ETSI ES 202 781 V1.7.1 (2019-04)

Methods for Testing and Specification (MTS);

The Testing and Test Control Notation version 3;

TTCN-3 Language Extensions:

Configuration and Deployment Support

**ETSI Standard**

Reference

RES/MTS-202781ConfDeplv171

Keywords

protocol, testing, TTCN-3

***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

***Important notice***

The present document can be downloaded from:  
<http://www.etsi.org/standards-search>

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the prevailing version of an ETSI deliverable is the one made publicly available in PDF format at [www.etsi.org/deliver](http://www.etsi.org/deliver).

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at <https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx>

If you find errors in the present document, please send your comment to one of the following services:  
<https://portal.etsi.org/People/CommiteeSupportStaff.aspx>

***Copyright Notification***

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.  
The content of the PDF version shall not be modified without the written authorization of ETSI.  
The copyright and the foregoing restriction extend to reproduction in all media.

© ETSI 2019.

All rights reserved.

**DECT**TM, **PLUGTESTS**TM, **UMTS**TM and the ETSI logo are trademarks of ETSI registered for the benefit of its Members.  
**3GPP**TM and **LTE**TM are trademarks of ETSI registered for the benefit of its Members and  
of the 3GPP Organizational Partners.  
**oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and  
of the oneM2M Partners.  
**GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Contents

Intellectual Property Rights 6

Foreword 6

Modal verbs terminology 6

1 Scope 7

2 References 7

2.1 Normative references 7

2.2 Informative references 7

3 Definition of terms, symbols and abbreviations 8

3.1 Terms 8

3.2 Symbols 8

3.3 Abbreviations 8

4 Package conformance and compatibility 8

5 Package Concepts for the Core Language 9

5.0 General 9

5.1 Static configurations 10

5.1.1 The special configuration type: configuration 10

5.1.2 The configuration function 11

5.1.3 Starting a static test configuration 12

5.1.4 Destruction of static test configurations 12

5.1.5 Creation of static test components 13

5.1.6 Establishment of static connections and static mappings 14

5.1.7 Test case definitions for static test configuration 14

5.1.8 Executing test cases on static test configurations 15

5.1.9 Further restrictions 17

5.1.10 Logging the status of static configurations 17

5.2 Ports with translation capability 17

5.2.0 General 17

5.2.1 Translation capability in port type declaration 19

5.2.2 Mapping and connecting ports 20

5.2.3 Translation functions 20

5.2.4 Translation state 21

5.2.5 Sending 23

5.2.6 Receiving 23

5.2.7 Address 25

5.2.8 Clear, start, stop and halt operation 25

5.2.9 The outer port reference 26

6 Package Semantics 27

6.0 General 27

6.1 Replacement of short forms 29

6.2 Order of replacement steps 29

6.3 Flow graph representation of TTCN-3 behaviour 30

6.4 Flow graph construction procedure 30

6.5 Flow graph representation of configuration functions 31

6.6 Retrieval of start nodes of flow graphs 32

6.7 Module state 32

6.8 Accessing the module state 33

6.9 Configuration state 33

6.10 Accessing the configuration state 33

6.11 Entity states 34

6.12 Accessing entity states 36

6.13 Handling of connections among ports 37

6.14 Handling of port states 37

6.15 Void 39

6.16 Evaluation phases, general 39

6.17 Phase I: Initialization 39

6.18 Phase II: Update 40

6.19 Phase III: Selection 40

6.20 Phase IV: Execution 41

6.21 Global functions 41

6.22 Clear port operation 41

6.23 Configuration function call 42

6.24 Connect operation 43

6.25 Create operation 44

6.26 Flow graph segment <disconnect-all> 46

6.27 Flow graph segment <disconnect-comp> 47

6.28 Flow graph segment <disconnect-port> 48

6.29 Flow graph segment <disconnect-two-par-pairs> 48

6.30 Execute statement 49

6.31 Flow graph segment <execute-without-config> 50

6.32 Flow graph segment <execute-on-config> 50

6.33 Flow graph segment <execute-on-config-without-timeout> 50

6.34 Flow graph segment <execute-on-config-timeout> 51

6.35 Flow graph segment <statement-block> 53

6.36 Halt port operation 54

6.37 Kill component operation 55

6.38 Flow graph segment <kill-mtc> 57

6.39 Flow graph segment <kill-all-comp> 57

6.40 Kill execution statement 59

6.41 Kill configuration operation 60

6.42 Map operation 60

6.43 Start port operation 61

6.44 Stop component operation 62

6.45 Flow graph segment <stop-mtc> 64

6.46 Flow graph segment <stop-config> 65

6.47 Flow graph segment <stop-tc-config> 66

6.48 Stop port operation 67

6.49 Flow graph segment <unmap-all> 68

6.50 Flow graph segment <unmap-comp> 69

6.51 Flow graph segment <unmap-port> 70

7 TRI Extensions for the Package 70

7.1 Changes and extensions to clause 5.5.2 of ETSI ES 201 873‑5 [3] Connection handling operations 70

7.2 Extensions to clause 6 of ETSI ES 201 873-5 [3] JavaTM language mapping 72

7.3 Extensions to clause 7 of ETSI ES 201 873-5 [3] ANSI C language mapping 72

7.4 Extensions to clause 8 of ETSI ES 201 873-5 [3] C++ language mapping 72

7.5 Extensions to clause 9 of ETSI ES 201 873-5 [3] C# language mapping 73

8 TCI Extensions for the Package 73

8.1 Extensions to clause 7.2.1.1 of ETSI ES 201 873-6 [4] Management 73

8.2 Extensions to clause 7.3.1.1 of ETSI ES 201 873-6 [4] TCI‑TM required 73

8.3 Extensions to clause 7.3.1.2 of ETSI ES 201 873-6 [4] TCI‑TM provided 75

8.4 Extensions to clause 7.3.3.1 of ETSI ES 201 873-6 [4] TCI‑CH required 75

8.5 Extensions to clause 7.3.3.2 of ETSI ES 201 873-6 [4] TCI CH provided 76

8.6 Extensions to clause 7.3.4 of ETSI ES 201 873-6 [4] TCI‑TL provided 77

8.7 Extensions to clause 8 of ETSI ES 201 873-6 [4] JavaTM language mapping 79

8.8 Extensions to clause 9 of ETSI ES 201 873-6 [4] ANSI C language mapping 81

8.9 Extensions to clause 10 of ETSI ES 201 873-6 [4] C++ language mapping 83

8.10 Extensions to clause 11 of ETSI ES 201 873-6 [4] W3C XML mapping 85

8.11 Extensions to clause 12 of ETSI ES 201 873-6 [4] C# language mapping 88

Annex A (normative): BNF and static semantics 90

A.1 Additional TTCN‑3 terminals 90

A.2 Modified TTCN‑3 syntax BNF productions 90

A.3 Additional TTCN‑3 syntax BNF productions 91

Annex B (informative): Library of useful types 93

B.1 Limitations 93

B.2 Useful TTCN‑3 types 93

B.2.1 Status values for port states 93

History 94

# Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables 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 (<https://ipr.etsi.org/>).

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.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

# Foreword

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

**The use of underline (additional text) and strike through (deleted text) highlights the differences between base document and extended documents.**

The present document relates to the multi-part series ETSI ES 201 873 covering the Testing and Test Control Notation version 3, as identified below:

Part 1: "TTCN‑3 Core Language";

Part 4: "TTCN‑3 Operational Semantics";

Part 5: "TTCN‑3 Runtime Interface (TRI)";

Part 6: "TTCN‑3 Control Interface (TCI)";

Part 7: "Using ASN.1 with TTCN‑3";

Part 8: "The IDL to TTCN-3 Mapping";

Part 9: "Using XML schema with TTCN‑3";

Part 10: "TTCN-3 Documentation Comment Specification";

Part 11: "Using JSON with TTCN-3".

# Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the [ETSI Drafting Rules](https://portal.etsi.org/Services/editHelp!/Howtostart/ETSIDraftingRules.aspx) (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

# 1 Scope

The present document defines the Configuration and Deployment Support package of TTCN‑3. TTCN‑3 can be used for the specification of all types of reactive system tests over a variety of communication ports. Typical areas of application are protocol testing (including mobile and Internet protocols), service testing (including supplementary services), module testing, testing of APIs, etc. TTCN‑3 is not restricted to conformance testing and can be used for many other kinds of testing including interoperability, robustness, regression, system and integration testing. The specification of test suites for physical layer protocols is outside the scope of the present document.

