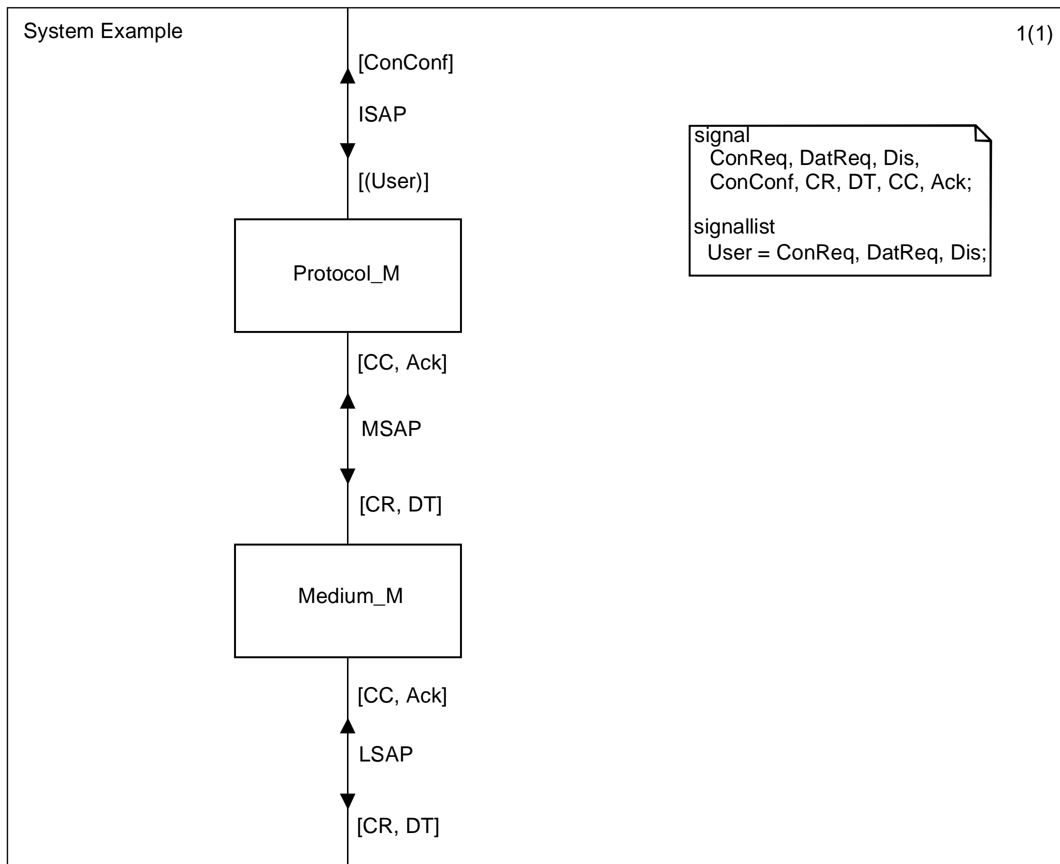

Figure A.11/Z.500 – The system diagram of the protocol

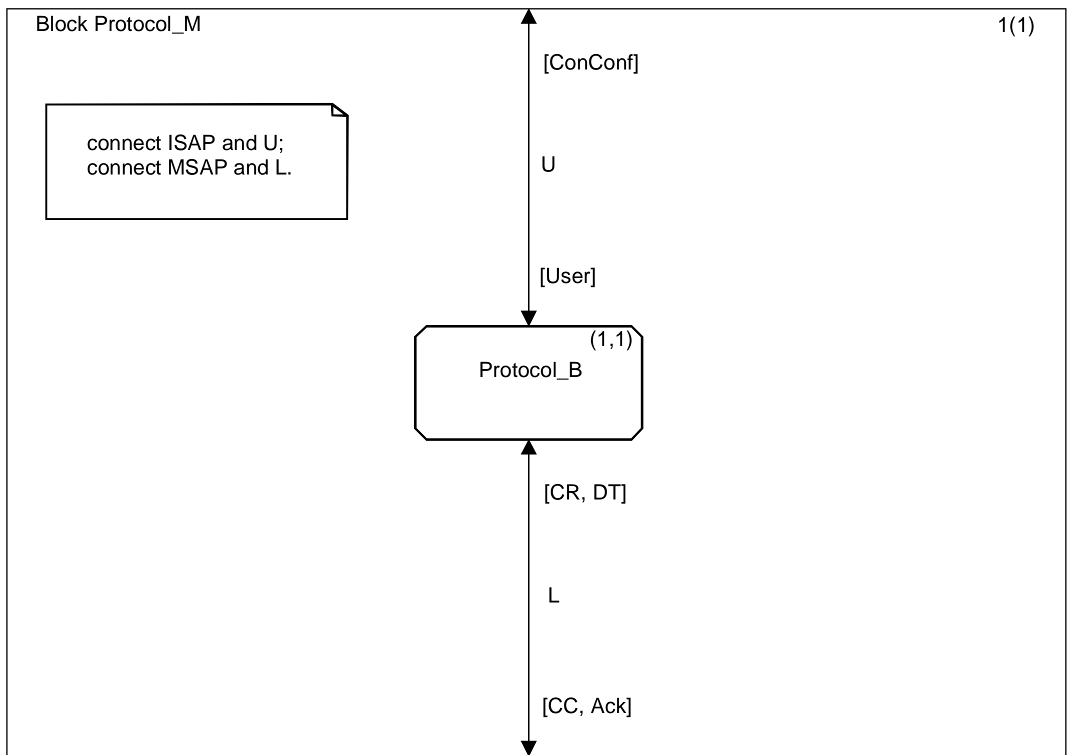

Figure A.12/Z.500 – The block of the sending process

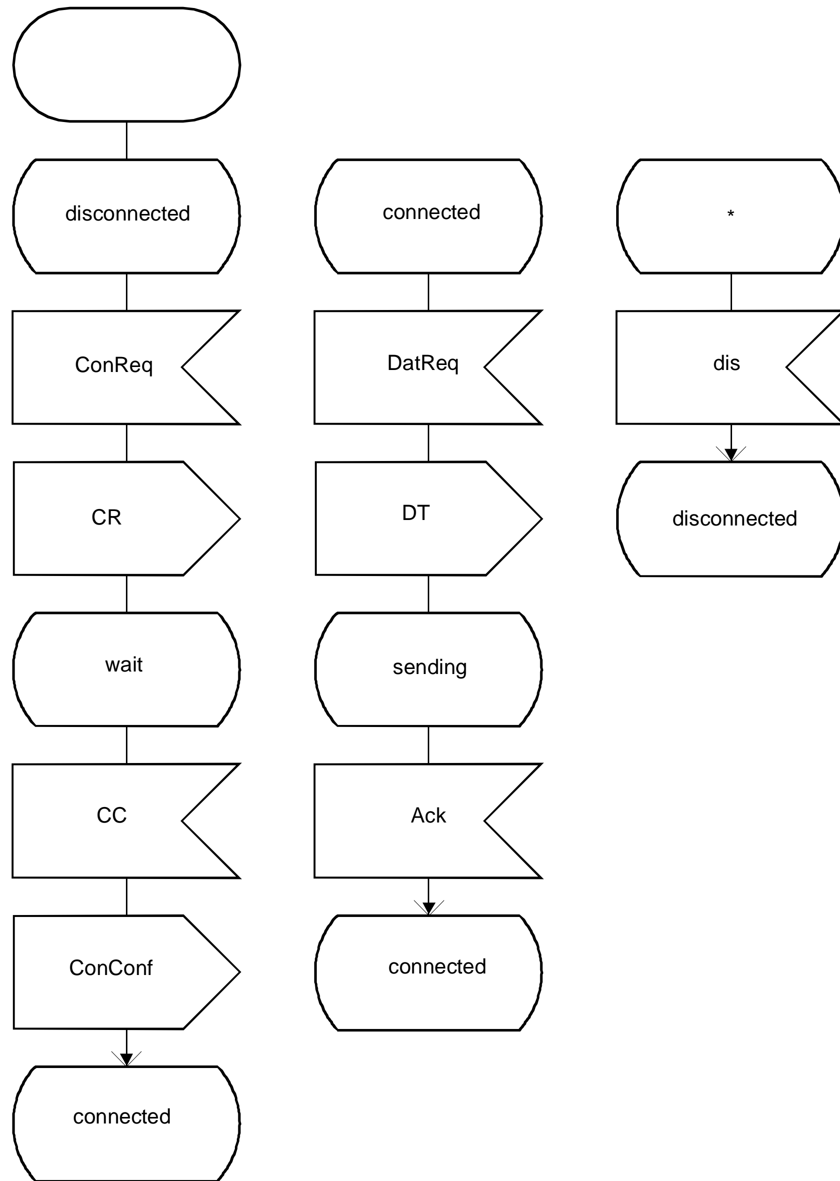

Figure A.13/Z.500 – The sending process

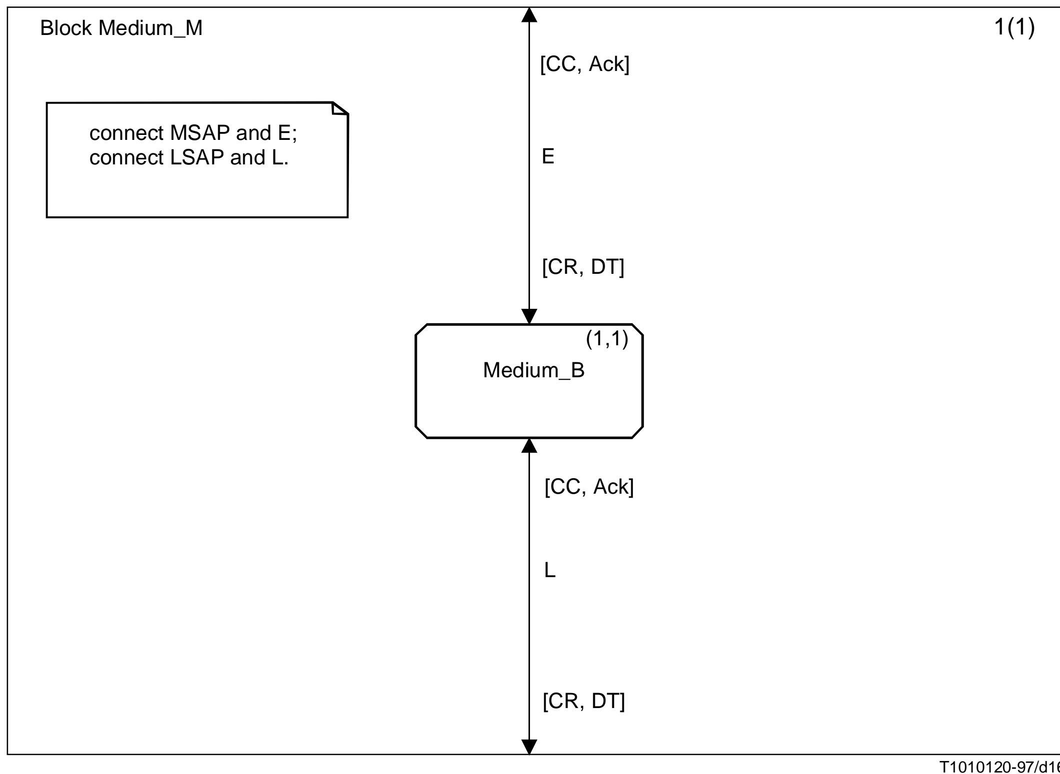

Figure A.14/Z.500 – The block modelling the unreliable medium

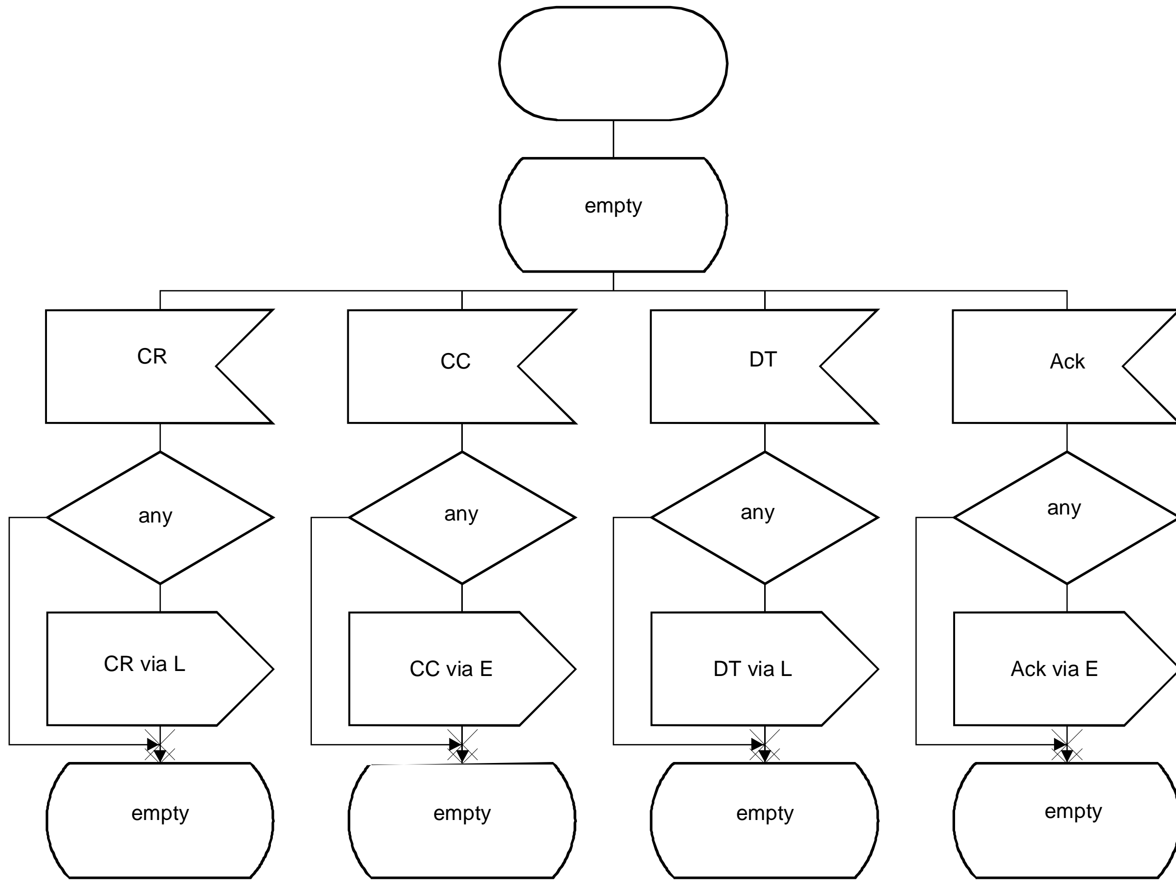

Figure A.15/Z.500 – The behaviour of the unreliable medium

A.8 References

[Bri 88] BRINKSMA (E.): A theory for the derivation of tests, *Protocol Specification*, Testing and Verification VIII, p. 63-74, North Holland, 1987.

