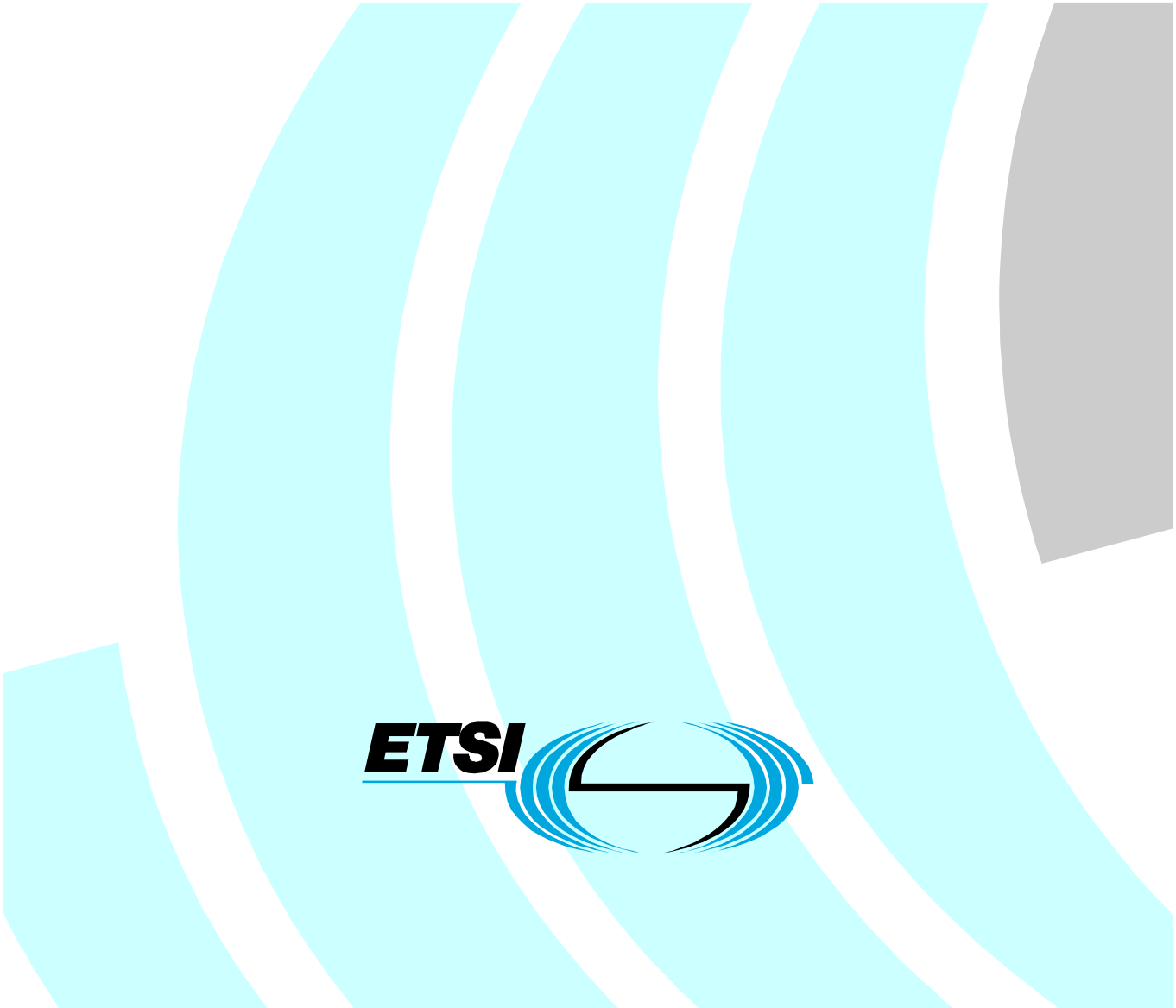
ETSI TS 102 351 V2.1.1 (2005-08)

Technical Specification

Methods for Testing and Specification (MTS); Internet Protocol Testing (IPT); IPv6 Testing: Methodology and Framework



Reference
RTS/MTS-IPT-001-IPV6-Fwk

Keywords
IP, interoperability, methodology, testing, TTCN

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://webapp.etsi.org/IPR/home.asp).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This Technical Specification (TS) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

1 Scope

The present document gives guidelines for the use of a common method for developing test specifications for IPv6. This method is applicable to all IPv6 categories including the core specification, mobility, security and transitioning to IPv6 from IPv4.

The underlying method is based on the methodologies specified in ISO/IEC 9646-1 [4] for conformance tests and TS 102 237-1 [1] for interoperability tests. It provides guidance on the development and use of the following key elements of the method:

- a Requirements Catalogue (RC);
- a Test Suite Structure (TSS) and Test Purposes (TP);
- Test Descriptions (TD) - interoperability;
- a TTCN-3 library of data types and values, templates and functions;
- an Abstract Test Suite (ATS) - conformance.

The methodology also offers guidance on naming conventions and other style-related issues.

Although the present document has been developed primarily for use in the testing of IPv6 standards, it could equally be used in other areas of protocol test specification.

2 References

The following documents contain provisions which, through reference in this text, constitute provisions of the present document.

- References are either specific (identified by date of publication and/or edition number or version number) or non-specific.
- For a specific reference, subsequent revisions do not apply.
- For a non-specific reference, the latest version applies.

Referenced documents which are not found to be publicly available in the expected location might be found at http://docbox.etsi.org/Reference.

[1] ETSI TS 102 237-1 (V4.1.1): "Telecommunications and Internet Protocol Harmonization Over Networks (TIPHON) Release 4; Interoperability test methods and approaches; Part 1: Generic approach to interoperability testing".

[2] ETSI EG 202 106 (V2.1.1): "Methods for Testing and Specification (MTS); Guidelines for the use of formal SDL as a descriptive tool".

[3] ETSI ES 201 873-6 (V3.1.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI)".

[4] ISO/IEC 9646-1 (1992): "Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework - Part 1: General concepts".

[5] IETF RFC 1035 (1997): "Domain names - implementation and specification".

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

behavioural function: TTCN-3 function which specifies actions which result in the sending of messages to one or more observed interface

computational function: TTCN-3 function which specifies actions which modifies data values but does not result in the sending of messages to one or more observed interface

Equipment Under Test (EUT): grouping of one or more devices which has not been previously shown to interoperate with previously Qualified Equipment (QE) TS 102 237-1 [1]

Qualified Equipment (QE): grouping of one or more devices that has been shown, by rigorous and well-defined testing, to interoperate with other equipment TS 102 237-1 [1]

NOTE: Once an EUT has been successfully tested against a QE, it may be considered to be a QE, itself.

3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

3GPP	3rd Generation mobile Partnership Project
API	Application Programming Interface
ATS	Abstract Test Suite
EUT	Equipment Under Test
IETF	Internet Engineering Task Force
IFS	Interoperable Functions Statement
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IUT	Implementation Under Test
MTC	Main Test Component
NGN	Next Generation Network
PICS	Protocol Implementation Conformance Statement
PTC	Parallel Test Component
QE	Qualified Equipment
RC	Requirements Catalogue
RFC	Request For Comments (IETF terminology for a draft standard)
RQ	Requirement
SCOP	Simple Control and Observation Protocol
SUT	System Under Test
TISPAN	ETSI technical body with responsibility for NGN standardization
TC	Test Case
TCI	TTCN-3 Control Interface
TD	Test Description
TE	Test Equipment
TP	Test Purpose
TSS	Test Suite Structure
TTCN-3	Testing and Test Control Notation edition 3
UDP	User Datagram Protocol
UT	Upper Tester
UTC	Upper Tester Client
UTS	Upper Tester Server

4 The TTCN-3 Framework

ETSI test specifications are usually developed for a single base protocol standard or for a coherent set of standards. As such, it is possible to follow the methodology specified for conformance test development in ISO/IEC 9646-1 [4] without much difficulty. However, the requirements of IPv6 are distributed across a wide range of documents and an adaptation of the ISO/IEC 9646 approach to test development is necessary. Also, for readability, consistency and to ease reusability of TTCN-3 code it is necessary to apply some guidelines on the use of TTCN-3.

It is this approach that is referred to as the "TTCN-3 Framework".

As its name implies, the framework is oriented towards the production of Abstract Test Suites (ATS) in the Testing and Test Control Notation edition 3 (TTCN-3). The TTCN-3 Framework comprises:

- a documentation structure:
  - catalogue of requirements;
  - Test Suite Structure (TSS);
  - Test Purposes (TP):
    - conformance;
    - interoperability;
- Abstract Test Suite (ATS):
  - Test Cases (TC) in TTCN-3 for conformance tests;
  - Test Descriptions (TD) in tabulated English for interoperability tests;
- library of TTCN-3 building blocks:
  - data types and values;
  - templates;
  - general computational functions;
  - TP functions (see clause 7.1.3.3);
- a methodology linking the individual documentation, library and ATS elements together:
  - style guidelines and examples;
  - naming conventions;
  - guidelines on the use and extension of the TTCN-3 library;
  - a structured notation for TPs.

The TTCN-3 Framework, particularly the methodology, draws heavily on the tried and tested ISO/IEC 9646-1 [4] but modifies it to suit the particular case of IPv6 testing. It also incorporates guidelines on interoperability testing taken from TS 102 237-1 [1].

5 The IPv6 test development process

The process to be followed when developing IPv6 test specifications is shown in figure 1.

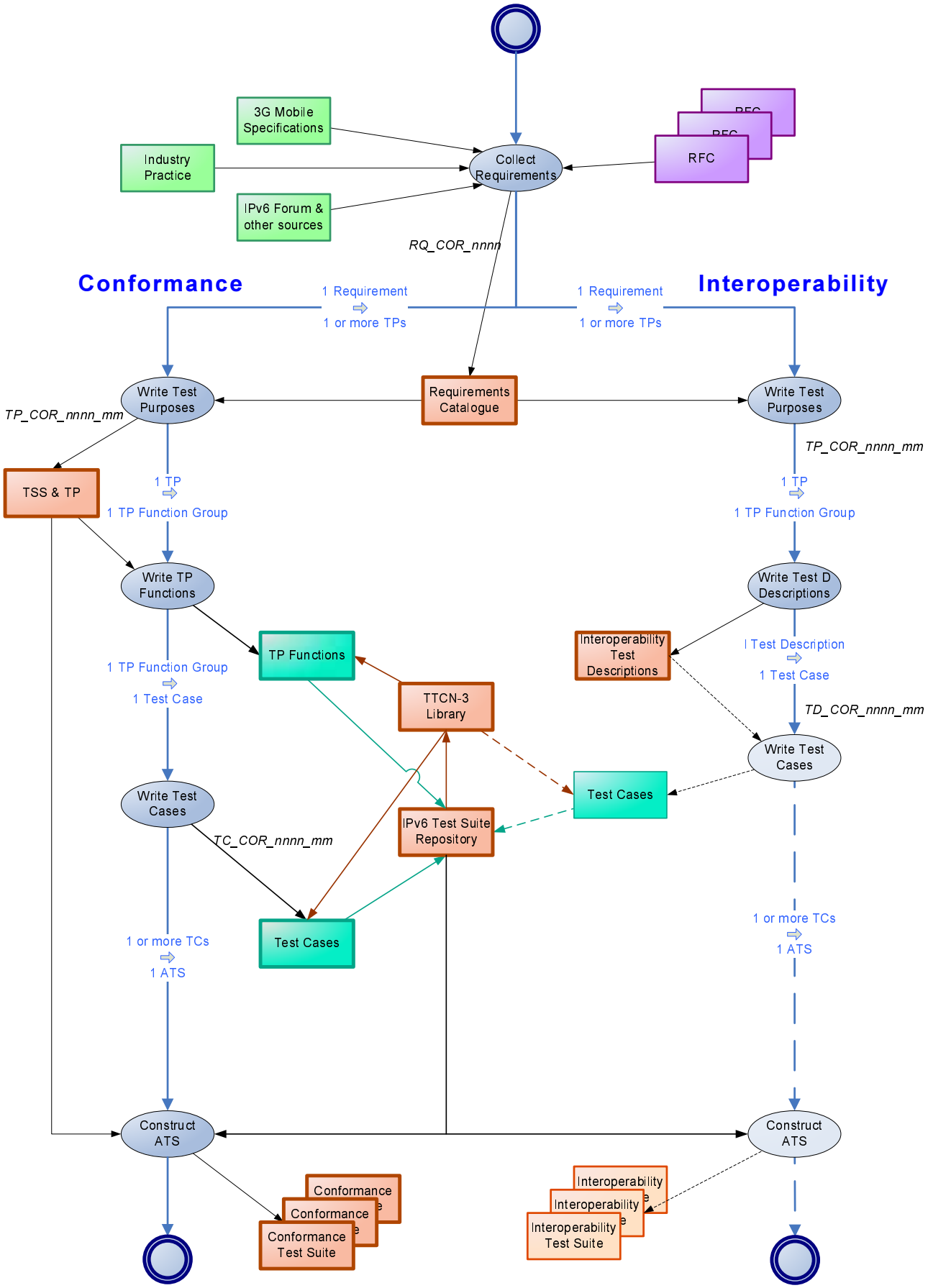


Figure 1: IPv6 test development process

The process begins with the analysis of the IETF RFCs related to IPv6 and a range of secondary inputs which include:

- current industry practice;
- existing test documentation from the IPv6 Forum and other established sources;
- specifications related to the use of IPv6 in 3rd Generation mobile networks.

The result of this analysis is the identification and classification of a full range of IPv6 requirements which is recorded in the Requirements Catalogue and used as the basis for both conformance and interoperability test specifications.

5.1 Conformance testing methodology

Conformance test specifications should be produced following the methodology described in ISO/IEC 9646-1 [4]. In summary, this methodology begins with the collation and categorization of the requirements to be tested into a tabular form which is normally referred to as the "Protocol Implementation Conformance Statement" (PICS). Each PICS relates to a specific protocol standard. As the requirements of IPv6 are distributed across a large number of documents, there would be very little benefit in producing a PICS for each document. Consequently, the IPv6 requirements will be collected together and categorized in a single document, the Requirements Catalogue.

For each requirement in the catalogue, one or more tests should be identified and classified into a number of groups which will provide a structure to the overall test suite (TSS). A brief Test Purpose (TP) should then be written for each identified test and this should make it clear what is to be tested but not how this should be done. Finally, a detailed Test Case (TC) is written for each TP. In the interests of test automation, TCs are usually combined into an Abstract Test Suite (ATS) using a specific testing language such as TTCN-3.