TTCN‑3 packages are intended to define additional TTCN-3 concepts, which are not mandatory as concepts in the TTCN-3 core language, but which are optional as part of a package which is suited for dedicated applications and/or usages of TTCN-3.

This package defines the TTCN-3 support for static test configurations.

While the design of TTCN‑3 package has taken into account the consistency of a combined usage of the core language with a number of packages, the concrete usages of and guidelines for this package in combination with other packages is outside the scope of the present document.

# 2 References

## 2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non‑specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

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

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.

[1] ETSI ES 201 873-1: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language".

[2] ETSI ES 201 873-4: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics".

[3] ETSI ES 201 873-5: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)".

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

[5] ISO/IEC 9646-1: "Information technology - Open Systems Interconnection -Conformance testing methodology and framework; Part 1: General concepts".

## 2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non‑specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI ES 201 873-7: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 7: Using ASN.1 with TTCN-3".

[i.2] ETSI ES 201 873-8: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 8: The IDL to TTCN-3 Mapping".

[i.3] ETSI ES 201 873-9: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 9: Using XML schema with TTCN-3".

[i.4] ETSI ES 201 873-10: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 10: TTCN-3 Documentation Comment Specification".

# 3 Definition of terms, symbols and abbreviations

## 3.1 Terms

For the purposes of the present document, the terms given in ETSI ES 201 873-1 [1], ETSI ES 201 873‑4 [2], ETSI ES 201 873-5 [3], ETSI ES 201 873-6 [4] and ISO/IEC 9646-1 [5] apply.

## 3.2 Symbols

Void.

## 3.3 Abbreviations

For the purposes of the present document, the abbreviations given in ETSI ES 201 873-1 [1], ETSI ES 201 873‑4 [2], ETSI ES 201 873-5 [3], ETSI ES 201 873-6 [4], ISO/IEC 9646-1 [5] and the following apply:

MTC Main Test Component

PTC Parallel Test Component

# 4 Package conformance and compatibility

The package presented in the present document is identified by the package tag:

"TTCN-3:2009 Static Test Configurations" - to be used with modules complying with the present document*.*

For an implementation claiming to conform to this package version, all features specified in the present document shall be implemented consistently with the requirements given in the present document, in ETSI ES 201 873‑1 [1] and in ETSI ES 201 873‑4 [2].

The package presented in the present document is compatible to:

* ETSI ES 201 873-1 [1] version 4.9.1;
* ETSI ES 201 873-4 [2] version 4.6.1;
* ETSI ES 201 873-5 [3] version 4.8.1;
* ETSI ES 201 873-6 [4] version 4.9.1;
* ETSI ES 201 873-7 [i.1];
* ETSI ES 201 873-8 [i.2];
* ETSI ES 201 873-9 [i.3];
* ETSI ES 201 873-10 [i.4].

If later versions of those parts are available and should be used instead, the compatibility to the package presented in the present document has to be checked individually.

# 5 Package Concepts for the Core Language

## 5.0 General

This package defines the TTCN-3 means to define *static test configurations*. A static test configuration is a test configuration with a lifetime that is not bound to a single test case. The test components of a static test configuration may be used by several test cases. This package realizes the following concepts:

* A special *configuration function* is introduced which can only be called in the control part of a TTCN-3 module to create *static test configurations*. The configuration function returns a handle of the predefined type**configuration** to access an existing static test configuration.
* A static test configuration consists of *static test components*, a test system interface, *static connections* and *static mappings*. These constituents have the following semantics:
* A *static test component* is a special kind of test component that can only be created during the creation of a static test configuration and can only be destroyed during the destruction of a static test configuration. By definition, the MTC of a static test configuration is a static test component.
* The test system interface of a static test configuration plays the same role as the test system interface of a test configuration created by a test case.
* A *static connection* is a connection between static test components. It can only be established during the creation of a static test configuration and only be destroyed during the destruction of a static test configuration.
* A *static mapping* is a mapping of a port of a static test component to a port of the test system interface of a static test configuration. Such a mapping can only be established during the creation of a static test configuration and only be destroyed during the destruction of a static test configuration.
* A static test configuration can be used by several test cases. For this the test case is started on a previously created static test configuration. This means:
* The body of the test case is executed on the MTC of the static test configuration.
* The MTC may start behaviour on other static test components of the static test configuration.
* Static test components may create, start, stop and kill normal and alive test components. The lifetime of these components is bound to the actual test case that is executed on the static test configuration. In case that a normal and alive test component is not destroyed explicitly by another test component, it is implicitly destroyed when the test case ends.
* During test case execution non-static connections and non-static mappings may be established. The lifetime of non-static connections and non-static mappings is bound to the actual test case that is executed on the static test configuration. In case that a non-static connection or a non-static mapping is not destroyed explicitly by another test component, it is implicitly destroyed when the test case ends.
* Component timers and variables of static test components are not reset or reininitialized when a test case is started on a static test configuration. They remain in the same state as when they were left after the creation of the static test configuration or after the termination of a previous test case. This allows to transfer information from one test case to another.
* Ports of static test components are not emptied or restarted when a test case is started on a static test configuration. For example, this allows a delayed handling of SUT responses like e.g. repetitive status messages, during the test campaign. In addition, all port operations (i.e. **clear**, **start**, **stop** and **halt**) are disallowed for ports of static test components. All ports of a static test component remain started during the whole lifetime of a static test configuration.
* In contrast to component timers, variables and ports, the verdict and the default handling is reset. This means all activated defaults are deactiviated, all local verdicts and the global verdict are set to **none**.

## 5.1 Static configurations

### 5.1.1 The special configuration type: configuration

The special configuration type **configuration** is a handle for static test configurations. The special value **null** is available to indicate an undefined configuration reference, e.g. for the initialization of variables to handle a static test configuration.

Values of type **configuration** shall be the result of configuration functions, they can be checked for equality, e.g. to check if two variables store the same value, and they can be used in **execute** statements for starting a test case on an existing static test configuration and in **kill** configuration statements to destroy an existing static test configuration.

Each successful execution of a configuration function results in a different configuration value which is only equal to itself.

***Restrictions***

The following restrictions apply to usages of the **configuration** type:

1. The **configuration** type cannot be subtyped or constrained.
2. The **configuration** type is not a data type, therefore, the **anytype** does not include the configuration type.
3. Module parameters shall not be of type **configuration**.
4. Signature parameters shall not be of type **configuration**.
5. Templates shall not be of type **configuration**.
6. Templates shall not be of a structured type that contains fields or elements of type **configuration** on any level of nesting.
7. External functions are not allowed to contain parameters or return values of type **configuration**.

EXAMPLES:

**var configuration** myStaticConfig := **null**; // Declaration and initialization of a

// configuration variable.

myStaticConfig := aStaticConfig(); // Assigns a value to the previously declared

// configuration variable. It is assumed that

//aStaticConfig() is a configuration function.

myStaticConfig.**kill** // Kills the static test configuration stored in

// variable myStaticConfig.

### 5.1.2 The configuration function

A configuration function allows the start of a static test configuration.

***Syntactical Structure***

**configuration** *ConfigurationIdentifier*

"(" [ { ( *FormalValuePar* | *FormalTemplatePar*) [","] } ] ")"

**runs** **on** *ComponentType*

[ **system** *ComponentType* ]

StatementBlock

***Semantic Description***

A configuration function allows the start of a static test configuration. A configuration function has to be defined in the definitions part of a TTCN-3 module and shall only be invoked in the control part of a TTCN-3 module. By definition, a configuration function returns a value of type **configuration** if the start of the configuration was successful, or **null** if the start of the configuration was not successful.

The invocation of a configuration function causes the creation of the MTC and the test system interface of the static test configuration. The types of MTC and test system interface shall be referenced in a **runs on** and a **system** clause. The **system** clause is optional and can be omitted, if the test system has exactly the same ports as the MTC and these ports are mapped one to one to each other.

The behaviour in the body of a configuration function shall be executed on the newly created MTC. During the start of a test configuration only behaviour on the MTC shall be executed and only static test components, static connections and static mappings shall be created or established. Communication with the SUT or with static PTCs is not allowed.

NOTE:The configuration function only returns a reference to a test configuration and no verdict. However, communication with the SUT might have to be checked. For this purpose, initial communication, e.g. for registration or coordination purposes, could be defined in form of a test case.

