
Methodology Guidelines

Introduction	iv
1. The TTCN Introduction	1
Introduction	2
Background	3
Black Box Implementations	3
Lower Tester and Upper Tester	3
Test Notation	4
Forms of TTCN	5
Requirements on TTCN	5
A Case Study	6
The Test Case	6
Case Study Road-Map	8
The Test Configuration	9
Specification of Test System Behaviour	13
Behaviour Trees	13
Statement Lines	15
Execution and Matching	16
TTCN Types and Values	19
Predefined Types	19
Value Denotation	20
Simple User Defined Types	20
Structured Types	22
ASN.1 Types and Values	23
Type References, Value References and Identifiers	23
Identifiers and Underscore	23
ASN.1 Simple Types	24
ASN.1 Constructors	26
PCOs and CPs	29
The Communication Model	29
Sending an ASP	29
Receiving an ASP	30
Declaring PCO Types	30
Declaring PCOs	31
Using PCOs and CPs	32
PCO and CP Snapshots	32
Declaring CPs	33
The SEND Statement	34
Sending an ASP	34

Executing a SEND Statement	34
Sending a PDU	35
Sending a Coordination Message	35
The RECEIVE Statement	35
Receiving an ASP	35
Executing a RECEIVE Statement	36
Receiving a PDU	36
Receiving a Coordination Message	37
The OTHERWISE Statement	37
Defining ASP, PDU and CM Types	38
Complex TTCN Types	38
Chaining	39
Complex ASN.1 Types	39
Local Type Definitions	40
Type Definitions by Reference	40
Defining ASPs	42
Defining PDUs	43
Substructuring ASPs and PDUs	46
Defining Coordination Message Types	48
Using ASPs and PDUs in Behaviour Trees	50
TTCN Expressions	51
TTCN Operators	51
TTCN Operations	53
Specifying ASP, PDU and CM Values	56
Static and Dynamic Chaining	56
Complex ASN.1 Values	57
ASP Constraints	58
PDU Constraints	60
Structured Type Constraints	62
CM Constraints	64
Constraint References	66
Parameterized Constraints	67
Sending and Receiving Constraints	68
Matching Received Constraint Values	73
Specific Values	73
Matching Mechanisms	76
Encoding	81
Encoding ASPs	81
Encoding PDUs	81
Manipulation of Encodings	81

Referencing Components of Complex Types	82
References in the Context of SEND and RECEIVE	82
Referencing ASN.1 Elements	83
Capturing Incoming ASPs and PDUs	85
Verdicts	86
The Result Variable	86
Preliminary Results	87
Final Verdicts	87
The GOTO Statement	89
Timer Statements	90
The Timeout List	91
The TIMEOUT Statement	91
Timer Snapshots	92
The START Timer Operation	92
The CANCEL Timer Operation	93
Constants and Variables	94
Test Suite Constants and Test Suite Parameters	95
Test Suite and Test Case Variables	97
Variables in Concurrent TTCN	98
Dynamic Behaviour Descriptions	99
Test case Identifiers and Test Group References	100
Test Purpose and Objective	100
Configuration	100
Default Behaviour	100
Using Aliases	102
Modularization of Test Cases	104
Test Steps	104
Default Behaviour	108
Parameter Lists in TTCN	112
Formal Parameter Lists	112
Actual Parameter Lists	112
Call-By-Reference	112
Call-By-Value	113
Test Case Selection	113
Selection Expressions	113
Structure of a TTCN Test Suite	114
Parts of a Test Suite	114
Suite Overview Part	114
Declarations Part	114
Constraints Part	116

Dynamic Part	117
Distributed Development	118
The Complete Case Study	120
Suite Overview Part	120
Declarations Part	121
Constraints Part	129
Dynamic Part	134

IBM Rational TTCN Suite 6.3

Methodology Guidelines

This edition applies to IBM Rational SDL Suite 6.3 and IBM Rational TTCN Suite 6.3 and to all subsequent releases and modifications until otherwise indicated in new editions.

Copyright Notice

© Copyright IBM Corporation 1993, 2009.

US Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send written license inquiries to the following:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions. Therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
1 Rogers Street
Cambridge, Massachusetts 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Additional legal notices are described in the `legal_information.html` file that is included in your software installation.

Copyright License

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_.

Trademarks

See <http://www.ibm.com/legal/copytrade.html>.

IBM, the IBM logo, and `ibm.com` are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.html.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

Introduction

About this Manual

This volume, [Methodology Guidelines](#), contains a selection of topics that we hope will assist you in understanding how to take advantage of the TTCN language in a TTCN Suite environment.

The TTCN language, concepts, and data types are described, including the use of ASN.1. The features of TTCN are introduced using a bottom-up approach with the help of a simple case study. Concurrent TTCN is also addressed.

Documentation Overview

A general description of the documentation can be found in [“Documentation” on page viii in the Release Guide](#).

Typographic Conventions

The typographic conventions that are used in the documentation are described in [“Typographic Conventions” on page x in the Release Guide](#).

How to Contact Customer Support

Detailed contact information for IBM Rational Customer Support can be found in [“How to Contact Customer Support” on page iv in the Release Guide](#).

Chapter

1

The TTCN Introduction

This chapter is intended to provide an easy – but not necessarily trivial – introduction to TTCN for the beginner.

Introduction

ISO/IEC 9646 (ITU X.290 series) is a five-part standard which defines a framework and methodology for conformance testing of implementations of OSI and ITU protocols. The test notation, the *Tree and Tabular Combined Notation* (TTCN), is the third part of this standard, i.e. ISO/IEC 9646-3.

The use of TTCN is increasing, and as the notation has now become an ISO International Standard and a ITU Recommendation, we believe there exists a need for a guideline on TTCN.

We do not describe the TTCN in the same order that it is presented in the standard, but instead have used a bottom-up approach. We begin by introducing some basic TTCN features developed round a simple example. Additional features are introduced as required. We have tried to concentrate on aspects that have been introduced in the IS version of the notation, especially concerning the use of ASN.1. We also address the concurrent TTCN.

We hope that readers will find this approach instructive. The guidelines intends to provide:

- an easy, but not necessarily trivial, introduction for the new-comer to TTCN;
- an overview of the TTCN for those users of the notation who are familiar with earlier versions and who require a quick up-date on the later features; note, for example, the summary of the extensions.

Background

Among other things, parts 1 and 2 of ISO/IEC 9646 define the basic concepts and abstract methods that are the cornerstone of standardized conformance testing. It is beyond the scope of this guideline to examine these concepts in detail, but a knowledge of the terms and concepts described below is helpful to understanding the TTCN.

Black Box Implementations

One of the basic premises of the conformance standard is that the implementation of the protocol, called an *implementation under test* (IUT) is a *black box*.

Any conclusions that we may draw about the conformance of that IUT will be made by observing and controlling the events that occur at the lower and upper service interfaces of the IUT. In ISO/IEC 9646 terms these interactions occur at *points of control and observation* (PCO) and are expressed in terms of *protocol data units* (PDUs) embedded in *abstract service primitives* (ASPs).

An IUT is tested by a *test system*. In TTCN the different parts of the test system are called *test components*. A test component created by the main test system is referred to as a Parallel Test Component (PTC).

Lower Tester and Upper Tester

In simple terms, the test components which communicate with the IUT via the PCOs at the lower interface are collectively called the *lower tester* (LT). The test components which communicate with the IUT via the PCOs at the upper interface are collectively called the *upper tester* (UT). There must be at least one test component always present in the test system. This is called the *master test component* (MTC) and it is responsible for coordinating and controlling the test and for setting the final verdict of the test.

Communication between test components in the LT is achieved via *coordination points* (CP). Similarly, UT test components may communicate with each other via CPs.

Coordination between the LT and the UT is achieved by *test coordination procedures* (TCP).

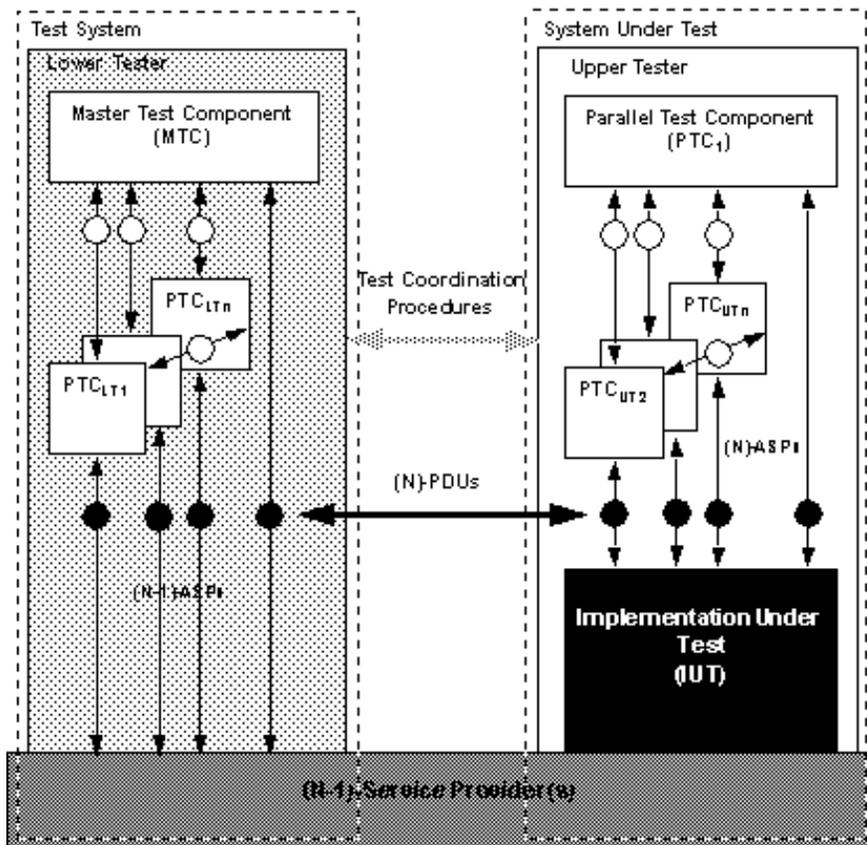


Figure 1: Generalized parallel test architecture (illustrated for a single-layer implementation)

The lower tester is the more complex of the two components as it is also responsible for the control and observation of the *protocol data units* (PDUs) embedded in the ASPs that it sends and receives. In fact, at any given time the LT, when executing a test case, is implementing a portion of the relevant protocol.

Test Notation

In order to test the IUT we need to specify the sequences of interactions, or *test events*, that we wish the test system to control and observe. A se-

Background

quence of such events that specify a complete *test purpose* is called a *test case*. A set of test cases for a particular protocol is called a *test suite*.

The TTCN is a notation that has been developed for the specification of test cases at a level that is abstracted from the architecture of any real test system that these test cases may eventually be run on.

The abstract test cases contain all the information that is necessary to fully specify the test purpose in terms of the protocol that the IUT is supposed to implement. It does not include test system specific information. However, this does not mean to imply that the notation itself is abstract - during the last few years the definition of TTCN has become very precise, with regard to both syntax and operational semantics, and is now close to a programming language.

Forms of TTCN

The main body of ISO/IEC 9646-3 defines the graphical form of the notation (TTCN-GR), where all information is presented using *tables*. There is also an underlying machine processable format (TTCN-MP) specified in an extended form of BNF (Backus-Naur Form). In this guideline we shall concentrate on the TTCN-GR.

Requirements on TTCN

From [Figure 1](#) it can be seen that, generally speaking, the ISO conformance standard requires that tests are specified in terms of (N-1)-layer ASPs, (N)-layer ASPs and (N)-layer PDUs. In order to fulfil these requirements the minimum functionality that the TTCN should provide is:

- the ability to specify the ASPs to be sent and/or received by the test system;
- the ability to specify the PDUs embedded in the ASPs;
- specification of the order in which ASPs are to be sent and/or received at specific PCOs.

In order to do this the TTCN allows:

- declaration of ASP and PDU *types*;
- declaration of PCOs;
- specification of *actual* ASPs and PDUs;
- specification of instances of behaviour.

We shall examine in detail how these, and other, features are supported in TTCN to make it a powerful notation for specifying abstract test cases.

A Case Study

The Test Case

For the purposes of this guideline we shall invent a simple case study for an imaginary protocol, which we shall call the *X-Protocol*. The case study is based on the architecture introduced in the previous section. The IUT is an implementation of the X-protocol.

We shall assume that there is an underlying service provider that provides a network service (N), over which we shall run the test. This leads to the following:

- the LT will be specified in terms of *N-SERVICE* primitives and *X-PDU*s,
- N_DATArequest and CR_PDU;
- the UT will be specified in terms of *X-SERVICE* primitives,
- X_CONNECTrequest;

Description of the Case Study

Our examples will introduce the TTCN features necessary to specify the simple scenario described below:

- The MTC initiates the test by CREATING the necessary PTCs. One lower PTC and one upper PTC for each connection.
- The lower PTC then establishes an X-connection with the upper PTC via the IUT. For the sake of simplicity we will assume that an (N)-connection has already been set up, and that the X-protocol does not allow an X_CONNECTrequest to be refused (the example

A Case Study

has been created to illustrate TTCN features, rather than to specify a sensible protocol).

- The test then continues with the data phase, where the lower PTC transmits a data packet which shall be returned by the upper PTC via the IUT. The packet shall be returned within a given period of time. This process is repeated a given number of times.
- After the data transfer the lower PTC disconnects and sends its preliminary result to the MTC which then computes the final verdict and the test terminates

Purpose of the Test Case

The case study has two test purposes, these can be stated as:

1. The IUT shall accept and return a given number of data packets within the time limitations of the protocol over a single X-connection.
2. The IUT shall accept and return a given number of data packets within the time limitations of the protocol over two simultaneous X-connections.

Each test purpose will be expressed as a separate test case.

Case Study Road-Map

We shall create a TTCN complete mini-test suite that contains all the TTCN necessary to specify the above test cases. The following table (this is not a TTCN table!) shows the main sections of this example in the order in which they would appear in a real test suite. The complete study can be found in [“The Complete Case Study” on page 120](#). The right-hand column of the table tells you where these sections are described in this guideline:

Section of test suite	Described in this guideline
Overview	“Suite Overview Part” on page 114
Configurations	“The Test Configuration” on page 9
Test Suite Parameterization	“Test Case Selection” on page 113
Global Type Definitions	“TTCN Types and Values” on page 19
Global Declarations	“Constants and Variables” on page 94
PCO and CP declarations	“PCOs and CPs” on page 29
Timer Declarations	“Timer Statements” on page 90
ASP, PDU and CM Definitions	“Defining ASP, PDU and CM Types” on page 38
ASP, PDU and CM Values	“Specifying ASP, PDU and CM Values” on page 56
Behaviour Descriptions	“Dynamic Behaviour Descriptions” on page 99

The Test Configuration

Let us start by specifying the test component configurations needed in the test suite. The conformance standard defines various abstract test methods. For the purposes of this guideline we shall assume that the IUT is a *single-layer* implementation and that we are testing with the *distributed method*. Also, we are testing in a *multi-party* context because our second test purpose requires more than one connection.

In this case, the architecture of [Figure 1](#) says that we need:

- one MTC;
- two lower PTCs
- two upper PTCs;
- one (N-1) service provider;
- four PCOs (two lower L1 and L2 and two upper U1 and U2);
- two coordination points (CP1 and CP2) between the lower PTCs and the MTC.

This is illustrated in the following figure:

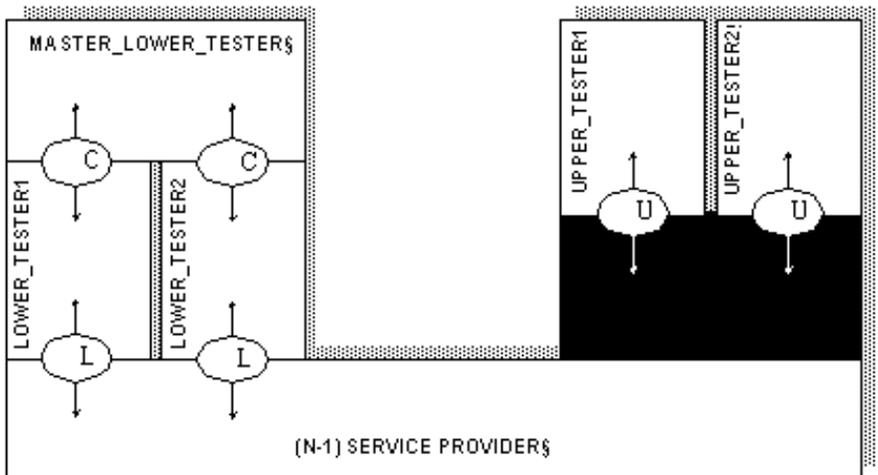


Figure 2: Illustration of the multi-party distributed test method

In concurrent TTCN this architecture is specified using the following tables:

- Test Component Declarations
- Test Component Configuration Declaration (one table per definition)

Case study 1: This table lists all the test components that may be used in the test suite. They can be thought of as building-blocks that can be used to construct different configurations. A test component may have the role of main test component (MTC), or parallel test component (PTC).

Test Component Declarations				
Component Name	Component Role	Nr PCOs	Nr OPs	Comments
MASTER_LOWER_TESTER	MTC	0	2	Main Test Component
LOWER_TESTER1	PTC	1	1	Parallel Test Component
LOWER_TESTER2	PTC	1	1	Parallel Test Component
UPPER_TESTER1	PTC	1	0	Parallel Test Component
UPPER_TESTER2	PTC	1	0	Parallel Test Component
Detailed Comments :				

Figure 3: Test Component Declarations

The Test Configuration

Case study 2: This table shows the configuration for the single-connection test case. In any one configuration there should never be more than one MTC.

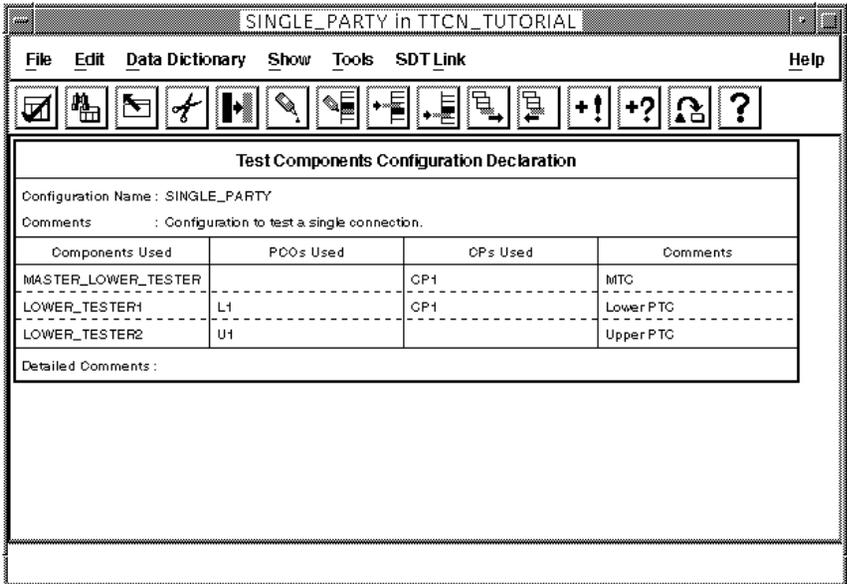


Figure 4: Test Component Configuration Declaration (SINGLE_PARTY)

Case study 3: This table shows the configuration for the multi-connection test case.

The screenshot shows a software window titled "MULTI_PARTY in TTCN_TUTORIAL". The window has a menu bar with "File", "Edit", "Data Dictionary", "Show", "Tools", "SDT_Link", and "Help". Below the menu bar is a toolbar with various icons for editing and navigation. The main content area is titled "Test Components Configuration Declaration" and contains the following information:

Configuration Name : MULTI_PARTY
 Comments : Configuration to test two simultaneous connections.

Components Used	PCOs Used	CPs Used	Comments
MASTER_LOWER_TESTER		CP1, CP2	MTC
LOWER_TESTER1	L1	CP1	Lower PTC
LOWER_TESTER2	L2	CP2	Upper PTC
UPPER_TESTER1	U1		Lower PTC
UPPER_TESTER2	U2		Upper PTC

Detailed Comments :

Figure 5: Test Component Configuration (MULTI_PARTY)

Specification of Test System Behaviour

Before we continue with the declaration of the test case let us now look at how TTCN describes the behaviour of the various test components. Many standardized service definitions and protocol specifications use state diagrams and/or state tables to describe the behaviour of the service or protocol.

Test cases are derived from these specifications. However, because conformance testing is concerned with observing and controlling sequences of interactions at service interfaces it is more appropriate that we specify test system behaviour as a *tree* which has branches for *all* the possible sequences of interactions that may occur between any two given protocol states.

Behaviour Trees

In TTCN a tree of interactions is called a *behaviour tree*. The tree structure is represented by using increasing levels of *indentation* to indicate progression into the tree with respect to time.

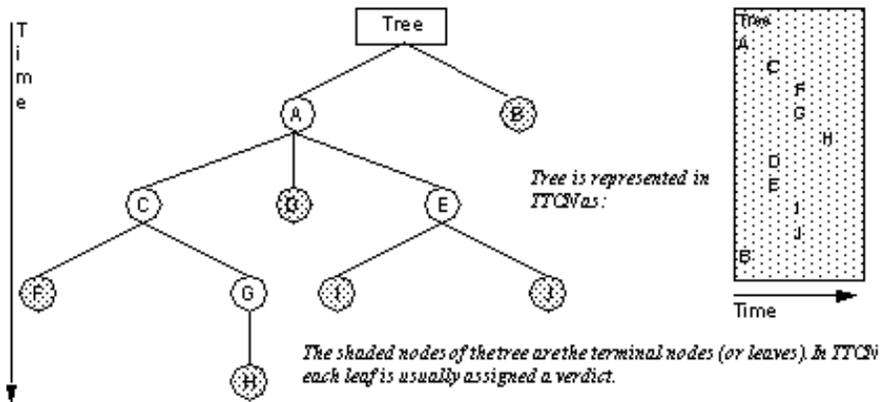


Figure 6: A tree is represented in TTCN using indentation

Note that an absolute level of indentation does not necessarily mean that nodes are siblings. For example, although the nodes F and G are numerically at the same level of indentation (i.e. 3) as I and J, the nodes F and G are in one branch of the tree and the nodes I and J are in another

Behaviour Lines

A node in a behaviour tree is called a *behaviour line*. A behaviour line consists of the following components:

- line number;
- label;
- statement line;
- constraint reference;
- verdict;
- behaviour line comment.

Exactly which components of the behaviour line are used at a specific time varies. For example, line numbers and comments are always optional and constraint references and verdicts shall only be used when required.

TTCN Behaviour Description

Behaviour lines are specified in *dynamic behaviour* tables. There are three kinds of behaviour tables, each consisting of a *header* and a *body*.

- Test Case Dynamic Behaviour;
- Test Step Dynamic Behaviour;
- Default Dynamic Behaviour.

The visual difference between these three tables is in the header. The format of the body is the same for all three. However, there is a significant difference, which will be explained later, in how these different tables are used.

Specification of Test System Behaviour

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		A			
2		C			
3		F			
4		D			
5		H			
6		G			
7		E			
8		I			
9		J			
10		B			

Figure 7: The body of a dynamic behavior table

The table body shows the columns for the line numbers, labels, statement lines, constraint references, verdicts and comments. The light shading indicates the extent of a single behaviour line. The dark shading indicates the extent of a single statement line.

Statement Lines

A sequence of one or more *statements*, together with the indentation information, in a single behaviour line is called a *statement line*. Statement lines appear in the behaviour description column of dynamic behaviour tables.

Statements

The behaviour of the test system, such as sending and receiving ASPs, is expressed using TTCN statements. Statements can be split into three distinct types:

- events;
- actions;
- qualifiers.

Events

Some statements will be successful, *i.e. match*, depending on the occurrence of certain *events*. There are two types of event: *input events* and *timer events*. An input event is the arrival of an ASP at a named PCO or a message at a named CP. A timer event is the expiry of a protocol timer. The TTCN statements that are events are:

- RECEIVE

- OTHERWISE
- TIMEOUT

Actions

Some statements will **always** be successful, *i.e.* execute. We shall call such statements *actions*, although this term is not used in ISO/IEC 9646-3. These are actions that are executed by the test system, and TTCN assumes that they can always be executed successfully. The TTCN statements that are actions are:

- SEND
- IMPLICIT_SEND
- ASSIGNMENT_LIST
- TIMER_OPERATION
- GOTO

Qualifiers

Statement lines may include a *qualifier* statement, *i.e.* a boolean expression. We call such statement lines *qualified* statement lines. No event can match, nor can any action be executed unless the qualifier included in the statement line evaluates to TRUE. An *unqualified* statement line is one that does not include a qualifier.

A TTCN qualifier is simply a:

- BOOLEAN_EXPRESSION

Combinations of Events, Actions and Qualifiers

The actual combinations of events, actions and qualifiers that are allowed are defined by the TTCN-MP. The different combinations will be described at the relevant points in this guideline.

Execution and Matching

We shall now consider how a behaviour tree is traversed and executed.

Alternatives

A set of statement lines at the same level of indentation, and in the same branch of the tree are called a set of *alternative statement lines*, or *alter-*

Specification of Test System Behaviour

natives for short. Thus, in the [Figure 6 on page 13](#) (A,B), (C, D, E), (F, G), (I, J) and (H) are all different sets of alternatives.

Because the ordering within any given set of alternatives is significant it is important that *all* events and qualified statements appear *before* any unqualified actions.