5.2 Interoperability testing methodology

For a certification (or branding or logo) scheme to be meaningful, it is necessary that interoperability testing is carried out in addition to conformance testing and that this is done in accordance with a comprehensive and structured suite of tests. In the context of the present document, it is this type of testing which is referred to as "Interoperability Testing". The purpose of interoperability testing is to prove that the end-to-end functionality between (at least) two communicating systems is as required by the standard(s) on which those systems are based. A methodology for developing such interoperability test specification is described in TS 102 237-1 [1] and this should be used as a guide when developing IPv6 test suites. This methodology is based extensively on ISO/IEC 9646-1 [4] but with some modifications to make it suitable for interoperability testing.

In TS 102 237-1 [1], the Interoperable Functions Statement (IFS) replaces the PICS and is a statement of which functions supported by the protocol have been implemented. However, in this framework these functions should be clearly identified in the Requirements Catalogue.

6 The Requirements Catalogue

The requirements which collectively specify and characterize IPv6 are taken from a wide range of specifications and other documentation. Building a coherent set of test specifications from these disperse requirements sources can be made simpler by gathering the requirements together into a single catalogue.

The Requirements Catalogue lists IPv6 requirements from the various sources and organizes them in a tree structure. Each node of the tree is an IPv6 function. These functions are either explicitly mentioned or implicit in the requirements source texts. Specific requirements are then associated to the relevant function node.

6.1 Entries in the Requirements Catalogue

Details of each requirement are entered in the Requirements Catalogue which is structured as a database. An example database is described in annex E. This annex also shows possible visualisation (presentation) of the data base.

For each requirement in the catalogue the following information should be present:

- the functional group to which the requirement belongs. Each functional group is a node in the requirements tree:
  - some functional groups may be created for structuring purposes; i.e. an "implicit" functional group. Such groups are not specifically mentioned in any of the sources but are derived during development of the catalogue for structuring and organizational purposes;
  - a functional group may not have any specific requirement associated with it. This indicates that the requirements can found in its descendents;
- a unique requirement identification number as defined in clause 6.2;
- the identification number(s) of the Test Purpose(s) written for this requirement, if any;
- the context of the requirement. This identifies the general conditions that are necessary for the requirement to exist. It may also be considered as a summary of the situation that leads to the specific requirement. It should not be confused with "preambles" specified for test cases;
- the requirement in text. This should be a direct quote from the source text. However, synthesis and simplification may be necessary in order to improve readability. However, in no event should the substance of the source's requirement be changed in transcribing it to the catalogue;
- reference(s) to the source of the requirement and the requirement type. There may be several source:type pairs for the same requirement. For example, a requirement may come from the RFCs and have a SHOULD type; but the same requirement may be in the IPv6 Logo criteria and have the MUST type. If so, there would be two pairs of references and type, one for the RFC(s) and another for the IPv6 Logo criteria. Each pair comprises the following:
  - reference(s) to the source document of the requirement (e.g. RFC(s), the IPv6 Logo test document or an ETSI 3GPP document) which should be precise and unambiguous. For example "RFC 2460, paragraph 3 ¶1" indicates Paragraph 1 of Section 3 of RFC 2460. A requirement may have several sources especially if it is specified in RFCs. The requirement reference is also automatically linked to the source document so that the source text can be easily located and read;
  - the type of requirement (MUST, SHALL, SHOULD, MAY) which is useful in determining whether a requirement is optional or mandatory:
    - some requirements may be "negative" requirements, for example "... MUST not do something ...". In such cases the requirement type should be indicated as MUST_NOT (SHALL_NOT, SHOULD_NOT, MAY_NOT);
    - the language for types may not always follow convention and may differ across different standards organizations. For example, "can", "ought", "will" and "could" may all be used in different sources to express a requirement. In such cases, the Requirement Type is implied from the text and marked by placing the type in square brackets. As a further example, "[SHOULD]" may be used for a requirement text that is specified in the source as "ought".

6.2 Naming IPv6 requirements

A unique name should be provided for each requirement in the catalogue. Each requirement name will begin with "RQ_" followed by three characters indicating which area of the IPv6 specification it refers to and a four-digit identifier, as follows:

- RQ_COR_nnnn IPv6 Core requirements (example: RQ_COR_1254).
- RQ_SEC_nnnn IPv6 Security requirements (example: RQ_SEC_0237).
- RQ_MOB_nnnn IPv6 Mobility requirements (example: RQ_MOB_1198).
- RQ_T46_nnnn IPv4 to IPv6 Transitioning requirements (example: RQ_T46_0471).

7 Developing test specifications

7.1 Conformance test specifications

7.1.1 Test configurations

For each test or group of tests specified in the IPv6 conformance test suites, a configuration should be defined to identify the IPv6 roles required for the test components and the communications paths between those components.

Figure 2 shows an example configuration for conformance testing.

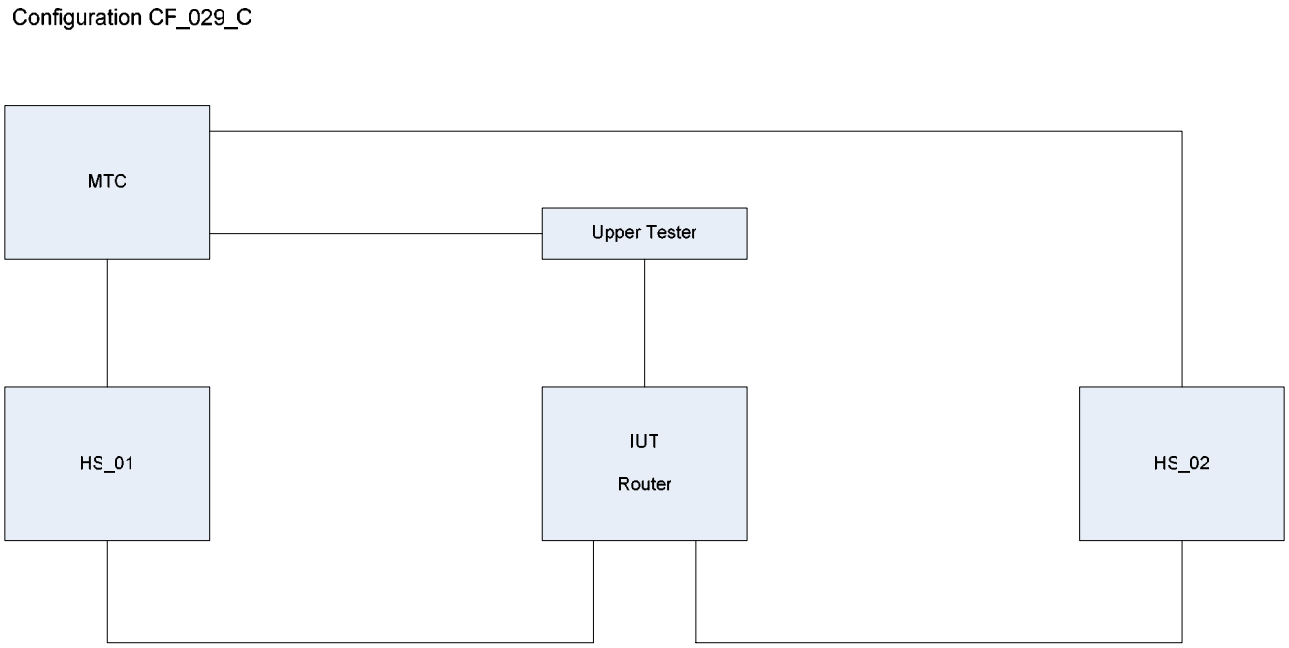


Figure 2: Example conformance testing configuration

7.1.1.1 Naming IPv6 conformance test configurations

Test configurations should be named so that they can be uniquely referenced in, for example, the TTCN-3 code or TP Language headers (7.1.2.2.2).

Configuration names should be of the form "CF_" followed by a three digit unique sequence number and the characters "_C", viz, "CF_023_C". This identifier should be included in any diagram associated with the configuration, as shown in figure 2.

7.1.1.2 Naming IPv6 test components

The components in each test configuration should be systematically and unambiguously identified. This naming is based on the role of each component, which include the following:

- HS Host;
- RT Router;
- ND Node;
- EN End Node.

Each role should be followed by a two-digit sequence number that uniquely identifies that component. This is necessary when more than one component plays the same role. The role and the sequence number should be separated by an underscore.

EXAMPLE: HS_01, HS_02, RT_01, RT_02.

7.1.2 Test Suite Structure and Test Purposes

7.1.2.1 Test Suite Structure

Test Suite Structure (TSS) groups should be chosen according to natural divisions in the base specification(s) and should follow the functionalities specified in the Requirements Catalogue. The architecture of the testing configuration should also be taken into account such that, for example, all test purposes explicitly requiring an Upper Tester may be collected into a single group. Other examples might be groupings of "Normal behaviour" and "Exceptional" behaviour.

7.1.2.1.1 Naming IPv6 test groups

TP groups have a short name (or identifier) and a longer, more readable title. The short name is derived from the longer title (i.e. it is a two or three letter abbreviation of the longer title). For example, if the long title is "Router", the short name should be: "RT". It is recommended that the title is followed by the short name in parentheses, for example: "Router (RT)"In the case of subgroups both the title and the short name should reflect the sub structuring, essentially making them path names. The group delimiter in the case of the title is "/". The delimiter in the case of the short name is: "_". As a further example, the group "Provide IPv6 Services (PS)" which is a sub group of the "Router (RT)" group, has the title:

- Router(RT)/Provide IPv6 Services(PS).

and the short name:

- RT_PS.

7.1.2.2 Test Purposes

A Test Purpose (TP) should be written for each potential test of an IPv6 requirement (as identified in the Requirements Catalogue) remembering that a requirement may need more than one TP to ensure that it is fully tested. As well as describing what is to be tested, the TP should identify the initial conditions to be established before testing can take place, the required status of the Implementation Under Test (IUT) from which testing can proceed and the criteria upon which verdicts can be assigned.

The contents of a TP should be limited to a description of what is to be tested rather than how that testing is to be carried out.

7.1.2.2.1 Naming IPv6 TPs

Each IPv6 requirement, as identified, in the Requirements Catalogue, will result in one or more TPs. Thus, RQs and TPs share a common numbering scheme but with a different prefix. For Test Purposes this prefix will, naturally, be "TP". This prefix will be followed by the four-digit sequence number taken from the requirement it corresponds to and a two digit sequence number to permit multiple TPs to be derived from a single requirement, thus:

- TP_COR_nnnn_mm IPv6 Core TPs.

EXAMPLE 1: TP_COR_0147_04.

- TP_SEC_nnnn_mm IPv6 Security TPs.

EXAMPLE 2: TP_SEC_0109_17.

- TP_MOB_nnnn_mm IPv6 Mobility TPs.

EXAMPLE 3: TP_MOB_0033_05.

- TP_T46_nnnn_mm IPv4 to IPv6 Transitioning TPs.

EXAMPLE 4: TP_T46_0006_32.

7.1.2.2.2 Using the TP Language

There is considerable benefit to be gained by having all Test Purposes written in a similar and consistent way. With this in mind, a simple, structured notation has been developed for the expression of TPs. This is described fully in annex A.

The benefits of using TPLan are:

- consistency in test purpose descriptions - less room for misinterpretation;
- simpler identification of preamble, test description and postamble;
- automatic test purpose syntax checking;
- a basis for a TP transfer format;
- possible TTCN-3 code stub generation;
- possibility to graphically or textually render TP descriptions for different users.

Examples of the use of the language to express conformance TPs are shown below.

EXAMPLE 1:

```
TP id   : TP_COR_0047_01
Summary : 'aligning PadN option'
RQ Ref  : RQ_COR_0047
Config  : CF_0_C
TC Ref  : TC_COR_0047_01
ensure that {
    when {  IUT receives 'Echo Request' from 'TN1'
                containing 'Hop-by-Hop Options Header'
                    indicating 'Header Ext Length field' set to 'ZERO'
            and IUT receives 'PadN option'
                containing 'Opt Data Len field' set to '4'
                and containing 'Option Data aligning the Hop-by-Hop Options Header to
                                a multiple of 8 octets' }
    then { IUT sends 'Echo Request' to 'TN2' } }
```

EXAMPLE 2:

```
TP id    : TP_COR_0047_02
Summary  : 'not aligning PadN option'
RQ Ref   : RQ_COR_0047
Config   : CF_005_C
TC Ref   : TC_COR_0047_02
ensure that {
  when {    IUT receives 'invalid Echo Request' from 'TN1'
                containing 'Hop-by-Hop Options Header'
                    indicating 'Header Ext Length field' set to 'ZERO'
            and IUT receives 'PadN option'
                containing 'Opt Data Len field' set to '3'
                and containing 'Option Data not aligning the Hop-by-Hop'
                        'Options Header to a multiple of 8 octets' }
  then { IUT sends 'PARAMETER PROBLEM' to 'TN1'
            containing 'the Code field'
                indicating 'code value 2'
            and containing 'the Pointer field'
                indicating 'pointer value' } }
```

7.1.3 Test Suite development in TTCN-3

7.1.3.1 Storage of TTCN-3 elements

In order to optimize the reuse of TTCN-3 code, it is necessary to separate those elements that are generic to IPv6 (data and behaviour for example) from those that are specific to a particular Test Suite.