A static test configuration is successfully started if the behaviour of the corresponding configuration function has been executed till its end or if a **return** statement in the corresponding configuration function is reached. In case of a successful start, a reference to the newly created configuration is returned. The usage of a **stop** or a **kill** statement allows to specify an unsuccessful start of a static test configuration. In case of an unsuccessful start, the value **null** is returned.

***Restrictions***

1. The rules for formal parameter lists for the configuration function shall be followed as defined in clause 5.4 of ETSI ES 201 873‑4 [2].
2. Configuration functions shall only be invoked in the module control part.
3. For the behaviour definition in the body of the configuration function and all functions directly or indirectly from the configuration function, the following restrictions shall hold:

* Only static test components, static connections and static mappings shall be created or established. All created test components, connections and mappings during the execution of a configuration function are static.
* Once created or established static test components, static connections and static mappings shall not be destroyed.
* It is not allowed to start behaviour on newly created static test components.
* Communication, timer and port operations are not allowed.

EXAMPLES:

// The following configuration function can be used to start a simple static test configuration

// which only consists of one MTC.

**configuration** simpleStaticConfig () **runs on** MyMTCtype{}

// The following configuration function starts a more complex static configuration.

// Configuration information is stored in MTC component variables. Further non-static

// connections and mappings may be established by the test cases that are executed

// on this configuration.

**configuration** aComplexStaticConfig (**in integer** NoOfPTCs) **runs on** MyMTCtype **system** MySystemType {

**var integer** i;

**if** (NoOfPTCs < 0) {

**log** ("Negative number of PTCs");

**kill**; // unsuccessful termination

}

**else** **if** (NoOfPTCs > MaxNoOfPTCs) { // MaxNoOfPTCs is a constant

**log** ("Number of PTCs is too high");

**kill**; // unsuccessful termination

}

**else** {

**for** (i := 1, i <= NoOfPTCs, i := i + 1) {

PTC[i] := PtcType.**create** **static**; // creation of static PTCs,

// Array PTC[] is a component variable

**connect** (**mtc**:SyncPort, PTC[i]:SyncPort) **static**; // static connection

}

**map**(**mtc**:PCO, **system**:PCO1) **static**; // static mapping of MTC.

**map**(PTC[1]:PCO, **system**:PCO2); // some static mappings of PTCs,

**map**(PTC[2]:PCO, **system**:PCO3); // further non-static mappings may be

// established during test runs

}

**return**; // successful termination

}

### 5.1.3 Starting a static test configuration

A static test configuration is started by calling a configuration function in the control part of a TTCN-3 module. In case of a successful start, a reference to the newly created static test configuration is returned. In case of an unsuccessful start, the special value null is returned.

EXAMPLES:

**control** {

**var configuration** myStaticConfig := **null**; // Declaration and initialization of a

// configuration variable.

myStaticConfig := aStaticConfig(); // Assigns a value to the previously declared

// configuration variable. It is assumed that

// aStaticConfig() is a configuration function.

**if** (myStaticConfig == **null**) {

**stop**; // Stop test campaign due to an unsuccessful start

}

**else** {

**execute**(MyTestCase(),myStaticConfig) // Successful start, continuation of test campaign

...

}

}

### 5.1.4 Destruction of static test configurations

A static test configuration can be destroyed by executing a **kill** configuration operation.

***Syntactical Structure***

*ConfigurationReference.***kill**

***Semantic Description***

The execution of a **kill** configuration operation causes the destruction of a static test configuration. The destruction is similar to stopping a test case by killing the MTC. This means, resources of all static PTCs shall be released and the PTCs shall be removed. The only difference is that no test verdict is calculated and returned. After executing the **kill** configuration operation, it is not possible to execute a test case on the killed static test configuration.

Executing the kill configuration operation with the special value **null** shall have no effect, executing a kill configuration operation with a reference to a non-existing static test configuration shall cause a runtime error.

***Restrictions***

1. The **kill** configuration operation shall only be executed in the control part of a TTCN-3 module.

EXAMPLES:

**control** {

**var configuration** myStaticConfig := **null**; // Declaration and initialization of a

// configuration variable.

myStaticConfig := aStaticConfig(); // Assigns a value to the previously declared

// configuration variable. It is assumed that

// aStaticConfig() is a configuration function.

myStaticConfig.**kill** // Destruction of the previously started static

// test configuration.

### 5.1.5 Creation of static test components

All create operations invoked directly or indirectly from configuration functions create static test components. The creation of static test components can be indicated by the additional optional keyword **static** in the **create** operation. The extension of the **create** operation in clause 21.2.1 of ETSI ES 201 873‑4 [2] required for the creation of static test components is described in the following clauses.

***Syntactical Structure***

*ComponentType* "." **create** [ "(" (*Name* | "-") ["," *HostId*] ")" ] [ **alive** | **static** ]

***Semantic Description***

The **create** operation in combination with the keyword **static** shall only be used to create static test components. Static test components can only be created by executing a configuration function and by functions directly or indirectly invoked by configuration functions. The keyword **static** in a **create** operation shall not be used in combination with the keyword **alive**.

NOTE 1: During the lifetime of a static test configuration, a static component behaves like an alive component.

Static test components are created in the same manner as normal test components that are not declared as alive components. Further details on this can be found in clause 21.2.1 of ETSI ES 201 873‑4 [2].

NOTE 2: Static test components can only be created directly or indirectly by a configuration function. This may be checkable at runtime and therefore the keyword static may not be required, but for having an explicit specification of static test configurations and for keeping the feature of static test configurations extendible, the keyword **static** has been introduced.

***Restrictions***

1. The **create** operation in combination with the keyword **static** shall only be invoked in configuration functions and in function that may be directly or indirectly called by such a configuration function.
2. The keyword **static** in a **create** operation shall not be used in combination with the keyword **alive**.

EXAMPLES:

// This example declares variables of type MyComponentType, which are used to store the

// references of newly created static component instances of type MyComponentType.

// An associated name is allocated to some of the created component instances.

:

**var** MyComponentType MyNewComponent;

**var** MyComponentType MyNewestComponent;

:

MyNewComponent := MyComponentType.**create static**;

MyNewestComponent := MyComponentType.**create**("Newest") **static**;

### 5.1.6 Establishment of static connections and static mappings

The **map** and **connect** operations called directly or indirectly from configuration functions establish static connections and static mappings. This can be indicated by the additional optional keyword **static** in **connect** and the **map** operations. The extension of the **connect** and **map** operation in clause 21.1.1 of ETSI ES 201 873‑4 [2] required for the establishment of static connections and mapping is described in the following clauses.

***Syntactical Structure***

**connect** "(" *ComponentRef* ":" *Port* "," *ComponentRef* ":" *Port* ")" [ **static** ]

**map** "(" *ComponentRef* ":" *Port* "," *ComponentRef* ":" *Port* ")"

[ **param** "(" [ { *ActualPar* [","] }+ ] ")" ] [ **static** ]

***Semantic Description***

The **connect** and **map** the operation in combination with the keyword **static** shall only be used to establish static connections and static mappings. Static connections and static mappings can only be established by executing the creator function of a configuration type and by functions directly or indirectly invoked by the creator functions of configuration type.

Static connections and static mappings are established in the same manner as normal connections and mappings. Further details on this can be found in clause 21.1.1 of ETSI ES 201 873‑4 [2].

NOTE:Static connections and mappings can only be established directly or indirectly by a creator function of a configuration type. This may be checkable at runtime and therefore the keyword **static** may not be required, but for having an explicit specification of static test configurations and for keeping the feature of static test configurations extendible, the keyword **static** has been introduced.

***Restrictions***

1. The **connect** and **map** operation in combination with the keyword **static** shall only be used in configuration functions and in functions that may be directly or indirectly called by a configuration function.
2. Static connections and static mappings shall only be established to connect ports of static test components and to map ports of a static component to the ports of the test system interface of a configuration type.

EXAMPLES:

// The following code fragment may be part of a creator function of a configuration type.

// It is assumed that the ports Port1, Port2, Port3 and PCO1 are properly defined and declared

// in the corresponding port type and component type definitions

:

**var** MyComponentType MyNewPTC;

MyNewPTC := MyComponentType.**create static**;

:

**connect**(MyNewPTC:Port1, **mtc**:Port3) **static**;

**map**(MyNewPTC:Port2, **system**:PCO1) **static**;

:

### 5.1.7 Test case definitions for static test configuration

Test cases that are executed on a static test configuration have to be defined in a special manner. Such test cases shall reference the configuration function that starts a static configuration on which the test case can be executed. The type of the MTC and the type of the test system interface are referenced in the configuration function and shall therefore not be specified in the test case header. The extension of the test case definition in clause 16.3 of ETSI ES 201 873‑4 [2] required for the execution of a test case on a static test configuration is described in the following clauses.