Execution of the Behaviour Tree

Execution starts at the root of the tree. That is, the first set of alternatives is repeatedly looped with each alternative being evaluated in the order of its appearance in the set. This looping continues until a statement line is successfully executed or *matched*. If a statement line is successful then the next set of alternatives (if any) is entered, and the process is repeated.

Execution stops when a leaf of the tree is reached. A final verdict will also halt execution, see [“Verdicts” on page 86](#).

In the example shown in [Figure 7 on page 15](#) execution starts by looping through the first set of alternatives (A, B). If B is successful then execution terminates. If A is successful then the next set of alternatives (C, D, E) is entered. Let’s assume that the statement line E is successful: then the next set of alternatives is (I, J). If either I or J is successful then execution terminates. Note that if no statement line in any set of alternatives is ever successful then execution gets ‘stuck’ as we repeatedly loop through those alternatives.

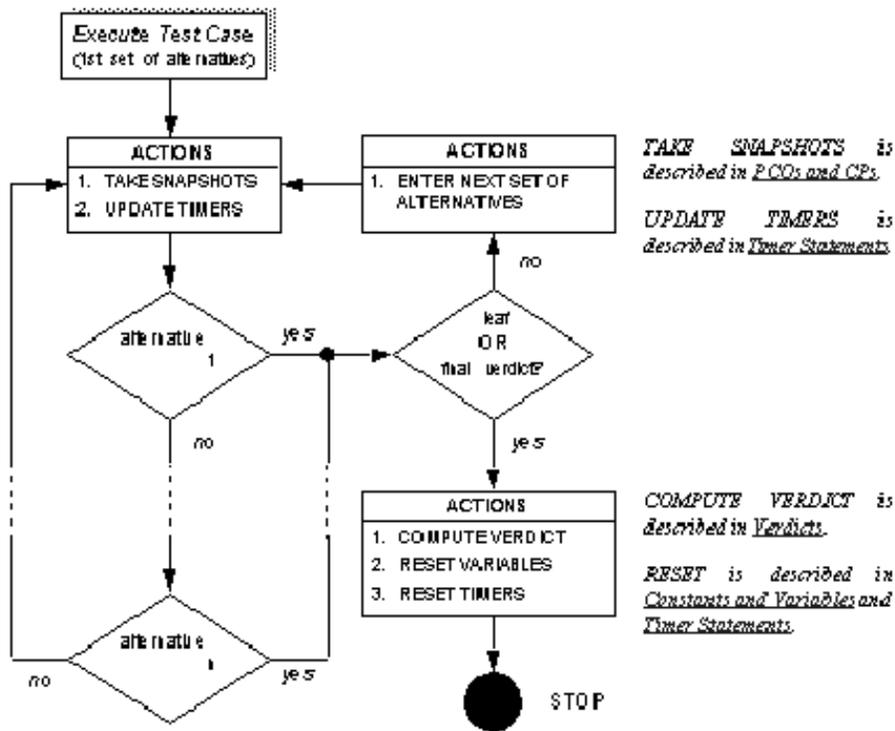


Figure 8: Cycle of execution of a test case behavior tree

TTCN Types and Values

We shall now discuss TTCN data types and their values. These are used to specify the data types, including ASPs and PDUs that are used in the behaviour descriptions.

TTCN has been tailored to interface with the *Abstract Syntax Notation One* (ASN.1, ISO/IEC 8824:1990). There is no clear boundary between TTCN types and ASN.1 types; the distinction is an artificial one. It is there, however, to allow test suite specifiers to build the types, ASPs, PDUs, etc. they need without using ASN.1 if they do not wish to do so. This is relevant, for example, in lower-layer protocols, where ASN.1 is not normally used in the protocol specifications.

TTCN contains a number of *predefined types*. It also allows the user to construct his own types from the predefined types. This may be done using the following tables:

- Simple Type Definitions
- Structured Type Definition (one table per definition)

Predefined Types

TTCN supports a rich set of predefined (built-in) types. The predefined TTCN types, with the exception of HEXSTRING, are a subset of the ASN.1 built-in types and are compatible with their ASN.1 counterparts. The HEXSTRING type does not exist in ASN.1. The remaining ASN.1 built-in types may also be used without being explicitly defined.

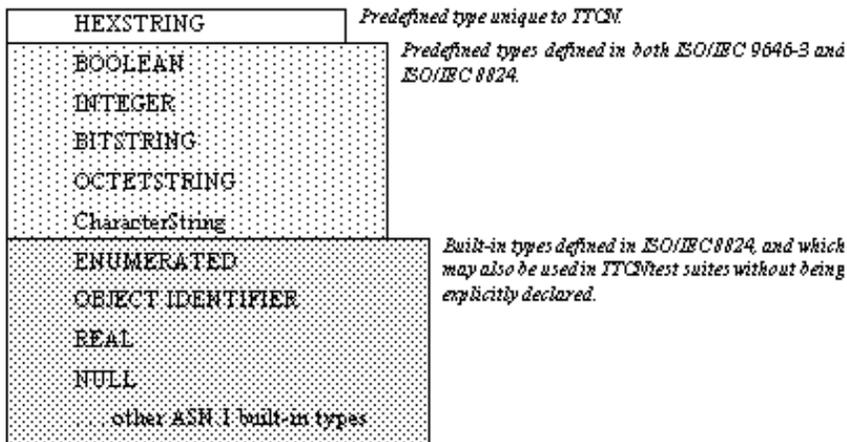


Figure 9: List of predefined types that may be used in TTCN Test Suites

Value Denotation

The value denotation for the predefined types is the same in both TTCN and ASN.1, see [“ASN.1 Types and Values” on page 23](#).

Simple User Defined Types

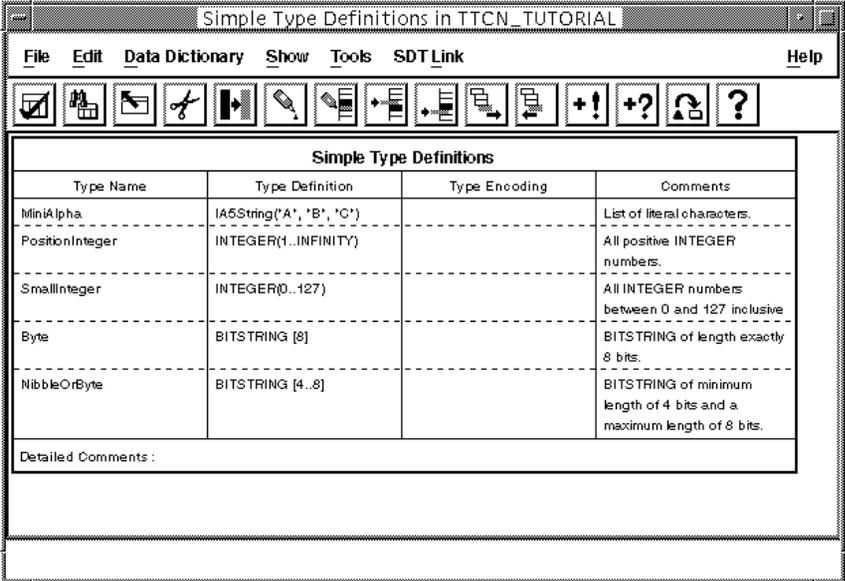
The TTCN user can construct other types *based* on the simple predefined types, without the need to resort to ASN.1 syntax. These subtypes are defined in the *Simple Type Definitions* table, and they may be used anywhere in the test suite. They are constructed by restricting the predefined types (and possibly previously declared subtypes) by specifying:

- value lists,
which are lists whose elements may consist of literal values only;
- ranges,
which may be used to restrict INTEGER types only;
- length restrictions,
which may be used to restrict string types only.

TTCN Types and Values

Note:

The TTCN syntax does allow the use of ASN.1 in the Simple Type Definitions table if wished. We recommend, however, that the special tables for ASN.1 types be used instead.



Type Name	Type Definition	Type Encoding	Comments
MiniAlpha	IA5String('A', 'B', 'C')		List of literal characters.
PositionInteger	INTEGER(1..INFINITY)		All positive INTEGER numbers.
SmallInteger	INTEGER(0..127)		All INTEGER numbers between 0 and 127 inclusive
Byte	BITSTRING [8]		BITSTRING of length exactly 8 bits.
NibbleOrByte	BITSTRING [4..8]		BITSTRING of minimum length of 4 bits and a maximum length of 8 bits.

Detailed Comments :

Figure 10: Some miscellaneous simple type definitions

Case study 4: Definition of a user type for test case results.

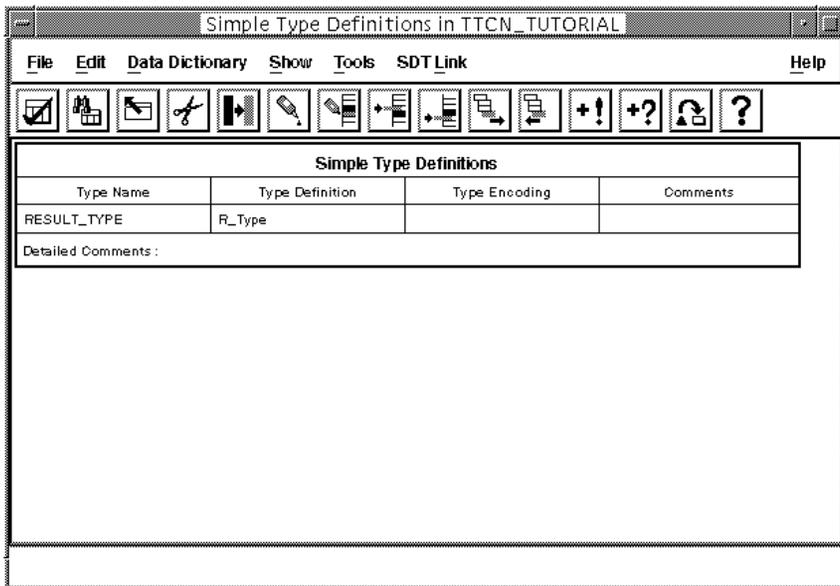


Figure 11: Simple Type Definitions

Structured Types

TTCN has tables for the declaration of structured (i.e., complex) types. These types, like the predefined types and the simple types may be used anywhere (that is why they are defined early on in the test suite). However, their main use is to substructure ASPs and PDUs and we shall therefore discuss them in [“Specifying ASP, PDU and CM Values” on page 56](#).

ASN.1 Types and Values

ASN.1 is used to describe data types. It is often used as a means for defining the PDU structures of many OSI protocols. In the following we shall give an overview of the ASN.1 type and value notation.

Type References, Value References and Identifiers

A *type reference* is the “name” of an ASN.1 type constructed by the ASN.1 user. The ASN.1 standard requires that the initial character of a type reference is **always** an *upper* case letter.

A *value reference* is the “name” of an ASN.1 value constructed by the ASN.1 user. The ASN.1 standard requires that the initial character of a value reference is **always** a *lower* case letter.

Note:

Many test suite specifiers ignore this convention, indeed, so does ISO/IEC 9646-3!

As you will see shortly, type references in SETs and SEQUENCEs *etc.* may be “labeled”. Such labels are called *identifiers*. The ASN.1 standard requires that the initial character of an identifier is **always** a *lower* case letter. Value references and identifiers are distinguished by context.

Identifiers and Underscore

The ASN.1 standard allows the character dash (-) in identifiers. TTCN does not (otherwise there would be ambiguity in arithmetic expressions between dash and minus). TTCN uses underscore (_) instead. If ASN.1 definitions are imported, copied, borrowed *etc.* from external specifications (e.g. the PDU definitions for a particular protocol) then all occurrences of dash in identifiers should be changed to underscore.

ASN.1 Simple Types

BOOLEAN

BOOLEAN is a type denoted by two distinguished values: TRUE and FALSE. The TTCN has operators for values of any type whose base type is BOOLEAN.

INTEGER

INTEGER is a type denoted by the distinguished values which are the positive and negative whole numbers, including zero. TTCN has operators for any values whose base type is INTEGER.

REAL

REAL is a data type specified as a triple of three INTEGERS consisting of:

- $\text{mantissa} * \text{base}^{\text{exponent}}$

The base is limited to 2 or 10.

- $\text{pi REAL} ::= \{ 314159, 10, -5 \}$

TTCN does not have any operators for values of REAL types. If REAL arithmetic is absolutely necessary this can be achieved by user defined operations.

BIT STRING

BIT STRING is a type whose distinguished values are the ordered sequences of zero, one, or more BITS. Individual BITS in the BIT STRING may be named.

- $\text{ABitString} ::= \text{BIT STRING } \{ \text{bit1 (0), bit2 (1), bit3 (2), bit4 (3)} \}$
- $\text{a-value ABitString} ::= '1001'B$

OCTET STRING

OCTET STRING is a type whose distinguished values are the ordered sequences of zero or an even number of HEX digits, each digit corresponding to an ordered sequence of four bits. Individual OCTETSs in the OCTET STRING may be named.

- AnOctetString ::= OCTET STRING { octet1 (0), octet2(1) }
- a-value AnOctetString ::= '0F'H

Note:

There is an incompatibility between TTCN and ASN.1 over the value denotation of OCTET STRING. TTCN terminates OCTET STRING values with the keyword 'O' rather than the keyword 'H'.

CharacterString

A variety of character sets are supported. For the purposes of this guideline we shall restrict ourselves to using the ITU character set IA5String.

ENUMERATED

ENUMERATED types represent the complete set of values (domain) that an instance of a data type may take.

- transport-classes ENUMERATED ::= { class1 (0), class2 (1), class3 (2), class4 (3), class5 (4) }

No TTCN operators can be applied to values of ENUMERATED type.

OBJECT IDENTIFIER

OBJECT IDENTIFIER denotes a named *object* as a sequence of non-negative INTEGERS. The naming hierarchy of specific objects is decided by the relevant authority (*e.g.* ISO or ITU). An OBJECT IDENTIFIER specifies a unique path in this hierarchy, *i.e.* all objects are uniquely named.

- ttcn-standard OBJECT IDENTIFIER ::= { iso (1) standard (0) 9646 3 }

The OBJECT IDENTIFIER for ISO/IEC 9646-3 would be 1.0.9646.3

Objects that TTCN needs to reference in this manner might be ASN.1 modules (PDUs etc.) and PICS, PXIT documents.

OBJECT DESCRIPTOR

OBJECT DESCRIPTOR denotes a text string that references an *object*. The difference between an OBJECT IDENTIFIER and an OBJECT DESCRIPTOR is that the former is unique, while the latter may not be.

- ttcn-standard OBJECT DESCRIPTOR ::= "The TTCN: ISO/IEC 9646, part 3"

ASN.1 Constructors

Complex data types can be built from the simple predefined types (excepting HEXSTRING) using ASN.1 *constructors*. This process is recursive, *i.e.* constructors can use other constructors (including themselves) to an arbitrary level of nesting. The constructor types are:

- SEQUENCE
- SEQUENCE OF
- SET
- SET OF
- CHOICE

SEQUENCE

SEQUENCE is a data type denoting an *ordered* set of elements. This set may be empty. The elements of this set may be of any ASN.1 type and may be of *different* types. The elements in the set may be named.

- ASequence ::= SEQUENCE {field1 INTEGER, field2 BOOLEAN}
- a-value ASequence ::= { field1 123, field2 TRUE }

SEQUENCE OF

SEQUENCE OF is a data type denoting an *ordered* set of elements. This set may be empty. The elements of this set may be of any ASN.1 type but they shall all be of the *same* type. The elements in the set may be named.

- ASequenceOf ::= SEQUENCE OF INTEGER { field1, field2 }

SET

SET is a data type denoting an *unordered* set of elements. This set may be empty. The elements of this set may be of any ASN.1 type and may be of *different* types. The elements in the set may be named.

- ASet ::= SET { field1 INTEGER, field2 BOOLEAN }

SET OF

SET OF is a data type denoting an *unordered* set of elements. This set may be empty. The elements of this set may be of any ASN.1 type but shall all be of the *same* type. The elements in the set may be named.

- ASetOf ::= SET OF INTEGER { field1, field2 }

At first glance SEQUENCE and SET may appear the same. The difference is that a SEQUENCE is ordered, a SET is not. For instance, in the previous examples when the SET is eventually encoded it can be transmitted by sending field2 before field1, if wished. In the case of SEQUENCE field1 must always precede field2. This has implications when testing, which will be discussed later.

The same applies to SEQUENCE OF and SET OF.

In short, SEQUENCE and SEQUENCE OF rely on ordering to avoid ambiguity. SET and SET OF rely on the data type and/or tag of each element to uniquely distinguish each element.

OPTIONAL

The keyword OPTIONAL in a SEQUENCE or SET indicates that the presence of that element in the SEQUENCE is not mandatory and may be included or omitted at will.

- ASequence ::= SEQUENCE { field1 INTEGER OPTIONAL, field2 BOOLEAN }
- a-value1 ASequence ::= { 123, TRUE }
- a-value2 ASequence ::= { TRUE }

The same applies for SEQUENCE OF, SET and SET OF.

DEFAULT

In certain cases it is useful to be able to specify a DEFAULT value to be used (in encoding) if the element is not present in the data type.

- ASequence ::= SEQUENCE { field1 INTEGER DEFAULT 0, field2 BOOLEAN }

CHOICE

A CHOICE type is a data type that defines the union of one or more data types. The alternatives in this union may be named. Any given instance, *i.e.* value, of a CHOICE shall be exactly one of the alternatives of the CHOICE. This has implications for testing which will be discussed later.

- AllPDUs ::= CHOICE { pdu1 SEQUENCE { }, pdu2 SEQUENCE { }, pdu3 SEQUENCE { } }
- a_pdu AllPDUs ::= pdu2

TAGGED TYPES

Tags are used to distinguish between different occurrences of the same type. Tags are denoted by a non-negative INTEGER enclosed in square brackets. Tags are included in the encoding of the data type. There are four classes of tags:

- UNIVERSAL

Universal tags are globally unique and are only defined in the ASN.1 standard. These tags have a meaning world-wide.

- PRIVATE

Private-use tags are unique within a given enterprise, and are defined by agreement of the parties involved in the enterprise. These tags have no meaning outside the scope of the enterprise.

- APPLICATION

Application-wide tags are unique within a specific ASN.1 module. These tags have no meaning outside of the ASN.1 module that they are used in.

- CONTEXT

Context-specific tags are unique within a specific constructor type. For example, elements in a particular SET may be tagged to unique-

ly distinguish them. These tags have no meaning outside of the ASN.1 type that they are used in.

IMPLICIT

The keyword IMPLICIT may be used together with the definition of the tagged type. IMPLICIT is an instruction to an ASN.1 encoder that only the tag need be encoded and thus transmitted over the network. This is done to reduce the amount of transmitted data. IMPLICIT may only be used where no loss of essential information would occur. For example, it should not be used with CHOICE.

EXTERNAL

Use of the EXTERNAL type is not allowed in TTCN.

PCOs and CPs

The TTCN supports an *asynchronous* communication model. Communication between the test components and the IUT or service provider is achieved via *points of control and observation* (PCOs). Communication between the test components themselves is achieved via *coordination points* (CPs).

The Communication Model

We shall use the same *queue* model to describe both PCOs and CPs:

- each PCO/CP has *two* unbounded *first-in-first-out* (FIFO) queues;
- one queue for SEND, and
- one queue for RECEIVE;
- exactly two parties must be connected to a single PCO or CP;
- the SEND queue for one party is the RECEIVE queue for the other, and vice versa.

Sending an ASP

The SEND action *appends* an ASP to the relevant PCO send queue. Even in the case of the IUT and the underlying service provider the send queue is considered to be unbounded, and that the IUT or service provider will *always* accept the ASPs sent by an LT or UT.

Receiving an ASP

A successful RECEIVE *pops* the ASP from the top of the RECEIVE queue. We shall see later that RECEIVE involves two steps:

- receipt of the ASP;
- checking its contents.

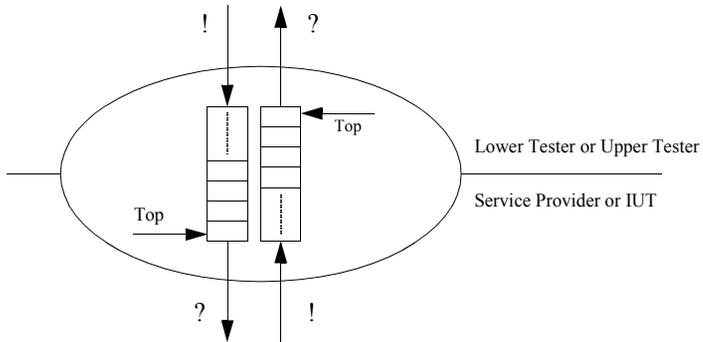


Figure 12: Illustration of the PCO and CP queue model

Declaring PCO Types

All PCO types that are used in the test suite must be declared in the

- *PCO Type Declarations*

Each PCO Type requires the following information:

- the name the PCO Type
- the role of the PCO, which is either of the keywords LT or UT;

Case study 5: Declaration of the PCO Types N_SAP and X_SAP.

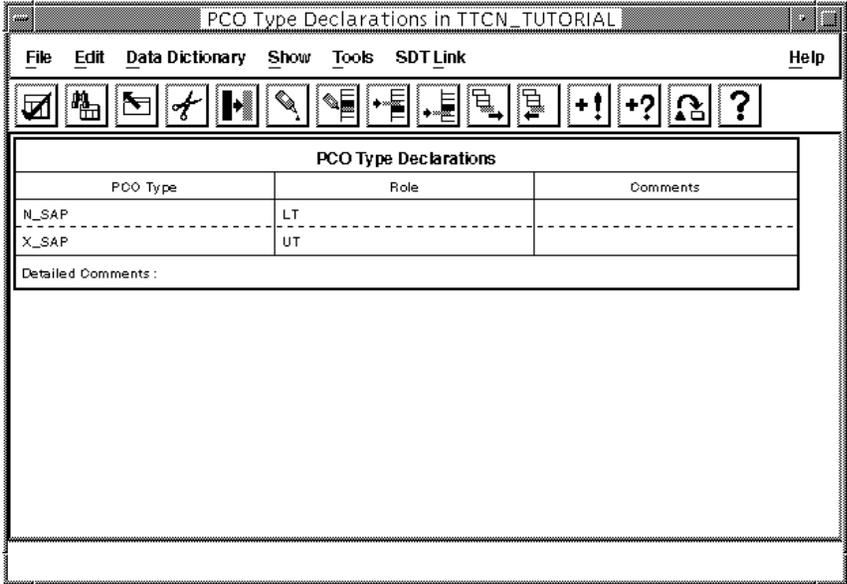


Figure 13: PCO Type Declarations

Declaring PCOs

All PCOs that are used in the test suite must be declared in the

- *PCO Declarations*
Each PCO requires the following information:
- the name the PCO
- the type of the PCO;
- the role of the PCO, which is either of the keywords LT or UT;

Case study 5: Declaration of the PCOs L and U.

PCO Declarations			
PCO Name	PCO Type	Role	Comments
L1	N_SAP	LT	N-service access points at the lower tester
L2	N_SAP	LT	
U1	X_SAP	UT	X-service access points at the upper tester
U2	X_SAP	UT	

Detailed Comments :

Figure 14: PCO Declarations

Using PCOs and CPs

If the test suite only uses one PCO it is allowed to omit the PCO name in the TTCN statements that use them. If there is more than one PCO used (*e.g.* as in the distributed method) then the PCO and CP names (if any) must appear in the TTCN statements that use PCOs or CPs.

PCO and CP Snapshots

We have already described how a behaviour tree is executed by repeatedly looping through a set of alternatives until a statement line is successful. At the beginning of each loop a *snapshot* is taken of each input queue in *every* PCO or CP. Statements are evaluated on the basis of the state of the snapshots, not on the actual state of the PCO or CP queues. This has the effect of freezing time while a set of alternatives is being processed *i.e.* “prevents” the occurrence of an event in between snapshots. This means that the arrival of an ASP, PDU or CM during pro-

cessing of a set of alternatives is not registered until the snapshots are updated.

Declaring CPs

All CPs that are used in the test suite must be declared. This is done in the *CP Declarations* table. Each PCO requires the following information:

- the name of the CP;
- the role of the CP,
i.e. the two test components which communicate with each other over the CP.

Case study 6: Declaration of the coordination points called CP1 and CP2.

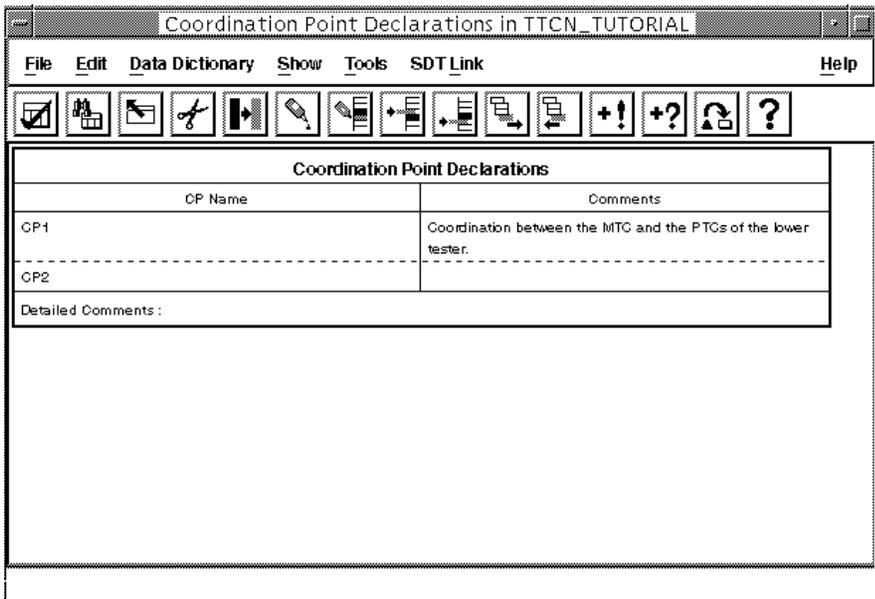


Figure 15: Test Component Declarations

The SEND Statement

The transmission of ASPs and/or PDUs to the IUT or messages to other test system components is one of the fundamental actions in a typical TTCN behaviour tree.