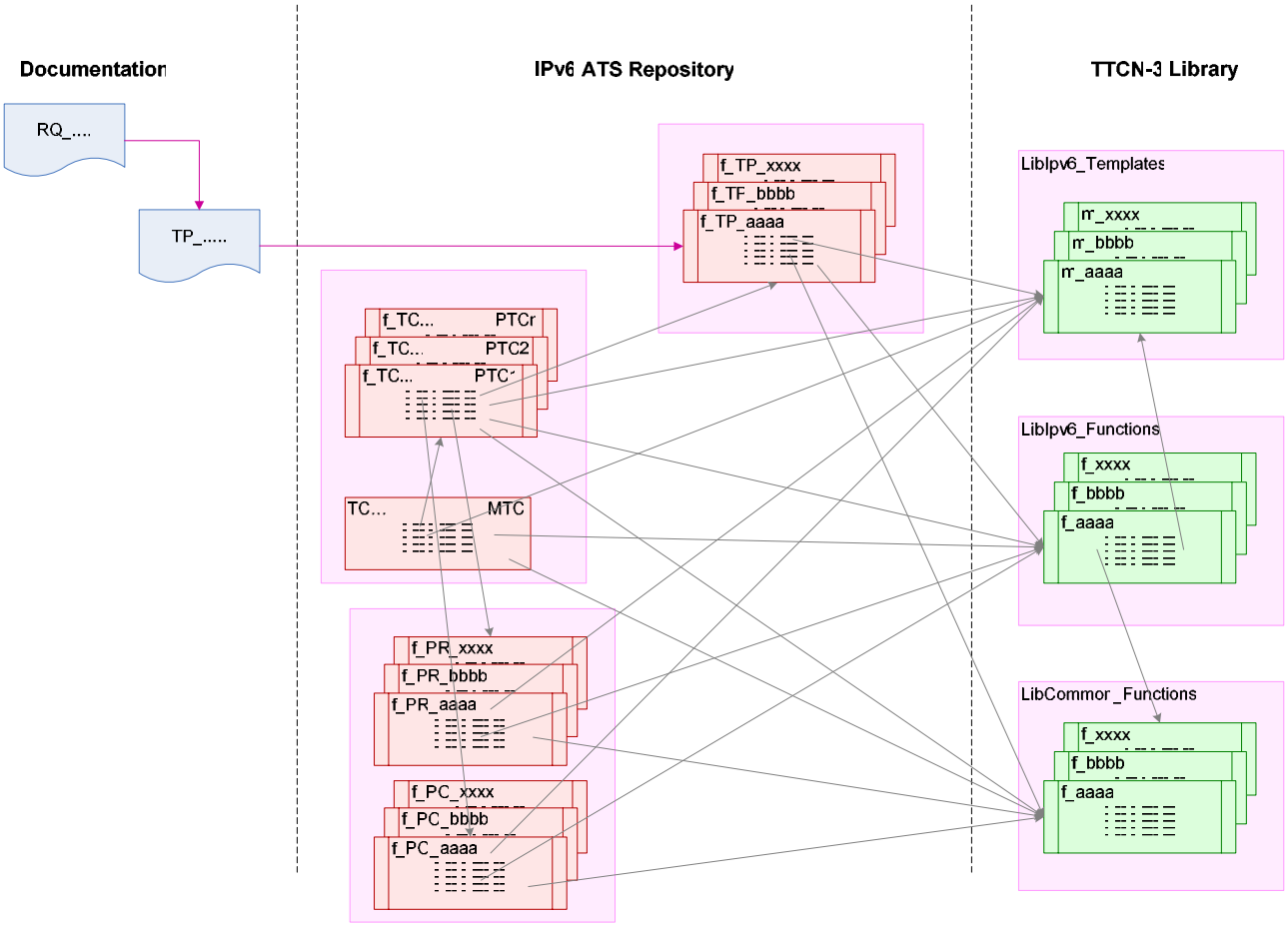


Figure 3: Allocation of TTCN-3 elements to the ATS Repository and the TTCN-3 module Library

The following elements are considered to be specific to the Test Suite and should be stored and maintained together in a Test Suite Repository as shown in figure 3:

- Test Cases (TC);
- Test Case functions (f_TC);
- Test Purpose functions (f_TP);
- Preambles (f_PR);
- Postambles (f_PO).

Although these functions can specify testing behaviour directly in TTCN-3, most should do little more than invoke reusable functions and use data and templates from the TTCN-3 Library which is described in more detail in clause 8.

7.1.3.2 Test Cases

From a TTCN-3 point of view, one of two basic testing configurations needs to be considered in each IPv6 test case implementation. The simpler of these is the non-concurrent arrangement, shown in figure 4, where there is only one TTCN-3 test component, the Main Test Component (MTC), which executes all aspects of the tests.

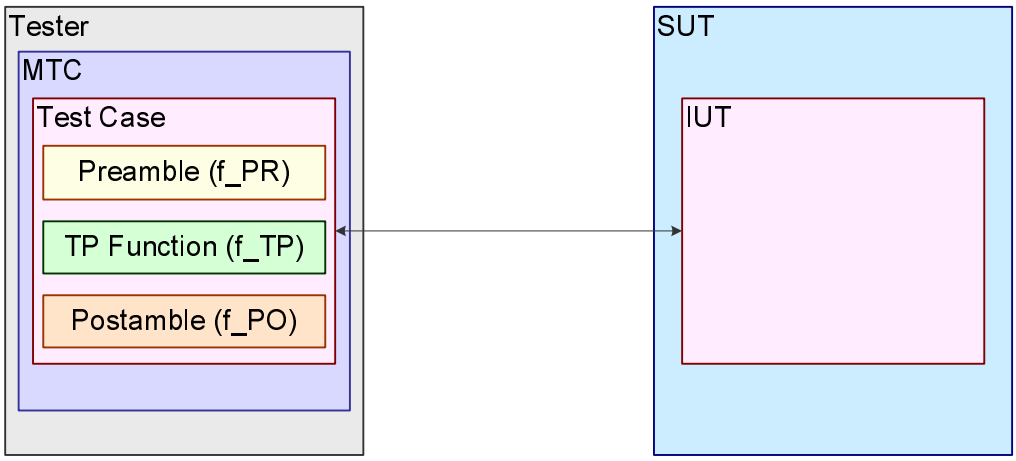


Figure 4: Non-concurrent TTCN-3 testing configuration

The more complex, and more usual, test arrangement distributes a test case implementation over two or more parallel components. This concurrent configuration is shown in figure 5.

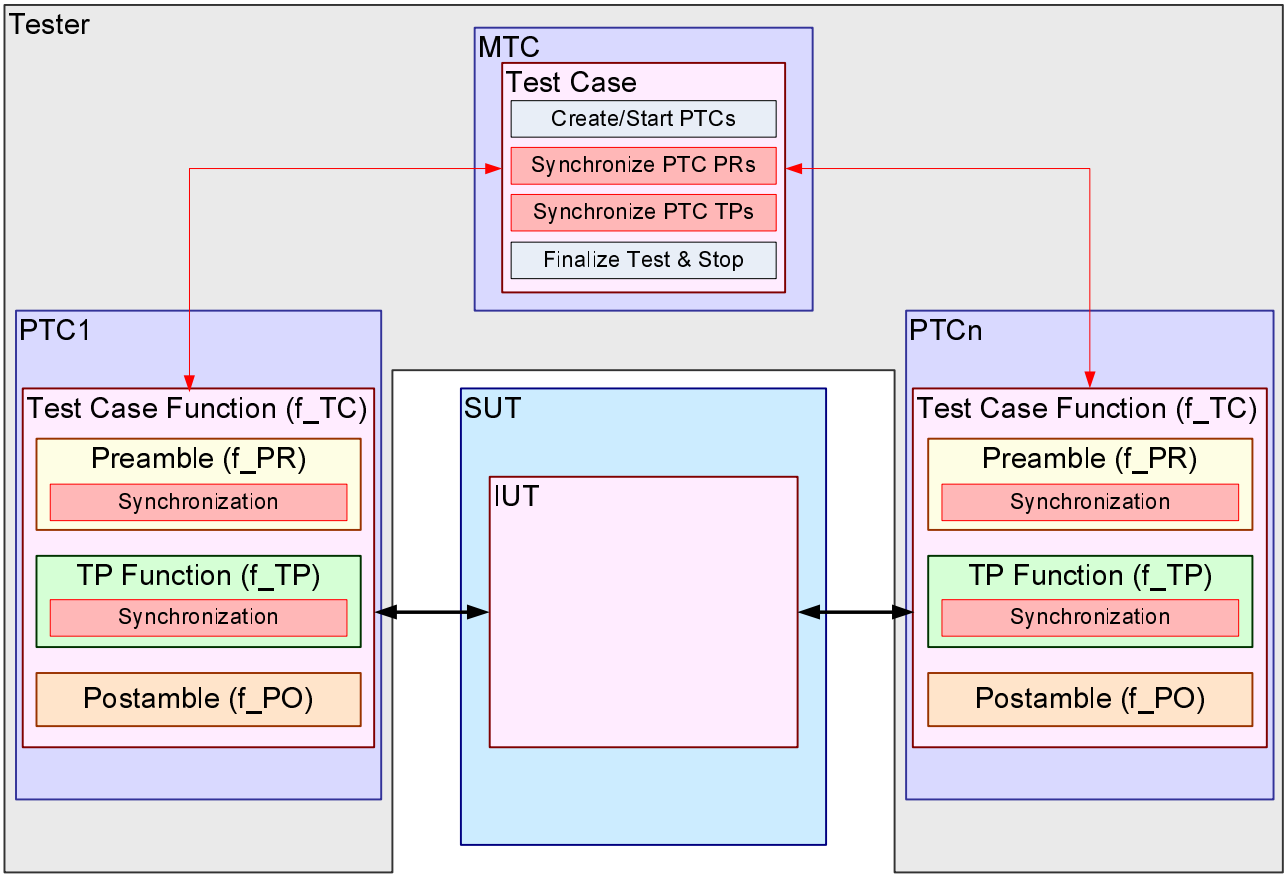


Figure 5: Concurrent TTCN-3 testing configuration

In this configuration the MTC only initiates the test by invoking a test case function on each Parallel Test Component (PTC) and coordinates the execution of PTCs using synchronization. Each PTC executes all aspects of a test related to its own particular role. Therefore, the MTC does not interact with the SUT.

Depending on the configuration used, the test case or each test case function should invoke a preamble (f_PR), test purpose (f_TP), and postamble (f_PO) function as described below:

- a preamble function should perform the actions required to place the IUT or EUT into the condition required by the test purpose function; then (within that function) the test component should set its verdict based on the success of these actions and synchronize with the MTC;
- a test purpose function should perform the actions required for the role of the test component (if applicable) to achieve the test as specified in the Test Purpose; then (within that function) the test component should set its verdict based on the success of these actions and synchronize with the MTC;
- a postamble function should perform the actions required to return the IUT to a known quiescent state after completing the test. Then, within that function, the test component should set its verdict based on the success of these actions.

7.1.3.2.1 Naming TCs

As there is a one-to-one relationship between TPs and TCs, they will share a common numbering scheme with a prefix to distinguish between them. For Test Cases this prefix will, naturally, be "TC". This prefix will be followed by the same numbering scheme as for TPs, thus:

- TC_COR_nnnn_mm IPv6 Core TCs.

EXAMPLE 1: TC_COR_0147_04.

- TC_SEC_nnnn_mm IPv6 Security TCs.

EXAMPLE 2: TC_SEC_0109_17.

- TC_MOB_nnnn_mm IPv6 Mobility TCs.

EXAMPLE 3: TC_MOB_0033_05.

- TC_T46_nnnn_mm IPv4 to IPv6 Transitioning TCs.

EXAMPLE 4: TC_T46_0006_32.

7.1.3.3 Test case functions

For each TP, one or more TC functions should be specified in TTCN-3. The TC function invokes the preamble, the TP function (clause 7.1.3.4) and the postamble. In non-concurrent test configurations (see figure 4) there will only be one TC function.

For configurations where the TC needs to be distributed over more than one parallel test component (see figure 5) there will be an equivalent number of TC functions. Each TC function will carry out the parts of the preamble and postamble specific to their particular PTC, as well as invoking the appropriate TP function. For example, in the configuration of figure 2 there would be corresponding TC functions running on HS_01, HS_02 and possibly the Upper Tester.

To summarize, the following guidelines apply to TC functions:

- TC Functions should only be necessary where there is more than one test component in the test architecture.
- Each PTC should have one TC Function defined for it.
- A TC Function is invoked in the "start test component" operation of a test case.
- TC Functions should be grouped with their associated test case.
- A TC Function should implement behaviour by invoking other functions rather than by expressing it directly. Any behaviour implemented directly in a TC Function would not be reusable in other test cases or functions.

7.1.3.3.1 Naming TC functions

As there is a one-to-one relationship between TCs and TC functions, they will share a common numbering scheme with the prefix "f_".

The name of a TC Function should include the role as well as the test case identifier, as shown in the following example:

- f_TC_COR_0051_12_HS.

The following abbreviations should be used to identify specific IPv6 roles:

- HS Host;
- RT Router;
- ND Node;
- EN End Node.

In those cases where more than one instance of a particular role is defined by a particular configuration, a 2-digit sequence number may be appended to the role as shown in the following examples:

- f_TC_T46_0298_03_ND_02.
- f_TC_SEC_0531_01_EN_01.

Optionally, a short string of free text may be appended to the TC Function name. This may be used to provide further classification and location information which would be useful for TTCN-3 programming purposes.

7.1.3.4 TP functions

For each TP, one or more TP functions should be specified in TTCN-3. The TP function should only specify the test behaviour related to the Test Purpose. A TP function should perform synchronisation as a final action and should include neither the preamble nor the postamble. TP functions are invoked from the TC function.

In test configurations where there is only one test component (see figure 4) there will only be one TP function.

In concurrent test configurations (see figure 5) there will be a number of TP functions which may be equal to or less than the number of components in the test configuration. For example, in the configuration shown in figure 2 it may be that the Upper Tester component is only used to configure the IUT but, as this behaviour is not part of the test body, no TP function will be associated with it. Consequently, there would be only two TP functions; one running on HS_01 and one running on HS_02.

To summarize, the following guidelines apply to TP functions:

- A TP Function should implement the test purpose for one component only.
- If there is more than one test component identified in the architecture associated with a TP, there should be one TP function for each of these components on which the TP behaviour is relevant.
- If there is only one test component identified in the test architecture, there should be only one TP function for each TP.
- A TP function should not call other behavioural functions although computational functions can be called.
- TP functions should not contain:
  - invocation of test configuration management;
  - implementation of test configuration management;
  - preamble aspects;
  - postamble aspects.

7.1.3.4.1 Naming of TP functions

The names (identifiers) of TP functions should begin with the prefix "f_TP_" followed by a descriptive name (text string). For example:

- f_TP_receiveEchoReplyAndTestChecksum.
- f_TP_sendHopLimitZeroAndReceiveTimeExceeded.
- f_TP_echoProcedure.

Additionally, where it is necessary to associate a TP function with a particular component in a test configuration TP function name should be suffixed with the role of the TP function in the test configuration.

The following abbreviations should be used to identify specific IPv6 roles:

- HS Host;
- RT Router;
- ND Node;
- EN End Node.

For example:

- f_TP_receiveEchoReplyAndSendRedirect_RT.

In those cases where more than one instance of a particular role is defined by a particular configuration, a 2-digit sequence number may be appended to the role as shown in the following example:

- f_TP_receiveEchoReplyAndSendRedirect_RT_01.