***Syntactical Structure***

**testcase** *TestcaseIdentifier*

"(" [ { ( *FormalValuePar* | *FormalTemplatePar*) [","] } ] ")"

( **runs** **on** *ComponentType* [ **system** *ComponentType* ] | **execute on** *ConfigurationType* )

StatementBlock

***Semantic Description***

A test case definition that includes an **execute on** clause will be executed on previously created static test configuration of the given configuration type. The type of the MTC and the type of the test system interface is defined in the referenced configuration type. A test case definition that includes an **execute on** clause shall not have a **runs** **on** or a **system** clause.

Apart from the execute on clause, the definition of test cases to be executed on a static test configuration follows the same rules as described in clause 16.3 of ETSI ES 201 873‑4 [2].

***Restrictions***

1. A test case definition that includes an **execute on** clause shall not have a **runs** **on** or a **system** clause.

EXAMPLES:

**configuration** aConfiguration () **runs on** MyMTCtype **system** MySystemType {

PeerComponent := MyPTCType.**create static**; // creation of a static PTC

// PeerComponent is a component variable

**connect**(**mtc**:syncPort, PeerComponent:syncPort); // static connection

**map** (**mtc**:PCO1, **system**:PCO1) // static mapping of MTC

**map** (PeerComponent:PCO2, **system**:PCO2); // static mapping of Peer Component

**return**  // successful start of test configuration

}

**testcase** MyTestCase () **execute on** aConfiguration {

default := activate(UnexpectedReceptions()); // activate a default

PeerComponent.**start** (PTCbehaviour()); // starting PTC behaviour

SyncPort.**send** (Ready); // synchronization with PTC

SyncPort.**receive**(Ready); // PTC ready

PCO1.**send** (stimulus); // test starts

... // test behaviour

}

### 5.1.8 Executing test cases on static test configurations

This clause only describes the syntax extensions of the **execute** statement to allow the execution of test cases with an **execute on** clause on static test configurations and the semantics for executing such test cases. The semantics of the **execute** statement for test cases without **execute on** clause remains unchanged.

***Syntactical Structure***

**execute** "(" *TestcaseRef* "(" [ { *TemplateInstance* [","] } ] ")"

[ "," (*TimerValue |* "-"*)*

["," (*HostId |* "-"*)*

[ "," *ConfigurationRef* ] ] ] ")"

***Semantic Description***

A test case definition that includes an **execute on** clause shall be executed on previously started static test configuration of a given configuration function. The reference of the previously started static test configuration shall be referenced in the **execute** statement.

Trying to execute a test case on a non-existing or unfitting static test configuration shall cause a run time error. Unfitting test configuration means that the referenced static test configuration has not been created by the configuration function referenced in the test case header.

If the execution of a test case on a static test configuration causes an **error** verdict, all following usages of this static test configuration in **execute** statements shall cause a runtime error.

NOTE:It is allowed to kill the possibly erroneous static test configuration and to start a new one by invoking the configuration function again.

A test case that shall be started on a fitting static test configuration can rely on the following things:

* All static test components, static connections and static mappings created or established by the referenced configuration function shall exist.
* No static test component is running.
* No non-static test components, non-static connections and non-static mappings shall exist.
* Component timers and variables of static test components shall not be reset or reininitialized when a test case is started on a static test configuration. They remain in the same state as when they were left after the creation of the static test configuration or after the termination of a previous test case, except for running timers which can change their state to timed out. This allows to transfer information from one test case to another. If a timer of a static component is running when a test case terminates, it can still time out even before the next test case starts. However, this can only be observed during the execution of a testcase.
* Ports of static test components shall not emptied or restarted when a test case is started on a static test configuration. For example, this allows a delayed handling of SUT responses like e.g. repetitive status messages, during the test campaign. Messages, calls, replies, exceptions and call-timeouts can still be enqueued at ports of static test components after the termination of a testcase, but they can only be observed and processed during a following testcase.
* In contrast to component timers, variables and ports, the verdict and the default handling shall be reset. This means all activated defaults are deactivated, all local verdicts and the global verdict are set to **none**.

Executing a test case on a static test configuration means that the body of the test case is executed on the MTC of the static test configuration. During test execution, all static PTCs behave like alive test components. This means, static PTCs may be stopped and started several times. During test case execution, non-static normal and alive components may be created, started, killed and stopped. In addition, non-static connections and mappings may be established and destroyed.

A test case that is executed on a static test configuration shall end when the behaviour of the MTC ends. In this case, the final test case verdict is returned. The final test case verdict shall be calculated based on the local verdicts of all static and non-static test components. Furthermore, all test components (static and non-static) shall be stopped, all non-static test components, non-static connections and all non-static mappings shall be discarded.

***Restrictions***

All restrictions mentioned in clause 26.1 of the core language document [1] apply.

EXAMPLES:

**var verdict** MyVerdict // local variable

**var configuration** MyConfiguration := aConfiguration(); // starting a static test configuration

MyVerdict := execute(MyTestCase (),MyConfiguration); // execution of a test case on a static

// test configuration

**if** (MyVerdict :== **pass**) {

MyVerdict := **execute** MyTestCase (), 10.0, MyConfiguration); // executing the same test case

// with time guard

}

... // further test behaviour

**stop**;

### 5.1.9 Further restrictions

Static test components, static connections and static mappings have a special semantics. Therefore, situations shall cause a runtime error:

* Applying a **kill** test component operation to a static test component.
* Applying port operations (**clear**, **start**, **stop** and **halt**) to a port owned by a static test component.
* Applying a **disconnect** operation to a static connection.
* Applying **unmap** operation to a static mapping.

### 5.1.10 Logging the status of static configurations

The **log** statement can be used to log the status of static configurations. Table 17 "TTCN 3 language elements that can be logged" of ETSI ES 201 873-1 [1] is to be extended as follows:

Table 1: TTCN‑3 language elements that can be logged

| Used in a log statement | What is logged | Comment |
| --- | --- | --- |
| … | … | … |
| configuration reference | actual state | Configurations states shall be logged according to note 9. |
| NOTE ..: …  NOTE 9: Configuration states that can be logged are: Started and Killed. | | |

## 5.2 Ports with translation capability

### 5.2.0 General

This clause describes an extension of a message port type definition adding translation capability into it.

Translation feature is a set of rules that allows to convert messages and/or addresses of one type into messages and/or addresses of different type during sending or receiving.

It can be used e.g. in situations where the test behaviour is defined on one set of data types but the system under test (or connected component) actually communicates using a different set of data types, i.e. if the test system works on a different layer of the protocol stack than the system under test.

To allow flexible adaptation to the system under test, the user shall have the means to control this translation in the abstract test suite.

***Syntactical Structure***

**type** **port** *PortTypeId*message

[ **map** **to** { *OuterPortType* [ "," ] }+ ]

[ **connect** **to** { *OuterPortType* [ "," ] }+ ] "{"

{

( **in** { *InnerInType* [ **from** { *OuterInType* **with** *InFunction* "(" ")" [ "," ] }+ ] [ "," ] }+ |

**out** { *InnerOutType* [ **to** { *OuterOutType* **with** *OutFunction* "(" ")" [ "," ]}+ ] [ "," ] }+ |

**inout** { *InOutType* [ "," ] }+ *|*

**address** *AddrType* [ **to** { *OuterAddrType*with *AddrOutFunction* "(" ")" [ "," ] }+ ]

[ **from** { *OuterAddrType*with *AddrInFunction*"("")"[","]}+ ] |

**map** **param** "(" { *FormalValuePar* [ "," ] }+ ")"|

**unmap** **param** "(" { *FormalValuePar* [ "," ] }+ ")" |

*VarInstance* ) ";"

}+

"}"

NOTE: Please note that the same *OuterInType* may appear in more than one **in** message specifications for different *InnerInType*-s. In each such clause the *InFunction* is different.

***Semantic Description***

*PortTypeId* is name of the type being defined.

