Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
TTCN-3 Language Extensions:
Configuration and Deployment Support
# 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
Annex A (normative): BNF and static semantics ......................................................... 90

A.1 Additional TTCN-3 terminals ............................................................................. 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 (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.


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

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:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTC</td>
<td>Main Test Component</td>
</tr>
<tr>
<td>PTC</td>
<td>Parallel Test Component</td>
</tr>
</tbody>
</table>

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 201 873-6 [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 reinitialized 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 deactivated, 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:

a) The configuration type cannot be subtyped or constrained.

b) The configuration type is not a data type, therefore, the anytype does not include the configuration type.

c) Module parameters shall not be of type configuration.

d) Signature parameters shall not be of type configuration.

e) Templates shall not be of type configuration.

f) Templates shall not be of a structured type that contains fields or elements of type configuration on any level of nesting.

g) 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

```plaintext
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

a) 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].

b) Configuration functions shall only be invoked in the module control part.

c) 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.

```
class 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.

```tcl
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:**

```tcl
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**

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

**EXAMPLES:**

```tcl
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**

a) 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.

b) The keyword `static` in a `create` operation shall not be used in combination with the keyword `alive`.

**EXAMPLES:**

```tcl
// 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**

- a) 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.
- b) 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:**

```c
// 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**

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

**EXAMPLES:**

```plaintext
classification 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**

```plaintext
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 reinitialized 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**


**EXAMPLES:**

```plaintext
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>
<thead>
<tr>
<th>Used in a log statement</th>
<th>What is logged</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>configuration reference</td>
<td>actual state</td>
<td>Configurations states shall be logged according to note 9.</td>
</tr>
</tbody>
</table>

**NOTE:** 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
    [ map to { OuterPortType [ "", ] }+ ]
    [ connect to { OuterPortType [ "," ] }+ ] "|
    [ ( in { InnerInType [ from { OuterInType with InFunction "(" ")" [ "," ] }+ ] [ "," ] }+ ]
    [ out { OuterOutType [ 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.

![Figure 1: Illustration of ports with translation capability](image)

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

\textbf{NOTE 1:} If these conditions are met, it is always safe to map TSI ports of \texttt{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 \texttt{out} message list of the \texttt{OuterPortType} shall be referenced either as \texttt{InnerInType}, \texttt{OuterInType} or \texttt{InOutType} in the port type with translation capability.
- All \texttt{InOutTypes} shall be present either in the \texttt{in} and \texttt{out} lists (at the same time) or in the \texttt{inout} message list of the \texttt{OuterPortType}.
- All \texttt{InnerOutTypes} shall be referenced in the \texttt{in} message list of the \texttt{OuterPortType} or if such a reference does not exist, the \texttt{OuterPortType} shall contain at least one reference to any of the \texttt{OuterOutTypes} associated with the \texttt{InnerOutType} in its \texttt{in} message list.

\textbf{NOTE 2:} If these conditions are met, it is always safe to connect ports with translation capability to ports of \texttt{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 \texttt{InFunctions} and \texttt{OutFunctions}. They are not visible outside of the translation procedure. The variables can be used e.g. for buffering data between individual calls of \texttt{InFunctions} and \texttt{OutFunctions}(e.g. in case of fragmented messages).

\textbf{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:

\begin{itemize}
\item[a)] If the \texttt{OuterPortType} is a port type with translation capability, it shall neither directly nor indirectly reference \texttt{PortTypeId} in its map or connect clause (i.e. port types with translation capability cannot reference each other).
\item[b)] All \texttt{OuterAddrTypes} shall be used as an address type at least in one of the \texttt{OuterPortTypes}.
\item[c)] All \texttt{InFunction}, \texttt{OutFunction} and \texttt{AddrFunction} identifiers shall be references to a translation function.
\end{itemize}

\textbf{EXAMPLE:}

\begin{verbatim}
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:

```plaintext
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**

```plaintext
function FunctionIdentifier"("in FormaValuePar ","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

a) Translation functions shall never return a value.

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

b) Translation functions shall not contain a runs on clause.

c) Translation function containing a `port` clause can be referenced only in the port type referenced in this port clause.

d) 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`.

e) 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.

f) 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.

g) 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`.

h) Translation functions and any behaviour invoked directly or indirectly from the translation function shall not contain any blocking operations.

i) Invoking a function with a `port` clause explicitly shall cause an error.

j) 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`.

k) 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:

```plaintext
type port DataPort map to TransportPort {
  in DataMessage from TransportMessage with transportToData();
  out DataMessage to TransportMessage with dataToTransport();
  var octetstring vp_remainings
}

function transportToData(inTransportMessage p_msg, outDataMessage p_res) port DataPort {
  ...
  port.setState(0, "Translated");
}

function dataToTransport(inDataMessage p_msg, outTransportMessage 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**

```plaintext
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**

a) 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.

b) Calling the **setstate** operation with an **integer** not listed in a) in the first parameter shall lead to an error.

c) Calling the **setstate** operation outside of a translation function or in a translation function translating an address shall cause a runtime error.

d) 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.
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:

```plaintext
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**

a) The outer port reference shall be present only in translation functions that contain a `port` clause.

b) 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:**

```plaintext
// 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);
    }
}

6 Package Semantics

6.0 General

The complete semantics of the using the package in TTCN-3 is defined by copying the following clauses in the following manner into ETSI ES 201 873-4 [2]: TTCN-3 Operational Semantics.