7.1.3.5 Preambles and Postambles

Both preamble and postamble functions should be constructed in a similar way to TC functions (clause 7.1.3.3) and TP functions (clause 7.1.3.4) in that they do not specify behaviour directly but use and invoke elements from the TTCN-3 Library. A preamble should also perform synchronisation as its final action.

7.1.3.5.1 Naming of Preambles and Postambles

Preamble and postamble functions should start with the prefix "f_" followed by "PR" for preambles and "PO" for postambles. These prefixes are followed by a text string specifying the role of the preamble or postamble. For example:

- f_PR_BasicInitialise.
- f_PO_BasicShutdown.

7.1.3.6 Test case selection

Every test case should be selectable by means a test case selection switch. This applies even to those test cases associated with mandatory requirements from the base specification. In order to facilitate this capability, each Test case should be preceded by a selection statement in the control part of the test suite module, as shown in the following example:

```
If (RQ_COR_0407)
    {
        execute (TC_COR_0407_35())
    }
```

NOTE: The conformance test suite is a TTCN-3 module comprising all the test cases relevant to a particular area of IPv6, Core IPv6 for example. This set of test cases may be subset by the requirement needs of a particular organization (e.g. IPv6 Forum or 3GPP) as defined in the requirements catalogue for the organization. This subset could be achieved using selection switches or even be specified as different test suite in its own right.

7.1.3.7 Test suite parameterization

It is often necessary to parameterize a test suite so that values not known at the time of writing the test cases can be used in testing. These values (input to the TTCN-3 ATS as module parameters) may depend on the IUT or the test system on which the test suite is being run.

NOTE: Test suite parameter values correspond to values normally found in a PICS or PIXIT.

Table 1 shows an example of how test suite parameters could be documented. The IUT Value column, highlighted in grey, is completed at the time of testing.

Table 1: Module (test suite) parameters

Organization: IPv6 Label				
Parameter Name	**Description**	**Reference**	**Type**	**IUT Value**
R_HOST	IP address for remote host	N/A	IPAddress	
T1	Response timer	RFC XYZ, 3.2	integer	
...				

7.2 Interoperability test specifications

7.2.1 Test configurations

For each test or group of tests specified in the IPv6 interoperability test suites, a configuration should be defined to identify the IPv6 roles required for the test components and the communications paths between those components.

Figure 6 shows an example configuration for interoperability testing.

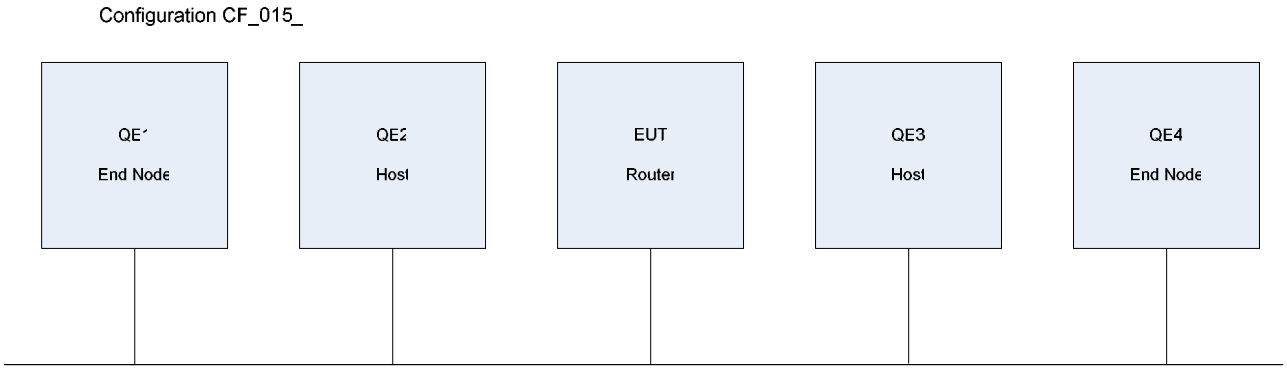


Figure 6: Example interoperability testing configuration

7.2.1.1 Naming IPv6 interoperability test configurations

Interoperability test configurations should be named so that they can be uniquely referenced in, for example, an interoperability test description or in the TP Language headers (7.1.2.2.2).

Configuration names should be of the form "CF_" followed by a three digit unique sequence number and the characters "_I", viz, "CF_023_I". This identifier should be included in any diagram associated with the configuration, as shown in figure 6.

7.2.1.2 Naming IPv6 test components

The components in each test configuration should be systematically and unambiguously identified. This naming is based on the role of each component, which includes the following:

- HS Host;
- RT Router;
- ND Node;
- EN End Node.

Each role should be followed by a two-digit sequence number that uniquely identifies that component. This is necessary when more than one component plays the same role. The role and the sequence number should be separated by an underscore. For example:

- HS_01, HS_02, RT_01, RT_02.

7.2.2 Test Suite Structure and Test Purposes

7.2.2.1 Test Suite Structure

Test Suite Structure (TSS) groups should be chosen according to natural divisions in the base specification(s) and should follow the functionalities specified in the Requirements Catalogue. The architecture of the testing configuration should also be taken into account such that, for example, all test purposes explicitly requiring an Upper Tester may be collected into a single group. Other examples might be groupings of "Normal behaviour" and "Exceptional" behaviour.

7.2.2.1.1 Naming IPv6 test groups

TP groups have a short name (or identifier) and a longer, more readable title. The short name is derived from the longer title (i.e. it is a two or three letter abbreviation of the longer title name). For example, if the long title is "Router", the short name should be: "RT". It is recommended that the title is followed by the short name in parentheses, for example: "Router (RT)"In the case of subgroups both the title and the short name should reflect the sub structuring, essentially making them path names. The group delimiter in the case of the title is "/". The delimiter in the case of the short name is: "_". As a further example, the group "Provide IPv6 Services (PS)" which is a sub group of the "Router (RT)" group, has the title:

- Router(RT)/Provide IPv6 Services(PS).

and the short name:

- RT_PS.

7.2.2.2 Test Purposes

A Test Purpose (TP) should be written for each potential test of an IPv6 requirement (as identified in the Requirements Catalogue) remembering that a requirement may need more than one TP to ensure that it is fully tested. As well as describing what is to be tested, the TP should identify the initial conditions to be established before testing can take place, the required status of the Implementation Under Test (IUT) or Equipment Under Test (EUT) from which testing can proceed and the criteria upon which verdicts can be assigned.

The contents of a TP should be limited to a description of what is to be tested rather than how that testing is to be carried out.

7.2.2.2.1 Naming IPv6 TPs

Each IPv6 requirement, as identified, in the Requirements Catalogue, will result in one or more TPs. Thus, RQs and TPs share a common numbering scheme but with a different prefix. For Test Purposes this prefix will, naturally, be "TP". This prefix will be followed by the four-digit sequence number taken from the requirement it corresponds to and a two digit sequence number to permit multiple TPs to be derived from a single requirement, thus:

- TP_COR_nnnn_mm IPv6 Core TPs.

EXAMPLE 1: TP_COR_0147_04.

- TP_SEC_nnnn_mm IPv6 Security TPs.

EXAMPLE 2: TP_SEC_0109_17.

- TP_MOB_nnnn_mm IPv6 Mobility TPs.

EXAMPLE 3: TP_MOB_0033_05.

- TP_T46_nnnn_mm IPv4 to IPv6 Transitioning TPs.

EXAMPLE 4: TP_T46_0006_32.

7.2.2.2.2 Using the TP Language

There is considerable benefit to be gained by having all Test Purposes written in a similar and consistent way. With this in mind, a simple, structured notation has been developed for the expression of TPs. This is described fully in annex A.

The benefits of using TPLan are identified in clause 7.1.2.2.2.

Examples of the use of the language to express conformance TPs are shown below.

EXAMPLE 1:

```
TP id   : TP_COR_1097_02
Summary : 'EUT forwards a traversed packet with its size equal to incoming link MTU'
RQ ref  : RQ_COR_1097
Config  : CF_021_I
TD ref  : TD_COR_1097_02

with { QE1 'configured with a unique global unicast address '
           and QE2 'configured with a unique global unicast address'
           and EUT 'configured with two unique global unicast addresses'
                   'on the link connecting QE1 and EUT and'
                   'on the link connecting QE2 and EUT, respectively'
           and QE1 'having a larger link MTU than EUT'
           and EUT 'having a larger or equivalent link MTU than QE2'
    }

ensure that {   when { EUT receives 'a packet with its size equals to its incoming link MTU'
                       containing 'QE1 as source address'
                       containing 'and QE2 as destination address' }
                then { EUT sends 'the packet' to QE2 }
            }
```

EXAMPLE 2:

```
TP id   : TP_COR_1130_01
Summary : 'EUT detects 2 packets with different hop-by-hop option contents'
          'but the same source and destination addresses in the flow label'
RQ ref  : RQ_COR_1130
Config  : CF_021_I
TD ref  : TD_COR_1130_01

with { QE1 'configured with a unique global unicast address '
           and QE2 'configured with a unique global unicast address'
           and EUT 'configured with 2 unique global unicast addresses'
                   'on the link connecting QE1 and EUT and the link '
                   'connecting QE2 and EUT, respectively'
    }
```

```
ensure that {   when { EUT receives 'two packets'
                       containing 'different hop-by-hop options'
                       and containing 'QE1 as the source address in the flow label'
                       and containing 'QE2 as the destination address in the flow label' }
                then { EUT sends 'an ICMP parameter problem message' to QE1
                       and EUT discards 'the packets' }
            }
```

7.3 Test Description development

Test Descriptions (TDs) specify the detailed steps that must be followed in order to achieve the stated purpose of each interoperability test. These steps should be specified in a clear and unambiguous way but without placing unreasonable restrictions on how the step is performed. TDs written in a structured and tabulated natural language are ideal when the tests themselves are to be performed manually. If, however, tests are to be automated, test cases should be written in TTCN-3. The development of TTCN-3 test cases does not mean that TDs should not also be produced because they have significant value as higher-level designs of the test cases.

NOTE: TDs should only be used in the specification of interoperability tests and not for conformance tests.

7.3.1 Naming Test Descriptions

As with TCs, there is a one-to-one relationship between TPs and TDs. Consequently, the naming of TDs is similar to that described for TCs in clause 7.1.3.2.1 except that the prefix "TD_" is used instead of "TC_", thus:

- TD_COR_nnnn_mm IPv6 Core TDs.

EXAMPLE 1: TD_COR_0147_04.

- TD_SEC_nnnn_mm IPv6 Security TDs.

EXAMPLE 2: TD_SEC_0109_17.

- TD_MOB_nnnn_mm IPv6 Mobility TDs.

EXAMPLE 3: TD_MOB_0033_05.

- TD_T46_nnnn_mm IPv4 to IPv6 Transitioning TDs.

EXAMPLE 4: TD_T46_0006_32.

7.3.2 Presentation of TDs

Test Descriptions should be presented as a sequence of activities and verdicts that can be followed manually by an operator, as described in TS 102 237-1 [1]. This sequence should be tabulated with header information and the associated Test Purpose as shown in table 2.

Table 2: Example Test Description

Test Description					
Identifier:	**TD_COR_001_02**				
Summary:	Autoconfigure QE using a unique address in a simple network				
Test Purpose:	TP_COR_001_02	**Reference:**	RQ_COR_01	**Configuration:**	CF_001_I

`ensure that {`
`  when {QE1 'has invoked stateless autoconfiguration' }`
`  then {EUT 'can address the QE'`
`  and QE1 'can address the EUT' } }`					

Pre-test conditions:	• QE1 configured to use a unique (within the network) link-local address during auto-configuration • EUT network interface is enabled • QE1 network interface disabled				
Step	**Test Sequence**			**Verdict**	
1	Enable QE1 network interface and wait a few seconds				
2	Cause QE1 to send an echo request to the Link-Local address of EUT				
3	*Check: Does protocol monitor show that an echo request was sent from QE1 to EUT?*			Yes	No
4	*Check: Does QE1 receive an echo reply from EUT?*			Yes	No
5	Cause EUT to send an echo request to the Link-Local address of QE1				
6	*Check: Does protocol monitor show that an echo request was sent from EUT to QE1?*			Yes	No
7	*Check: Does EUT receive an echo reply from QE1?*			Yes	No
Observations					

8 The TTCN-3 ATS Repository and Library

In order to facilitate the rapid and consistent production of both abstract and executable test suites, an (extensible) library of generally reusable TTCN-3 definitions is maintained by ETSI TC-MTS. The relation of this TTCN-3 library to the frame work is shown figure 1 and figure 3. This library is publicly available so that manufacturers, operators, testing organizations and other standards bodies can make use of it in constructing IPv6 test suites specific to their needs.

NOTE: The following clauses specify a number of rules on the basic library structure and the addition of modules to the library. These rules are presented as strong recommendations ("should") because they are considered to be based on good test programming practice. However, any TTCN-3 segments submitted for inclusion in the IPv6 TTCN-3 Library will be expected to comply with these recommendations as if they were mandatory.

Any TTCN-3 definition residing in the TTCN-3 Library should be independent of a test suite. Thus, definitions such as test case functions, preamble functions, test purpose functions and postamble functions should not appear in the library but be stored in an IPv6 Abstract Test Suite (ATS) Repository which is described in clause 8.3.

8.1 TTCN-3 Library structure overview

Within the library, elements are grouped into a number of different modules, for example:

- Modules for different kinds of data type and constant value definitions, e.g. common subtypes of basicTTCN-3 types, types specifying the protocol message structure in a particular RFC.
- Modules for template definitions which are based on some data type definitions, e.g. templates for specifying neighbourhood discovery messages.
- Modules for different kinds of function definitions, e.g. functions implementing basic protocol behaviour, verdict control, test component synchronization or algorithms.

8.1.1 Data types and values modules

Commonly used subtypes of TTCN-3 types are defined in separate modules based on their type kind or application context. For example, subtypes reflecting common encoding restrictions for *integer* or *boolean* values are stored within the TTCN-3 Common Library module for Basic Types And Values. Similarly, such subtypes for *bitstring* or *octetstring* are stored in the Common Library module for Data Strings. These subtype definitions use the TTCN-3 *encode* attribute to provide additional information to codecs because the TCI [3] currently does not support access to subtyping information.

```
type UInt2 integer (0..3) with { encode "2 bits" } // defined in LibCommon_BasicTypesAndValues
type Octet2 octetstring length(2) with { encode "2 octets" } // defined in LibCommon_DataStrings
```

Also, types, constants, and module parameter definitions related to a particular protocol, IPv6 for example, should be organized into one module per RFC. For each protocol a special module defines a message "meta-type" which includes all protocol messages. Another module collects common information element types which is imported in each of the RFC modules.