Port in translation mode

**Translation behaviour**

**Test System Interface**

**Standard port behaviour**

***OutFunction*** is implicitly invoked

**Outer** in message (of type *OuterInType*)

**SUT**

*InFunction* is implicitly invoked

Inner out message (of type *InnerOutType*)

Inner in message (of type *InnerInType*)

IN

IN

OUT

OUT

Outer out message (of type *OuterOutType*)

Inner queue

Outer queue

Figure : Illustration of ports with translation capability

* *OuterPortType* references the outer message port type this port is mapped to. If the referenced port is a mapped port, it shall not contain direct or indirect reference to the *PortTypeId* in the list of its *OuterPortTypes*.
* *InnerInType* references a type that can be received over such a port.
* *OuterInType* references a type that is actually received and which shall be translated to *InnerInType*.
* *InFunction* references a function which shall be used to translate *OuterInType* to *InnerInType*.
* *InnerOutType* references a type that can be sent over such a port.
* *OuterOutType* references a type that is actually sent which has been translated from *InnerOutType*.
* *OutFunction* references a function which shall be used to translate *InnerOutType* to *OuterOutType*.
* *InOutType* references a type that can be sent and received by the port.
* *AddrType* is the address type bound to the port type being defined.
* *OuterAddrType* is the address type into which the *AddrType* is translated.
* *AddrOutFunction* references a function which shall be used to translate the *AddrType* to the*OuterAddrType.*
* *AddrInFunction* references a function which shall be used to translate the *OuterAddrType* to the*AddrType*.
* *VarInstance* is a declaration of a port variable.

### 5.2.1 Translation capability in port type declaration

If a port type declaration includes translation capability, it shall always contain at least one map or connect clause. These clauses define one or more port types for which translation mechanism is defined.

If a port type is referenced in the map clause, the following applies:

* All types from the **in** message list of the *OuterPortType* shall be referenced either as *InnerInType, OuterInType* or *InOutType* in the port type with translation capability.
* All *InOutTypes* shall be present either in the **in** and **out** lists (at the same time) or in the **inout** message list of the *OuterPortType.*
* All *InnerOutTypes* shall be referenced in the out message list of the *OuterPortType* or if such a reference does not exist, the *OuterPortType* shall contain at least one reference to any of the *OuterOutTypes* associated with the *InnerOutType* in its **out** message list.

NOTE 1:If these conditions are met, it is always safe to map TSI ports of *OuterOutType* to instances of the port type with translation capability.

If a port type is referenced in the connect clause, the following applies:

* All types from the out message list of the *OuterPortType* shall be referenced either as *InnerInType, OuterInType* or *InOutType* in the port type with translation capability.
* All *InOutTypes* shall be present either in the **in** and **out** lists (at the same time) or in the **inout** message list of the *OuterPortType.*
* All *InnerOutTypes* shall be referenced in the **in** message list of the *OuterPortType* or if such a reference does not exist, the *OuterPortType* shall contain at least one reference to any of the *OuterOutTypes* associated with the *InnerOutType* in its **in** message list.

NOTE 2:If these conditions are met, it is always safe to connect ports with translation capability to ports of *OuterOutType*.

Port types with translation capability can contain variable declarations. These variables are created and initialized when a port instance is created and have the same lifetime as the port instance itself. Every port instance has its own copy of these variables. Port variables can be accessed only from *InFunctions* and *OutFunctions.* They are not visible outside of the translation procedure. The variables can be used e.g. for buffering data between individual calls of *InFunctions* and *OutFunctions*(e.g. in case of fragmented messages).

***Restrictions***

In addition to the general static rules of TTCN-3 restrictions specified in clause 6.2.9 of ETSI ES 201 873-1 [1], the following restrictions apply:

1. If the *OuterPortType* is a port type with translation capability, it shall neither directly nor indirectly reference *PortTypeId* in its map or connect clause (i.e. port types with translation capability cannot reference each other).
2. All *OuterAddrTypes* shall be used as an address type at least in one of the *OuterPortTypes*.
3. All *InFunction*, *OutFunction* and *AddrFunction* identifiers shall be references to a translation function.

EXAMPLE:

**type port** TransportPort

{

**inout** TransportMessage;

}

**type port** DataPort **map to** TransportPort

{

**in** DataMessage **from** TransportMessage **with** transportToData();

**out** DataMessage **to** TransportMessage **with** dataToTransport();

}

### 5.2.2 Mapping and connecting ports

Ports with translation capability can work in two different modes: normal and translation mode. In normal mode, the port behaves as a standard message port according to the rules specified in ETSI ES 201 873-1 [1]. In translation mode, the port uses rules described in the following clauses of the present document to convert messages and addresses when communicating with linked ports.

The translation mode is activated in these cases:

* A map operation is applied to a component port and TSI port and the component port type contains a reference to the TSI port type in its map clause.
* A port type of one operands of a connect operation contains a reference to the port type of the other operand in its connect clause.

In all other cases, normal mode is activated.

EXAMPLE:

**type port** TransportPort {

...

}

**type port** DataPort **map to** TransportPort {

...

}

**type component** SystemComponent{

**port** DataPort dataPort;

**port** TransportPort transportPort;

}

**type component** TestComponent{

**port** DataPort dataPort;

}

**testcase** TC **runs on** TestComponent **system** SystemComponent

{

**if** (PX\_TRANSPORT\_USED){

// activate translation mode (TransportPort is implicitly referenced via transportPort

// in the map operation)

**map**(**mtc**:dataPort, **system**:transportPort);

}

**else**{

// activate normal mode (TransportPort is not referenced in the map operation)

**map**(**mtc**:dataPort, **system**:dataPort);

}

}

### 5.2.3 Translation functions

Translation functions are used by ports working in translation mode for converting incoming and outgoing messages and addresses from one type to another.

***Syntactical Structure***

**function** FunctionIdentifier"("**in** FormalValuePar ","**out** FormalValuePar ")"

[**port** PortTypeId]

StatementBlock

***Semantic Description***

Translation functions have always two parameters. The first one is always an **in** parameter and it is used to pass in a value that shall be translated by the function. The second one is always an **out** parameter and it shall be used to pass the result of the translation to the translation procedure (see clauses 5.2.5, 5.2.6 and 5.2.7) in case of successful translation.

Unlike standard functions described in clause 16.1 of ETSI ES 201 873-1 [1], translation functions can contain a **port** clause. If the port clause is present, all variables defined in the referenced port type become visible in the function body.

***Restrictions***

1. Translation functions shall never return a value.

NOTE: The **setstate** operation is used to inform the test system about the success of translation.

1. Translation functions shall not contain a runs on clause.
2. Translation function containing a **port** clause can be referenced only in the port type referenced in this port clause.
3. The type of the **in** parameter of a translation function referenced as an *InFunction* in an **in** clause shall be the *OuterInType* immediately preceding the *InFunction* reference and the type of its **out** parameter shall be the *InnerInType*.
4. The type of the **in** parameter of a translation function referenced as an *OutFunction* in an **out** clause shall be the *InnerOutType* and the type of its **out** parameter shall be the *OuterOutType* immediately preceding the *OutFunction* reference.
5. The type of the **in** parameter of a translation function referenced as an *AddrOutFunction* in a port **address** declaration shall be the *AddrType* and the type of its **out** parameter shall be the *OuterAddrType* that immediately precedes the *AddrFunction* reference.
6. The type of the **in** parameter of a translation function referenced as an *AddrInFunction* in a port **address** declaration shall be the *OuterAddrType* that immediately precedes the *AddrFunction* reference and the type of its **out** parameter shall be the *AddrType*.
7. Translation functions and any behaviour invoked directly or indirectly from the translation function shall not contain any blocking operations.
8. Invoking a function with a **port** clause explicitly shall cause an error.
9. Translation functions and any behaviour invoked directly or indirectly form the translation function shall not contain the following port operations: **start**(port), **stop**(port), **halt**, **connect**, **disconnect**, **map** and **unmap.**
10. The rules for functions called from special places defined in clause 16.1.4 of ETSI ES 201 873-1 [1] are valid for receiving translation functions (i.e. the functions referenced in the *OutFunction* part of a translation port type definition). The only exception to this rule is the send operation which is allowed in receiving translation functions. When executing the send operation initiated from receiving translation function, the TE temporarily stores sent messages and places them on an outgoing message port after snapshot evaluation is finished.

EXAMPLE:

**type port** DataPort **map to** TransportPort

{

**in** DataMessage **from** TransportMessage **with** transportToData();

**out** DataMessage **to** TransportMessage **with** dataToTransport();

**var octetstring** vp\_remainings

}

**function** transportToData(**in**TransportMessage p\_msg, **out**DataMessage p\_res) **port** DataPort {

...