[Hogrefe 88] HOGREFE (D.): Automatic generation of test cases from SDL specifications, *SDL Newsletter*, No. 12, 1988.

[Hopcroft 79] HOPCROFT (J.) and ULLMAN (J.): Introduction to automata theory, languages and computation, Addison-Wesley publishing company, 1979.

[ISO 8073] *Information technology – Telecommunications and information exchange between systems – Open Systems Interconnection – Protocol for providing the connection-mode transport service*, International Standard IS-8073, 3rd edition, 1992.

[ISO 8807] *Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*, International Standard IS-8807, ISO, 1989.

[ISO 9646] *Information technology – Open Systems Interconnection – Conformance testing methodology and framework*, International Standard IS-9646, ISO.

[Luo 93] LUO (G.), PETRENKO (A.) and BOCHMANN (G.V.): Selecting test sequences for partially-specified nondeterministic finite state machines, Publication No. 864, Université de Montréal, février 1993.

[Phalippou 92] PHALIPPOU (M.): The limited power of testing, *Proceedings of the 5th International Workshop on Protocol Test Systems*, Montréal, September 1992.

[Phalippou 94] PHALIPPOU (M.): Relations d'implantation et hypothèses de test sur des automates à entrées et sorties, *Thèse de* l'Université *de Bordeaux I*, September 1994.

[R1072] ITACA, IBCN Testing Architecture for Conformance Assessment, R1072 ITACA – No. 365, 1992.

[Tret 92] TRETMANS (J.), A Formal Approach to Conformance Testing, *PhD thesis*, University of Twente, 1992.

[TrVe 92] TRETMANS (J.) and VERHAARD (L.): A queue model relating synchronous and asynchronous communication, *Memorandum INF-92-04*, University of Twente, Enschede, The Netherlands. TFL RR 1992-1, *Tele Danmark Research*, 1992.

[Z105] ITU-T Recommendation Z.105, *SDL Combined with ASN.1 (SDL/ASN.1)*, ITU, 1995.

ITU-T RECOMMENDATIONS SERIES

Series A	Organization of the work of the ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks and open system communication
Series Z	**Programming languages**