More details about the modularization of data types and values in the TTCN-3 Library can be found in the electronic attachment of annex B.

8.1.2 Templates modules

Template definitions should follow the same modularization as the data types they are based on. The following rules apply to template definitions themselves:

- Templates should be identified with meaningful names rather than numbers.
- Templates should not modify other modified templates. Base templates (which are modified by other templates) must be identified in their naming.
- Template definitions should avoid using matching attributes such as "*" or "?" for complete structured values, e.g. record or set of values.

8.1.3 Modules of TTCN-3 functions

The IPv6 library differentiates between protocol dependent functions, computational functions and common functions such as those for synchronization, verdict handling or data type manipulation. Again, functions should be modularized according to a particular application context. Function modules may also include data types and templates which are only used by the functions defined in that module. The following general rules apply:

- Functions should use the *runs on* statement wherever this is possible.
- Each function should provide a return value using the return value enumeration defined in the Common Library Verdict Control module.
- The *stop* statement should be used with care in functions (controlled test component shutdown should be always ensured).

8.1.3.1 Verdict control modules

The Verdict Control module is part of the Common Library and defines the TTCN-3 *FncRetCode* type which should be used by a function in the TTCN-3 Library (as well as in an ATS) to indicate its execution status to the calling function in the return value. This type is defined as an enumeration with the values *e_success, e_error* and *e_timeout*.

In addition, this module includes three verdict setting functions which set a test component verdict based on the *FncRetCode* value passed into the functions. The function *f_setVerdict* maps the values to directly to a pass, fail, and inclusive verdict which is useful, e.g., in test purpose functions. Other functions may be used to assign the verdict more appropriately for behaviour which is part of a preamble or postamble function. The functions in this module should only be called from functions in the ATS repository, but not by functions of the TTCN-3 library.

8.1.3.2 Synchronization module

The Sync Module is part of the TTCN-3 Common Library and specifies a generic mechanism which can be used to synchronize one or more test components in a test case. The main concepts underlying this mechanism are that one component acts as a synchronization server (usually the MTC) and all other components, i.e. the PTCs, act as synchronization clients. In a test case the synchronization may then be used to synchronize clients on one or more synchronization points after, for example, completing actions in a preamble or test body function. Depending on the test case there may be a need for more synchronization(s), e.g. in a test purpose function. A special case of synchronization is the one with only a single test component which is called "self synchronization".

The Sync module provides a number of basic functions which realize this mechanism.

- Server synchronization functions:

  - require specification of name and number of synchronization points as well as the number of clients, which are to participate in each synchronization, in their parameters;

NOTE 1: For preamble and test body synchronization the synchronization point names *c_prDone* and *c_tbDone* should be used. These are defined as part of the Sync module.

  - are implemented to await reports from the specified number of clients about their success in reaching each specified synchronization point, i.e. READY or STOP messages sent by clients; after that the server reports to all clients the success or failure of one or more of them to reach the synchronization point, i.e. again by sending a READY or STOP message to each client; if one of the specified clients fails to report to the server within a given time limit the function sends a STOP to all clients;

  - can be used to overwrite the default time limit the server waits for clients to initiate their synchronization;

  - always set the verdict of the test component acting as the server, i.e. to fail if one of the clients reports a failure of reaching a synchronization point or fails to reply within a given time limit and a pass verdict in all other cases.

- Client synchronization functions:

  - require the specification of only one synchronization point name and the current execution status, i.e. as a *FncRetCode* value, in their parameters;

NOTE 2: The execution status will be used by the client synchronization function to report to the server if a synchronization point was successfully reached or not, i.e. *e_success* will report a success whereas other values will report a failure.

NOTE 3: For preamble and test body synchronization the predefined synchronization points *c_prDone* and *c_tbDone* should be used.

  - report to the server the success or failure to reach a synchronization point based on the execution status; in all cases the client then awaits a response by server; upon a READY response the function exits normally; upon a STOP response the function will wait for some time and then stop test component execution;

NOTE 4: In order to prevent the test component from stopping without the execution of a postamble, a TTCN-3 altstep should be defined which (when receiving the server STOP message) invokes that postamble. This altstep shall then be activated as a default before the client synchronization function invocation.

  - can also be used set verdict of the test component acting as the client before the synchronization with the server, e.g. to fail if the execution status is *error, inconclusive* if timeout, or *pass* if success.

- Self synchronization functions:
  - allow the reuse of TTCN-3 shutdown altsteps also in non-concurrent test case configurations, i.e. the triggering of a postamble based on a (server) STOP message; in self synchronization a test component acts in essence as both, a client and a server;
  - require the specification of one specific synchronization point and the current execution status, i.e. as a *FncRetCode* value, in their parameters;
  - report the success or failure to reach a synchronization point to the *same* test component based on the execution status; the behaviour upon receipt of synchronization messages is exactly the same as in the case of client synchronization functions;
  - can be used to set the verdict of the test component, e.g. to fail if the execution status is *error*, *inconclusive* if *timeout*, or *pass* if *success*.

The use of the synchronization functions imposes certain requirements upon the ATS code:

- The test component types defined in an ATS for the MTC and PTCs must be type compatible to the *ServerSyncComp* and *ClientSyncComp* types defined in the Sync module. In the case of self synchronization the MTC type should be type compatible to the *SelfSyncComp* type.
- As part of the establishment of the test configuration in a test case and before the possible execution of any synchronization function, the *syncPort* of all test components acting as synchronization clients should be connected to the *syncPort* of the component acting as the synchronization server. In the case of self synchronization the *syncSendPort* needs to be connected to the *syncPort* of the MTC. Similarly, these ports should be disconnected as part of the test configuration tear down. The Sync module contains function definitions for this purpose.

All Sync module functions are well documented in the electronic TTCN-3 Library provided in annex A. Also part of TTCN-3 Library is a Sync Example TTCN-3 module which contains executable example test cases illustrating client/server and self synchronization as well as the triggering of postambles based on server STOP messages.

8.1.3.3 IPv6 behaviour modules

A number of modules specify IPv6 behaviour. Each module describes behaviour for one particular RFC. Functions in IPv6 behaviour modules should:

- neither set the test component verdict nor stop test execution but instead indicate their execution status using a *FncRetCode* return value;
- be test configuration independent, i.e. they should define information such as IPv6 addresses as function parameters;
- only use the IPv6 interface, i.e. they should neither configure the test system adapter nor invoke synchronization functions;
- include external IPv6 related computation functions which may be external functions, e.g. to compute the checksum of an Ipv6 packet.

8.1.4 Adding modules to the TTCN-3 Library

Users or organizations may submit their own modules for addition to the TTCN-3 Library. Such modules should be submitted to ETSI Technical Committee MTS for review. Details of the submission process can be obtained from the ETSI Secretariat at mtssupport@etsi.org.

This library itself can be accessed from http://www.ipt.etsi.org

8.1.5 ATS Repository structure overview

The ATS Repository is conceptually located above of the TTCN-3 library. It contains test suite specific TTCN-3 definitions including test case functions, test component types, test configuration functions, configuration message types and Ipv6 Node component types which reuse definitions in the TTCN-3 library as illustrated in figure 3.

The TTCN-3 definitions part of the ATS Repository can be further classified as being reusable or not reusable within the context of an ATS. The definitions which are reusable are stored in several ATS common TTCN-3 modules. Such definitions include:

- module parameters for a test suite;
- common test suite type definitions, e.g. test component types, configuration message types;
- preamble, test purpose, and postamble functions which invoke and evaluate the return values of functions implementing Ipv6 behaviour and then synchronize;
- configuration functions which establish or tear down specific TTCN-3 configurations, i.e. connect and map ports; they include also functions which compose address information for test components and test adapter for a specific test configuration (used by a test purposes);
- altstep definitions which are used as defaults by test components, e.g. to filter out unwanted IP packets, handle neighbourhood advertisements or a shutdown in test component synchronization.

TTCN-3 definitions which are not reusable in an ATS repository are:

- TTCN-3 test case statements;
- test case functions which invoke the relevant configuration, preamble, test purpose, and postamble function for a given test component role;
- the control part which should use Boolean test selection switches to invoke test cases.

More details about these modules can be found in the ATS Repository provided as part of the electronic annex B.

9 TTCN-3 naming conventions

The IPv6 TTCN-3 library will be publicly available for test developers to use and, in a controlled way, extend. It is, therefore, desirable to specify a naming convention to cover each of the TTCN-3 elements which require an identifier.

The naming convention is based on the following underlying principles:

- when constructing meaningful identifiers, the general guidelines specified for naming in clause 6 of EG 202 106 [2] should be followed;
- in most cases, identifiers should be prefixed with a short alphabetic string (specified in table 3) indicating the type of TTCN-3 element it represents;
- prefixes should be separated from the body of the identifier with an underscore ("_"):

EXAMPLE 1: `c_sixteen.`

- only module names, data type names and module parameters should begin with an upper-case letter. All other names (i.e. the part of the identifier following the prefix) should begin with a lower-case letter;
- the start of second and subsequent words in an identifier should be indicated by capitalizing the first character. Underscores should not be used for this purpose:

EXAMPLE 2: `f_authenticateUser()`.

Table 3 specifies the naming guidelines for each element of the TTCN-3 language indicating the recommended prefix and capitalization.

Table 3: IPv6 TTCN-3 naming convention

Language element	Naming convention	Prefix	Example	Notes
Module	Use upper-case initial letter	*none*	IPv6Templates	
TSS grouping	Use all upper-case letters as specified in clause 7.1.2.1.1	*none*	TP_RT_PS_TR	
Item group within a module	Use lower-case initial letter	*none*	messageGroup	
Data type	Use upper-case initial letter	*none*	SetupContents	
Message template	Use lower-case initial letter	m_	m_setupInit m_setupBasic	Note 1
Message template with wildcard or matching expression	Use lower-case initial letters	mw_	mw_anyUserReply	Note 2
Signature template	Use lower-case initial letter	s_	s_callSignature	
Port instance	Use lower-case initial letter	*none*	signallingPort	
Test component ref	Use lower-case initial letter	*none*	userTerminal	
Constant	Use lower-case initial letter	c_	c_maxRetransmission	
External constant	Use lower-case initial letter	cx_	cx_macId	
Function	Use lower-case initial letter	f_	f_authentication()	
External function	Use lower-case initial letter	fx_	fx_calculateLength()	
Altstep (incl. Default)	Use lower-case initial letter	a_	a_receiveSetup()	
Test case	Use numbering as specified in clause 7.1.3.2.1	TC_	TC_COR_0009_47_ND	
Variable (local)	Use lower-case initial letter	v_	v_macId	
Variable (defined within a component)	Use lower-case initial letters	vc_	vc_systemName	
Timer (local)	Use lower-case initial letter	t_	t_wait	
Timer (defined within a component)	Use lower-case initial letters	tc_	tc_authMin	
Module parameter	Use all upper case letters	*none*	PX_MAC_ID	Note 3
Parameterization	Use lower-case initial letter	p_	p_macId	
Enumerated Value	Use lower-case initial letter	e_	e_syncOk	

NOTE 1: This prefix must be used for all template definitions which do *not* assign or refer to templates with wildcards or matching expressions, e.g. templates specifying a constant value, parameterized templates without matching expressions, etc.
NOTE 2: This prefix must be used in identifiers for templates which either assign a wildcard or matching expression (e.g., ?, *, value list, ifpresent, pattern, etc) or reference another template which assigns a wildcard or matching expression.
NOTE 3: In this case it is acceptable to use underscore as a word delimiter.

10 TTCN-3 comment tags

Any TTCN-3 definition in the Test Suite Repository or Library should contain embedded comment tags. These comment tags can be used by tools to extract information from the TTCN-3 code to create, for example, a HTML-based reference documentation.

Comment tags which cover one or more lines should be specified using block comments, as illustrated:

```
/* ----------------------
 * @desc This line of text is now identified as a description
 *       which covers multiple lines
 * ----------------------*/
```

Comments tags specified within a single line may be specified using line comments, as illustrated:

```
// @author John Doe
```

Table 4 lists the tags that can be used in ETSI TTCN-3 test specifications with a short description of the intended use of each tag. Tools may support other, non standard tags. Such tags should not be used in TTCN-3 modules standardized by ETSI.

NOTE: Tools may also extract other information from the TTCN-3 code based, for example, on TTCN- 3 keywords. The definition of that extraction is beyond the scope of the present document.

Table 4: TTCN-3 Comment Tags

Tag	Description
@author	This tag should be used to specify the names of the authors or an authoring organisation which either has created or is maintaining a particular piece of TTCN-3 code.
@desc	This is probably the most import of all the tags. It should be used to describe the purpose of a particular piece of TTCN-3 code . The description should be concise yet informative and describe the function and use of the construct.
@remark	This tag may be used to add additional information, such as highlighting a particular feature or aspect not covered in the description.
@img	This tag may be used to associate images with a particular piece of TTCN-3 code.
@see	This tag may be used to refer to other TTCN-3 definitions in the same or another module.
@url	This tag should be used to associate references to external files or web pages with a particular piece of TTCN-3 code, e.g. a protocol specification or standard.
@return	This tag should only be used with functions. It is used to provide additional information on the value returned by the given function
@param	This tag is used to document the parameters of parameterized TTCN-3 definitions.
@version	This tag is used to state the version of a particular piece of TTCN-3 code.