**port**.**setstate**(0, "Translated");

}

**function** dataToTransport(**in**DataMessage p\_msg, **out**TransportMessage p\_res) **port** DataPort {

...

**port**.**setstate**(0, "Translated");

}

### 5.2.4 Translation state

In addition to port state dimensions defined ETSI ES 201 873-1 [1], all ports working in translation mode have an additional port state dimension called translation state. The translation state always contains the result of the last executed translation function performed by the port.

There are five possible translation states:

* **unset** is the default state before invoking a translation error. If a translation function ends with this state, an error is generated;
* **not translated** means that the translation function has not been successful;
* **fragmented** indicates the translation function did not finish translation, because the input data did not contain a complete message (i.e. more fragments are needed to finish translation);
* **translated** means that the translation function successfully performed translation and there are no non‑translated data left;
* **partially translated** is used when the translation function successfully performed translation, but there are additional data which has not been translated yet (i.e. the input data contained more than one message);
* **discarded** is used when the translation function finished successfully, by discarding the message.

Translation state is set implicitly to *unset* whenever a translation function is called to translate a sent or received message. The translation state can be changed by a **setstate** operation.

***Syntactical Structure***

**port.setstate**"("SingleExpression { "," ( FreeText | TemplateInstance ) } ")"

***Semantic Description***

The **setstate** operation can be used only inside a function that is called during a translation procedure to translate a sent or received a message. It changes the translation state of the related port.

The optional parameters allow to provide information that explains the reasons for setting a port translation state. This information is composed to a string and might be used for logging purposes.

***Restrictions***

1. The value passed to the **setstate** operation in the first parameter shall be of the **integer** type and shall have one of the following values:

* 0 (meaning *translated*)
* 1 (meaning *not translated*)
* 2 (meaning *fragmented*)
* 3 (meaning *partially translated*)
* 4 (meaning *discarded*)

NOTE 1:Numeric parameter values 0, 1 and 2 are the same as results of the predefined **decvalue** function.

NOTE 2: Clause B.2.1 of the present document includes the type definition translation state and the constant definitions TRANSLATED, NOT\_TRANSLATED, FRAGMENTED, PARTIALLY\_TRANSLATED, DISCARDED.

1. Calling the **setstate** operation with an **integer** not listed in a) in the first parameter shall lead to an error.
2. Calling the **setstate** operation outside of a translation function or in a translation function translating an address shall cause a runtime error.
3. For *FreeText* and *TemplateInstance*, the same rules and restrictions apply as for the parameters of the log statement. See clause 19.11 of ETSI ES 201 873-1 [1] for more details.

NOTE 3:The *unset* state cannot be set by the setstate operation, it is reserved for TE internal use only.

### 5.2.5 Sending

When a message is to be sent over a port, working in translation mode, the following shall apply:

* If no *OutFunction* is specified for the given *InnerOutType,* it is simply sent over the port transparently.
* If an *OutFunction* is specified for the *InnerOutType*, the translation procedure first sets the translation state to *Unset*. Then the *OutFunction* is automatically invoked to translate the *InnerOutType* to the *OuterOutType.* When the function execution is finished, then depending on the current translation state one of the following actions is taken:
* The *unset* state shall cause an error (i.e. if there is no **setstate** operation is invoked in the translation function).
* If the state is *not translated*, the translation procedure tries to translate the message using the next *OutFunction* specified for the given *InnerOutType*. *OutFunction*-s are tried according to their textual order in the port type definition. If there is no such a function, an error is generated.
* If the state is *fragmented*, the translation procedure ends but no data is sent to the connected or mapped port (the port will wait for the next fragment to complete translation). The **to** clause of the following send operation shall be the same as the **to** clause of the current send operation or missing if the current send operation does not contain any to clause.
* If the state is *translated*, the translation procedure sends the translated message (retrieved from the out parameter of the *OutFunction*) to the port it is mapped or connected to.
* If the state is *partially translated*, the sent message of the*InnerOutType* contains several messages (or message fragments) of the*OuterOutType.* In this case, the translation procedure sends the translated message to the mapped or connected port. The translation function is then called again, with the same **in** parameter value, to enable sending of the remaining messages.
* If the state is *discarded*, the translation procedure ends, with no data sent to the connected or mapped port (the message was intentionally discarded).

NOTE: In the *fragmented* case the non-translated part of *InnerOutType* has to be explicitly assigned to port variables.

### 5.2.6 Receiving

Unlike a port working in standard mode, ports working in translation mode maintain two different queues. The outer queue is used to keep not translated messages that are either enqueued or sent to the port working in translation mode. The inner message queue contains already translated messages. Receiving operations access this inner queue. In case of successful receiving (see clause 22.2.2 of ETSI ES 201 873-1 [1]), the successfully received message is removed from the inner queue. Messages stored in the outer queue can be removed from it only by the translation procedure as described below.

The TTCN‑3 Executable (TE, see ETSI ES 201 873-6 [4]) shall control the translation process and the normal decoding algorithm (see note 1) in co-operation, as specified below. But yet, the normal decoding algorithm itself is not changed.

**decode (TRI message, decoding hypothesis: B**)

TE

**Port in translation mode**

**System**

**adaptor**

TRI message

**p.receive(A:?)**

Co**dec**

**decoded value**

**Outer queue**

**Inner** **queue**

***InFunction***  
 (**in B**, **out A**)

Figure 2: Illustration of the interworking of decoding and translation procedure during receiving

NOTE 1: In this clause the "normal decoding algorithm" refers to the process that the TE invokes decoding the received bitstring as specified in clauses 7.3.2 and C.5.4 of ETSI ES 201 873-6 [4].

The translation procedure for receiving operations is invoked by the snapshot mechanism. This procedure iterates through all **in** clauses (*InnerInType*-s) defined in the port type definition. The **in** clauses are iterated according to their textual order. During this iteration, the following shall apply:

* If no *InFunction* is specified for the given *InnerInType*, the translation procedure checks, if the top item of the outer queue is of *InnerInType* (i.e. invokes the normal decoding algorithm, and the check is successful if the decoding is successful). If the result of the check is positive, the message is moved from the outer queue into the inner queue (i.e. the port will relay the message from the outer port to the inner port transparently) and iteration ends.
* Otherwise (if the *InFunction* is present for the *InnerInType*), then the translation procedure checks if the top item of the outer queue is of the *OuterInType,* by invoking the normal decoding algorithm, as described above. If the check is successful, the translation procedure automatically executes the *InFunction*: first sets the translation state to *Unset* and passes the message of the *OuterInType* to it, in the first parameter. When the function execution is finished, the translation procedure checks the translation state of the port:
* The *unset* state shall cause an error (i.e. if there is no **setstate** operation is invoked in the translation function).
* If the state is *not* *translated*, the iteration shall continue with the next *InFunction* for the same *OuterInType*. If there is no more such *InFunction*, the translation procedure shall continue with the next *OuterInType*. If there is no more *OuterInType*-s for the given *InnerInType*, the iteration process shall continue with the next *InnerInType*. The order is determined by the textual order in the port type definition.
* If the state is *fragmented*, the top item of the outer queue is removed and the iteration shall be restarted to process the next message in the outer queue. The next message shall have the same address as the current one (including a missing address). If there is no such message, the iteration shall continue with the next *InnerInType*.
* If the state is *translated*, the top item of the outer queue is removed and the translated message (retrieved from the out parameter of the *InFunction*) is inserted into the inner queue. This ends the whole iteration.
* If the state is *partially translated*, the received message of the *OuterInType* contains several messages (or message fragments) of the *InnerInType.* In this case, the translated message (retrieved from the out parameter of the *InFunction*) is inserted into the inner queue. Unlike in the *translated* case, the top message is not removed from the outer queue. Instead, it is kept in its decoded form in the queue to enable translation of the remaining messages embedded in the outer message in subsequent receive calls.
* If the state is *discarded*, the top item of the outer queue is removed. No new message is inserted into the inner queue. The iteration shall be restarted to process the next message in the outer queue.

NOTE 2: In the *fragmented* case the non-translated part of *OuterInType*has to be explicitly assigned to port variables.

* If the iteration has processed all **in** clauses without any success (no transparently relayed message was successfully moved from the outer to inner queue and all *InFunction* calls ended with the *not translated* state), the iteration process returns.
* In case the iteration produces a successful result, the translation procedure might restart the iteration in order to translate the remaining messages in the outer queue (if there are any), or it might for performance consideration postpone this translation to the moment when the next snapshot is taken. For the same performance reasons, the snapshot mechanism is not required to start the translation procedure in case the inner queue already contains some messages.