Sending an ASP

The SEND statement allows the test suite specifier to express that an ASP of a certain type is to be transmitted over a named PCO. The SEND statement is denoted by:

- *PCO_Identifier ! ASP_Identifier*

The SEND statement may be qualified and it may be followed by an ASSIGNMENT_LIST and/or TIMER_OPERATION. The order in which these statements may appear in the statement line is fixed, as shown below; the square brackets indicate that the presence of the statement in the statement line is optional:

- SEND³ [QUALIFIER]¹ [ASSIGNMENT_LIST]²
[TIMER_OPERATION]⁴

Executing a SEND Statement

The numbers on the line above indicate the order, with respect to time, in which the statements should be executed: the QUALIFIER (if any) is evaluated first. If it evaluates to FALSE processing stops and the statement line is not successful. If it evaluates to TRUE then the ASSIGNMENT_LIST (if any) is executed. Only then can the SEND statement be executed. Finally, the TIMER_OPERATION (if any) is executed.

- L! N_DATArequest
means: send the Network data request service primitive to the PCO named L;
- L! N_DATArequest [B=1]
means: if B is equal to 1 then execute the SEND;
- L! N_DATArequest [B=1] (X:=3)
means: if B is equal to 1 then assign the value 3 to X and then perform the SEND.

The RECEIVE Statement

Sending a PDU

Normally PDUs are embedded in ASPs, and will not be explicitly named in the SEND statement. However, not all protocols have a service definition (e.g. X.25) and TTCN therefore permits the SEND statement to be used explicitly with PDUs instead of ASPs. The action of sending a PDU is denoted by:

- *PCO_Identifier ! PDU_Identifier*

Other statements that may be associated with sending a message, and the order in which the statement line is processed, is exactly the same as for an ordinary SEND statement line.

Sending a Coordination Message

The SEND statement is also used to send messages to coordination points. The action of sending a CM is denoted by:

- *CP_Identifier ! CM_Identifier*

Other statements that may be associated with sending a message, and the order in which the statement line is processed, is exactly the same as for an ordinary SEND statement line.

The RECEIVE Statement

The receipt of ASPs and/or PDUs from the IUT or messages from other test system components is one of the fundamental events in a typical TTCN behaviour tree.

Receiving an ASP

The RECEIVE statement allows the test suite specifier to express that an ASP of a certain type is to be received over a named PCO. The RECEIVE statement is denoted by:

- *PCO_Identifier ? ASP_Identifier*

The RECEIVE statement may be qualified and it may be followed by an ASSIGNMENT_LIST and/or TIMER_OPERATION. The order in which these statements appear in the statement line is fixed, as shown below; the square brackets indicate that the presence of the statement in the statement line is optional:

- RECEIVE¹ [QUALIFIER]² (ASSIGNMENT_LIST)³
[TIMER_OPERATION]⁴

Executing a RECEIVE Statement

The numbers on the line above indicate the order, with respect to time, in which the statements should be executed: the RECEIVE is evaluated first, and succeeds if an ASP of the correct type is at the head of the PCO queue. If the RECEIVE fails then processing stops and the statement line is not successful. If the RECEIVE is successful then the QUALIFIER (if any) is evaluated. If the QUALIFIER evaluates to FALSE processing stops and the statement line is not successful. If it evaluates to TRUE then the ASSIGNMENT_LIST (if any) is executed. Finally, the TIMER_OPERATION (if any) is executed. For example:

- L? N_DATArequest
means: the statement line matches if a Network data request primitive is at the head of the PCO named L;
- L? N_DATArequest [B=1]
means: the statement line matches if the correct ASP is at the head of the PCO L *and* if B is equal to 1;
- L? N_DATArequest [B=1] (X:=3)
means: the statement line matches if the correct ASP is at the head of the PCO L *and* if B is equal to 1. Only when the match has occurred can the ASSIGNMENT_LIST be executed.

Receiving a PDU

Normally PDUs are embedded in ASPs, and will not be explicitly named in the RECEIVE statement. However, not all protocols have a service definition (e.g. X.25) and TTCN therefore permits the RECEIVE statement to be used explicitly with PDUs instead of ASPs. Receipt of a PDU is denoted by:

- *PCO_Identifier ? PDU_Identifier*

Other statements that may be associated with sending a message, and the order in which the statement line is processed, is exactly the same as for an ordinary RECEIVE statement line.

Receiving a Coordination Message

The RECEIVE statement is also used to accept messages from coordination points. Receipt of a CM is denoted by:

- *CP_Identifier ? CM_Identifier*

Other statements that may be associated with receiving a message, and the order in which the statement line is processed, is exactly the same as for an ordinary RECEIVE statement line.

The OTHERWISE Statement

The OTHERWISE statement allows the test suite specifier to express that an ASP or PDU of any type is to be received over a named PCO. Note that this includes objects that may not normally be recognized as proper ASPs or PDUs, due to the fact that the IUT may not be working correctly, *i.e.* OTHERWISE is a catch-all. The OTHERWISE statement is denoted by:

- *PCO Identifier ? OTHERWISE*

OTHERWISE should not be used at coordination points.

Note:

Always have an OTHERWISE as an alternative to a RECEIVE event.

Defining ASP, PDU and CM Types

ASPs are derived from the relevant standardized service definitions. When using the distributed method, for example, ASP definitions are needed for both the (N) and the (N-1) service. There should be one ASP definition for each ASP used in the test suite.

PDUs are derived from the relevant protocol specifications. There should be one PDU type definition for each PDU used in the test suite. If, for the purposes of testing, it is required to use non-standard PDUs then these too should also be defined in the test suite.

Coordination Messages are also defined by the test suite specifier.

TTCN has tables that allow the definition of ASPs, PDUs and CMs using either the simple TTCN tabular format or ASN.1.

Complex TTCN Types

TTCN has tables for the declaration of the following complex types:

- ASP Type Definitions;
- PDU Type Definitions;
- Structured Type Definitions;
- CM Type Definitions.

Using these complex types we can define arbitrarily structured ASPs and PDUs (structured types are substructures of ASPs and PDUs). In essence there is no real difference in TTCN between the composition of the body of an ASP, PDU or structured type. Note the following:

- an ASP has *parameters*,
where the type of each parameter may be of any type *except* ASP type;
- a PDU has *fields*,
where the type of each field may be of any type *except* ASP type;
- a structured type has *elements*,
where the type of each element may be of any type *except* ASP type.

Chaining

Normally, the ASP parameters, PDU fields and structure elements will be predefined or simple types (note that this includes the use of ASN.1, if wished). However, as noted above, the parameter, field or element types may also be PDUs or structures to allow the *chaining* of these types to build complex definitions.

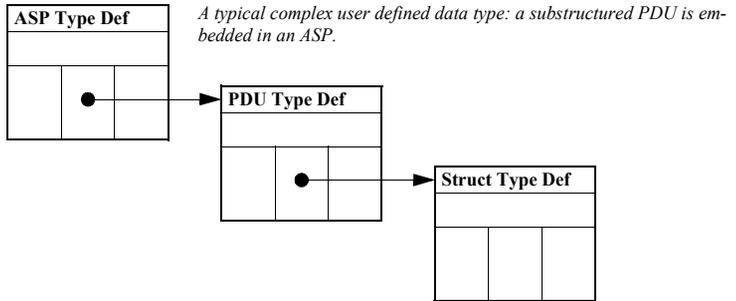


Figure 16: Structuring ASPs and PDUs

Complex ASN.1 Types

In ASN.1, constructors such as SEQUENCE and SET are used to build arbitrarily complex types. ASN.1 definitions may be used in the following tables:

- ASN.1 Type Definitions;
- ASN.1 ASP Type Definitions;
- ASN.1 PDU Type Definitions;
- ASN.1 CM Type Definitions.

It is always possible to express the TTCN tabular format in ASN.1, but not vice versa. The two formats can be used in combination, if wished. A common example is to use a tabular ASP to carry a structured PDU defined in ASN.1.

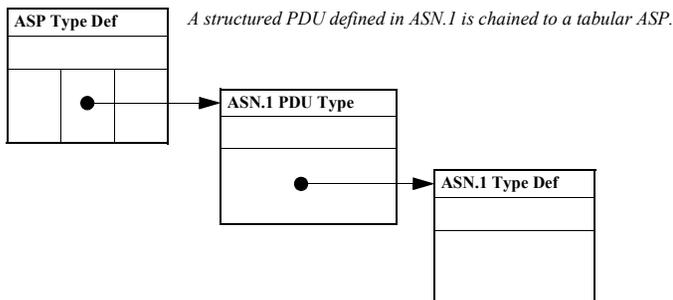


Figure 17: Structuring ASPs and PDUs using ASN.1

Local Type Definitions

There will always be at least one type definition in an ASN.1 table. This is the *main* definition, and it is named in the table header. However, these ASN.1 tables may also include any number of *local* definitions which are only available to the type definitions defined within the table itself, *i.e.* the main definition and other local definitions (if any).

Note that local definitions begin with *typereference ::=*. This is not the case for the main definition as the type identifier already appears in the header.

Type Definitions by Reference

In order to save repeating PDU and other type definitions that are specified in another standard TTCN allows the following types to be declared *by reference* rather than explicitly:

- ASN.1 ASP Definitions;
- ASN.1 PDU Definitions;
- ASN.1 Type Definitions.

A single table is used for all references to a particular type. The reference tables are:

- ASN.1 ASP Definitions by Reference;
- ASN.1 PDU Definitions by Reference;
- ASN.1 Type Definitions by Reference.

Defining ASP, PDU and CM Types

Note that because the entries in the *Type Reference* column and the *Module Identifier* column follow ASN.1 syntax they may contain the dash character. Note also that the module identifier may be followed by an optional object identifier.

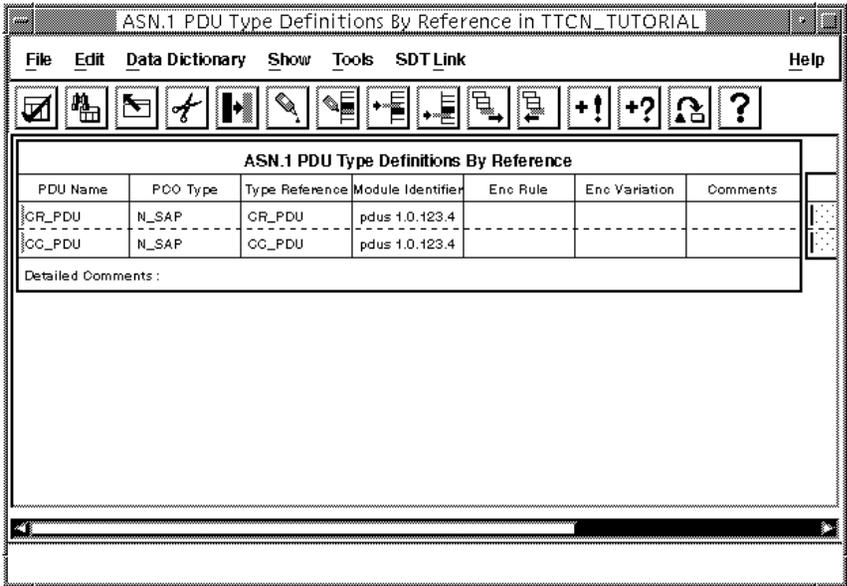


Figure 18: Example of PDU definitions by reference

Defining ASPs

OSI service primitives are often defined in a standard as a tuple, *i.e.* the primitive name followed by a list of parameters. Each parameter is defined using natural language descriptions and may represent service control information or service user data. Some parameters are mandatory (*i.e.* must always be present) while others are optional and, under certain circumstances, may be omitted.

In TTCN service primitives are called Abstract Service Primitives (ASPs) and are declared in *ASP Type Definition* tables.

Case study 7: A service provider ASP type definition.

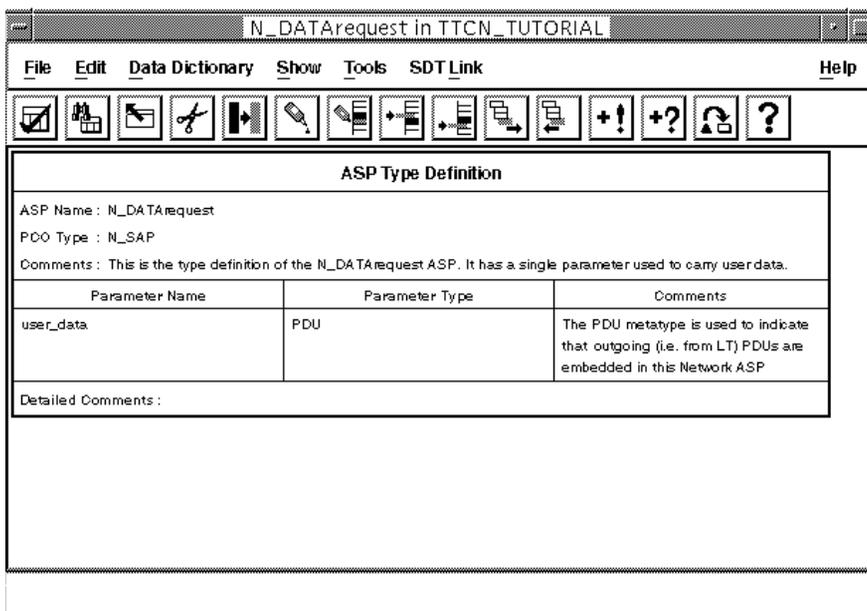


Figure 19: ASP Type Definition (N_DATArequest)

The PDU Metatype

The above example uses the PDU *metatype*. This indicates that any type of PDU, and not just a particular type of PDU may be embedded in this ASP.

Defining PDUs

In most OSI standards PDUs are usually defined using either:

- a simple tabular-like format, together with informal text; or
- ASN.1 together with informal text.

In the first case the specification may be rather loose, and typing of PDU fields and substructuring of the PDUs is not always obvious. The test suite specifier must transpose these definitions to the more powerful and precise formats available in TTCN.

For example, a standard may describe a particular field as comprising 8-bits, implying that it shall be encoded as a BITSTRING. If none of the bits in this BITSTRING need to be referenced individually, it may be adequate for testing purposes (and easier to understand) if this field is defined as an OCTETSTRING.

In the case of ASN.1 the types and structure of PDUs and their fields is usually complete and well-defined, and may be taken directly from the protocol standard, either by copying them or by reference.

Case study 8: Type definition of an X_PDU using the TTCN format.

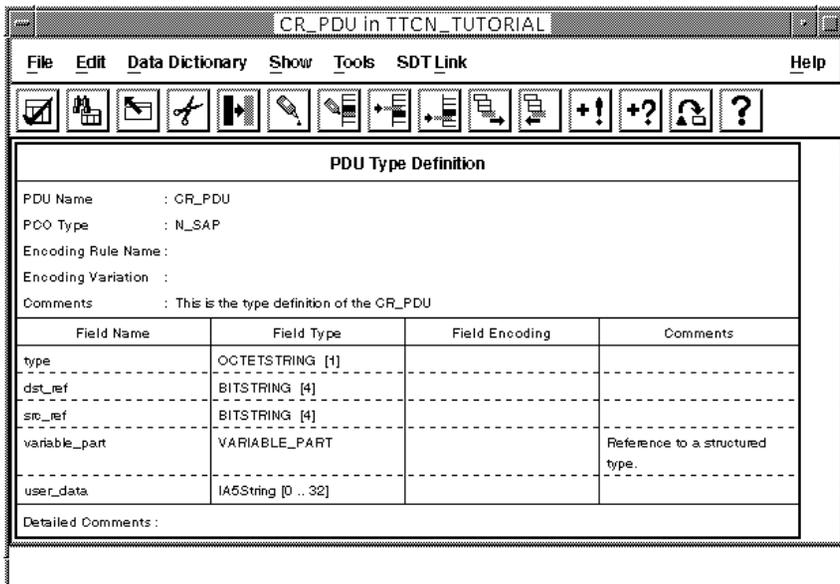


Figure 20: PDU Type Definition (CR_PDU)

Defining ASP, PDU and CM Types

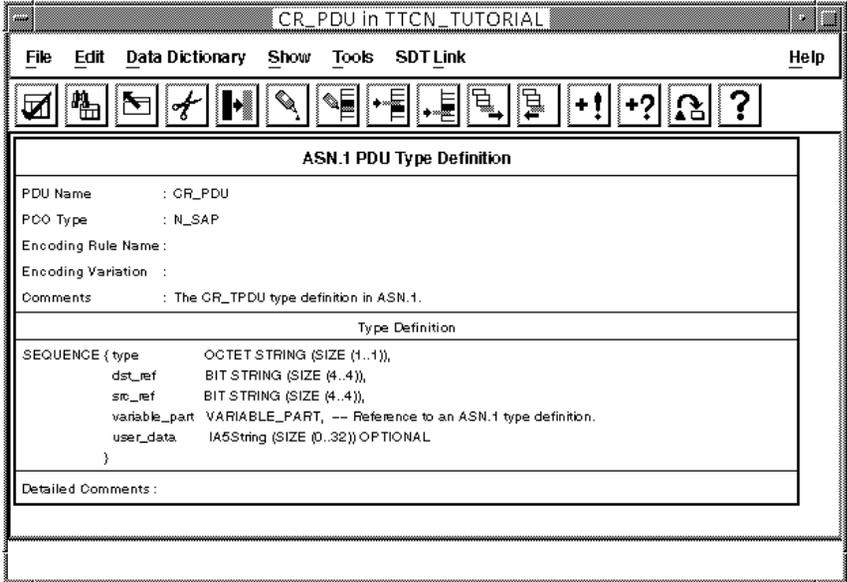


Figure 21: Type definition of the same X_PDU using ASN.1

Substructuring ASPs and PDUs

TTCN structured types (which we will sometimes refer to simply as *structures*) are only used to substructure ASPs, PDUs, CMs and other structured types.

If ASN.1 is used then the ASN.1 type definition table may be used not only to substructure ASPs, PDUs *etc.* but also to define types general to the entire test suite.

Case study 9: Type definition of a PDU substructure using the TTCN format.

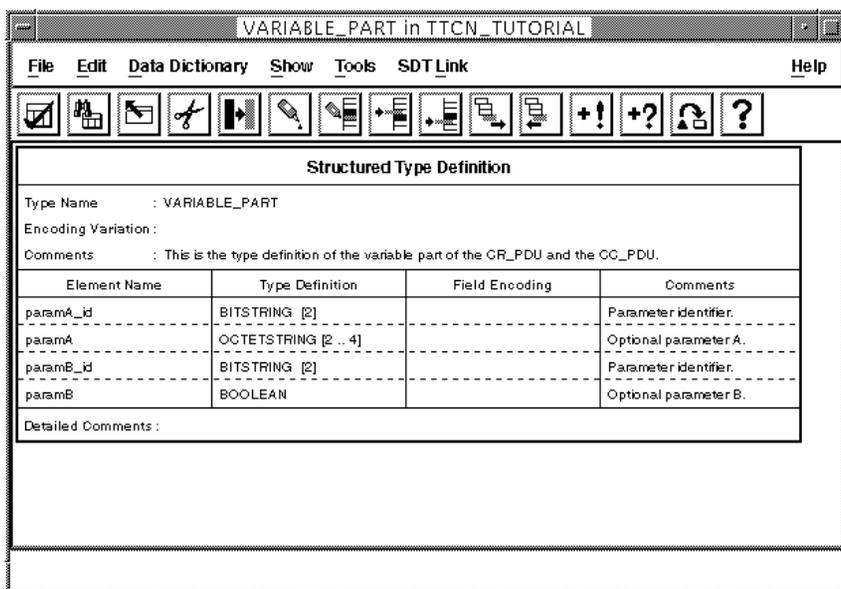


Figure 22: Structure Type Definition (VARIABLE_PART)

Defining ASP, PDU and CM Types

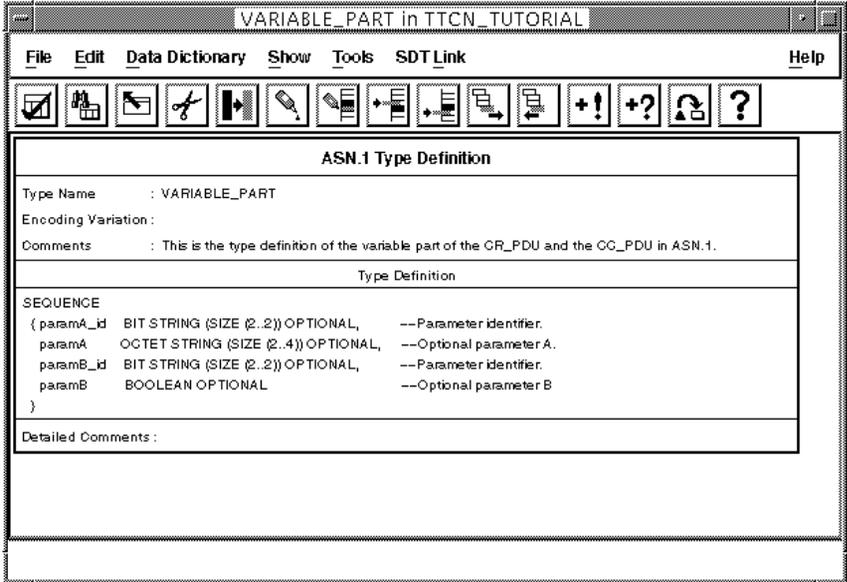


Figure 23: Type definition of the same substructure using ASN.1

Defining Coordination Message Types

Coordination messages are special to each test suite and are created by the test suite specifier. Either the tabular format or ASN.1 may be used.

Case study 10: Type definition of a coordination message.

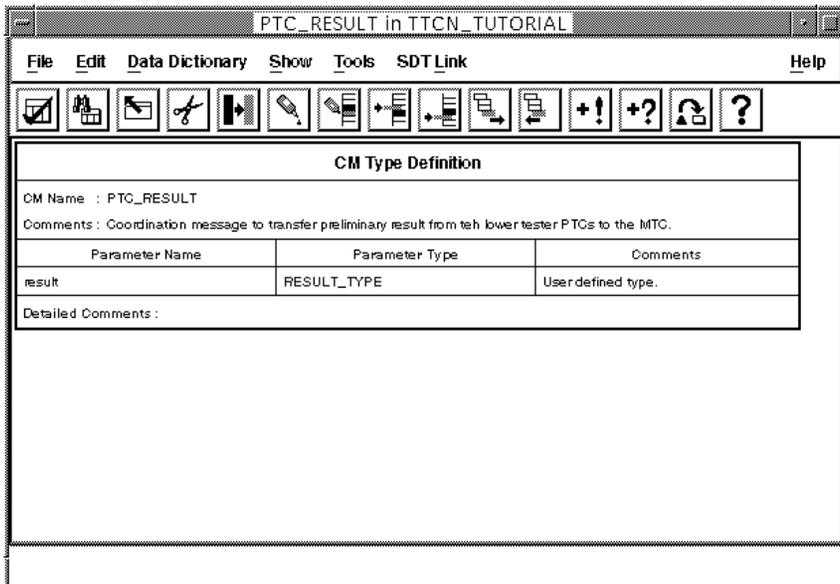


Figure 24: CM Type Definition (PTC_RESULT)

Defining ASP, PDU and CM Types

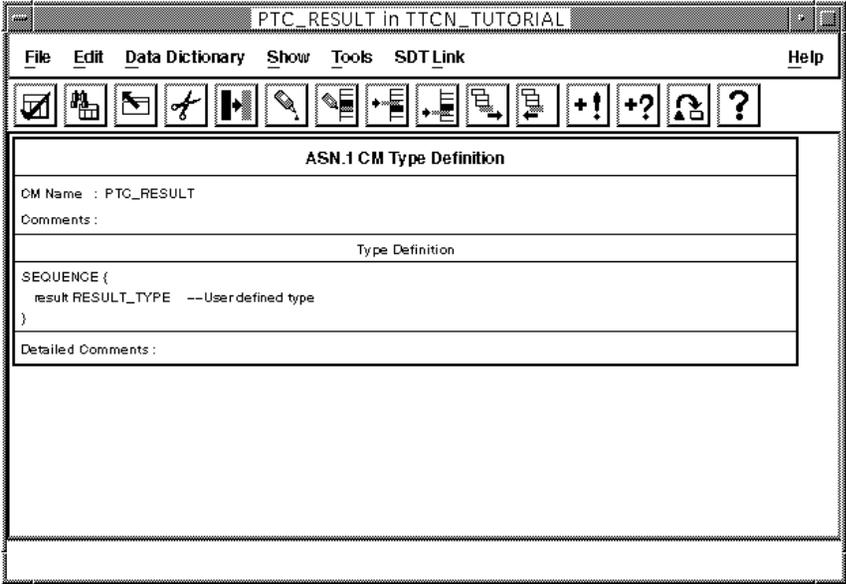


Figure 25: Type definition of the same coordination message using ASN.1

Using ASPs and PDUs in Behaviour Trees

In this section we will expand our example a bit further and show how the sequencing of ASPs w.r.t. time is expressed in TTCN behaviour descriptions.