The following provides some basic guidelines on the usage of tags for specific TTCN-3 definitions:

- each TTCN-3 module should use the *@author*, *@version* and *@desc* tags;
- the *@desc* tag should be used with all TTCN-3 definitions. However, this should not be taken to the extreme. For example, it is probably not useful to tag literally every single constant or template declaration. It is left to the discretion of the writer to find the right level of use. At least all major constructs such as test cases and functions should have a comprehensive description:
  - when a TTCN-3 definition uses module parameters, it is also recommended to mention this explicitly in the description;
  - descriptions for behavioural constructs should mention if they set the test component verdict and also all known limitations of the construct;
  - descriptions for type definitions, e.g. component types, should mention if the type has been designed to be type compatible to another type or vice versa to be used as a basis for other type definitions;
- the @see tag should be used to make dependencies between TTCN-3 definitions which are described by a @desc tag more explicit in the documentation, e.g. if some TTCN-3 definition uses a module parameter then its TTCN-3 definition should be referenced to using a @see tag;
- where applicable, parameterized constructions such as functions, altsteps and templates should use the *@param* and *@return* tags. The @param tags should first list the parameter name and then a brief description of how this parameter is used by the construct;
- the *@url* tag should be used to refer to the specification from which the TTCN-3 definition was derived from, e.g. a type definition could refer to a particular RFC IETF page. In some cases it may be necessary to use the *@desc* tag instead for this purpose as documents often are hard to access internally, i.e. it may only be possible to specify a reference to a complete document but impossible to point to a very specific clause in this document;
- the *@url* and *@img* tag may be used to link to relevant documentation such as Test Purposes or original requirements or even drawings of test configurations. Generally, the corresponding Test Purpose (in the TSS&TP) and to the corresponding Requirement (in the Requirements Catalogue) should be linked from the relevant TTCN-3 test case definition;
- the *@remark* tag may be used with any TTCN-3 definition. It should be used sparingly, e.g. possibly to indicate how a TTCN-3 definition should not be used.

11 Interaction between the test system and the SUT

In order to be able to completely automate conformance and interoperability testing, the upper interface of an IPv6 IUT (or in the case of interoperability testing, the EUT or QE) needs to be accessible to the TTCN-3 test system. Of special importance is the automated and controlled start up of the IUT prior to the execution of TTCN-3 code.

The specification of interfaces for performing interactions with an IPv6 stack is *not* standardized in IETF RFCs. Consequently, implementations of this interface are vendor specific and may vary between different IUTs. In order to implement a test suite independent of some given vendor's API, a test system needs to observe and control that API indirectly via an Upper Tester Server (UTS) which resides in the SUT as shown in figure 7. The test system communicates with the UTS using an abstract, client-server based protocol, the Simple Control and Observation Protocol (SCOP).

The client part of the protocol resides in the TTCN-3 test system and is called the Upper Tester Client (UTC).

The purpose of the UTS is to transform SCOP messages received from the UTC into interactions via the non-standardized IUT interface and vice versa.

NOTE: The UTS implementation is *not* part of the TTCN-3 test system executable. It is expected to be provided either by the implementer of the IUT or the party which intends to execute test cases in the TTCN-3 IPv6 test suite which require interaction via the upper interface. This clear separation of UTS and the test system implementation ensures, for example, that the test system will not be directly affected if the SUT crashes.

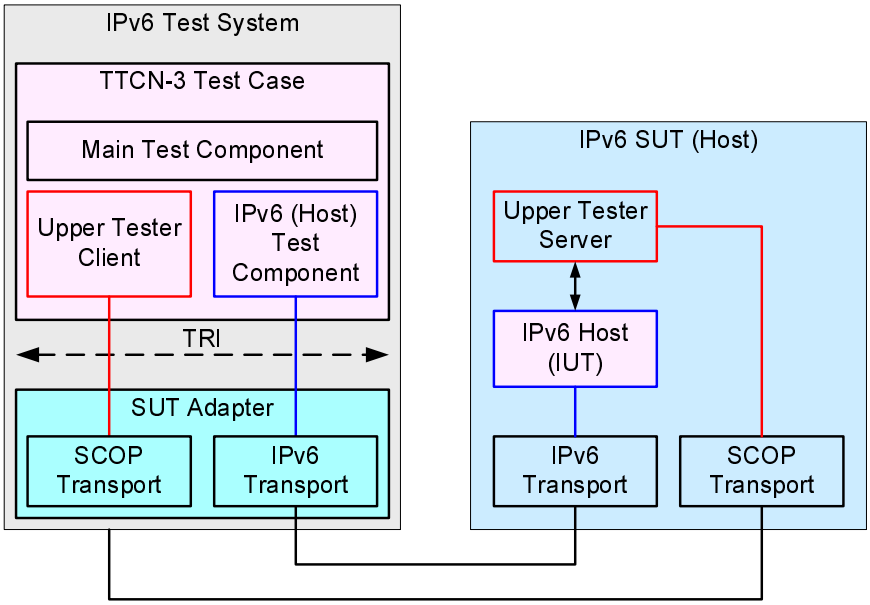


Figure 7: An example test configuration with an upper tester

11.1 The Simple Control and Observation Protocol (SCOP)

The set of SCOP messages is as follows:

```
ScopInitRequest (TestCaseName, [Params])
ScopResetRequest ([Params])
ScopResponse (Code, Description)

ScopDataRequest (Command, [Params], [Data])
ScopDataResponse (Code, Data)
```

The full definition of these primitives and their encodings can be found in annex C.

11.1.1 Upper Tester Server (UTS)

The UTS provides a SCOP interface towards the TTCN-3 test system (always acting in a server role). It should also support the SCOP transport protocol on the IUT-UTS interface by faithfully transforming the SCOP primitives to ensure the following interactions:

- Upon receipt of a `ScopInitRequest` primitive the UTS should configure and (re)start the IPv6 IUT. The test case name specified in the primitive should be used to select IPv6 stack configuration parameters necessary to start the IUT in the desired configuration. After the completion of this action the UTS should respond with a `ScopResponse` primitive where the response *`Code`* parameter should be either "OK" or an Error code (if the IUT did not start correctly).

- Upon receipt of a `ScopResetRequest` primitive the UTS should reset the IPv6 IUT which may involve restarting the IUT. After the completion of this action the UTS should respond with a `ScopResponse` primitive where the response *`Code`* parameter should be either "OK" or an Error code (if the IUT did not reset correctly).

- Upon the receipt of a `ScopDataRequest` primitive that specifies a `sendViaIut` command, the UTS should perform the actions(s) required for the IUT to send an IPv6 packet equivalent to the packet provided in the `ScopDataRequest` primitive i.e. the provided IPv6 packet is used to set the parameters in the required action(s). After the completion of this task the UTS should respond with an `ScopDataResponse` primitive where the response *`Code`* parameter should be either "OK" or an Error code (if the action could not be completed correctly) and with the *Data* parameter empty.

- Upon the receipt of a `ScopDataRequest` primitive that specifies a `getViaIut` command, the UTS should perform the action(s) required to the IUT retrieve the IPv6 packet that it received. An IPv6 stack may implement the notification of IPv6 packet arrival to applications either via a polling or interrupt scheme. As SCOP uses a polling scheme, the UTS is responsible for converting any interrupt driven mechanism to a polling scheme. For example, the UTS may use a polling timer to wait for a notification by the IUT. Note that a polling scheme implies that there is a FIFO buffer in either the IUT or the UTS. After the completion of this task the UTS must respond with a `ScopDataResponse` primitive where the response *`Code`* parameter should be either "OK" or an Error code (if the action could not be completed correctly). In the case that no message was received the *`Data`* parameter should be empty.

NOTE: The actual implementation of SCOP message transformations towards the IUT, e.g. into IUT function calls, and vice versa is beyond the scope of this document. A UTS may even be implemented for a manual transformation, e.g. as a graphical user interface which displays the SCOP message sent by the test system in some format to a human user. In the interest of automated test system execution, however, it is strongly recommended that the transformation is implemented in software.

11.2 The Upper Tester Client (UTC)

The upper IUT interface is driven via the UTS by a dedicated test component that acts as the TTCN-3 Upper Tester Client (UTC). This component is created in all test cases which require interaction(s) via the non-standardized upper IPv6 stack interface. It uses SCOP primitives to communicate with the UTS. These primitives and their encodings (transmitted over the SCOP transport) are defined in annex B. The TTCN-3 test system codecs and SUT adaptation layer handle the encoding, decoding, and transport of the TTCN-3 SCOP messages.

NOTE 1: It is outside the scope of the UTC to start the UTS in the SUT. Nevertheless this condition has to be fulfilled before any UTC behaviour executes.

NOTE 2: If a UTC participates in a test case then a separate synchronization by the MTC with all other test components must be included to prevent that the other PTCs attempt to interact with the IUT before it is initialized.

The UTC mainly interacts by sending an `ScopDataRequest` to the UTS and then waiting on an `ScopDataResponse` primitive from the UTS where the response *`Code`* parameter should be either "OK" or an Error code (if the action could not be completed correctly). Should the UTS fail to respond or respond with a `ScopResponse` primitive which contains an error code, the UTC should attempt to reset the IUT using the `ScopResetRequest` primitive.

NOTE 3: IPv6 packets returned by the UTS as a result of repeated requests for data (i.e. ScopDataRequest with the `getViaIut` command) may not always arrive in the order expected.

NOTE 4: It is possible to realize SCOP based message exchanges in TTCN-3 test suite implementations using either message-based communication or procedure based communication.

Annex A (normative): A formal notation for expressing test purposes

A.1 Introduction to TPLan

This annex defines a simple but formal notation for the expression of Test Purposes in a consistent and structured form. This notation, called TPLan, provides structure through the use of keywords defined in table A.1. However, because Test Purposes need to be expressive, TPLan allows the TP writer considerable freedom in the use of unstructured text between the keywords.

The syntax of this notation applies to Test Purposes for both conformance and interoperability testing. The only difference being that in the former case the keywords **IUT** and **TESTER** should be used. In the latter case keywords **EUT** and **QE** should be used.

The notation provides for header information to be included with the actual description of each TP which is referred to as the TP body. It also allows TPs to be grouped to provide the Test Suite Structure (TSS).

Table A.1: TP notation keywords

TSS header keywords	TP body keywords
author	accepts
date	after
title	and
version	before
	containing
TP grouping keywords	discards
end	ensure
group	EUT
id	from
objective	generates
	indicating
TP header keywords	IUT
config	not
id	or
ref	QE (optionally numbered QE1, QE2 etc.)
RQ	receives
summary	rejects
TC	sends
TD	set
TP	state
	TESTER
	that
	then
	to
	when
	with
	within

For conciseness and with a few notable exceptions, the keywords in table A.1 are shown in lower-case. However, these keywords are not fully case sensitive and may be capitalized. For example, the keyword **group** may also be written as **Group.** Other combinations of upper- and lower-case letters in keywords should be strictly avoided. Certain keywords such as **TP**, **IUT**, **EUT**, **RQ**, **TC**, **TD**, **TESTER** and **QE** should always appear in upper case.

Comments shall be introduced by the string"--" and terminated at the end of the same line.

A.2 TSS Header

The complete TSS&TP specification shall begin with the following header:

- **`Title`**

  name of the TSS&TP as quoted free text;

- **`Version`**

  version number as three numeric values separated by dots (".");

- **`Date`**

  date as three numeric values separated by dots (".") or 'forward slash ("/");

- **`Author`**

  document author as quoted free text.

Note that each keyword may optionally be followed by a colon (":").

An example TSS&TP header:

```
Title   : 'My TSS&TP as an example'
Version : 1.0.0
Date    : 29.11.2004 -- could also be written as 29/11/2004
Author  : 'ETSI STF276'
```

A.3 Grouping

TP groups may be nested to provide sub-grouping. The contents of the group may be other groups (sub-groups) or TPs or both sub-groups and TPs. A TSS&TP does not have to be structured but if it is then each group in that structure shall have the following group header:

- **`Group`**

  - start of a new group. Includes group number (e.g. 1.1) and a long form of the group identifier as described in clause 7.1.2.1.1;

- **`Objective`**

  - a short quoted free text description of the objective of the test group.

The TSS (Test Suite Structure) shall be expressed using the **Group** keyword followed by the group number and by an optional string of free text. The end of a group shall be indicated by the keyword pair **end Group** followed by the group number.

Indentation may be used to indicate a sub group. But in cases of deep sub-grouping this should be avoided for readability reasons.

In order to aid readability the **end group** keywords shall be followed by the group number.

An example of one group and a sub group:

```
Group 1 'Router (RT)' -- some optional free text can go here
Objective 'Test Purposes for Router'
    Group 1.1 ' Router(RT)/Provide IPv6 Services(PS)'
    Objective 'Test Purposes for Provide IPv6 Services'
        ... TPs or more subgroups can go here ...
    End Group 1.1
    ... TPs or more subgroups can go here ...
End Group 1
```

A.4 TP Header

Each TP shall begin with the following header.:

- **`tp id`**

  the TP Identifier of the form TP_aaa_nnnn_mm, as defined in clause 7.1.2.2.1;

- **`summary`**

  free text descriptive title of the TP;

- **`RQ ref`**

  a reference to the original requirement of the form RQ_aaa_nnnn, as defined in clause 6.2;

- **`config`**

  a reference to the relevant testing configuration of the form CF_nnn, as defined in clause 7.1.1;

- **`TC ref`** or **`TD ref`**

  a reference to the corresponding Test Case (for conformance) or Test Description (for interoperability).

Note that each keyword may optionally be followed by a colon (":").

For example:

```
TP id   : TP_COR_0001
Summary : 'Pad1 option'
RQ Ref  : RQ_COR_0001
Config  : CF_001_C
TC Ref  : TC_COR_0001
```

A.5 TP body

The main body of the TP follows the header and it is here that the test purpose is described in detail. The TP is generally written from the viewpoint of the **IUT** or **EUT**. Each TP description shall begin with the keywords **ensure that** followed by the remainder of the description enclosed in curly braces.