### 5.2.7 Address

When an address type associated with a mapped port working in the translation mode contains a **to** or **from** clause and one of the *OuterAddrType*-s is the same as the address type of the mapped TSI port, the translation procedure is applied to all addresses used by sending or receiving calls of the port.

In case of sending a message, the translation procedure automatically invokes the *AddrOutFunction* passing the address value defined in the **to** clause to it, in its first parameter. In case of receiving a message, the translation procedure automatically invokes the *AddrInFunction* passing the received address value to it, in its first parameter. When the function execution is over, the translation procedure retrieves the translated address from the **out** parameter of the translation function and the control is returned to the calling sending or receiving procedure to finish the operation using the translated address value.

NOTE:Unlike translation functions used for translating sent or received messages, the translation functions for addresses do not use translation states.

EXAMPLE:

**type port** TransportPort

{

...

**address** TransportAddress;

}

**type port** DataPort **map to** TransportPort

{

...

**address** DataAddress **to** TransportAddress **with** toTransportAddress()

**from** TransportAddress **with** fromTransportAddress;

}

**function** toTransportAddress(DataAddress p\_addr, **out** TransportAddress p\_translated) { ...}

**function** fromTransportAddress(TransportAddress p\_addr, **out** DataAddress p\_translated) { ... }

### 5.2.8 Clear, start, stop and halt operation

The **clear** and **start** operations clean messages both from inner and outer message queues. In addition to that, all port variables are reset in the following way: if a variable declaration contains an assignment, the assignment operation will be performed as a part of the clear or start operation restoring the initial value of the variable. Otherwise (if the variable declaration does not contain an assignment part), the value of the variable will be uninitialized after the clear or start operation.

The **halt** operation affects the outer queue only. The translation procedure can still insert translated messages into the inner queue of a halted port, provided that there are available messages in the outer queue.

Since the **stop** port operation requires all communication operations to cease before the port is stopped, all unfinished translation operations shall be completely performed before the working of the port is suspended.

### 5.2.9 The outer port reference

If the **port** clause of a translation function is present, it is possible to reference the mapped or connected outer port using the **port** keyword.

***Semantic Description***

The **port** keyword is used in expressions and port operations as a reference to the mapped or connected outer port.

***Restrictions***

1. The outer port reference shall be present only in translation functions that contain a **port** clause.
2. The port type referenced in the **port** clause of the translation function shall contain an *OuterPortType* reference.

NOTE: Translation functions can use only a limited set of port operations (see clause 5.2.3 for more details). These restrictions are valid for the outer port reference as well.

EXAMPLE:

// The example shows a way how to support service messages (in this case a simple handshake  
 // procedure) that are required for correct communication with the target, but not directly  
 // related to the translated content.  
   
 **type** **record** HelloMessage {…}  
 **type** **record** DataRequest {…}  
 **type record** HelloMessageReply {…}  
 **type** **record** DataReply {…}  
 **type** **union** TransportLayerReply {  
 HelloMessageReply helloReply,  
 DataReply dataReply  
 }  
 **type** **port** TransportLayerPort **message** {  
 **out** HelloMessage, DataRequest;  
 **in** TransportLayerReply;  
 }

**type** **port** DataLayerPort **message** **map** **to** TransportLayerPort {  
 **in charstring from**  TransportLayerReply **with** ft\_replyToString();  
 **out charstring to** DataRequest **with** ft\_stringToRequest();  
 **var** ETransportLayerStatus vp\_status := handshakeNotStarted**;**  
 **var** **charstring** vp\_storedMsg**;**  
 }

**type component** MyComp {  
 **port** DataLayerPort p;  
 }

**type component** System {  
 **port** TransportLayerPort p;  
 }  
  
 **type enumerated** ETransportLayerStatus { handshakeNotStarted, handshakeStarted, handshakeDone }  
  
 **function** f\_createDataRequest(**charstring** p\_msg) **return** DataRequest {…}  
  
 **function** f\_extractPayloadFromReply (DataReply p\_msg) **return** **charstring** {…}  
  
 **function** ft\_stringToRequest (**in charstring** p\_input, **out** DataRequest p\_output)   
 **port** DataLayerPort {  
 if (vp\_state == handshakeNotStarted) { // handshake required  
 **port**.**send**(TransportLayerMessageHelloMessage:{…}); // send the handshake message  
 vp\_state := handshakeStarted; // change the translation port state  
 vp\_storedMsg := p\_input;  
 **port**.**setstate**(2); // notify the TE that translation hasn’t been finished yet  
 } else if (vp\_state == handshakeDone) {   
 // handshake done, transform the input payload to a standard message  
 p\_output = f\_createDataRequest(p\_input);  
 **port**.**setstate**(0);  
 } else { // unexpected state: handshake not initialized yet  
 **port**.**setstate**(1);   
 }  
 }

**function** ft\_replyToString (**in** TransportLayerReply v\_input, **out charstring** v\_output)  
 **port** DataLayerPort {  
 if (vp\_state == handshakeStarted) { // handshake reply: finish handshake  
 if (ischosen(v\_input.helloReply) {  
 vp\_state := handshakeDone; // change the translation port state  
 **port**.**send**(f\_createDataRequest(vp\_storedMsg)); // send the stored data request  
 **port**.**setstate**(2); // notify the TE that the reply is not ready yet  
 } else { // unexpected message  
 **port**.**setstate**(1);   
 }  
 } else if (vp\_state == handshakeDone) { // handshake done; data reply expected  
 if (ischosen(v\_input.dataReply) {  
 v\_output := extractPayloadFromReply(v\_input.dataReply);  
 **port**.**setstate**(0); // notify the TE about successful translation  
 } else { // unexpected message  
 **port**.**setstate**(1);   
 }  
 } else { // unexpected state: handshake not initialized yet  
 **port**.**setstate**(1);   
 }  
 }

**testcase** TC\_TEST\_01() **runs on** MyComp **system** System {  
 **map**(**self**:p, **system**:p);  
 p.**send**("Test"); // starts the handshake procedure  
 p.**receive**(charstring:?); // finishes the handshake procedure, then dispatches the request  
 // and receives a reply  
 p.**send**("Test"); // dispatches the request (handshake already done)  
 p.**receive**(charstring:?); // receives the reply  
 }

## 5.3 Parallel Control Components

### 5.3.0 General

This clause describes an extension of capabilities of control behaviour definitions, allowing coordinated parallel execution of test cases.

The main control function is the control function or control block that is started from outside the test system.

Before the main control function is started, first a main control component (MCC) is implicitly instantiated which will execute the control function behaviour, similar to the MTC in a test case that is implicitly created when a test case is executed.

Inside the behaviour definition being executed by a control component it is allowed to dynamically create additional parallel control components (PCCs) and start them with other control behaviour similar to how a test component inside test case behaviour is allowed to create and start PTCs. All restrictions applying to control behaviour in general apply also to control behaviour executed on PCCs.

PCCs are allowed to execute test cases independently and in parallel with test cases being executed on other PCCs and the MCC.

The restriction that in every configuration there shall be one (and only one) MTC is amended to that in every test case configuration shall be exactly one MTC and no MTC exists inside control behaviour. Additionally, a new restriction is added that in every test system configuration, there shall be exactly one MCC in case that the test system is started by executing a control function.

NOTE1: The feature of PCCs has multiple possible applications from coordinated parallel execution of independent test cases to co-execution of a test case with its mirror test case simulating the system under test for test case validation.

NOTE2: Since the test cases are running in parallel in the same test environment, the values for module parameters are the same for all test cases and cannot be configured per test case independently.

At any point in time, every component of the set of currently existing components inside the test system shall have a globally unique id. Each MTC and PTC has a test case local id. The globally unique id of the MTC or any PTC existing in the scope of a running test case are a pairing of the global id of the control component that is executing the test case and the test case local id of the component. So, in parallelly running test cases, a component in one test case might have the same local id as a component in the other test case. In test cases executed sequentially by the same control component, the test case local ids may be the same as in a previous test case executed by the same control component, since these do not exist at the same point in time.