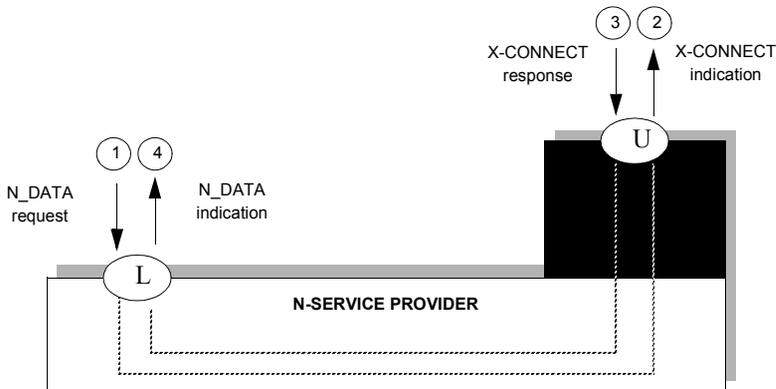


Figure 26: An N_DATArequest, carrying a CR_PDU, is sent over the network

It results in the IUT generating an X_CONNECTindication, which is responded to by the UT tester sending an X_CONNECTresponse. This results in an N_DATAindication, carrying a CC_PDU, appearing at the LT.

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		L! N_DATArequest			
2		L? N_DATAindication			
3		L? OTHERWISE			

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		U? X_CONNECTindication			
2		U! X_CONNECTresponse			
3		U? OTHERWISE			

Figure 27: The above scenario can be expressed as two TTCN behavior trees

Note the introduction of the OTHERWISE statement.

TTCN Expressions

All values in TTCN are quite simply *expressions*. Note that the TTCN syntax allows the *operands* of expressions to be:

- literal values;
- constant or variable identifiers;
- formal parameter identifiers;
- ASP parameters;
- PDU or CM fields;
- structure elements;
- predefined and user defined operations;
- expressions, *i.e.* the syntax is recursive.

Exactly which variables *etc.* may be used in an expression depends on the context in which the expression is used. This aspect will be discussed in the relevant sections.

TTCN Operators

TTCN supports the following kinds of operators for use in expressions:

- arithmetic;
- relational
- logical.

Arithmetic Operators

TTCN supports the following *arithmetic* operators for use only with operands of INTEGER type or derivations of INTEGER type:

- +, -, *, /, MOD

Expressions that use these operators are called *arithmetic expressions*.

- $3*(Z+9)$

Equality Operator

The *equal to* and *not equal to* operators may be used on values of any type:

- =, <>

Expressions that use these operators must always evaluate to a BOOLEAN value:

- B_string = '01'B
- H_string <> 'FF'H

Other Relational Operators

TTCN supports the following *relational* operators for use only with operands of INTEGER type or derivations of INTEGER type:

- <, >, >=, <=

Expressions that use these operators must always evaluate to a BOOLEAN value:

- X <= 3*Y

Boolean Operators

TTCN supports the following *logical* operators for use only with operands of BOOLEAN type or derivations of BOOLEAN type:

- AND, OR, NOT

Expressions that use these operators must always evaluate to a BOOLEAN value:

- A AND NOT (B OR C)

Qualifiers

A qualifier is an expression enclosed in square brackets:

- [expression]

The expression must evaluate to a BOOLEAN value.

- [X < 6 AND H_string <> 'FF'H]

Assignment Lists

A TTCN statement may be an ASSIGNMENT_LIST, *i.e.* a list of assignments, separated by commas and enclosed in parentheses:

- (assignment₁, . . . , assignment_n)

The left-hand side (l.h.s.) of an assignment must resolve to a variable. In the context of SEND and RECEIVE statements the l.h.s. of an assignment may resolve to an ASP parameter reference, a PDU field reference or a structure element reference. The right-hand side (r.h.s.) of the assignment is an expression, which must evaluate to a value of a type compatible with the type of the l.h.s.

- (X := 3, A := “a string”, Y := 3*(Z+9), H := ‘FF’H)

Note:

By *type compatibility* we quite simply mean that a value, *a*, of type *A* is type compatible with type *B* if *a* is a legal value of both type *A* and type *B*.

TTCN Operations

TTCN supports both a number of predefined operations and a mechanism that allows the definition of user operations. Operations may be used as operands in expressions.

Predefined Operations

TTCN now supports a number of *predefined operations*. More are expected to be added by the work on TTCN extensions. Currently these operations are:

- HEX_TO_INT (data_object_reference),
converts a HEXSTRING value to an INTEGER value;
- BIT_TO_INT (data_object_reference),
converts a BITSTRING value to an INTEGER value;
- INT_TO_HEX (data_object_reference),
converts an INTEGER value to an HEXSTRING value;
- INT_TO_BIT (data_object_reference),
converts an INTEGER value to an BITSTRING value;

- `LENGTH_OF (data_object_reference)`,
returns the length of the *data object reference*, which must be of string type, in units of that string type, *e.g.* number of bits, number of characters *etc.*;
- `NUMBER_OF_ELEMENTS (data_object_reference)`,
returns the number of elements in the *data object reference* (*e.g.* PDU field) which must be of type `SEQUENCE OF` or `SET OF`;
- `IS_PRESENT (data_object_reference)`,
returns `TRUE` if an `OPTIONAL` or `DEFAULT` *data object reference* (*e.g.* PDU field) is present in a received PDU; otherwise returns `FALSE`;
- `IS_CHOSEN (data_object_reference)`,
this operation is used to indicate that we wish to accept a particular element from a `CHOICE`. It returns `TRUE` if the *data object reference* (*e.g.* PDU field), which must be of `CHOICE` type, matches the received value.

User Defined Operations

TTCN allows the informal definition of user define operations. A possible approach is to use a programming language to ‘describe’ the operation.

Note:

The TTCN amendment (PDAM 2) is currently exploring ways of how user operation descriptions can be made more ‘formal’.

Like the predefined operations user defined operations may be used in both behaviour trees and as ‘values’ in constraints.

Each user defined operations is declared in a *Test Suite Operations* table.

Case study 11: Definition of a user defined operation.

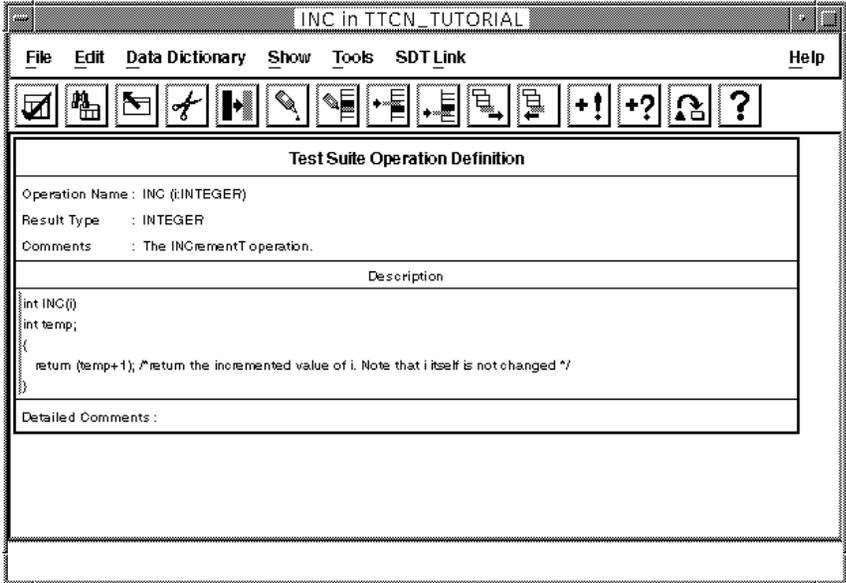


Figure 28: Test Suite Operation Definition

If an operator does not have any arguments it should be called with an empty actual parameter list, e.g. DATE ().

Specifying ASP, PDU and CM Values

The previous section showed how to define structured ASP, PDU and CM types. However, when a tester SENDs or RECEIVES an ASP, PDU or CM it is necessary to specify in detail actual values of these complex types.

Values, or instances, of complete ASPs, PDUs and CMs are called *constraints*. For each ASP, PDU or CM definition table there should be *at least one* corresponding constraint table.

The constraint declaration tables are:

- ASP Constraint Declaration;
- PDU Constraint Declaration;
- Structured Type Constraint Declarations;
- CM Constraint Declaration.

Static and Dynamic Chaining

ASPs, PDUs and structured types may be chained to allow the construction of arbitrarily complex ASPs and PDUs. *Static chaining* means that the actual name of a PDU constraint or structure constraint appears as the value of an ASP parameter, PDU field or structure element, *i.e.* the structure is *hardwired* by symbolic references. *Dynamic chaining* means that the linking occurs when the actual constraint is passed as a parameter in the constraints reference.

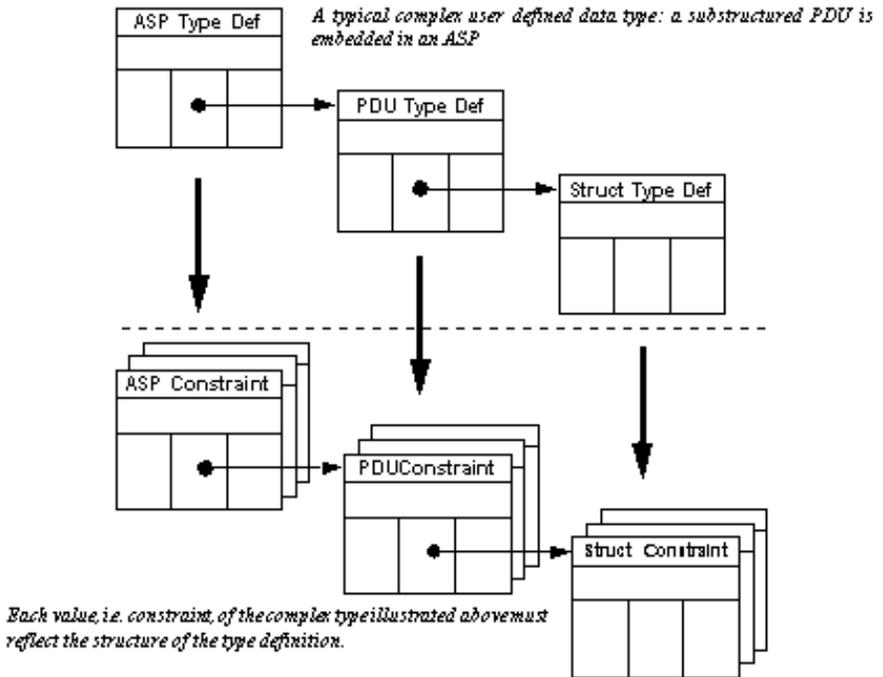


Figure 29: Relation between structured ASP and PDU types and their constraints

Complex ASN.1 Values

The concept of chaining is an integral part of ASN.1, although it is not described in those terms. It is expressed by the use of type references. If a reference is made from one type definition to another then there should be a corresponding value for that reference in the relevant constraints.

ASP Constraints

Generally, for every ASP type definition, there will be at least one ASP constraint declaration. However, some service definitions include ASPs that do not have parameters. In such cases, a constraint is not necessary. The same may apply to coordination messages, but it does not apply to PDUs. A PDU without fields is a nonsense.

ASP constraints are very similar to PDU constraints, which are more fully described in the next section. The rules that apply to PDU constraints, also apply to ASP constraints.

Case study 12a: A typical ASP constraint. Note that the constraint is parameterized - more about that later.

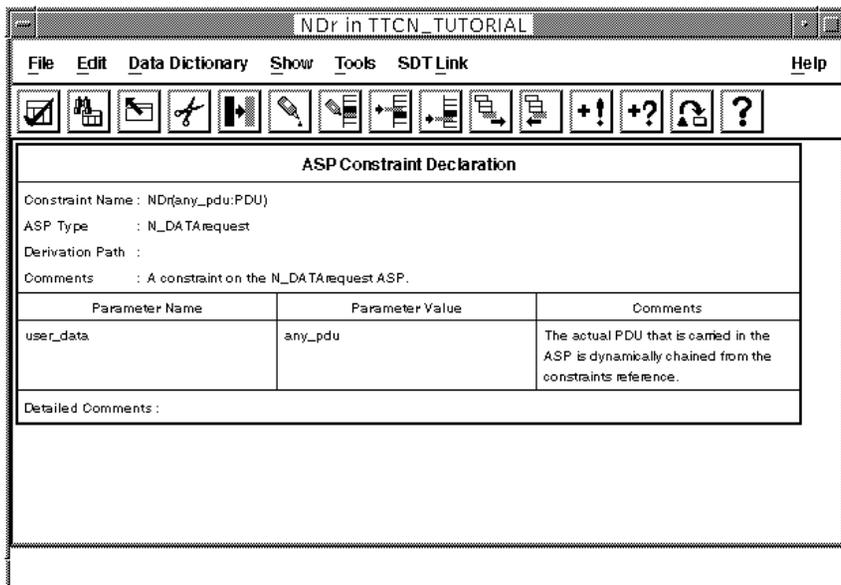


Figure 30: ASP Constraint Declaration (NDr)

The DefCon Utility

The DefCon utility is a TTCN Access application that traverses a test suite in `.itex` format and generates default constraints for all ASN.1 ASP:s found in the test suite. The output is directed to the standard terminal output in `.mp` format.

Case study 12b: A test suite named `MyTest.itex` contains some ASN.1 ASP:s. To generate default constraints for these ASP:s and to store the constraints in a file named `MyTestConstraints.mp`, the DefCon utility should be called like this:

```
c:\> defcon MyTest.itex > MyTestConstraints.mp
```

The generated constraints can then be merged into the `.itex` file with the Autolink Merge utility. See [“Merging TTCN Test Suites in the TTCN Suite” on page 1398 in chapter 35, *TTCN Test Suite Generation*](#) for more information.

Naming generated constraints

When calling DefCon, optional arguments may be provided. These arguments specify which pre- and/or postfixes to use for naming the generated constraints.

The default naming scheme is defined like this:

If there is an ASN.1 ASP named "MyASP" in the test suite, DefCon generates a constraint named "MyASPConstraint" from it. In other words, the default prefix is empty and the default postfix is "Constraint".

The syntax for changing pre- and postfixes is:

```
defcon [-pre <prefix>] [-post <postfix>] <testsuite>
```

Note:

Pre- and postfix strings are given without quotes on the command line.

PDU Constraints

Generally, for each field in the PDU type definition, there will be a corresponding field in the constraint. The value of the constraint field must be compatible with the type definition for that field. We shall see later how fields may be *omitted* or *replaced*, and how the *derivation path* entry should be used. We shall also see how constraint values are matched.

Case study 13: Declaration of an X_PDU constraint using the TTCN tabular format.

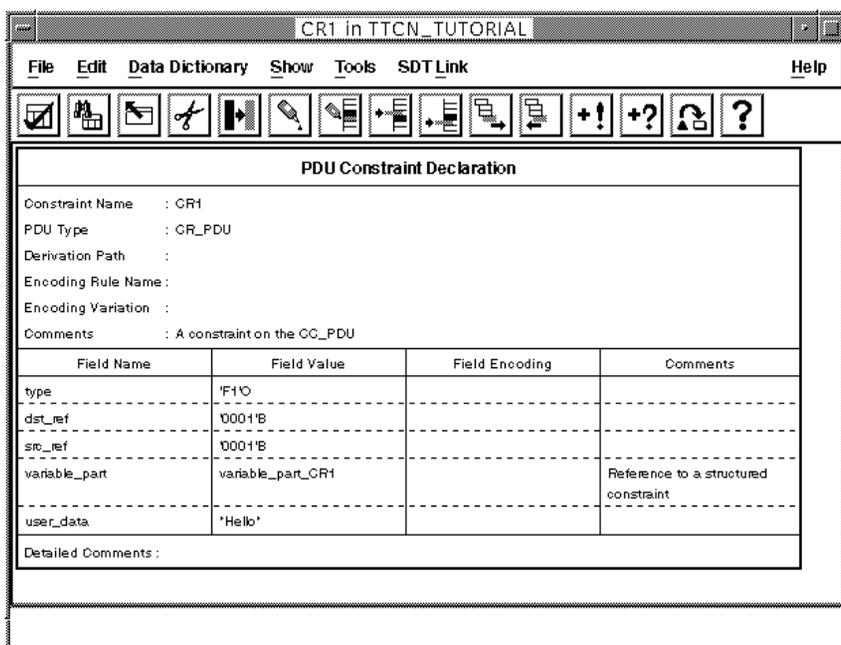


Figure 31: PDU Constraint Declaration (CR1)

Specifying ASP, PDU and CM Values

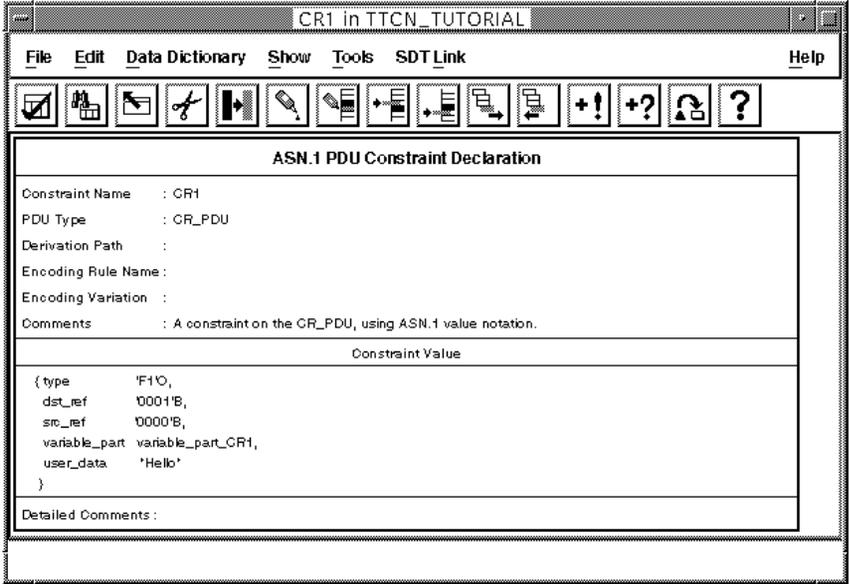


Figure 32: Declaration of the same constraint using ASN.1

Structured Type Constraints

Constraints on structured type definitions and ASN.1 type definitions are constructed in the same way as ASPs and PDUs. Just as the type definitions may be used by both ASP and/or PDU type definitions, so also may the constraints.

When the TTCN tabular format is used the structure of the constraints shall be the same as the structure of the type definitions. That is, if a PDU field is defined as being of structure type then there will be one constraint for the PDU and one for the structure.

This rule is relaxed in ASN.1. The structure must be compatible but there need not necessarily be a one-to-one correspondence between the type tables and the constraint tables.

Case study 14: Declaration of a structured type constraint using TTCN tabular format.

The screenshot shows a software window titled "variable_part_CR1 in TTCN_TUTORIAL". The window has a menu bar with "File", "Edit", "Data Dictionary", "Show", "Tools", "SDT Link", and "Help". Below the menu bar is a toolbar with various icons. The main content area is titled "Structured Type Constraint Declaration" and contains the following information:

Constraint Name : variable_part_CR1
 Structured Type : VARIABLE_PART
 Derivation Path :
 Encoding Variation :
 Comments : A constraint on the structure type VARIABLE_PART for teh CR_PDU

Element Name	Element Value	Element Encoding	Comments
paramA_id	-		Omit this field
paramA	-		Omit this field
paramB_id	'01'B		
paramB	TRUE		

Detailed Comments :

Figure 33: Structured Type Constraint Declaration (variable_part_CR1)

Specifying ASP, PDU and CM Values

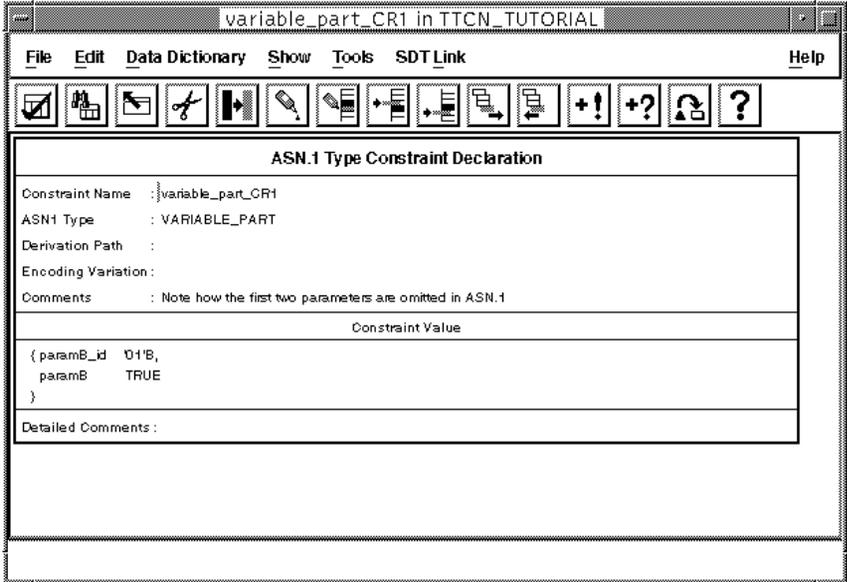


Figure 34: Declaration of the same substructure constraint using ASN.1

CM Constraints

Coordination message constraints are also similar to PDU constraints.

Case study 15: Declaration of a CM constraint using TTCN tabular format

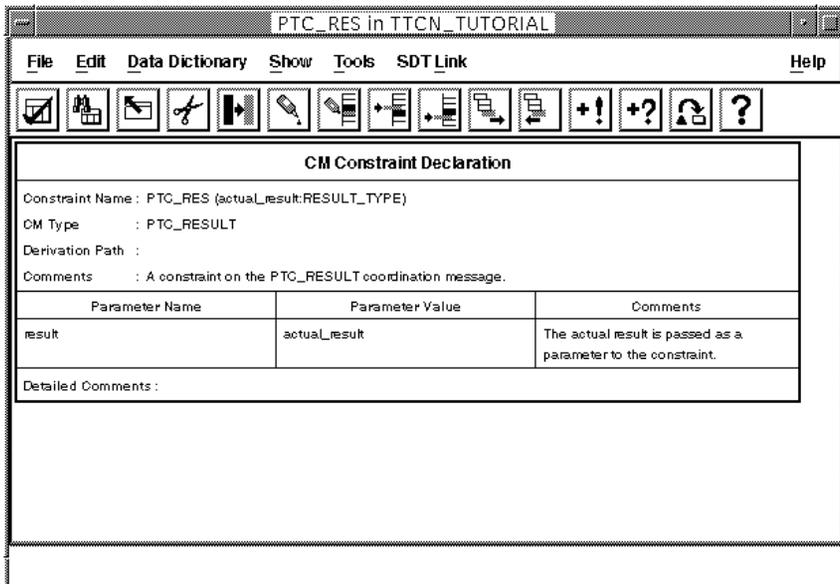


Figure 35: CM Constraint Declaration (PTC_RES)

Specifying ASP, PDU and CM Values

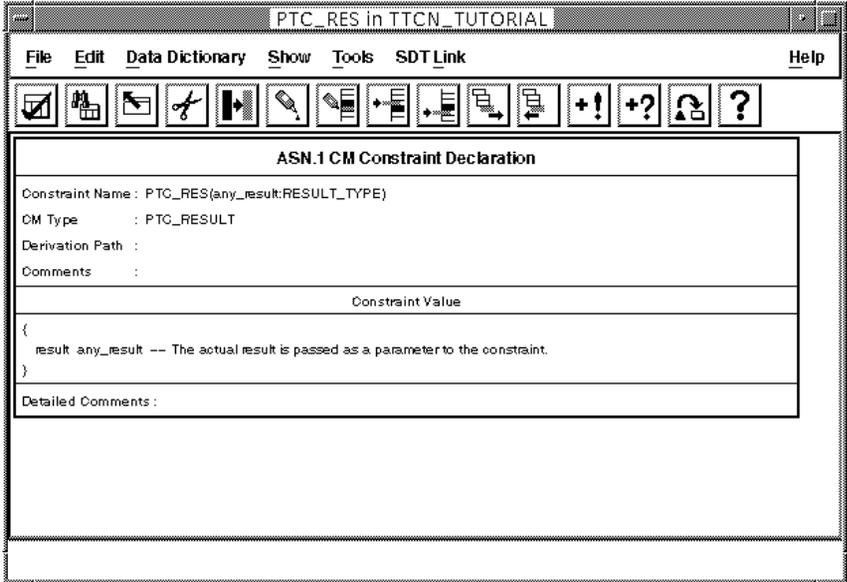


Figure 36: Declaration of the same CM constraint using ASN (we shall assume that `RESULT_TYPE` is an ASN.1 `ENUMERATED` type)

Constraint References

The TTCN SEND and RECEIVE statements indicate only which ASP or PDU type is to be transmitted or received. The constraints column in dynamic behaviour tables is used to state exactly which ASP or PDU value is to be sent, or is expected to be received. In other words, each SEND or RECEIVE statement must be accompanied by a constraints reference.

Note:

This rule can be relaxed for parameterless ASPs.

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		L! N_DATArequest	NDr		
2		L? N_DATAindication	NDi		
3		L? OTHERWISE			

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		U? X_CONNECTindication	CONind		
2		U! X_CONNECTresponse	CONrsp		
3		U? OTHERWISE			

Figure 37: Using constraint references in behavior lines

Parameterized Constraints

Constraints can be *parameterized*. That is, a constraint name may be followed by an optional formal parameter list. The formal parameters can be used in the value column of the constraint.