For example:

```
……
ensure that {

    -- TP description goes here

}
……
```

The **when** and **then** statements describe stimuli and responses (interactions) as seen from the point of view of the IUT (or EUT). Generally these are of the form:

```
ensure that {
                when { ... }   -- actions described from the viewpoint of the IUT or EUT.
                then { ... }   -- IUT or EUT responses and other behaviour
            }
```

These statements may be repeated in any order and any number of times, for example:

```
ensure that {
    when { ... }
    then { ... }
    when { ... }
    then { ... }
    ……
}
```

A.5.1 The **with** statement

The **with** statement may be optionally used to express the initial state or condition of the **IUT** or **EUT** from which the TP description begins. If used, the **with** statement shall precede the **ensure that** statement. The **with** statement does not define the steps or actions needed to reach the starting condition, only the condition itself. The conditions shall be expressed as free text. Multiple conditions shall be logically concatenated using the Boolean operators **and**, **or**, **not**. The general format of the **with** statement is:

```
with { IUT condition 1 and condition 2 and not ...etc...}
```

For example:

```
with { IUT 'in idle state' and 'port80 open' }
ensure that {
    when { ... }
    then { ... }
    when { ... }
    then { ... }
    …… }
```

A.5.2 The **when** statement

The **when** statement shall express an action, in most cases performed by the tester (or **QE**) and observed by the **IUT** (or **EUT**). Typically this will be a **receives** statement (i.e. the IUT has received a stimulus) with the name or description (expressed as free text) of the received message.

```
IUT receives 'a message'
```

In cases where there is more than one test interface in the test configuration the keyword **from** may be added to the **receives**.

```
IUT receives 'a message' from 'some interface'
```

Optionally, the expected message fields may be described using the **containing** keyword followed by a free text description. Also optionally, the values of these fields may be described in free text following the **indicating** keyword or the **set to** keywords.

```
IUT receives 'a message' containing 'description of a field'
                         indicating 'expected value of a field'
```

Other keywords that may appear in the **when** statement are **and**, **or**, **not**. For example:

```
when { IUT receives 'a message' from 'node 1'
            containing 'field 1' indicating 'any valid value'
            and containing 'field 2' set to '33'
            and IUT receives 'a second message'
     }
```

A.5.3 The **then** statement

The **then** statement shall express the expected response to the previous **when** statement. In most cases the response is performed by the **IUT** (or **EUT**) and observed by the tester (or **QE**). Typically this will be a **sends** statement followed by the name or description (expressed as free text) of the sent message.

```
IUT sends 'a message'
```

In cases where there is more than one test interface in the test configuration the keyword **to** may be added to the **sends**. For example:

```
IUT sends 'a message' to 'some interface'
```

The syntax of the contents of sent messages is the same as that for the receive statement. For example:

```
IUT sends 'a message' containing 'description of a field' indicating 'expected value of a field'
```

Other keywords that may appear in the **then** statement are **and**, **or**, **not**. For example:

```
then   { IUT sends 'another message' to 'node 1'
                containing 'field 3' indicating 'any valid value'
                and containing 'field 4' indicating 'any valid value'
                and IUT sends 'yet another message'
       }
```

A.5.4 Other behavioural statements

The keywords **accepts**, **discards**, **ignores**, **rejects** may be used to describe other possible response to a received message. For example:

```
IUT accepts 'the message'
-- or
IUT ignores 'the message'
-- or
IUT discards 'the message'
-- or
IUT rejects 'the message'
```

The **state** keyword shall be used to express state information. For example:

```
IUT state 'changes from IDLE to ACTIVE'
-- or
IUT state 'remains in IDLE'
```

The keywords **before, within** and **after** are used to express ordering, especially in the context of timers.

For example:

```
before 'timer T1 expires'
-- or
within 'two minutes'
-- or
after '15 seconds'
```

The following example shows a more complete use of the **then** statement using the above constructs:

```
then {  EUT accepts 'incoming request'
            and EUT sends 'a message to Node 1' within '15 seconds'
            and EUT state 'changes to ERROR'
     }
```

A.6 The TPLan Grammar

The TPLan grammar is defined below. This can be used either as a reference or as input to parser generator tools. Table A.2 defines the syntactic conventions that should be used when reading the TPLan BNF.

Table A.2: The syntactic metanotation

::=	is defined to be
abc	the non-terminal symbol abc
abc xyz	abc followed by xyz
abc | xyz	alternative (abc or xyz)
[abc]	0 or 1 instances of abc
{abc}	1 or more instances of abc
[{abc}]	0 or more instances of abc
(...)	textual grouping
"abc"	the terminal symbol abc

```
// BNF grammar for TSS & TP language (TPLan)
// Version: 1.0
// TSS part
1. tssandtp         ::= KWD_tssandtp qstring
                    tss_version
                    date
                    author
                    tss;
2. tss_version      ::= KWD_version
                    numeric "." numeric "." numeric;
3. date             ::= KWD_date
                    (numeric "." numeric "." numeric)
```

```
                    | (numeric "/" numeric "/" numeric);
4. author           ::= KWD_author
                    qstring;
5. tss              ::= {group | tp};
6. group            ::= group_id
                    [group_objective]
                    [{group | tp}]
                    end_group;
7. group_id         ::= KWD_group
                    numeric [{"." numeric}]
                    qstring;
8. group_objective  ::= KWD_group_objective
                    qstring;
// TP part
9. tp               ::= header
                    body;
// TP Header
10. header          ::= tp_id
                    summary
                    catalogue_ref
                    config_ref
                    tc_ref;
11. tp_id           ::= KWD_tp_id area numeric<4> "_" numeric<2>;
12. summary         ::= KWD_tp_summary qstring;
13. catalogue_ref   ::= KWD_catalogue_ref area numeric<4>;
14. config_ref      ::= KWD_config "CF_" numeric<2>;
15. tc_ref          ::= KWD_tc_ref area numeric<4> "_" numeric<2>;
16. area            ::= tp_area | tc_area | td_area | rq_area;
17. body            ::= [conditions]
                    KWD_ensure KWD_that
                    begin_tp
                    [{actions | responses}]
                    end_tp;
18. conditions      ::= KWD_condition
                    begin_conditions
                    [condition [{KWD_Boolean condition}]]
                    end_conditions;
19. condition       ::= [KWD_iut] [KWD_state] qstring;
20. actions         ::= KWD_action
                    begin_actions
                    [action [{KWD_Boolean action}]]
                    end_actions;
21. action          ::= KWD_IUT
                    [KWD_receive] qstring
                    [KWD_from qstring]
                    [{contents}]
                    [{time_constraints}];
22. contents        ::= [KWD_Boolean] KWD_containing
                    content [{content}];
23. content         ::= qstring
                    [{indications}];
24. indications     ::= KWD_indication
                    indication [{KWD_Boolean indication}];
25. indication      ::= qstring;
26. responses       ::= KWD_response
                    begin_responses
                    [response [{KWD_Boolean response}]]
                    end_responses;
26. response        ::= KWD_IUT
                    [(KWD_send | KWD_other_rsp)] qstring
                    [KWD_to (KWD_IUT [qstring] | qstring)]
                    [{contents}]
                    [{time_constraints}];
28. time_constraints::= KWD_timeout qstring;
qstring         ::= "'" *("'") "'";

// Keywords etc.
29. tp_area     ::=     "TP_COR_"  | "TP_SEC_" | "TP_MOB_" | "TP_TRA_";
30. tc_area     ::=     "TC_COR_"  | "TC_SEC_" | "TC_MOB_" | "TC_TRA_";
31. td_area     ::=     "TD_COR_"  | "TD_SEC_" | "TD_MOB_" | "TD_TRA_";
32. rq_area     ::=     "RQ_COR_"  | "RQ_SEC_" | "RQ_MOB_" | "RQ_TRA_";
33. KWD_action          ::= "when";
34. KWD_author          ::= "author" Delim;
35. KWD_boolean         ::= "and" ["not"] | "or" | "not";
36. KWD_catalogue_ref   ::= "RQ" "ref" Delim;
37. KWD_condition       ::= "with";
38. KWD_config          ::=  "config" Delim;
39. KWD_containing      ::= "containing";
```

```
40. KWD_date            ::= "date" Delim;
41. KWD_ensure          ::= "ensure";
42. KWD_from            ::= "from";
43. KWD_group           ::= "group";
44. KWD_group_objective ::= "objective" Delim;
45. KWD_group_title     ::= "title";
46. KWD_indication      ::= "indicating" | "set" "to";
47. KWD_iut             ::= "IUT" | "EUT" | "TESTER" | KWD_QE;
48. KWD_other_rsp       ::= "rejects" | "discards" | "accepts" | "ignores" | KWD_state;
49. KWD_QE              ::= "QE" [numeric];
50. KWD_receive         ::= "receives" | "generates";
51. KWD_response        ::= "then";
52. KWD_send            ::= "sends";
53. KWD_state           ::= "state";
54. KWD_tc_ref          ::= ("TC" | "TD") "ref" Delim;
55. KWD_that            ::= "that";
56. KWD_timeout         ::= "within" | "after" | "before";
57. KWD_to              ::= "to";
58. KWD_tp_id           ::= "TP" "id" Delim;
59. KWD_tp_summary      ::= "summary" Delim;
60. KWD_tssandtp        ::= "title" Delim;
61. KWD_version         ::= "version" Delim;

// Delimiters etc.
62. Delim               ::=     [":"];
63. end_group           ::=     "end" "group" [numeric [{"." numeric}]];
64. begin_tp            ::=     L_BRACE;
65. end_tp              ::=     R_BRACE;
66. begin_conditions    ::=     L_BRACE;
67. end_conditions      ::=     R_BRACE;
68. begin_contents      ::=     L_BRACE;
69. end_contents        ::=     R_BRACE;
70. begin_actions       ::=     L_BRACE;
71. end_actions         ::=     R_BRACE;
72. begin_responses     ::=     L_BRACE;
73. end_responses       ::=     R_BRACE;
74. L_BRACE             ::=     "{";
75. R_BRACE             ::=     "}";

// Whitespace and comments
76. space_symbol <TERMINAL,HIDDEN> ::=
        {   '[\32\r\n\t]'       // regular whitespace
            | "--" *("\n")          // ASN.1 style comment
            ,0,0
        };
```

Annex B (informative): TTCN-3 library modules

B.1 Electronic annex, zip file with TTCN-3 code

The TTCN-3 library modules are contained in archive ts_102351v020101p0.zip which accompanies the present document.

Annex C (normative): SCOP type definitions and encodings

The objective of the Simple Control & Observation Protocol (SCOP) is to offer an intermediate, abstract interface for controlling and observing non-standardized interfaces via test systems. It is intentionally kept very simple and readable in its encoding to make it easy to use as well as flexible.

In its core the protocol follows a client server protocol. Here, an SCOP client is the controlling entity, i.e. a test component with a test system, and an SCOP server is the controlled entity which resides in the SUT. SCOP defines a set of primitives which can be exchanged between clients and servers.

SCOP can be easily extended using SCOP parameters. They allow to add additional information (in a textually encoded format) which can be helpful when to implement more advanced SCOP client and server implementations. They could be used, for example, to specify configuration parameters and values in the SCOP initialization primitive instead of associating them with specific test case name in the SCOP server. Another example is to use them to exchange information about the state of the SCOP server.

In the specific case of the ETSI IPv6 test system SCOP parameters are actually *not* used since it has not been designed for a specific IUT. Therefore these parameters do not need to be handled in a SCOP client implementation for this particular test system.

C.1 The Protocol Type Definition

In the following we present the available SCOP primitives in more detail. Although we have chosen TTCN-3 types in our presentation one can also define similar type structures in other languages, e.g. ASN.1 or C.

```
module scopTypes {

    group scopPrimitives {

        /*
        ** @desc Requests the SCOP server to configure and start up the IUT
        **       implementation for a specific test case. The scopParams
        **       field may be used to provide some additional information
        **       for the configuration of the IUT.
        **       This primitive is sent from client to server.
        */
        type record ScopInitRequest {
            ScopString    testCaseId,
            ScopParamList params optional
        }

        /*
        ** @desc Requests the SCOP server to reset the IUT implementation. It
     **       is intended to be used by a SCOP client at any time to
     **       attempt 'recovering' the IUT, e.g. if the IUT
     **       is not responding anymore during a test case.
        **       This primitive is sent from client to server.
        */
        type record ScopResetRequest { }

        /*
        ** @desc Requests the SCOP server to perform the action specified in
        **       the cmd field on the IUT. The data field may contain the encoded
        **       data to be produced by the IUT as a result of the command.
        **       Parameters may be used to convey information from the
     **       client to the server which is not contained in the encoded data,
     **       e.g. to set the state of a SCOP client.
        **       This primitive is sent from client to server.
        */
        type record ScopDataRequest {
            ScopAction    command,
            ScopParamList params optional,
            Data          data optional
        }
```

```
        /*
        ** @desc Indicates result of performing any SCOP request except
     **       a ScopDataRequest on the IUT
        **       A code 0 and desc 'OK' must be used for successful
     **       return from the IUT operation.
        **       The codes 1+ should be used for errors where the
     **       corresponding description should describe the problem
     **       that occurred.
     **       Should be returned for each ScopInitRequest and ScopResetRequest
        **       This primitive is sent from server to client.
        */
        type record ScopResponse {
            integer      code,
            ScopString   desc
        }

        /*
        ** @desc Indicates the result of performing a ScopDataRequest
     **       on the IUT..
        **       A code 0 must be used for successful
     **       return from the IUT operation. The codes 1+ should be used
        **       for errors.
     **       In a response to a ScopDataRequest with a 'getViaIut' action
     **       the data field may contain the encoded data, e.g. a message,
        **       that was received by the SCOP client from the IUT. In all
     **       other cases this filed should be omitted.
     **       The parameters may be used to convey information to the
        **       client which is not part of the encoded data (e.g. about
        **       new state of SCOP server).
        **       This primitive is sent from server to client.
        */
        type record ScopDataResponse {
            ScopAction    command,
            integer       code,
            ScopParamList params optional,
            Data          data optional
        }

    } // end group scopPrimitves

    group otherScopTypes {

        /*
        ** @desc Currently identified commands. May be subject to further extension
        */
        type ScopString ScopAction ("sendViaIut", "getViaIut");

        type record of ScopParam ScopParamList;

        type record ScopParam { ScopString pName, ScopString pValue };

            type charstring ScopString length(1..255);
        type octetstring Data length(1..255);

    } // end group otherScopTypes

} // end module scopTypes
with {
    encode "SCOP/1.0"
}
```

C.2 Encoding of SCOP

The encoding of SCOP primitives has been designed to be as simple and humanly readable as possible in order to simplify the analysis of communication between the SCOP clients and servers. Both, type and value information, is encoded using text strings. Each string is encoded with one octet specifying the length of the string followed by the textually encoded value. The only exception is the encoded data string which uses a four octet length field (in network byte order). This form of text string encoding has been adapted from other IETF protocols, e.g. the DNS protocol [5] clause 3.3.

The protocol identifier "SCOP/1.0" refers to the version of the protocol and encoding rules defined in clause 11 and annex C of the present document.