***Examples***

EXAMPLE 1:

testcase T() runs on MtcType system SystemType { … }

testcase TMirror() runs on MtcMirrorType system SystemMirrorType { … }

function @control executeT(float maxTime) {

execute(T(), maxTime);

}

function @control executeTMirror() {

execute(TMirror());

}

control {

var PccType mirror := PccType.create;

var PccType tc := PccType.create;

mirror.start(executeTMirror());

tc.start(executeT(200.0));

alt {

[] tc.done { mirror.stop }

[] mirror.done { tc.stop }

}

}

### 5.3.1 Component Operations

The component operations **create, start, call, stop, kill, running, alive, done, killed** shall be allowed to be used also in control behaviour with the same semantics and the same restrictions as for test components inside test behaviour with the following differences:

When the MCC terminates, all running PCCs are killed implicitly which will in turn stop all currently executed test cases.

The usages of the **any component** and **all component** operationsinside control behaviour reference only the created and started PCCs, not the PTCs that are running inside test cases started on these PCCs. Control hehaviour has no way of referencing components created for an executed test case.

The usages of the **any component** and **all component** operations when used inside test case behaviour reference only the set of PTCs started by the behaviours associated with the test case behaviour that is using these operations. One test case can not reference the components of another test case.

***Restrictions***

1. The use of the operations **any component** and **all component** when used inside control behaviour are only allowed inside the control behaviour running on the MCC.
2. The behaviour definitions used with the **start** or **call** operation when used inside control behaviour shall be control behaviour definitions.

### 5.3.2 Port Operations

The port operations **map, unmap, send, receive, trigger, call, getcall, reply, getreply, raise, catch, start, halt, clear** shall be allowed for the ports of control components with the same semantics and restrictions as for ports of test components.

NOTE: Control components are allowed to communicate with the SUT. This communication is intended to be used for setting up the testing environment and it is not a part of the actual tests.

### 5.3.3 Alt and Interleave Statements

**alt** statements and **interleave** statements used in control behaviour shall be allowed to use all port, timer and component operations allowed inside control behaviour. Default alternatives started during control behaviour shall also be allowed to use these operations. The semantics of **alt** statements and **interleave** statements as well as the **activate** and **deactivate** operations are the same as for behaviour executed on a test component.

### 5.3.4 Test Case Execution

Control components are allowed to use the **execute** operation to execute test cases. The **execute** operation inside PCC behaviour will also block (same as inside MCC behaviour) until either the executed test case has terminated or the given timeout has occurred and test execution has been stopped. Thus, there can still be only at most one test case executed per control component. If a PCC is stopped (explicitly by the MCC or implicitly when the MCC terminates) while executing a test case, test case execution will be stopped before the PCC terminates.

Restrictions

1. The types of parameters of test cases executed by control behaviour shall be data types.

### 5.3.5 MTC Reference and Clause

The **mtc** operation inside control behaviour shall reference the MCC. Likewise, the type of the **mtc** clause of control behaviour definitions shall be the type of the MCC and thus mtc compatibility restrictions apply in reference to the MCC inside control behaviour.

### 5.3.5 Changes to the Test Control Inferface TCI

#### In section 7.1.1.2, Table 2 shall be replaced with the following table:

Table 2: Correlation between TTCN‑3 test case, test component and  
port operations and TCI operation invocations

| TTCN‑3 Operation Name | TCI Operation Name | TCI Interface Name |
| --- | --- | --- |
| create | tciCreateTestComponentReq | TCI‑CH Provided |
| tciCreateTestComponent | TCI‑CH Required |
| start (a component) | tciStartTestComponentReq | TCI‑CH Provided |
| tciStartTestComponent | TCI‑CH Required |
| stop (a component) | tciStopTestComponentReq | TCI‑CH Provided |
| tciStopTestComponent | TCI‑CH Required |
| kill | tciKillTestComponentReq | TCI‑CH Provided |
| tciKillTestComponent | TCI‑CH Required |
| connect | tciConnectReq | TCI‑CH Provided |
| tciConnect | TCI‑CH Required |
| disconnect | tciDisconnectReq | TCI‑CH Provided |
| tciDisconnect | TCI‑CH Required |
| map | tciMapReq (see note 1) | TCI‑CH Provided |
| tciMapParamReq (see note 2) |  |
| tciMap (see note 1) | TCI‑CH Required |
| tciMapParam (see note 2) |  |
| unmap | tciUnmapReq (see note 1) | TCI‑CH Provided |
| tciUnmapParamReq (see note 2) |  |
| tciUnmap (see note 1) | TCI‑CH Required |
| tciUnmapParam (see note 2) |  |
| running | tciTestComponentRunningReq | TCI‑CH Provided |
| tciTestComponentRunning | TCI‑CH Required |
| alive | tciTestComponentAliveReq | TCI‑CH Provided |
| tciTestComponentAlive | TCI‑CH Required |
| done | tciTestComponentDoneReq | TCI‑CH Provided |
| tciTestComponentDone | TCI‑CH Required |
| killed | tciTestComponentKilledReq | TCI‑CH Provided |
| tciTestComponentKilled | TCI‑CH Required |
| mtc | tciGetMTCReq, tciGetParallelMTCReq | TCI‑CH Provided |
| tciGetMTC, tciGetParallelMTC | TCI‑CH Required |
| execute | tciTestCaseExecuteReq | TCI‑CH Provided |
| tciTestCaseExecute | TCI‑CH Required |
| NOTE 1: For statement without configuration parameter.  note 2: For statement with configuration parameter. | | |

#### The Following sections shall be added

##### 7.3.3.1.25 tciGetParallelMTC

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Signature | TriComponentIdType tciGetParallelMTC(in TriComponentIdType component) | | | |
| In Parameters | | component | Identifier of the control component executing the test case for which the MTC shall be determined. |
| Return Value | A TriComponentIdType value of the MTC associated with the given parallel control component if the MTC executes on the local TE, the distinct value null otherwise. | | | |
| Constraint | This operation can be called by the CH at the appropriate local TE when at a remote TE a *provided* tciGetParallelMTCReq has been called. | | | |
| Effect | The local TE determines whether the MTC of the given component is executing on the local TE. If the MTC executes on the local TE the component id of the MTC is being returned. If the MTC is not executed on the local TE the distinct value null will be returned. The operation will have no effect on the execution of the MTC. After the operation returns, the CH will communicate the value back to the remote TE. | | | |

##### 7.3.3.2.33 tciGetParallelMTCReq

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Signature | TriComponentIdType tciGetParallelMTCReq(in TriComponentIdType component) | | | |
| In Parameters | | component | Identifier of the control component executing the test case for which the MTC shall be determined. |
| Return Value | A TriComponentIdType value of the MTC. | | | |
| Constraint | This operation shall be called by the TE when it executes a TTCN‑3 mtc operation. | | | |
| Effect | The CH determines the component id of the MTC. | | | |

#### In the following section:

#### In 8.5.3.1 TCI‑CH provided

Add the following methods to TciCHProvided:

public void tciGetParallelMTCReq(TriComponentId comp) ;

#### In 8.5.3.2 TCI‑CH required

Add the following methods to TciCHRequired:

public void tciGetParallelMTC(TriComponentId comp) ;

#### In 9.4.3.1 TCI‑CH provided

Add the following declaration:

void tciGetParallelMTCReq(TriComponentId component)

#### In 9.4.3.2 TCI‑CH required

Add the following declaration:

void tciGetParallelMTC(TriComponentId component)

#### In 10.6.3.1 TciChRequired

Add the following declaration:

//This operation can be called by the CH at the appropriate local TE when at a remote TE a //provided tciGetParallelMTCReq has been called

virtual const TriComponentId \* tciGetParallelMTC (const TriComponentId \*component) const =0;

#### In 10.6.3.2 TciChProvided

Add the following declaration:

//Called by the TE when it executes a TTCN-3 mtc on a parallel control component operation

virtual const TriComponentId \* tciGetParallelMTCReq (const TriComponentId \*component) const =0;

#### In 12.5.3.1 TCI-CH provided

Add the following declaration to ITriCHProvided:

ITriComponentId TciGetParallelMTC (ITriComponentId component);

#### In 12.5.3.2 TCI-CH required

Add the following declaration to ITriCHRequired:

ITriComponentId TciGetParallelMtc(ITriComponentId comp);

#### In Annex A (normative) IDL Specification of TCI

Add the following declaration to TCI\_CD\_Required;

TriComponentIdType tciGetParallelMTC (

in TriComponentIdType comp

);

Add the following declaration to TCI\_CD\_Provided;

TriComponentIdType tciGetParallelMTCReq (

in TriComponentIdType comp

);