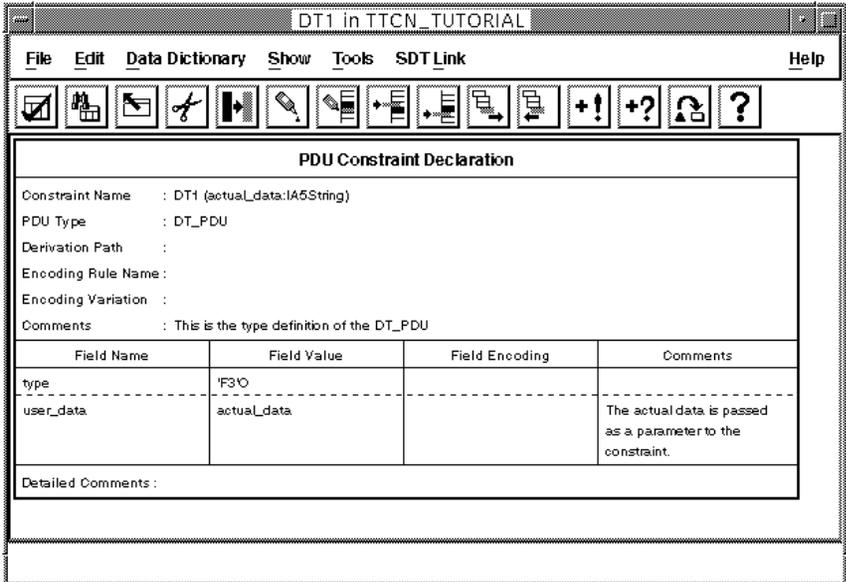


Figure 38: A parameterized constraint

The actual parameters are passed to the constraint when it is invoked from the constraints column in a behaviour description.

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
2		L! DT_PDU	DT1("A string")		

Figure 39: Invocation of a parameterized constraint

The actual parameter must always resolve to a specific value. In a SEND constraint this is the value that will eventually be encoded and transmitted.

In a RECEIVE constraint the actual parameter resolves to the value that will be matched against the received value. No binding occurs, *i.e.* the received value is not bound to the actual parameter. If it wished to capture received values, then this should be done by explicit assignment statements in the behaviour descriptions.

Dynamic Chaining

A common use of parameterized constraints is to link ASPs, PDUs and structures dynamically rather than statically, as we have described earlier. The linking occurs when the actual constraint is passed as a parameter in the constraints reference.

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
2		L! N_DATArequest	NDr (DT1)		

Figure 40: Dynamic chaining of a parameterized PDU in an ASP

An *N_DATArequest* is used to carry the *DT_PDU* of the previous example.

Sending and Receiving Constraints

The rules for sending a constraint are not the same as those for receiving one. We shall examine each of these aspects in turn.

Constraints and the SEND Statement

A constraint in the context of SEND specifies the values that will eventually be transmitted over the network (at this point in time we will ignore encoding issues). In TTCN this transmitted object is called the *Send Object* which is built from information in the relevant constraint. Note that assignments may override values derived from the constraint in the Send Object, which is why BUILD occurs before ASSIGNMENT_LIST.

- SEND³ BUILD² [QUALIFIER]¹ [ASSIGNMENT_LIST]³
[TIMER_OPERATION]⁴

Constraint Values and SEND

In the context of SEND we shall use the term received *constraint value* to mean the value of an ASP parameter, PDU field or CM field of the ASP, PDU or CM constraint that the test specifier wishes to transmit. The type of the constraint value is defined in the relevant ASP, PDU or CM definition.

Constraint values for Send Objects should always be fully specified at the time of transmission of the object.

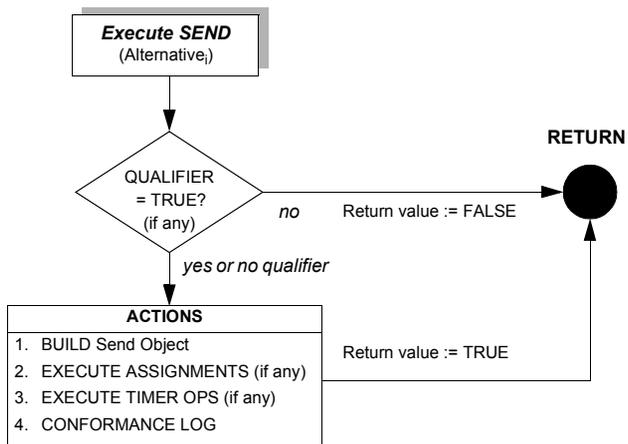


Figure 41: Execution of an alternative that contains SEND

Constraints and the RECEIVE Statement

The receipt of an ASP, PDU or CM is more complex than simply receiving an ASP, PDU or CM of the correct type. Testing often requires that the composition of the ASP, PDU or CM is checked in detail. This is achieved in TTCN by specifying a constraint that the ASP, PDU or CM is expected to match. The RECEIVE event can be considered successful only if all the conditions stipulated in the constraint are satisfied. We shall, therefore, extend our description of the RECEIVE statement line of [“The RECEIVE Statement” on page 35](#) to be:

- RECEIVE¹ MATCH² [QUALIFIER]³ [ASSIGNMENT_LIST]⁴ [TIMER_OPERATION]⁵

Received Object

TTCN uses the term *Received Object* to mean the ASP, PDU or CM that is currently at the top of the relevant incoming PCO or CP queue, and is being checked during evaluation of a RECEIVE statement.

Constraint Values and RECEIVE

In the context of RECEIVE we shall use the term received *constraint value* to mean the value of an ASP parameter, PDU field or CM field of the ASP, PDU or CM field that the test specifier wishes the received value to match. Sometimes the received constraint value is called the *expected value*. The type of the expected value, defined in the relevant ASP, PDU or CM definition, is called the expected type.

Received Value

We shall use the term *received value* to mean the value of received object element. A received value is always, of course, a literal value, of a type compatible with the type of the corresponding element in the ASP, PDU or CM definition.

Constraint References

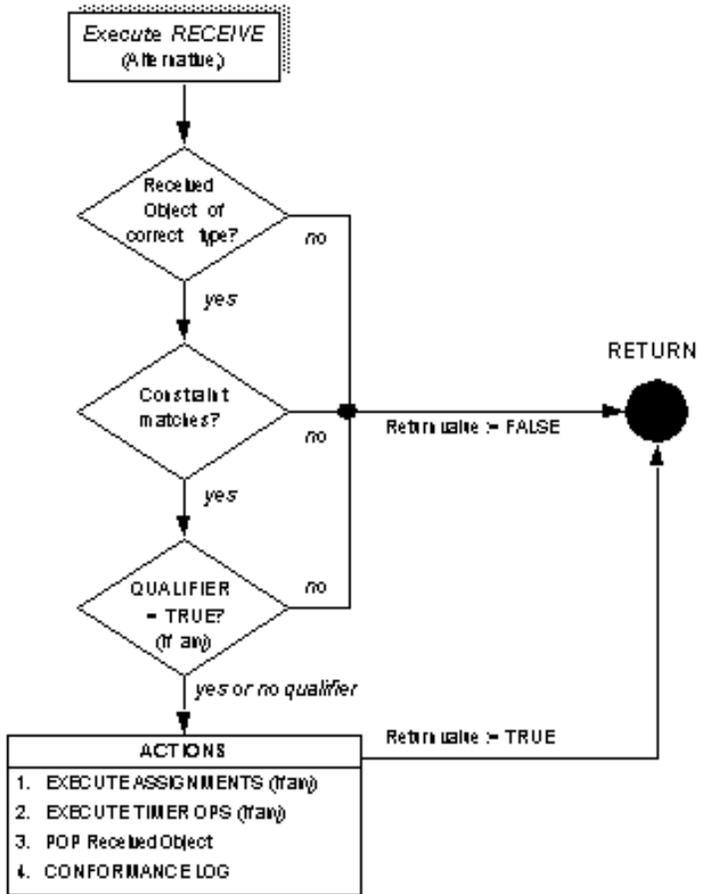


Figure 42: Execution of an alternative that contains RECEIVE

Constraints and the OTHERWISE Statement

Constraints are not used with the OTHERWISE statement. Remember, OTHERWISE will always match if the named PCO incoming queue is not empty. No other checking is required.

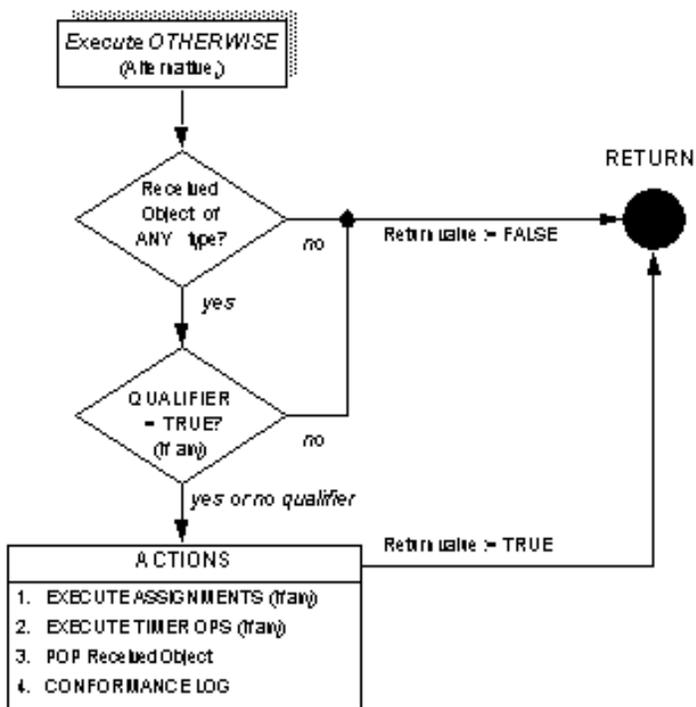


Figure 43: Execution of an alternative that contains OTHERWISE

Matching Received Constraint Values

In this section we shall take a closer look at the RECEIVE statement and how it is used to check the received values against the specified constraint values.

Specific Values

In most cases a constraint value will be a *specific value*. Note that this is not necessarily always a literal value. In TTCN a specific value is an expression which evaluates to a value compatible with the corresponding element type in the relevant ASP, PDU or CM definition. The TTCN syntax allows the operands of these expressions to be:

- literal values;
- constant identifiers;
- formal parameter identifiers;
- predefined and user defined operations;
- expressions, *i.e.* the syntax is recursive.

When a specific value is used as a constraint value a successful match means that the received value is exactly equal to the value to which the constraint expression evaluates. Specific values can of course, be used to specify constraint values of all types.

Note:

We will talk about matching in different contexts. For example, a received value can match a constraint value. This does not mean of course that the entire constraint matches. For that to happen all received values must match all component values specified in the constraint.

Omitting Values

In many cases it may be necessary to *omit* ASP parameters or PDU fields. In the tabular format all parameters or fields are considered to be optional and may be omitted. This is denoted by writing a dash (-) instead of value.

Case study 16: Omitting values.

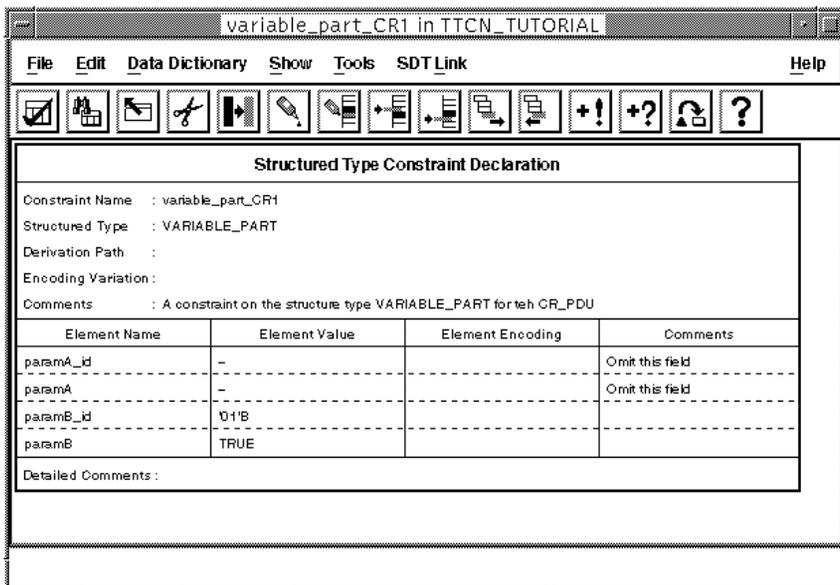


Figure 44: Structured Type Constraint Declaration (*variable_part_CR1*)

In the ASN.1 only parameters or fields that are defined as being OPTIONAL or DEFAULT may be omitted. This can be indicated either by explicitly using the OMIT keyword, or by not including the parameter or field in the constraint.

Matching Received Constraint Values

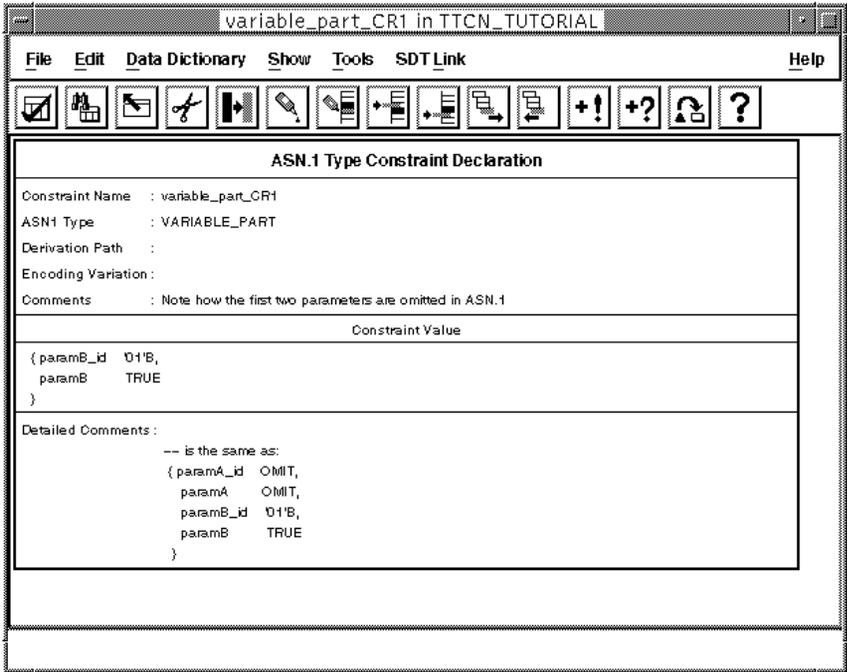


Figure 45: Omitting values in ASN.1 constraints

Replacing Values

In ASN.1 constraints may be constructed from previously defined constraints by using the REPLACE keyword.

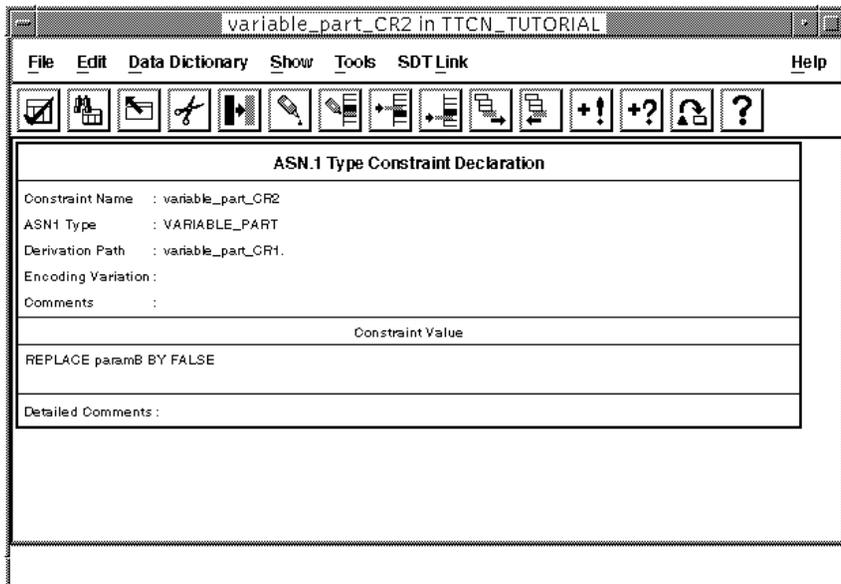


Figure 46: This table indicates that the constraint `variable_part_CR2` is exactly the same as `variable_part_CR1`, except that the value of `paramB` is set to `FALSE`

Matching Mechanisms

In many instances it is not possible, or even desirable, to specify that the field of a received PDU shall have a specific value. It may be more appropriate to say that a match occurs if the received value falls within certain boundaries or fulfils certain conditions.

TTCN supports a number of matching mechanisms: *matching symbols*, *matching operations* and *attributes* that allow the test specifier to express these matching conditions instead of specific values. These mechanisms include:

- lists of values
- complemented lists of values
- ranges of INTEGER values
- any value
- any value or omit value
- wildcards

Matching Received Constraint Values

- if present attribute
- length attributes

Matching Value Lists

A constraint value may be a list of one or more specific values (remember that specific values include expressions *etc.* so the elements in the list may be quite complex). A match occurs only if the received value is equal to any one of the values in the constraint value list, otherwise the match fails.

- ('00'B, '11'B) will match if the received value is either '00'B or '11'B.

Complementing Value Lists

If a value list is preceded by the keyword **COMPLEMENT** then a match occurs only if the received value is not equal to any of the values in the constraint value list, otherwise the match fails. Complement can be used on values of any type.

- **COMPLEMENT** ('00'B, '11'B) will match if the received value is either '01'B or '10'B. Note that this is the same as the list: (NOT '00'B, NOT '11'B).

Matching Ranges

Ranges may only be used to match values of **INTEGER** compatible types. The keywords **INFINITY** and **-INFINITY** may be used to specify ranges that may be unbounded in the positive and/or negative direction.

A range matches if the received value is within the range, including the upper and lower boundary.

- the range (8 .. **INFINITY**) matches any **INTEGER** value greater than 7.

Matching Any Value

In many cases the test suite specifier is prepared to accept *any* single value for a particular field, provided that the actual value is compatible with the corresponding element type.

The matching symbol *AnyValue* is denoted by “?”. A match will occur if the received value is any value that is compatible with the expected type.

- suppose that we have declared a BITSTRING of length exactly 2; then *AnyValue* would match one of ‘00’B, ‘01’B, ‘10’B and ‘11’B but nothing else;
- suppose that we have declared a value of SEQUENCE OF INTEGER type; then *AnyValue* will match any SEQUENCE OF INTEGER, except an empty sequence.

Matching Any Value, or Omitting It Altogether

The *AnyOrOmit* matching symbol, denoted by “*”, is similar to *AnyValue*, except that the value may be omitted altogether. If there is a value present then a match will occur if the received value is any value that is compatible with the expected type; otherwise the value must be omitted. This is only allowed with optional fields.

- suppose that we have declared a BITSTRING of length exactly 2; then *AnyOrOmit* would match one of ‘00’B, ‘01’B, ‘10’B and ‘11’B, or a value could be missing altogether;
- suppose that we have declared a value of SEQUENCE OF INTEGER type; then *AnyOrOmit* will match any SEQUENCE OF INTEGER, including an empty sequence.

Wildcards Within Values

There are two wildcards that may be used within values:

- *AnyOne*;
- *AnyOrNone*.

The *AnyOne* symbol, denoted by “?”, is used to replace single elements *within* all the string types, and *within* SEQUENCE, SEQUENCE OF, SET and SET OF types. However, the element may not be omitted.

- ‘?0’B would match either ‘00’B or ‘10’B;
- “ab?z” will match any character string of length 4 that begins with *ab* and ends with *z*;
- A value of SEQUENCE OF INTEGER such as: {1, 2, ?, 3} means that the third element matches any INTEGER value.

Matching Received Constraint Values

Note that the denotation is the same as for AnyValue, but the semantics of the symbol are not the same.

The *AnyOrNone* symbol, denoted by “*”, is used to replace single elements or a consecutive number of elements *within* all the string types, and *within* SEQUENCE, SEQUENCE OF, SET and SET OF types. Also, the element may be omitted.

- ‘*0’B would match any BITSTRING value that ended with a zero bit;
- “ab*z” will match any character string that begins with *ab* and ends with *z*, including the string “abz”;
- A value of SEQUENCE OF INTEGER such as: {1, 2, *, 3} means that any SEQUENCE OF INTEGER that begins with 1, 2 and ends with 3, including {1,2,3}, will match.

The If_Present Attribute

The *If_Present* attribute is intended for use with OPTIONAL fields. The test suite specifier may not know beforehand whether the IUT will be including an OPTIONAL value or not in a particular PDU - the protocol allows either or. The test then has to specify that if the optional value is present it should be checked.

- 3 IF_PRESENT means that either the INTEGER value of 3 will be accepted for that particular field or no value shall be present.

Note that in the tabular format all fields are considered to be OPTIONAL. The match will occur if the received value is any value that is allowed by the specified expected type, *i.e.* they need not be explicitly declared as such. In ASN.1 this is not the same case; any fields that are OPTIONAL have to be declared as such.

Length Restrictions

Length restrictions apply to the following types:

- BITSTRING
- HEXSTRING
- OCTETSTRING
- CharacterString
- SEQUENCE OF
- SET OF

Essentially these are the same length restrictions that may be placed on the type definitions. The restriction may state the precise length of the string:

- HEXSTRING [8]
- or it may define a range:
- HEXSTRING [4 .. 8]

The screenshot shows a window titled "a_constraint in TTCN_TUTORIAL" with a menu bar (File, Edit, Data Dictionary, Show, Tools, SDT_Link, Help) and a toolbar. The main content is a "PDU Constraint Declaration" table.

PDU Constraint Declaration

Constraint Name : a_constraint (PAR:BITSTRING)
PDU Type : A_TTCN_PDU
Derivation Path :
Encoding Rule Name :
Encoding Variation :
Comments : This is a constraint on an incoming PDU

Field Name	Field Value	Field Encoding	Comments
fieldA	{'00'B, '11'B}		Accept either BITSTRING '00'B or '11'B
fieldB	COMPLEMENT {'00'B, '11'B}		Accept either BITSTRING '00'B or '11'B
fieldC	{8..INFINITY}		Accept any INTEGER value ≥ 8
fieldD	{'0'B}		Accept either BITSTRING '00'B or '10'B
fieldE	*'0'B		Accept any BITSTRING of any length that ends with a zero bit
fieldF	PAR [4..8]		Accept any BITSTRING of length ≥ 4 and length ≤ 8

Detailed Comments :

Figure 47: Examples of matching mechanisms used in a PDU constraint

Encoding

The TTCN standard says nothing about the actual *encoding* of values that are to be transmitted and received over the network. This aspect is being addressed in the second TTCN amendment, WDAM 2. It will allow the TTCN user to specify encodings at several levels:

- for all ASPs and/or PDUs;
- for individual ASP types and/or PDU types;
- for individual ASP parameters and/or PDU fields;
- for individual ASP constraints and/or PDU constraints;
- for individual ASP constraint parameters and/or PDU constraint fields.

Encoding ASPs

It is unusual for a standard to specify the types of the parameters that constitute an ASP. How ASPs are realized is an implementation issue, outside the scope of the standard. The types, therefore, that are given to TTCN ASPs should not be considered binding - they are there to give a consistent representation in the test suite, and are mainly for documentation purposes.

In other words, checking of ASP parameters should be consistent with the implementation of those ASPs in the test system, rather than the exact TTCN specification.

Encoding PDUs

In contrast to ASPs, PDU fields are typed in the relevant protocol standard and it is essential that these types are implemented correctly in the ETS. As far as encoding is concerned, TTCN currently refers to the standards that the PDUs are derived from. For example, if ASN.1 is used it is probable that the ASN.1 basic encoding rules (BER) rules apply, but not necessarily. Work needs to be done on this issue.

Manipulation of Encodings

In some cases of testing it may be necessary to manipulate the encoding of values, *e.g.* in testing of the *presentation* layer. This aspect, too, is addressed in WDAM 2.

Referencing Components of Complex Types

TTCN allows the use of individual components of complex types as operands in expressions or as the l.h.s. of an assignment. Such components include:

- a single ASP parameter;
- a single PDU field;
- a single structure element;
- a single CM field.

In the context of ASN.1 it is possible to access:

- an individual BIT in a BITSTRING;
- an element in a SEQUENCE or SEQUENCE OF;
- an element in a SET or SET OF;
- an element in a CHOICE.

These references may be made either:

- in the context of a SEND or RECEIVE statement; or
- by capturing an incoming ASP or PDU for later reference.

References in the Context of SEND and RECEIVE

These are references to ASP parameters, PDU fields or structure elements made from a statement line that contains a SEND or RECEIVE and, most importantly, an associated constraint. In their simplest form these references are denoted by:

- ASP_Identifier . ParameterIdentifier
- PDU_Identifier . FieldIdentifier
- CM_Identifier . FieldIdentifier
- StructuredTypeIdentifier . ElementIdentifier

Suppose that a substructured PDU is chained to an ASP. To reference the k^{th} element in the structure from a statement line we could write:

- ASP_Identifier . Parameter _{i} . PDU_Identifier . field _{j} . Structure-Identifier . element _{k}

However, because the ASP, PDU and structure identifiers are unique within the test suite, it is allowed to simply use:

Referencing Components of Complex Types

- StructureIdentifier . element_k

For example, if we wish to preserve the value of the *user_data* field of the incoming *DT_PDU*, embedded in an *N_DATAindication*. This could be done by writing:

- A := N_DATAindication . user_data . DT_PDU . user_data

This is rather verbose and because the PDU identifier is unique it is enough to write:

- A := DT_PDU . user_data

In other words the ‘dotted path’ need only contain the identifiers that are enough to give a complete and unique reference.