The following algorithm describes the encoding process in more detail:

1. Create new empty encoded message.
2. Append text string 'SCOP/1.0' as protocol identifier.
3. Append length in bytes of message type name as single octet to encoded message.
4. Append message type name as text string to encoded message.
5. For each field in the message value:
    a. if field value is present:
        i. append length in bytes of message field name as single octet to encoded message.
        ii. Append message field name as text string to encoded message.
        iii. If field type is ScopParamList:
            1. for each parameter in parameter list value:
                a. encode the fields of the parameter like message fields, i.e. append 0x05 and 'pName' for first field, then append the length of the name value as a single octet, append the name value, etc.
            2. Append a single octet with value 0x00 to indicate the end of the list.
        iv. Else if field type is integer kind:
            1. convert to integer value to text string, e.g. 0x01 into '1' (i.e, 0x31).
            2. Append length in bytes of the *text integer value* as single octet to encoded message.
            3. Append text integer value to encoded message.
        v. Else if field type is octetstring kind:
            1. convert to octetstring value to text string, e.g. 0x010A into '010A' (i.e, 0x3031); where letter hexadecimals should always we converted to upper case letters.
            2. Append length in bytes of *text octetstring value* as four octets (in network byte order) to encoded message, e.g. the length of the encoded '010A' is 0x00000004.
            3. Append the text octetstring value to encoded message.
        vi. Otherwise:
            1. append length of field text string value in bytes to encoded message as single octet.
            2. Append the text string value to encoded message.

EXAMPLES:

TTCN-3 message (type see type definition above):

```
template ScopInitIutRequest m_startTc_001 := {
    testCaseId := 'TC_0001',
    params     := omit
}
```

Encoding with non-character octets shown in \xhh format:

```
SCOP/1.0\x0FScopInitRequest\x0AtestCaseId\x07TC_0001
```

Encoding with non-character octets shown as '?':

```
SCOP/1.0?ScopInitRequest?testCaseId?TC_0001
```

TTCN-3 message (type see type definition above):

```
template ScopDataRequest m_sendIpv6Pkt := {
    command := 'sendViaIut',
    params  := omit,
    data    := "60000000000103A4020010660550327 6a00000000000000042001066055032 76aaeacacfffe276a21
                8000085d46a4000042369c39000648a7"O // IPv6 packet with ICMP echo request
   }
```

Encoding with non-character octets shown in \xhh format:

```
SCOP/1.0\x0FScopDataRequest\x07command\x0AsendViaIut\x04data\x00\x00\x00\x706000000000103A4020010660
5503276a0000000000000004200106605503276aaeacacfffe276a218000085d46a4000042369c39000648a7
```

Encoding with non-character octets shown as '?':

```
SCOP/1.0?ScopDataRequest?command?sendViaIut?data????60000000000103A4020010660550327 6a0000000000000004
200106605503276aaeacacfffe276a218000085d46a4000042369c39000648a7
```

Annex D (informative): The IPv6 requirements database

The information discovered during and after requirements identification and cataloguing will be placed into a database. This database will be used as the source of data for visual presentations of the catalogue in either HTML format, ETSI specifications format, or any other presentation format. It will also contain the various hypertext links developed during the requirements cataloguing process.

Fields can be empty. Required data fields are indicated by an * in the discussion below.

Figure D.1 shows fields for the database with the field names separated from the data by a colon. The grouping of the fields into records appropriate to the type of DBMS is not discussed here. For example, each `RqmtTupleList` may be a record in a separate database file using `RqmtID` as a key in a relational database management system (DBMS).

```
RqmtID:  RQ_COR_1254
RqmtSubject:  Node
ParentFncNode: Process IPv6 Header
ParentFncType: SHALL
ParentFncRef:  RFC 2460
Function_node:  Process Oversized Packet
FunctionType:  SHALL
FunctionRef:  RFC 1981, §3 ¶1
RqmtContext:  The Implementation received a packet for forwarding. The packet's
size is larger than the PMTU.
Rqmt: The Implementation discards the packet and returns an ICMPv6 Packet Too
Big message to the packet's source address.
RqmtTupleList:  {RFC 1981, §3 ¶1; RFC 2463, §3.2::SHALL}
Conf_TP_ID:
Conf_TC_ID:
Interop_TP_ID:
Interop_TC_ID:
Link_2srce:
Link_2rqmt:
Link_2ConfTP:
Link_2InteropTP:
Link_2ConfTC:
Link_2InteropTC:
```

Figure D.1: Sample Record from the Catalogue Database

Note that figure D.1 is an abstract representation of the record. Its implementation depends upon the DBMS; e.g. Microsoft Access, text-oriented, etc.

A discussion of each field follows.

- `*RqmtID`: IPv6 Core requirements Identification number, (example: RQ_COR_1254). Data type is ASCII string.
- `RqmtSubject`: The subject of the requirement; i.e {Node, Host, Router, etc}. Data type is enumerated.
- `*ParentFncNode`: The parent function node above the node of this specific requirement in the function tree. This field allows construction of the complete function tree if necessary. Data type is enumerated or ASCII string.
- `*ParentFncType`: The parent function node's type; i.e. {SHALL, SHOULD, …}. Data type is enumerated. If the source document does not explicitly state the type, the catalogue will implicitly define its type and denote this by surrounding it with brackets "[ ]".

- `*ParentFncRef`: The source document reference for the parent's function type. Data type is ASCII string.
- `*Function_node`: The requirement's function and node on the IPv6 function tree. This node is a child of the parent node in the same record. The entire function tree can be correctly built in the leaf-to-root node direction from the "Function_node" and "ParentFncNode" in the database. Data type is enumerated or ASCII string.
- `*FunctionType`: The function node's type; i.e. {SHALL, SHOULD, …}. Data type is enumerated. If the source document does not explicitly state the type, the catalogue will implicitly define its type and denote this by surrounding it with brackets "[ ]".
- `*FunctionRef`: The source document reference for the parent's function type. Data type is ASCII string.
- `RqmtContext`: the general conditions that are necessary for the requirement to exist. Another view of the context is that it is the situation that leads to the specific requirement. It is not to be confused with "preambles" used for test cases. Data type is ASCII string.
- `Rqmt`: the requirement for the given function. Data type is ASCII string.
- `RqmtTupleList`: the list of requirement 2-tuples. A requirements 2-tuple is a list of requirements sources and one requirement type. Requirement's sources are references to the source documents. There may be more than one requirement's source for the same requirement. Data type of requirement's source is ASCII string. Requirement's type is {SHALL, SHOULD, …}…}. Data type is enumerated. If the source document does not explicitly state the type, the catalogue will implicitly define its type and denote this by surrounding it with brackets "[ ]".
- `Conf_TP_ID`: The ETSI conformance test purpose ID number corresponding to this requirement. Data type is ASCII string.
- `Conf_TC_ID`: The ETSI conformance test case ID number corresponding to this requirement. Data type is ASCII string.
- `Interop_TP_ID`: The ETSI interoperability test purpose ID number corresponding to this requirement. Data type is ASCII string.
- `Interop_TC_ID`: The ETSI interoperability test case ID number corresponding to this requirement. Data type is ASCII string.
- `Link_2srce`: The hypertext link to the location in the source document of the requirement. This link can be used in the Requirements Catalogue to go to the source of the requirement. Data type is hyperlink.
- `Link_2rqmt`: The hypertext link to the location in the Requirements Catalogue of the requirement. This link can be used in a version of the reference source document that allows hyperlinks in order to see the Requirements Catalogue entry. Data type is hyperlink.
- `Link_2ConfTP`: The hypertext link to the location in the ETSI conformance TP specification of the TP associated with Conf_TP_ID. This link can be used in a version of the Requirements Catalogue to see the conformance TP associated with this requirement. Data type is hyperlink.
- `Link_2InteropTP`: The hypertext link to the location in the ETSI interoperability TP specification of the TP associated with Interop_TP_ID. This link can be used in a version of the Requirements Catalog to see the interoperability TP associated with this requirement. Data type is hyperlink.
- `Link_2ConfTC`: The hypertext link to the location in the ETSI conformance test case specification of the test case associated Conf_TC_ID. This link can be used in a version of the Abstract Test Suite. Data type is hyperlink.
- `Link_2InteropTC`: The hypertext link to the location in the ETSI interoperability test case specification of the test case associated Interop_TC_ID. Data type is hyperlink.

  Note: * indicates mandatory field entries for each database record. It may seem odd that each record may not contain a requirement. This is because some nodes in the tree function do not have any requirements attached to them. In this case, the requirements are attached to the descendants.

Example display of the Requirements Catalogue.

One possible way of displaying the Requirements Catalogue using information in the data base is shown in figures D.2 and D.3.

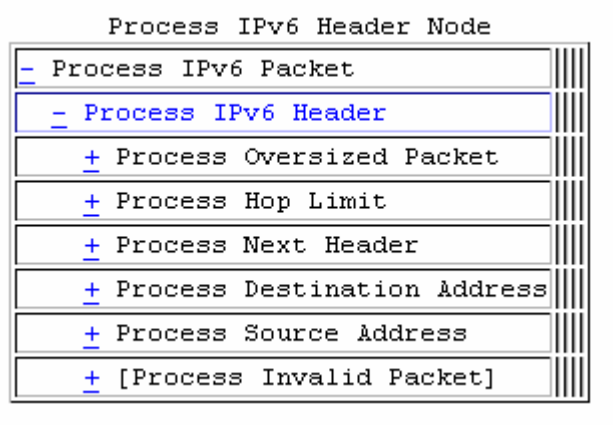


Figure D.2: Process IPv6 Header Node

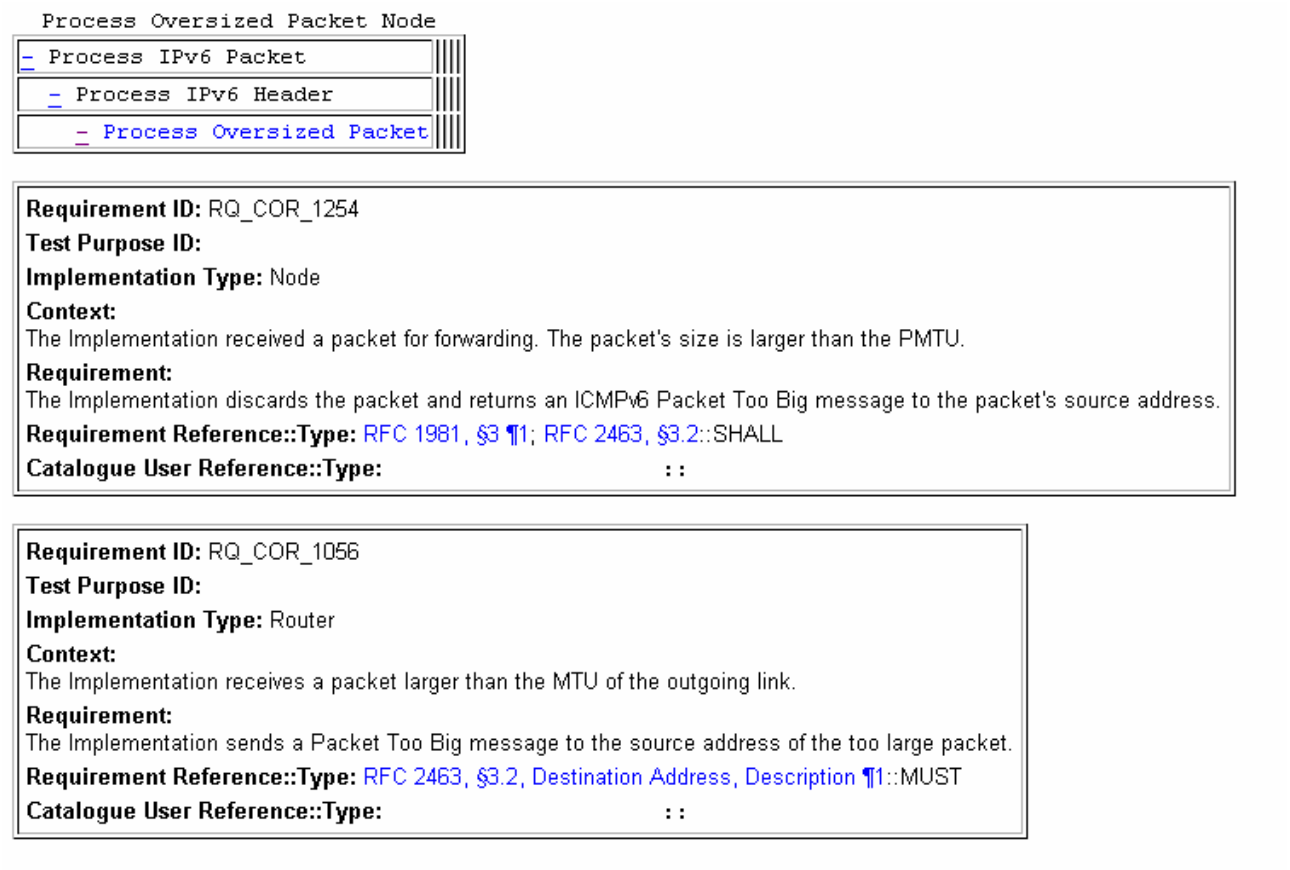


Figure D.3: Process Oversized Packet Node

Figure D.2 shows the "Process IPv6 Header" node of the Requirements Catalogue. The function associated with this node is "Process Oversized Packet". The parent node's function is "Process IPv6 Packet". The node has six children whose functions are:

- "Process Oversized Packet"";
- "Process Hop Limit";
- "Process Next Header";
- "Process Destination Address";
- "Process Source Address"; and
- "Process Invalid Packet".

The "+" symbol indicates that the child nodes can be expanded. The "-" symbol indicates the nodes that can be collapsed. There are no specific requirements associated with this node. The display for each node shows all the node's ancestors (parents, grandparents, etc.) and the node's children; i.e. a branch.

Figure D.3 shows the result of expanding figure D.2's "Process Oversized Packet" node. The branch display shows that this is a leaf node of the tree. There are no children nodes below it. Figure D.3 also shows that there are two specific requirements associated with the "Process Oversized Packet" function.

Other information that may be extracted from the data base could be a TP checklist. Such a checklist maps a specific requirement to one or more Test Purposes for that requirement. The numbering and naming conventions of clauses 6.2 and 7.1.2.2.1 ensure consistency between requirements numbering and Test Purpose numbering. Figure D.4 serves as a useful summary as well as acting as a checklist for a particular IUT to indicate what requirements are supported. The IUT Support column is filled in at the time of testing (hence shown in grey).

Organization: IPv6 Label		
Requirement	**TP**	**IUT Support**
RQ_COR_0001	TP_COR_0001_01 TP_COR_0001_02	Yes
RQ_COR_0002	TP_COR_0002_01	No
RQ_COR_0003	TP_COR_0003_01 TP_COR_0003_02 TP_COR_0003_03	Yes
...	...	...

Figure D.4: Example TP checklist

History

Document history		
V1.1.1	September 2004	Publication
V2.1.1	August 2005	Publication