Referencing ASN.1 Elements

The same mechanism can be used to reference elements in ASN.1 constraints that use SEQUENCE, SEQUENCE OF etc.

Suppose that we have defined the following PDU:

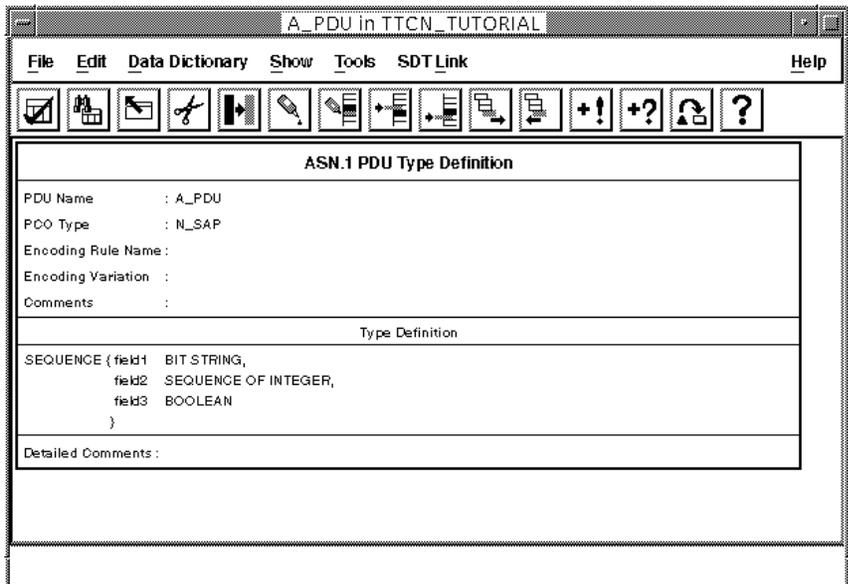


Figure 48: A TTCN PDU Type

A constraint on that PDU may be:

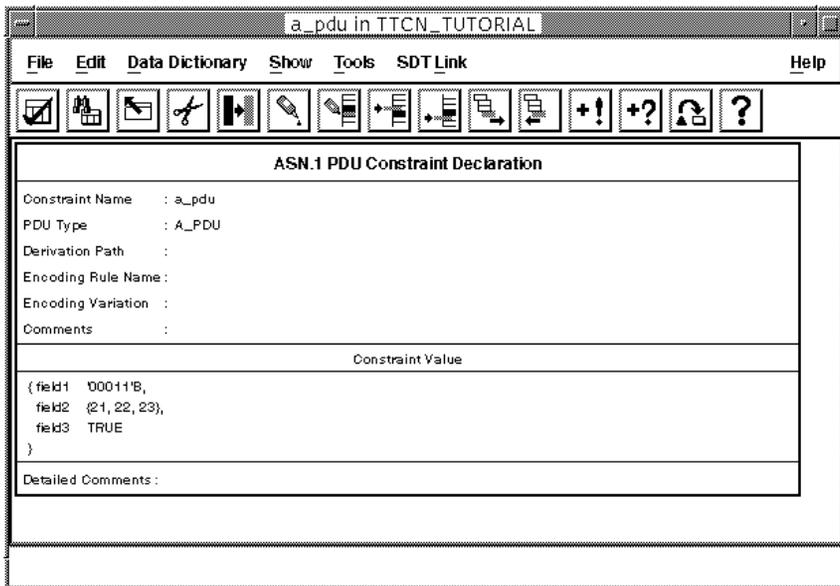


Figure 49: A TTCN PDU Constraint

Then writing:

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
		L! A_PDU (A_PDU.field3 := FALSE)	a_pdu		

Figure 50: A Send Statement

means that the third field in the constraint is overridden and that the send object *a_pdu* is transmitted with *field3* having the value *FALSE*.

Note:

Where possible values should be set using parameters, rather than by this mechanism.

Referencing Components of Complex Types

In cases where the elements are not named, such as the elements in the SEQUENCE OF INTEGER that compose *field2* in our previous examples, it is possible to reference the element by position. For example, if we wish to override the value 22 in *field2* we simply write:

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
		L! A_PDU (A_PDU.field2.(2) := 33)	a_pdu		

Figure 51: A Send statement

Individual bits in a BIT STRING can also be accessed in a similar manner. If we wish to change the third bit in the value of *field1* from 0 to 1 we write:

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
		L! A_PDU (A_PDU.field1[3] := 1)	a_pdu		

Figure 52: A Send statement

This mechanism cannot be used with other string types.

Capturing Incoming ASPs and PDUs

An incoming ASP or PDU (*i.e.* received object) is only preserved for the duration of a RECEIVE statement, *i.e.* components of the received object cannot be accessed on statement lines subsequent to the RECEIVE event. It is possible, however, to declare variables of ASP, PDU or structure type. These variables are then bound to the received object. Suppose the variable *temp_pdu* is of type *A_PDU*:

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
		L? A_PDU temp_pdu := A_PDU	a_pdu		

Figure 53: A Receive statement

We can now access components of *a_pdu* on subsequent statement lines and not just on the statement line that contains the RECEIVE statement:

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
		[temp_pdu.field3]			

Figure 54: A Qualifier statement

Verdicts

There are two mechanisms in TTCN that provide assignment of *verdicts* to a test case. These mechanisms are:

- preliminary results;
- explicit final verdicts.

A preliminary result or explicit final verdict may be associated with any TTCN statement except for the following:

- IMPLICIT SEND;
- ATTACH;
- GOTO;
- REPEAT.

The Result Variable

TTCN has a predefined test case variable, known as the *result variable*, called *R*. This variable may be used in expressions and the verdict column of a behaviour description. It is used to store preliminary results and has the following characteristics:

- A preliminary verdict does not terminate execution of a test case;
- it may appear in expressions as a read-only variable, i.e. it may not be used on the l.h.s. of an assignment;
- it may only take one of the values: *pass*, *fail*, *inconc* or type definition. These values are predefined identifiers, and are case sensitive;
- changes are made to its value by entries in the verdicts column;
- at the start of a test case *R* is bound to the value type definition.

Preliminary Results

The value of R is changed by recording a preliminary result in the verdicts column. A preliminary result may be one of the following:

- (P) or (PASS), meaning that some aspect of the test purpose has been achieved;
- (I) or (INCONC), meaning that something has occurred which makes the test case inconclusive for some aspect of the test purpose;
- (F) or (FAIL), meaning that a protocol error has occurred or that some aspect of the test purpose has resulted in failure.

For example:

- writing (FAIL) in the verdict column will bind R to the value *fail*.

Preliminary results have an order of precedence, for example:

- if R has the value *fail* and a preliminary result (PASS) is encountered in the verdict column, then R cannot be changed to *pass* and it will remain bound to *fail*. On the other hand, if R has the value *pass* and a preliminary result (FAIL) is encountered in the verdict column, then R is bound to the value *fail*.

The table below shows how R may be changed according to the precedence rules:

Current value of R	Preliminary verdict		
	(PASS)	(INCONC)	(FAIL)
none	pass	inconc	fail
pass	pass	inconc	fail
inconc	inconc	inconc	fail
fail	fail	fail	fail

Figure 55: Calculation of the preliminary result variable R

Final Verdicts

Execution of a test case is terminated either by:

- reaching a leaf of the test case behaviour tree; and/or
- an explicit final verdict on the behaviour line (i.e. in the verdict column).

A final verdict may be one of the following:

- P or PASS, meaning that a *pass* verdict is to be recorded;
- I or INCONC, meaning that an inconclusive verdict is to be recorded;
- F or FAIL, meaning that a *fail* verdict is to be recorded;
- the predefined variable *R*, meaning that the value of *R* is to be taken as the final verdict, unless the value of *R* is *none* in which case a test case error is recorded instead of a final verdict.

If no explicit final verdict is reached, then the final verdict is the value of *R*. If *R* is still bound to the value *none* then this is a test case error.

The final verdict must be consistent with the value of *R*. For example:

- if *R* has the value *fail* and an explicit final verdict PASS is encountered in the verdict column, then a final verdict of *fail* and not *pass* should be recorded. On the other hand, if *R* has the value *pass* and an explicit final verdict FAIL is encountered in the verdict column, then a final verdict of *fail* should be recorded.

The table below shows how the final verdict should be recorded according to the value of *R*:

Current value of R	Final verdict			
	(PASS)	(INCONC)	(FAIL)	R
none	pass	inconc	fail	*error*
pass	pass	inconc	fail	pass
inconc	*error*	inconc	fail	inconc
fail	*error*	*error*	fail	fail

Figure 56: Final verdict

The GOTO Statement

In order to be able to express repetitive behaviour in a convenient way, TTCN allows statement lines to be *labelled* so that jumps may be made to them from later points in the tree. A GOTO is denoted either by:

- -> LabelIdentifier

or:

- GOTO LabelIdentifier

Infinite loops should be avoided, *i.e.* entering the GOTO loop should always depend on some event occurring or condition being fulfilled.

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
4 5 : 7	LAB	: L! N_DATArequest (count := count+1) [count <= max] : -> LAB : :	NDr		

Figure 57: Using GOTO in a behavior tree

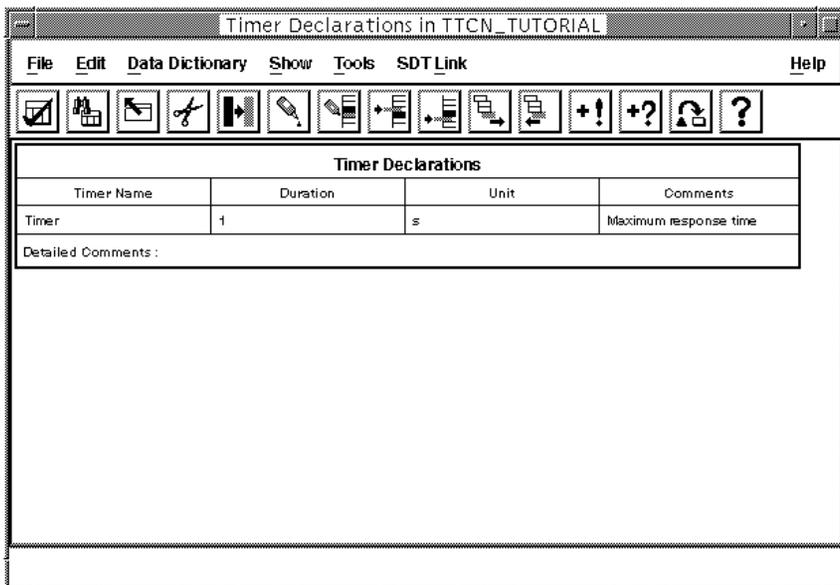
The following rules should be followed:

- a GOTO can only be made within a *single* tree in a behaviour description;
- the label should be *unique* within the behaviour description;
- line numbers may not be used as labels;
- the label must always be associated with the *first* statement line in a given set of alternatives, *i.e.* a GOTO cannot cause a jump to the middle of a set of alternatives;
- a result of the previous rule means that a GOTO to the first level of alternatives in a test step (*i.e.* the test step root) is not allowed;
- a GOTO may only be made to an ancestor node in the behaviour tree, *i.e.* a jump to a part of the tree that has previously been executed;
- no other statements may be used in conjunction with a GOTO.

Timer Statements

TTCN timers are used to test timer events in the IUT. This is usually done by timing an expected response from the IUT using the START timer operation and the TIMEOUT event. The CANCEL timer operation is used to stop and reset a running or expired timer. All timers are declared in the *Timer Declarations* table. The *duration* is the period of time that will pass from the moment a timer is started to the moment it expires. Duration is measured in one of the following units:

- ps (i.e. picosecond);
- ns (i.e. nanosecond);
- us (i.e. microsecond);
- ms (i.e. millisecond);
- s (i.e. second);
- min (i.e. minute).



The screenshot shows a software window titled "Timer Declarations in TTCN_TUTORIAL". The window has a menu bar with "File", "Edit", "Data Dictionary", "Show", "Tools", "SDT Link", and "Help". Below the menu bar is a toolbar with various icons for editing and navigation. The main content area contains a table titled "Timer Declarations". The table has four columns: "Timer Name", "Duration", "Unit", and "Comments". There is one row in the table with the following values: "Timer", "1", "s", and "Maximum response time". Below the table is a section labeled "Detailed Comments:" which is currently empty.

Timer Declarations			
Timer Name	Duration	Unit	Comments
Timer	1	s	Maximum response time

Detailed Comments :

Figure 58: Declaration of timers

The Timeout List

TTCN maintains a *timeout list*. If a timer expires its name is added to the timeout list. Three things can remove the name of the timer from the timeout list:

- a successful TIMEOUT statement;
- a START timer operation;
- a CANCEL timer operation.

The TIMEOUT Statement

The test suite specifier may state that a named timer be checked to see if it has timed-out. This is denoted by:

- ?TIMEOUT TimerIdentifier

When this statement is encountered while processing a statement line the TIMEOUT will match if the named timer is in the timeout list, otherwise the TIMEOUT fails.

An alternative use of TIMEOUT is simply:

- ?TIMEOUT

i.e. no *TimerIdentifier* is given. In this case the TIMEOUT statement will succeed as long as the timeout list is *not* empty.

The TIMEOUT statement may be qualified and it may be followed by an ASSIGNMENT_LIST and/or TIMER_OPERATION. The order in which these statements may appear in the statement line is fixed, as shown below; the square brackets indicate that the presence of the statement in the statement line is optional:

- TIMEOUT² [QUALIFIER]¹ [ASSIGNMENT_LIST]³
[TIMER_OPERATION]⁴

Note:

TIMEOUT should not be used to guard against a faulty IUT not sending a required response. It is the responsibility of the test system to implement detection of such an occurrence.

Timer Snapshots

We have already mentioned that at the beginning of each cycle through a set of alternatives a snapshot is taken of the incoming PCO queues. The alternatives are then checked against this snapshot. The same thing is done for the timeout list. A snapshot is taken of this list at the start of each cycle and if a TIMEOUT alternative is encountered in the set of alternatives it is checked against the timeout snapshot rather than the actual timeout list. This means that the expiry of a timer during processing of a set of alternatives is not registered until the timer snapshot is updated.

The START Timer Operation

A named timer is started using the START timer operation. This is denoted by:

- START TimerIdentifier

The duration for this timer is taken from the timer declaration. Alternatively, an explicit duration may be given, which overrides the declared duration:

- START TimerIdentifier (Duration)

If the timer is already running when the START is invoked then the timer is cancelled, reset and then started, *i.e.* the timer is re-started.

If the timer has expired then its name is removed from the timeout list before it is re-started.

The START_TIMER statement may be qualified and it may be followed by an ASSIGNMENT_LIST. The order in which these statements may appear in the statement line is fixed, as shown below:

- [QUALIFIER]¹ [ASSIGNMENT_LIST]² [START_TIMER]³

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
5		:			
:		START a_timer			
8		?TIMEOUT a_timer			

Figure 59: Using START and TIMEOUT in a behavior tree

The CANCEL Timer Operation

A named timer is cancelled using the CANCEL operation. This is denoted by:

- CANCEL TimerIdentifier

An alternative use of CANCEL is simply:

- CANCEL

i.e. no *TimerIdentifier* is given. In this case all running timers are cancelled and reset and the timeout list is cleared.

Cancelling a timer that is expired will result in the timer being reset and its identifier is removed from the timeout list.

The CANCEL_TIMER statement may be qualified and it may be followed by an ASSIGNMENT_LIST. The order in which these statements may appear in the statement line is fixed, as shown below:

- [QUALIFIER]¹ [ASSIGNMENT_LIST]² [CANCEL_TIMER]³

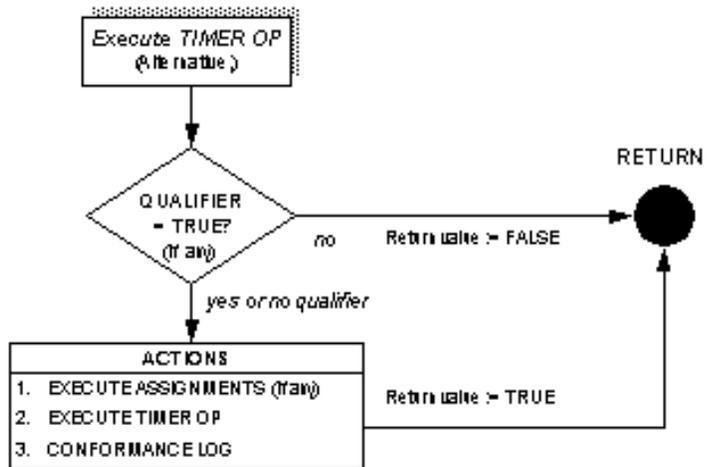


Figure 60: Execution of an alternative that contains a stand-alone timer operation

Constants and Variables

TTCN supports both constants and variables. There are two types of constants:

- test suite parameters;
- test suite constants;

and two types of variables:

- test suite variables;
- test case variables.

The tables used are:

- Test Suite Constant Declarations
- Test Suite Parameter Declarations
- Test Suite Variable Declarations
- Test Case Variable Declarations

Test Suite Constants and Test Suite Parameters

Test suite constants are declared globally and may be used anywhere in the test suite, including the constraints part. The value of the constant is specified at its point of declaration and may not be changed.

Case study 17: Declaration of test suite constants.

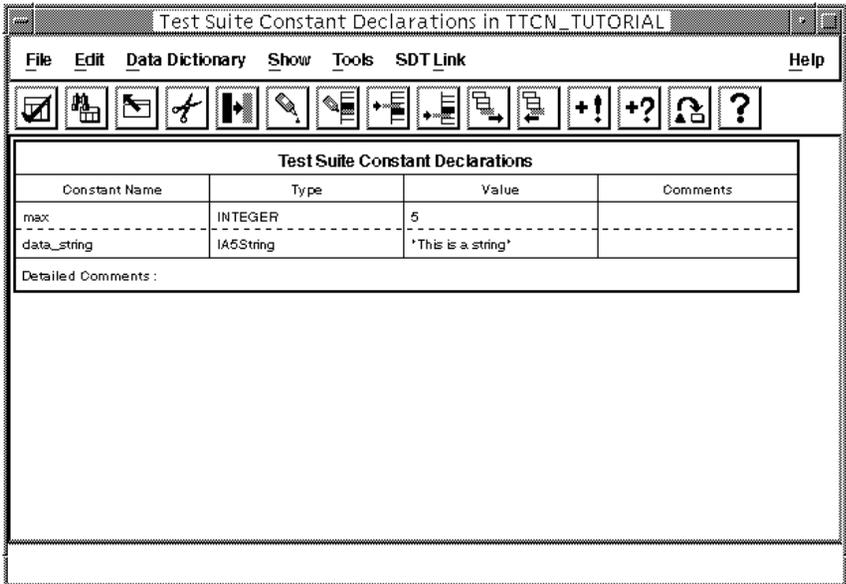


Figure 61: Test suite constant declarations

Test Suite Parameters

Test suite parameters are also constants, but their actual values are not known to the abstract test suite specifier. These values will depend on which IUT is being tested, and possibly on the test system itself. In this sense the values of test suite parameters will be different from IUT to IUT, but during the testing of any given IUT they will remain constant.

Case study 18: Declaration of test suite parameters.

Test Suite Parameter Declarations			
Parameter Name	Type	PICS/PIXIT Ref	Comments
it_address	IA5String	PIXIT question xx	Lower tester address
ut_address	IA5String	PIXIT question yy	upper tester address
Detailed Comments :			

Figure 62: Test suite parameter declarations

The test suite parameter values are derived from the *Protocol Implementation Conformance Statement* (PICS) and the *Protocol Implementation eXtra Information for Testing* (PIXIT). These documents are like checklists that are filled-in according to the characteristics of the IUT.

Prior to executing the tests the PICS and PIXIT are used to bind values to the test suite parameters. This process is called *test suite parameterization*.

Test Suite and Test Case Variables

Both test suite variables and test case variables are declared globally *i.e.* they may be used by test cases, test steps and defaults throughout the test suite. A default value may be specified for each variable, if wished. If no default value is specified, then the variable is said to be *unbound*.

Variables should be bound before use, unless they appear on the l.h.s. of an assignment.

Case study 19: Declaration of test case variables.

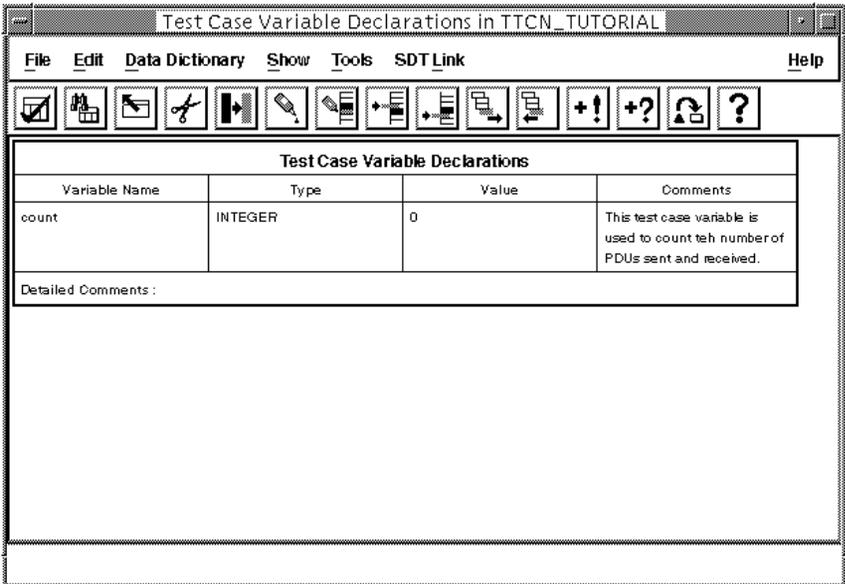


Figure 63: Test case variable declaration

Resetting Default Values

The difference between the two kinds of variable is when they are *reset* to their default values (if no default value is specified for a variable, then resetting means that the variable becomes unbound):

- test suite variables are reset at the end of execution of the *test suite*, which means that information may be retained between test case execution;
- Test case variables are reset at the end of execution of each *test case*, *i.e.* test case variables begin each test case bound to their default values.

Variables in Concurrent TTCN

When more than one test component exists, as does with concurrent TTCN, then each test component is supplied with its own copy of each test case variable.

- In the case study we declare the test case variable *count*. This variable is available to both the lower tester and the upper tester as a separate copy of *count* to each, *i.e.* if the lower tester changes the value of *count* it only changes its copy of *count*, and not the upper tester's copy.

Test suite variables behave the same way in concurrent TTCN as they do in the non-concurrent version.

Dynamic Behaviour Descriptions

There are three types of tables for specifying the behaviour descriptions:

- Test Case Dynamic Behaviour;
- Test Step Dynamic Behaviour;
- Default Dynamic Behaviour.

We have already noted that the difference between the different behaviour tables is in the header, rather than in the body of the tables.

MP_DATA_TRANSFER in TTCN_TUTORIAL

File Edit Data Dictionary Show Tools SDT Link Help

Test Case Dynamic Behaviour

Test Case Name : MP_DATA_TRANSFER
 Group : MULTI/DATA/
 Purpose : IUT shall receive and send a data within time limit, a given number of times over two simultaneous X-connections.
 Configuration : MULTI_PARTY
 Default : T_DEFAULT
 Comments : This test case creates the otehr PTCs in the configuration necessary for a two-connection configuration.
 Selection Ref :
 Description : Data transfer multi-connection

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		CREATE(LOWER_TESTER1: LTS (L1, CP1), LOWER_TESTER2: LTS (L2, CP2), UPPER_TESTER1: UTS (U1), UPPER_TESTER2: UTS (U2))			1)
2		CP1?PTC_RESULT	PTC_RES (pass)		2)
3		CP2?PTC_RESULT	PTC_RES (pass)	PASS	2), 3)
4		CP2?PTC_RESULT	PTC_RES (fail)	FAIL	2), 3)
5		CP1?PTC_RESULT	PTC_RES (fail)		2)
6		CP2?PTC_RESULT	PTC_RES (pass)	FAIL	2), 3)
7		CP2?PTC_RESULT	PTC_RES (fail)	FAIL	2), 3)

Detailed Comments : 1) The CREATE command binds specific test steps to the relevant PTCs in the Test Component Declarations table. It also implicitly starts execution of each test step (i.e. two instances of LT and one instance of UT)
 2) The preliminary results from the lower tester PTCs are picked up here.
 3) Final verdict

Figure 64: Fragment of a test case behavior table, showing the header for the test case

Test case Identifiers and Test Group References

The *test case identifier* appears in the first field and, like most TTCN identifiers, it should be a name unique to the entire test suite. The second field contains the *test case reference*, which is a path name that specifies the test case's *location* in the test suite structure.

In the case of test steps this path specifies the test step's location in the test step library. In the case of defaults it specifies the location of the default in the default library. These references have the general format:

- SuiteIdentifier / GroupIdentifier₁ / . . . / GroupIdentifier_n /

Note the terminating slash, which is the last group name in the path. The path may begin with the first *GroupIdentifier*, *i.e.* the *SuiteIdentifier* is optional. If the test suite has no hierarchy then the reference is empty.

Test Purpose and Objective

In the test case table the third field is used to specify the test purpose. The corresponding field in test steps and defaults is called the *objective*.

Configuration

The configuration entry is introduced by the concurrent TTCN to state the configuration in which this test case behaviour description is used. This field does not appear in test steps and defaults.

Default Behaviour

The default entry is used to state the default behaviour which should be used, if any.

Dynamic Behaviour Descriptions

Case study 20: Test step dynamic behaviour.

The screenshot shows a window titled "LT_DATA_TRANSFER in TTCN_TUTORIAL". The menu bar includes "File", "Edit", "Data Dictionary", "Show", "Tools", "SDT_Link", and "Help". The toolbar contains various icons for editing and navigation. The main content area is titled "Test Step Dynamic Behaviour" and contains the following text:

Test Step Name : LT_DATA_TRANSFER (L:_N_SAP; CPP:CP)
 Group : TEST_STEP_LIB/LOWER/
 Objective : Test data transfer
 Default : LT_DEFAULT(L)
 Comments : This test step implements the test body of our example test case on the lower tester side.
 Description :

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1	LAB	LIDATAout (count=INC(count))	NDriDT1(data_string)		
2		[count <= max] START Timer			
3		L?DATAin	NDijDT1(data_string)	(PASS)	
4		->LAB			
5		?TIMEOUT Timer		(FAIL)	
6		CPP!PTC_RESULT	PTC_RES(fail)		
7		CPP!PTC_RESULT	PTC_RES(pass)		

Detailed Comments :

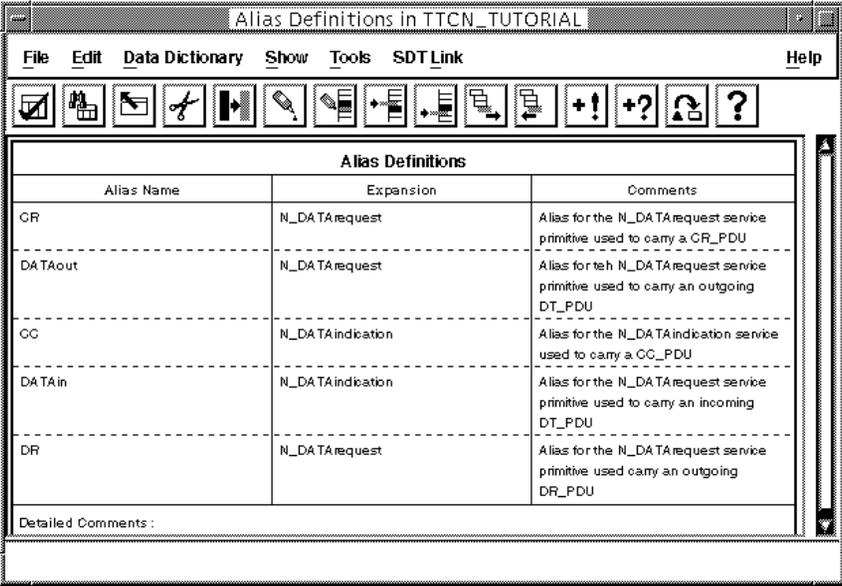
Figure 65: Test step dynamic behaviour (LT_DATA_TRANSFER)

Using Aliases

One of the main aims of TTCN is to specify behaviour descriptions so that the human reader can easily understand the TTCN specification of the test purpose.

The conformance standard requires that behaviour be expressed in terms of (N) and (N-1) ASPs. However, a behaviour tree consisting of mostly (N-1)-data requests and indications says very little to the reader. What is important are the PDUs embedded in these service primitives. If static chaining is used the reader will have no idea, without turning to the constraints, what PDU interactions are specified in the test.

The alias mechanism allows ASPs (and if necessary PDUs) to be re-named to reflect the different PDUs that they carry. The (N-1)-data request and indication may have several aliases, depending on which (N)-PDU they are carrying.



The screenshot shows a window titled "Alias Definitions in TTCN_TUTORIAL" with a menu bar (File, Edit, Data Dictionary, Show, Tools, SDT_Link, Help) and a toolbar with various icons. The main content is a table with the following data:

Alias Definitions		
Alias Name	Expansion	Comments
GR	N_DATArequest	Alias for the N_DATArequest service primitive used to carry a GR_PDU
DATAout	N_DATArequest	Alias for the N_DATArequest service primitive used to carry an outgoing DT_PDU
CC	N_DATAindication	Alias for the N_DATAindication service used to carry a CC_PDU
DATAin	N_DATAindication	Alias for the N_DATArequest service primitive used to carry an incoming DT_PDU
DR	N_DATArequest	Alias for the N_DATArequest service primitive used to carry an outgoing DR_PDU

Below the table is a section labeled "Detailed Comments :

Figure 66: Declaring aliases

Using Aliases

In ISO/IEC 9646-3 aliases are defined as textual expansions. However, it is probably easier to think of alias identifiers as alternatives to ASP or PDU identifiers in the SEND and RECEIVE statements. The effect is exactly the same, *i.e.*

- PCO_Identifier ! AliasIdentifier
- PCO_Identifier ? AliasIdentifier

Test Step Dynamic Behaviour

Test Step Name : LT_DATA_TRANSFER (L:N_SAP; CPP:CP)

Group : TEST_STEP_LIB/LOWER/

Objective : Test data transfer

Default : LT_DEFAULT(L)

Comments : This test step implements the test body of our example test case on the lower tester side.

Description :

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1	LAB	L!DATAout (count:=INC(count))	NDr(DT1 (data_string))		
2		[count <= max] START Timer			
3		L?DATAin	NDi(DT1 (data_string))	(PASS)	
4		->LAB			
5		?TIMEOUT Timer		(FAIL)	
6		CPP!PTC_RESULT	PTC_RES(fail)		
7		CPP!PTC_RESULT	PTC_RES(pass)		

Detailed Comments :

Figure 67: Test Step dynamic behavior using aliases

Modularization of Test Cases

Test cases can be long and complex. There exist two mechanisms that allow test cases to be modularized: *test steps* and *defaults*.

Test Steps

Behaviour trees can be modularized by splitting them into sub-trees called test steps. Test steps are either:

- *local* to a behaviour description; or
- reside in the *test step library*.

Test steps may be parameterized, *i.e.* the calling tree can pass PCOs, variables, literal values, constraints *etc.* to the attached test step.

Local test steps

Local test steps may only be used within the behaviour description in which they appear:

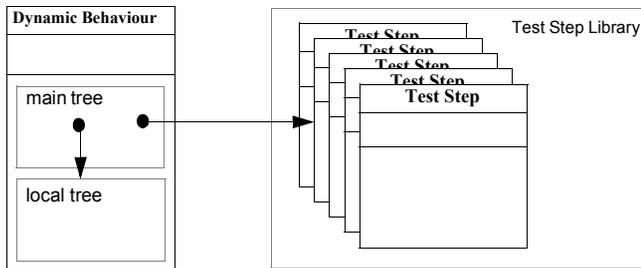


Figure 68: Illustration of local test trees and test step library

The Test Step Library

Test steps that belong to the test step library are specified in Test Step behaviour tables. These steps may be called by *any* test case, test step or default.

The ATTACH Statement

The ATTACH statement is used to invoke a test step, and is denoted by:

- + TreeIdentifier ActualParameterList
for attachment of a local test step; or
- + TestStepIdentifier ActualParameterList
for attachment of a test step in the test step library.

In both cases the actual parameter list should only be used if the test step has a formal parameter list. Note that a parameter may also be a PCO or CP.

Case study 21: The following test step:

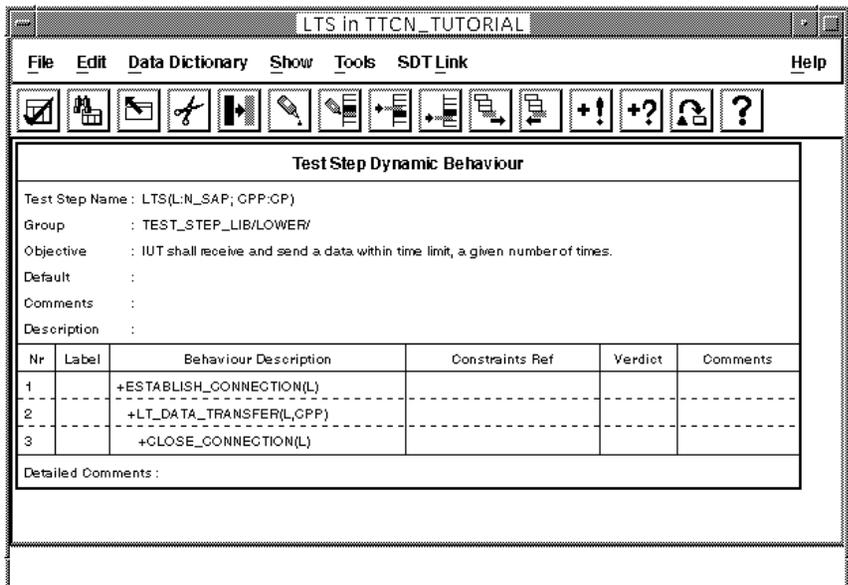


Figure 69: Test Step Dynamic Behaviour (LTS)

Case study 22: Is the same as:

The screenshot shows a window titled "LTS in misc" with a menu bar (File, Edit, Data Dictionary, Show, Tools, SDT_Link, Help) and a toolbar. The main content area is titled "Test Step Dynamic Behaviour" and contains the following information:

Test Step Name : LTS(L:N_SAP; CPP:CP)
 Group : TEST_STEP_LIB/LOWER/
 Objective : IUT shall receive and send a data within time limit, a given number of times.
 Default : LT_DEFAULT(L)
 Comments :
 Description :

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		+ESTABLISH_CONNECTION(L)			
2	LAB	L!DATAOut (count=INC(count))	NDr(DT1(data_string))		
3		[count <= max] START Timer			
4		L?DATAin	NDi(DT1(data_string))	(PASS)	
5		->LAB			
6		+CLOSE_CONNECTION(L)			
7		?TIMEOUT Timer		(FAIL)	
8		CPP!PTC_RESULT	PTC_RES(fail)		
9		+CLOSE_CONNECTION(L)			
10		CPP!PTC_RESULT	PTC_RES(fail)		
11		+CLOSE_CONNECTION(L)			

Detailed Comments :

Figure 70: Test Step Dynamic Behaviour (LT)

Tree Attachment as a Subroutine Call

TTCN defines tree attachment as the actual *expansion* of the called tree, *i.e.* the test step, into the calling tree, which may be a test case or another test step. While this is a sensible approach, adequately described in the TTCN standard, we feel that the view of treating test steps as *subroutines* is a valid one, and one that implements the TTCN semantics for tree attachment correctly. These semantics are easily understood by anyone with a programming background.

Modularization of Test Cases

A test step can be considered as a subroutine when handled in the following manner:

- when an attach statement is reached when looping through a set of alternatives, control is passed to the attached test step;
- if *no* alternative in the *first* set of alternatives in the test step is successful during the *first* loop through that set of alternatives, then control returns from the test step to the calling tree, and evaluation *continues* with the other alternatives, if any, in the same set of alternatives as the attach statement; this has the same effect as if the test step was actually expanded into the calling tree;
- if an alternative in the first level of the test step *is* successful then execution continues in the test step tree;
- if a final verdict is reached in the test step tree then the execution of the test step (*i.e.* the entire test case) is halted and control is *not* returned to the calling tree;
- if *no* final verdict is encountered, before a leaf of the test step tree is reached then control returns to the calling tree, and execution continues with the next set of alternatives (if any) *subsequent* to the attach statement, *i.e.* the next level of indentation.

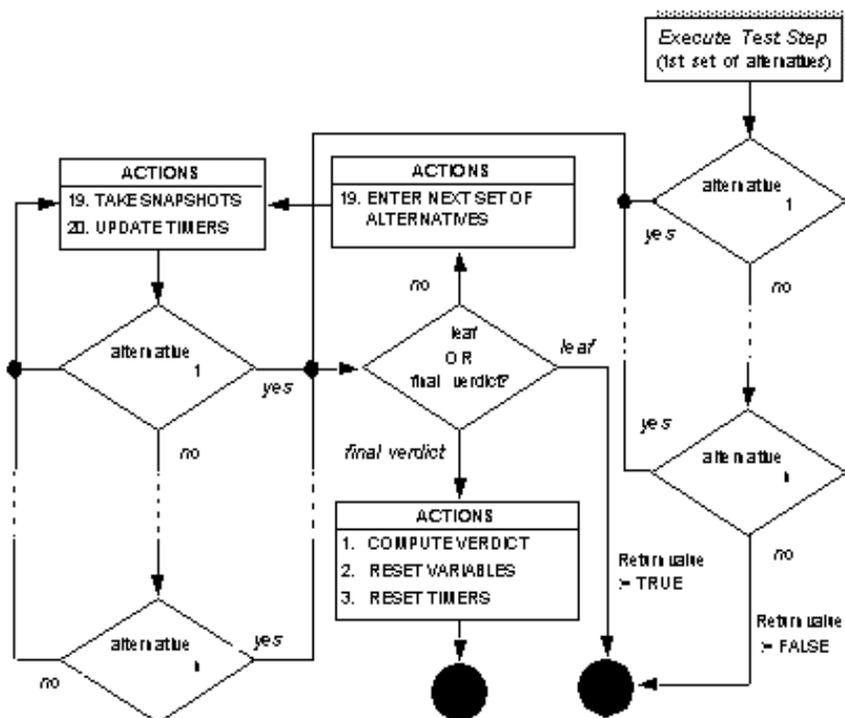


Figure 71: Execution of a test step

Default Behaviour

The conformance standard requires that a TTCN test case must be fully specified, in terms of behaviour. This means that at any point in time a tester must be prepared to accept all possible incoming ASPs or PDUs. This includes not only the ASPs or PDUs that are allowed by the protocol but also any ASP, legal or otherwise, that the IUT or service provider may issue

The easiest way to take care of this is by using the OTHERWISE statement. However, TTCN requires that the OTHERWISE statement leads to a fail verdict and this may not always be desirable. For example, it may be perfectly legal for the underlying network service to issue an N_DISCONNECT indication at any time. Certainly, an OTHERWISE

Modularization of Test Cases

would pick this up, but a verdict of FAIL in such an instance would be quite wrong. The only verdict that should be assigned in this case is INCONCLUSIVE, *i.e.* use with care!

Specifying all possible combinations tends to clutter up the main behaviour description, detracting from the readability of the test case. The default behaviour can be used to specify this peripheral behaviour in a precise manner. It will often comprise the set of ASPs or PDUs that are allowed by the protocol at any given time but which are not part of the test purpose, and an OTHERWISE to catch all other unspecified events. It is also common practice to include a general TIMEOUT in the default.

Case study 23: If we specify the following default behaviour.

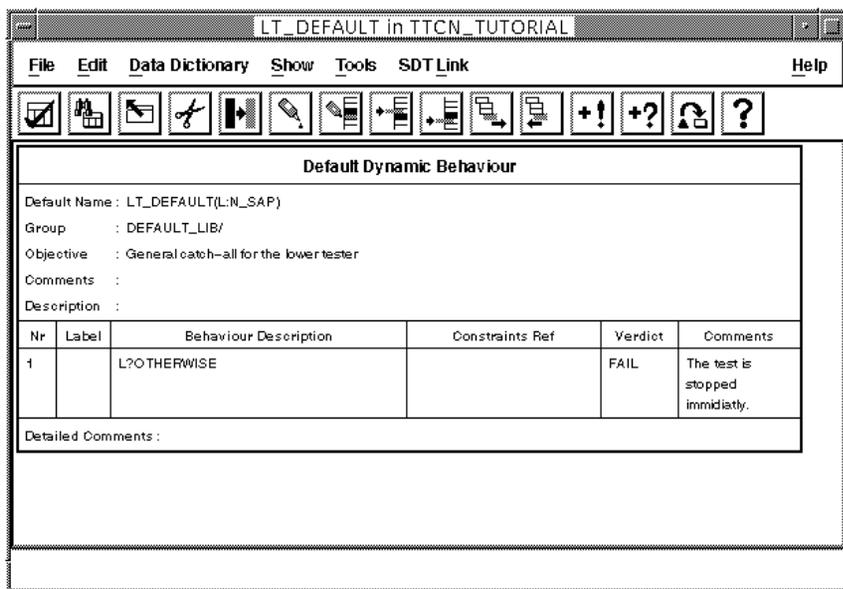


Figure 72: Default Dynamic Behaviour (LT_DEFAULT)

Modeling Default Behaviour

Default behaviour can be modeled as a tree attachment that is implicitly called as the last alternative in every set of alternatives.

Defaults Reference

A test case or test step references this default behaviour in the *Defaults* entry in its header. If this entry is empty then no default behaviour is applicable.

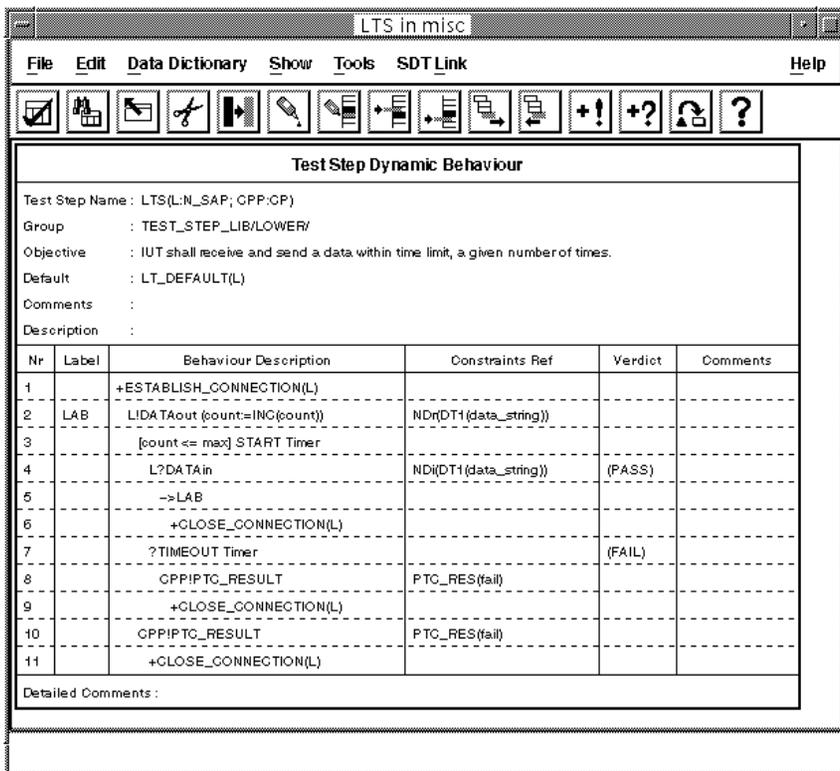


Figure 73: The test step with a default

Modularization of Test Cases

LTS in misc

File Edit Data Dictionary Show Tools SDT Link Help

Test Step Dynamic Behaviour

Test Step Name : LTS(L:N_SAP; GPP:GP)

Group : TEST_STEP_LIB/LOWER/

Objective : IUT shall receive and send a data within time limit, a given number of times.

Default : LT_DEFAULT(L)

Comments :

Description :

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		+ESTABLISH_CONNECTION(L)			
2	LAB	LIDATAout (count=INC(count))	NDr(DT1(data_string))		
3		[count <= max] START Timer			
4		L?DATAin	NDi(DT1(data_string))	(PASS)	
5		->LAB			
6		+CLOSE_CONNECTION(L)			
7		L?OTHERWISE		FAIL	
8		L?OTHERWISE		FAIL	
9		L?OTHERWISE		FAIL	
10		L?OTHERWISE		FAIL	
11		?TIMEOUT Timer		(FAIL)	
12		GPPPTC_RESULT	PTC_RES(fail)		
13		L?OTHERWISE		FAIL	
14		+CLOSE_CONNECTION(L)			
15		L?OTHERWISE		FAIL	
16		L?OTHERWISE		FAIL	
17		GPPPTC_RESULT	PTC_RES(fail)		
18		+CLOSE_CONNECTION(L)			
19		L?OTHERWISE		FAIL	
20		L?OTHERWISE		FAIL	
21		L?OTHERWISE		FAIL	
22		L?OTHERWISE		FAIL	

Detailed Comments :

Figure 74: Is the same as the previous test step

Parameter Lists in TTCN

The following TTCN objects may be parameterized:

- test suite operations
- constraints
- test steps
- defaults

Formal Parameter Lists

In all cases parameterization is indicated by the relevant TTCN object identifier being followed by a *formal parameter list*. For example:

- `an_identifier (fpar1, fpar2:INTEGER, fpar3:HEXSTRING)`

Actual Parameter Lists

Parameterized objects are invoked with an *actual parameter list*. For example:

- `an_identifier(1, 2, FALSE)`

The following rules apply:

- the number of parameters in the actual parameter list must be the same as the number of parameters in the formal parameter list;
- the actual values in the actual parameter list must be of a type that is compatible with the type of the corresponding formal parameters;
- all actual parameters shall be bound at the time of invocation of the test suite operation, test step, constraint or default;
- all actual parameters must resolve to specific values.

Call-By-Reference

The TTCN uses *textual substitution* to define the passing of actual parameters in test steps and defaults. An alternative, and more intuitive, way of describing parameter passing for test steps and defaults and yet retain TTCN semantics is to describe the mechanism in terms of *call-by-reference*, in which the called routine (test step or default) has access to the *original argument*, not a local copy. All operations that effect that argument have the same effect on the original.

Call-By-Value

The TTCN standard states that neither *user defined operations* nor *constraints* may change the values of any actual parameters that are passed to them, *i.e.* they shall have no side-effects. Thus, for user defined operations and constraints it is more suitable to describe the parameter passing mechanism in terms of *call-by-value* in which the called routine (user defined operations or constraint) works on a local copy of the argument. The original argument is not affected by the routine.

Test Case Selection

A test suite contains many hundreds, perhaps thousands, of test cases. In most cases of testing it will only be necessary to choose and run a selection of tests taken from the test suite. This choosing process is called *test case selection*. Depending on values and answers obtained from the PICS and PIXIT only a subset of the entire test suite need be executed.

Selection Expressions

TTCN allows each test case to be associate with a *selection expression*. These expressions are predicates that will evaluate to TRUE or FALSE depending on the answers given to the relevant PICS and PIXIT questions. If no selection predicate is given then the test will *always* be selected.

The predicates are defined in the *Test Case Selection Expression Definitions* table, and references are made to them from the *Test Case Index*.

Groups of test cases may be selected in a similar manner by making references to selection expressions from the *Test Suite Structure* table.

Structure of a TTCN Test Suite

Each TTCN object has a specific position in the hierarchy of the test suite.

Parts of a Test Suite

The different test suite components may only appear in a specific order. A TTCN test suite consists of four *parts*:

- Overview
- Declarations
- Constraints
- Behaviour

Each part contains a number of TTCN tables. The order in which the tables appear is shown in the following list. Each bulleted item in this list represents a TTCN table. The tables that have number subscripts are tables for *single* TTCN objects, *e.g.* PDUs and test cases. The tables that do not have a subscript are *multiple* TTCN object tables, *e.g.* simple type definitions or test suite variables.

Some tables may be displayed in a *compact* format. Tables printed in italic font are defined in the TTCN extensions.

Suite Overview Part

The test suite overview consists of four tables:

- Test Suite Structure
- Test Case Index
- Test Step Index
- Default Index.

Declarations Part

The declarations part is concerned both with the *definition* of new (*i.e.* not predefined) data types and operations and the *declaration* of all the test suite components.

- Test Component Declarations
- Test Component Configuration Declarations
- Simple Type Definitions
- Structured Type Definition₁

Structure of a TTCN Test Suite

- :
- ASN.1 Type Definition₁
- :
- ASN.1 Type Definitions By Reference
- Test Suite Operation Definition₁
- :
- Test Suite Parameter Declarations
- Test Case Selection Expression Definitions
- Test Suite Constant Declarations
- Test Suite Variable Declarations
- Test Case Variable Declarations
- PCO Declarations
- CP Declarations
- Timer Declarations
- ASP Type Definition₁
- :
- ASN.1 ASP Type Definition₁
- :
- ASN.1 ASP Type Definitions By Reference
- PDU Type Definition₁
- :
- ASN.1 PDU Type Definition₁
- :
- ASN.1 PDU Type Definitions By Reference
- TTCN CM Type Definition₁
- :
- ASN.1 CM Type Definition₁
- :
- Alias Declarations

Constraints Part

The constraints part contains the tables for all the ASP, PDU, structure and CM constraints. Both in the tabular form and the ASN.1.

- ASP Constraint Declaration₁
- :

Note:

ASP Constraints may displayed in a compact format if wished.

- ASN.1 ASP Constraint Declaration₁
- :

Note:

ASN.1 ASP Constraints may displayed in a compact format if wished.

- PDU Constraint Declaration₁
- :

Note:

PDU Constraints may displayed in a compact format if wished.

- ASN.1 PDU Constraint Declaration₁
- :

Note:

ASN.1 PDUConstraints may displayed in a compact format if wished.

- Structured Type Constraint Declaration₁
- :

Note:

Structured Type Constraints may displayed in a compact format if wished.

- ASN.1 Type Constraint Declaration₁
- :

Note:

ASN.1 Type Constraints may displayed in a compact format if wished.

- CM Constraint Declaration₁
- :
- ASN.1 CM Constraint Declaration₁
- :

Dynamic Part

The dynamic part contains all the test cases, all the test steps in the test step library and the all the defaults in the default library.

- Test Case Dynamic Behaviour₁
- :

Note:

Test groups, *i.e.* the test suite structure, are not represented here.
Test cases may displayed in a compact format if wished.

- Test Step Dynamic Behaviour₁
- :

Note:

Test step groups are not represented here.

- Default Dynamic Behaviour₁
- :

Note:

Default groups are not represented here.

Distributed Development

In the TTCN Suite, the implemented version of TTCN does support modularization of a TTCN document. This support is conveniently used to concurrently produce multiple documents with some definitions in common or to cooperatively produce one large TTCN document.

The mechanism implemented assumes that each user have a private target directory and all collaborating users having the same source files.

The following figures depict how this is accomplished. In the figures *TS* denotes a test suite and *M* denotes a TTCN module. Numbers are used to distinguish separate documents.

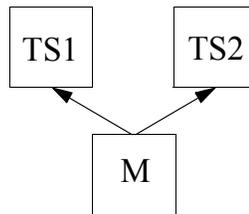


Figure 75: Two Modular Test Suites using a common Module

[Figure 75](#) depicts the case where two test suites are developed in parallel by two different users with both test suites referencing objects defined in a common module.

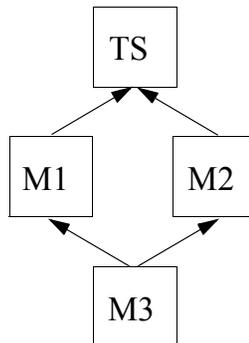


Figure 76: A large Modular Test Suite

[Figure 76](#) depicts the case where a large test suite (TS) is developed in parallel by two different users, each responsible for one module of the test suite (M1 and M2), with both modules referencing objects defined in a common module (M3).

In the case where more than one user is to concurrently develop different parts of a large test suite, here is a simple way to distribute the document files in the file system.

1. First create or select a suitable readable and writable directory accessible to all users. This directory will be used to store all files common to all users and so need to be accessible to all users with the same path.
2. Next create a template system file in this directory containing references to all documents (all document files referenced could conveniently be present in this directory), creating empty documents for those documents that will be produced later, set the directory representation to the absolute form, save everything, and finally make this template system file unwritable to protect it from inadvertent modifications.

At this point it may be wise to assign ownership of the documents to the users that are responsible for them and inhibit write access for others, and also to remove superfluous files (e.g. all TTCN files that have an extra hash-sign, '#', prepended to the file name).

3. Finally inform all users that they should follow these steps when they start their work:
 - Select or create a personal target directory, preferably on a local disk for optimum speed
 - Copy the template system file to it (i.e. create a personal copy)

The user may now change the directory representation back to the relative form if so wished.

The Complete Case Study

Suite Overview Part

Test Suite Structure			
Suite Name : TTCN_TUTORIAL			
Standards Ref : ISO/IEC xxxx			
PICS Ref : ISO/IEC aaaa			
PIXIT Ref : ISO/IEC bbbb			
Test Method(s) : Distributed single layer (DSE)			
Comments :			
Test Group Reference	Selection Ref	Test Group Objective	Page Nr
SINGLE/		Tests run over single connection	33
SINGLE/DATA/			33
MULT/		Tests run over multiple connections	34
MULT/DATA/			34
Detailed Comments :			

Figure 77: Test suite structure

Test Step Index			
Test Step Group Reference	Test Step Id	Description	Page Nr
TEST_STEP_LIB/LOWER/	LTS		35
TEST_STEP_LIB/LOWER/	ESTABLISH_CONNECT		35
TEST_STEP_LIB/LOWER/	LT_DATA_TRANSFER		35
TEST_STEP_LIB/LOWER/	CLOSE_CONNECTION		35
TEST_STEP_LIB/UPPER/	UTS		36
TEST_STEP_LIB/UPPER/	ACCEPT_CONNECTION		36
TEST_STEP_LIB/UPPER/	UT_DATA_TRANSFER		37
Detailed Comments :			

Figure 78: Test step index

Default Index			
Default Group Reference	Default Id	Description	Page Nr
DEFAULT_LIB/	LT_DEFAULT		38
DEFAULT_LIB/	UT_DEFAULT		38
DEFAULT_LIB/	T_DEFAULT		38
Detailed Comments :			

Figure 79: Default index

Declarations Part

Simple Type Definitions			
Type Name	Type Definition	Type Encoding	Comments
RESULT_TYPE	R_Type		
Detailed Comments :			

Figure 80: Simple type definitions

Structured Type Definition			
Type Name : VARIABLE_PART			
Encoding Variation :			
Comments : This is the type definition of the variable part of the GR_PDU and the CC_PDU.			
Element Name	Type Definition	Field Encoding	Comments
paramA_id	BITSTRING [2]		Parameter identifier.
paramA	OCTETSTRING [2..4]		Optional parameter A.
paramB_id	BITSTRING [2]		Parameter identifier.
paramB	BOOLEAN		Optional parameter B.
Detailed Comments :			

Figure 81: Definition of VARIABLE_PART

Test Suite Operation Definition	
Operation Name : INC (i:INTEGER)	
Result Type : INTEGER	
Comments : The INCREMENT operation.	
Description	
<pre>int INC(i) int temp; { return (temp+1); /*return the incremented value of i. Note that i itself is not changed */ }</pre>	
Detailed Comments :	

Figure 82: Definition of INC operation

Test Suite Parameter Declarations			
Parameter Name	Type	PICS/PIXIT Ref	Comments
it_address	IA5String	PIXIT question xx	Lower tester address
ut_address	IA5String	PIXIT question yy	upper tester address
Detailed Comments :			

Figure 83: Test suite parameter declarations

Test Suite Constant Declarations			
Constant Name	Type	Value	Comments
max	INTEGER	5	
data_string	IA5String	'This is a string'	
Detailed Comments :			

Figure 84: Test suite constant declarations

Test Case Variable Declarations			
Variable Name	Type	Value	Comments
count	INTEGER	0	This test case variable is used to count the number of PDUs sent and received.
Detailed Comments :			

Figure 85: Test case variable declarations

PCO Type Declarations		
PCO Type	Role	Comments
N_SAP	LT	
X_SAP	UT	
Detailed Comments :		

Figure 86: PCO type declarations

PCO Declarations			
PCO Name	PCO Type	Role	Comments
L1	N_SAP	LT	N-service access points at the lower tester
L2	N_SAP	LT	
U1	X_SAP	UT	X-service access points at the upper tester
U2	X_SAP	UT	
Detailed Comments :			

Figure 87: PCO declarations

The Complete Case Study

Coordination Point Declarations	
CP Name	Comments
CP1	Coordination between the MTC and the PTCs of teh lower tester.
CP2	
Detailed Comments :	

Figure 88: Coordination point declarations

Test Component Declarations				
Component Name	Component Role	Nr POCs	Nr CPs	Comments
MASTER_LOWER_TESTER	MTC	0	2	Main Test Component
LOWER_TESTER1	PTC	1	1	Parallel Test Component
LOWER_TESTER2	PTC	1	1	Parallel Test Component
UPPER_TESTER1	PTC	1	0	Parallel Test Component
UPPER_TESTER2	PTC	1	0	Parallel Test Component
Detailed Comments :				

Figure 89: Test component declarations

Test Components Configuration Declaration			
Configuration Name : SINGLE_PARTY			
Comments : Configuration to test a single connection.			
Components Used	POCs Used	CPs Used	Comments
MASTER_LOWER_TESTER		CP1	MTC
LOWER_TESTER1	L1	CP1	Lower PTC
LOWER_TESTER2	U1		Upper PTC
Detailed Comments :			

Figure 90: Declaration of SINGLE_PARTY

Test Components Configuration Declaration			
Configuration Name : MULTI_PARTY			
Comments : Configuration to test two simultaneous connections.			
Components Used	PCOs Used	CPs Used	Comments
MASTER_LOWER_TESTER		CP1, CP2	MITC
LOWER_TESTER1	L1	CP1	Lower PTC
LOWER_TESTER2	L2	CP2	Upper PTC
UPPER_TESTER1	U1		Lower PTC
UPPER_TESTER2	U2		Upper PTC
Detailed Comments :			

Figure 91: Declaration of MULTI_PARTY

ASP Type Definition		
ASP Name : N_DATArequest		
PCO Type : N_SAP		
Comments : This is the type definition of the N_DATArequest ASP. It has a single parameter used to carry user data.		
Parameter Name	Parameter Type	Comments
user_data	PDU	The PDU metatype is used to indicate that utgoing (i.e. from LT) PDUs are embedded in this Network ASP
Detailed Comments :		

Figure 92: Definition of IN_DATArequest

ASP Type Definition		
ASP Name : N_DATAindication		
PCO Type : N_SAP		
Comments : This is the type definition of the N_DATAindication ASP. It has a single parameter used to carry user data.		
Parameter Name	Parameter Type	Comments
user_data	PDU	The PDU metatype is used to indicate that utgoing (i.e. from LT) PDUs are embedded in this Network ASP
Detailed Comments :		

Figure 93: Definition of IN_DATAindication

The Complete Case Study

ASP Type Definition		
ASP Name : X_CONNECTIndication		
PCO Type : N_SAP		
Comments : This is the type definition of the X_CONNECTIndication ASP. These ASPs are issued by IUT to the upper tester.		
Parameter Name	Parameter Type	Comments
called_address	IA5String	
calling_address	IA5String	
user_data	IA5String [0 .. 32]	
Detailed Comments :		

Figure 94: Definition of X_CONNECTIndication

ASP Type Definition		
ASP Name : X_CONNECTResponse		
PCO Type : N_SAP		
Comments : This is the type definition of the X_CONNECTResponse ASP. These ASPs are issued by IUT to the upper tester.		
Parameter Name	Parameter Type	Comments
called_address	IA5String	
calling_address	IA5String	
user_data	IA5String [0 .. 32]	
Detailed Comments :		

Figure 95: Definition of X_CONNECTResponse

ASP Type Definition		
ASP Name : X_DATArequest		
PCO Type : X_SAP		
Comments : This is the type definition of the X_DATArequest ASP. These ASPs are issued by the upper tester to the IUT.		
Parameter Name	Parameter Type	Comments
called_address	IA5String	
Detailed Comments :		

Figure 96: Definition of X_DATArequest

ASP Type Definition		
ASP Name : X_DATAIndication		
PCO Type : X_SAP		
Comments : This is the type definition of the X_DATAIndication ASP. These ASPs are issued by the upper tester to the IUT.		
Parameter Name	Parameter Type	Comments
called_address	IA5String	
Detailed Comments :		

Figure 97: Definition of X_DATAIndication

PDU Type Definition			
PDU Name : CR_PDU			
PCO Type : N_SAP			
Encoding Rule Name :			
Encoding Variation :			
Comments : This is the type definition of the CR_PDU			
Field Name	Field Type	Field Encoding	Comments
type	OCTETSTRING [1]		
dst_ref	BITSTRING [4]		
src_ref	BITSTRING [4]		
variable_part	VARIABLE_PART		Reference to a structured type.
user_data	IA5String [0 .. 32]		
Detailed Comments :			

Figure 98: Definition of CR_PDU

The Complete Case Study

PDU Type Definition			
PDU Name : CC_PDU			
PCO Type : N_SAP			
Encoding Rule Name :			
Encoding Variation :			
Comments : This is the type definition of the CC_PDU			
Field Name	Field Type	Field Encoding	Comments
type	OCTETSTRING [1]		
dst_ref	BITSTRING [4]		
src_ref	BITSTRING [4]		
variable_part	VARIABLE_PART		Reference to a structured type.
user_data	IA5String [0..32]		
Detailed Comments :			

Figure 99: Definition of CC_PDU

PDU Type Definition			
PDU Name : DT_PDU			
PCO Type : N_SAP			
Encoding Rule Name :			
Encoding Variation :			
Comments : This is the type definition of the DT_PDU			
Field Name	Field Type	Field Encoding	Comments
type	OCTETSTRING [1]		
user_data	IA5String		
Detailed Comments :			

Figure 100: Definition of DT_PDU

CM Type Definition		
CM Name : PTC_RESULT		
Comments : Coordination message to transfer preliminary result from teh lower tester PTCs to the IMTC.		
Parameter Name	Parameter Type	Comments
result	RESULT_TYPE	User defined type.
Detailed Comments :		

Figure 101: Definition of PTC_RESULT

Alias Definitions		
Alias Name	Expansion	Comments
CR	N_DATArequest	Alias for the N_DATArequest service primitive used to carry a CR_PDU
DATAout	N_DATArequest	Alias for the N_DATArequest service primitive used to carry an outgoing DT_PDU
CC	N_DATAindication	Alias for the N_DATAindication service used to carry a CC_PDU
DATAin	N_DATAindication	Alias for the N_DATArequest service primitive used to carry an incoming DT_PDU
DR	N_DATArequest	Alias for the N_DATArequest service primitive used carry an outgoing DR_PDU
Detailed Comments :		

Figure 102: Alias definitions

Constraints Part

Structured Type Constraint Declaration			
Constraint Name : variable_part_CR1			
Structured Type : VARIABLE_PART			
Derivation Path :			
Encoding Variation :			
Comments : A constraint on the structure type VARIABLE_PART for teh CR_PDU			
Element Name	Element Value	Element Encoding	Comments
paramA_id	--		Omit this field
paramA	--		Omit this field
paramB_id	01'B		
paramB	TRUE		
Detailed Comments :			

Figure 103: Declaration of variable_part_CR1

Structured Type Constraint Declaration			
Constraint Name : variable_part_CR2			
Structured Type : VARIABLE_PART			
Derivation Path :			
Encoding Variation :			
Comments : A constraint on the structure type VARIABLE_PART for teh CR_PDU			
Element Name	Element Value	Element Encoding	Comments
paramA_id	00'B IF_PRESENT		Accept if present.
paramA	*		Any value, or none.
paramB_id	01'B IF_PRESENT		Accept if present.
paramB	*		Any value, or none
Detailed Comments :			

Figure 104: Declaration of variable_part_CR2

ASP Constraint Declaration		
Constraint Name : NDr(any_pdu:PDU)		
ASP Type : N_DATArequest		
Derivation Path :		
Comments : A constraint on the N_DATArequest ASP.		
Parameter Name	Parameter Value	Comments
user_data	any_pdu	The actual PDU that is carried in the ASP is dynamically chained from the constraints reference.
Detailed Comments :		

Figure 105: Declaration of NDr

ASP Constraint Declaration		
Constraint Name : NDi (any_pdu:PDU)		
ASP Type : N_DATAindication		
Derivation Path :		
Comments : This is the type definition of the N_DATAindication ASP. It has a single parameter used to carry user data.		
Parameter Name	Parameter Value	Comments
user_data	any_pdu	The actual PDU that is carried in the ASP is dynamically chained from the constraints reference.
Detailed Comments :		

Figure 106: Declaration of NDi

ASP Constraint Declaration		
Constraint Name : CONInd		
ASP Type : X_CONNECTindication		
Derivation Path :		
Comments : A constraint on the X_CONNECTindication ASP.		
Parameter Name	Parameter Value	Comments
called_address	ut_address	From test suite parameters.
calling_address	it_address	From test suite parameters.
user_data	*	Accept any value, or none
Detailed Comments :		

Figure 107: Declaration of CONInd

The Complete Case Study

ASP Constraint Declaration		
Constraint Name : CONrsp		
ASP Type : X_CONNECTresponse		
Derivation Path :		
Comments : A constraint on the X_CONNECTresponse ASP.		
Parameter Name	Parameter Value	Comments
called_address	ut_address	From test suite parameters.
calling_address	rl_address	From test suite parameters.
user_data	^	Omit optional user data.
Detailed Comments :		

Figure 108: Declaration of CONrsp

ASP Constraint Declaration		
Constraint Name : DATreq		
ASP Type : X_DATArequest		
Derivation Path :		
Comments : A constraint on the X_DATArequest ASP.		
Parameter Name	Parameter Value	Comments
called_address	data_string	
Detailed Comments :		

Figure 109: Declaration of DATreq

ASP Constraint Declaration		
Constraint Name : DATind		
ASP Type : X_DATAindication		
Derivation Path :		
Comments : A constraint on the X_DATAindication ASP		
Parameter Name	Parameter Value	Comments
called_address	data_string	
Detailed Comments :		

Figure 110: Declaration of DATind

PDU Constraint Declaration			
Constraint Name : CR1			
PDU Type : CR_PDU			
Derivation Path :			
Encoding Rule Name :			
Encoding Variation :			
Comments : A constraint on the CC_PDU			
Field Name	Field Value	Field Encoding	Comments
type	'F1O		
dst_ref	'0001'B		
src_ref	'0001'B		
variable_part	variable_part_CR1		Reference to a structured constraint
user_data	'Hello'		

Figure 111: Declaration of CR1

PDU Constraint Declaration			
Constraint Name : CC1			
PDU Type : CC_PDU			
Derivation Path :			
Encoding Rule Name :			
Encoding Variation :			
Comments : A constraint on the CC_PDU			
Field Name	Field Value	Field Encoding	Comments
type	'F2O		
dst_ref	'0001'B		
src_ref	'0001'B		
variable_part	variable_part_CR2		Reference to a structured constraint
user_data	^		

Figure 112: Declaration of CC1

The Complete Case Study

PDU Constraint Declaration			
Constraint Name : DT1 (actual_data:IA5String)			
PDU Type : DT_PDU			
Derivation Path :			
Encoding Rule Name :			
Encoding Variation :			
Comments : This is the type definition of the DT_PDU			
Field Name	Field Value	Field Encoding	Comments
type	'F3O		
user_data	actual_data		The actual data is passed as a parameter to the constraint.
Detailed Comments :			

Figure 113: Declaration of DT1

CM Constraint Declaration		
Constraint Name : PTC_RES (actual_result:RESULT_TYPE)		
CM Type : PTC_RESULT		
Derivation Path :		
Comments : A constraint on the PTC_RESULT coordination message.		
Parameter Name	Parameter Value	Comments
result	actual_result	The actual result is passed as a parameter to the constraint.
Detailed Comments :		

Figure 114: Declaration of PTC_RES

Dynamic Part

Test Case Dynamic Behaviour

Test Case Name : SP_DATA_TRANSFER
 Group : SINGLE/DATA/
 Purpose : IUT shall receive and send a data within time limit, a given number of times over a single X-connection
 Configuration : SINGLE_PARTY
 Default : T_DEFAULT
 Comments : This test case creates the other PTCs in the configuration necessary for a single-connection configuration.
 Selection Ref :
 Description : Data transfer single connection

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		CREATE(LOWER_TESTER1:LTS (L1, CP1), UPPER_TESTER1:UTS (U1))			1)
2		CP1?PTC_RESULT	PTC_RES (pass)	PASS	2), 3)
3		CP1?PTC_RESULT	PTC_RES (fail)	FAIL	2), 3)

Detailed Comments : 1) The CREATE command binds specific test steps to the relevant PTCs in the Test Component Declarations table. It also implicitly starts execution of each test step (i.e. one instance of LT and one instance of UT)
 2) The preliminary results from the lower tester PTCs are picked up here.
 3) Final verdict

Figure 115: Definition of SP_DATA_TRANSFER

The Complete Case Study

MP_DATA_TRANSFER in TTCN_TUTORIAL

File Edit Data Dictionary Show Tools SDTLink Help

Test Case Dynamic Behaviour

Test Case Name : MP_DATA_TRANSFER

Group : MULT/DATA/

Purpose : IUT shall receive and send a data within time limit, a given number of times over two simultaneous X-connections.

Configuration : MULTI_PARTY

Default : T_DEFAULT

Comments : This test case creates the otehr PTCs in the configuration necessary for a two-connection configuration.

Selection Ref :

Description : Data transfer multi-connection

Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		CREATE(LOWER_TESTER1:LTS (L1, CP1), LOWER_TESTER2:LTS (L2, CP2), UPPER_TESTER1:UTS (U1), UPPER_TESTER2:UTS (U2))			1)
2		CP1?PTC_RESULT	PTC_RES (pass)		2)
3		CP2?PTC_RESULT	PTC_RES (pass)	PASS	2), 3)
4		CP2?PTC_RESULT	PTC_RES (fail)	FAIL	2), 3)
5		CP1?PTC_RESULT	PTC_RES (fail)	FAIL	2)
6		CP2?PTC_RESULT	PTC_RES (pass)	FAIL	2), 3)
7		CP2?PTC_RESULT	PTC_RES (fail)	FAIL	2), 3)

Detailed Comments : 1) The CREATE command binds specific test steps to the relevant PTCs in the Test Component Declarations table. It also implicitly starts execution of each test step (i.e. two instances of LT and one instance of UT)

2) The preliminary results from the lower tester PTCs are picked up here.

3) Final verdict

Figure 116: Definition of MP_DATA_TRANSFER

Test Step Dynamic Behaviour					
Test Step Name : LTS(L:N_SAP; GPP:GP)					
Group : TEST_STEP_LIB/LOWER/					
Objective : IUT shall receive and send a data within time limit, a given number of times.					
Default :					
Comments :					
Description :					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		+ESTABLISH_CONNECTION(L)			
2		+LT_DATA_TRANSFER(L,GP)			
3		+CLOSE_CONNECTION(L)			
Detailed Comments :					

Figure 117: Definition of LTS

Test Step Dynamic Behaviour					
Test Step Name : ESTABLISH_CONNECTION (L:N_SAP)					
Group : TEST_STEP_LIB/LOWER/					
Objective : To establish a connection.					
Default : LT_DEFAULT(L)					
Comments : This is a preamble test step used by the lower tester(s) to set up a connection between the lower tester(s) and the upper tester(s). For teh sake of simplicity we shall assume that the connection cannot be refused.					
Description :					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		LICR	NDn(CR1)		
2		L?CG	NDi(CG1)		
Detailed Comments :					

Figure 118: Definition of ESTABLISH_CONNECTION

The Complete Case Study

Test Step Dynamic Behaviour					
Test Step Name : LT_DATA_TRANSFER (L:N_SAP; GPP:GP)					
Group : TEST_STEP_LIB/LOWER/					
Objective : Test data transfer					
Default : LT_DEFAULT(L)					
Comments : This test step implements the test body of our example test case on the lower tester side.					
Description :					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1	LAB	L!DATAout (count=INC(count))	NDr(DT1(data_string))		
2		[count <= max] START Timer			
3		L?DATAin	NDi(DT1(data_string))	(PASS)	
4		->LAB			
5		?TIMEOUT Timer		(FAIL)	
6		GPP!PTC_RESULT	PTC_RES(fail)		
7		GPP!PTC_RESULT	PTC_RES(pass)		
Detailed Comments :					

Figure 119: Definition of LT_DATA_TRANSFER

Test Step Dynamic Behaviour					
Test Step Name : CLOSE_CONNECTION(L:N_SAP)					
Group : TEST_STEP_LIB/LOWER/					
Objective : Close teh connection to the IUT					
Default :					
Comments : This is a postamble test step that closes a connection between the lower tester(s) and the upper tester(s).					
Description :					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		L!DR	NDr(CR1)		
Detailed Comments :					

Figure 120: Definition of CLOSE_CONNECTION

Test Step Dynamic Behaviour					
Test Step Name : UTS(U:X_SAP)					
Group : TEST_STEP_LIB/UPPER/					
Objective : Accept connection and receive/send DATA a certain number of times					
Default :					
Comments :					
Description :					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		+ACCEPT_CONNECTION(U)			
2		+UT_DATA_TRANSFER(U)			
Detailed Comments :					

Figure 121: Definition of UTS

Test Step Dynamic Behaviour					
Test Step Name : ACCEPT_CONNECTION(U:X_SAP)					
Group : TEST_STEP_LIB/UPPER/					
Objective : Accept an X_CONNECTIndication from the lower tester					
Default : UT_DEFAULT(U)					
Comments : This is a preamble test step used by the upper tester(s) to accept an incoming connection request from the lower tester(s).					
Description :					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		U?X_CONNECTIndication	CONInd		
2		UIX_CONNECTResponse	CONrsp		
Detailed Comments :					

Figure 122: Definition of ACCEPT_CONNECTION

The Complete Case Study

Test Step Dynamic Behaviour					
Test Step Name : UT_DATA_TRANSFER(U:X_SAP)					
Group : TEST_STEP_LIB/UPPER/					
Objective : Respond to incoming data.					
Default : UT_DEFAULT(U)					
Comments : This test step implements the test body of our example test case on the upper tester side.					
Description :					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1	LAB	U?N_DATAindication (count := INC(count))	NDi(DT1 (data_string))		
2		U!N_DATArequest (count <= max)	NDr(DT1 (data_string))		
3		->LAB			
Detailed Comments :					

Figure 123: Definition of UT_DATA_TRANSFER

Default Dynamic Behaviour					
Default Name : UT_DEFAULT(U:X_SAP)					
Group : DEFAULT_LIB/					
Objective : General catch—all for the upper tester.					
Comments :					
Description :					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		U?OTHERWISE		(FAIL)	
Detailed Comments :					

Figure 124: Definition of UT_DEFAULT

Default Dynamic Behaviour					
Default Name : UT_DEFAULT(U:X_SAP)					
Group : DEFAULT_LIB/					
Objective : General catch—all for the upper tester.					
Comments :					
Description :					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		U?OTHERWISE		(FAIL)	
Detailed Comments :					

Figure 125: Definition of UT_DEFAULT

Default Dynamic Behaviour					
Default Name : T_DEFAULT					
Group : DEFAULT_LIB/					
Objective : General catch-all					
Comments :					
Description :					
Nr	Label	Behaviour Description	Constraints Ref	Verdict	Comments
1		L1?OTHERWISE		FAIL	
2		L2?OTHERWISE		FAIL	
3		U1?OTHERWISE		FAIL	
4		U2?OTHERWISE		FAIL	
Detailed Comments :					

Figure 126: Definition of T_DEFAULT

D

DefCon Utility: [59](#)