- Clause 6.1 replaces clause 7 in ETSI ES 201 873-4 [2].
- Clause 6.2 replaces clause 7.1 in ETSI ES 201 873-4 [2].
- Clause 6.3 replaces clause 8.2 in ETSI ES 201 873-4 [2].
- Clause 6.4 replaces clause 8.2.1 in ETSI ES 201 873-4 [2].
- Clause 6.5 is a new clause. It would become clause 8.2.6a in ETSI ES 201 873-4 [2].
- Clause 6.6 replaces clause 8.2.7 in ETSI ES 201 873-4 [2].
- Clause 6.7 replaces clause 8.3.1.0 in ETSI ES 201 873-4 [2].
- Clause 6.8 replaces clause 8.3.1.1 in ETSI ES 201 873-4 [2].
- Clause 6.9 replaces clause 8.3.1a.0 in ETSI ES 201 873-4 [2].
- Clause 6.10 replaces clause 8.3.1a.1 in ETSI ES 201 873-4 [2].
- Clause 6.11 replaces clause 8.3.2.0 in ETSI ES 201 873-4 [2].
• Clause 6.12 replaces clause 8.3.2.1 in ETSI ES 201 873-4 [2].
• Clause 6.13 replaces clause 8.3.3.1 in ETSI ES 201 873-4 [2].
• Clause 6.14 replaces clause 8.3.3.2 in ETSI ES 201 873-4 [2].
• Clause 6.16 replaces clause 8.6.1 in ETSI ES 201 873-4 [2].
• Clause 6.17 replaces clause 8.6.1.1 in ETSI ES 201 873-4 [2].
• Clause 6.18 replaces clause 8.6.1.2 in ETSI ES 201 873-4 [2].
• Clause 6.19 replaces clause 8.6.1.3 in ETSI ES 201 873-4 [2].
• Clause 6.20 replaces clause 8.6.1.4 in ETSI ES 201 873-4 [2].
• Clause 6.21 replaces clause 8.6.2 in ETSI ES 201 873-4 [2].
• Clause 6.22 replaces clause 9.9 in ETSI ES 201 873-4 [2].
• Clause 6.23 is a new clause. It would become clause 9.9a in ETSI ES 201 873-4 [2].
• Clause 6.24 replaces clause 9.10 in ETSI ES 201 873-4 [2].
• Clause 6.25 replaces clause 9.12 in ETSI ES 201 873-4 [2].
• Clause 6.26 replaces clause 9.14.2 in ETSI ES 201 873-4 [2].
• Clause 6.27 replaces clause 9.14.3 in ETSI ES 201 873-4 [2].
• Clause 6.28 replaces clause 9.14.4 in ETSI ES 201 873-4 [2].
• Clause 6.29 replaces clause 9.14.5 in ETSI ES 201 873-4 [2].
• Clause 6.30 replaces clause 9.17 in ETSI ES 201 873-4 [2].
• Clause 6.31 is a new clause. It would become clause 9.17.0 in ETSI ES 201 873-4 [2].
• Clause 6.32 is a new clause. It would become clause 9.17.3 in ETSI ES 201 873-4 [2].
• Clause 6.33 is a new clause. It would become clause 9.17.4 in ETSI ES 201 873-4 [2].
• Clause 6.34 is a new clause. It would become clause 9.17.5 in ETSI ES 201 873-4 [2].
• Clause 6.35 replaces clause 9.22 in ETSI ES 201 873-4 [2].
• Clause 6.36 replaces clause 9.28a in ETSI ES 201 873-4 [2].
• Clause 6.37 replaces clause 9.29a in ETSI ES 201 873-4 [2].
• Clause 6.38 replaces clause 9.29a.1 in ETSI ES 201 873-4 [2].
• Clause 6.39 replaces clause 9.29a.3 in ETSI ES 201 873-4 [2].
• Clause 6.40 replaces clause 9.29b in ETSI ES 201 873-4 [2].
• Clause 6.41 is a new clause. It would become clause 9.29c in ETSI ES 201 873-4 [2].
• Clause 6.42 replaces clause 9.32 in ETSI ES 201 873-4 [2].
• Clause 6.43 replaces clause 9.47 in ETSI ES 201 873-4 [2].
• Clause 6.44 replaces clause 9.49 in ETSI ES 201 873-4 [2].
• Clause 6.45 replaces clause 9.49.1 in ETSI ES 201 873-4 [2].
• Clause 6.46 is a new clause. It would become clause 9.49.4 in ETSI ES 201 873-4 [2].
• Clause 6.47 is a new clause. It would become clause 9.49.5 in ETSI ES 201 873-4 [2].
• Clause 6.48 replaces clause 9.51 in ETSI ES 201 873-4 [2].
• Clause 6.49 replaces clause 9.56.1 in ETSI ES 201 873-4 [2].
• Clause 6.50 replaces clause 9.56.2 in ETSI ES 201 873-4 [2].
• Clause 6.51 replaces clause 9.56.3 in ETSI ES 201 873-4 [2].

6.1 Replacement of short forms
Short forms have to be expanded by the corresponding complete definitions on a textual level before this operational semantics can be used for the explanation of TTCN-3 behaviour.

TTCN-3 short forms are:

• lists of module parameter, constant and variable declarations of the same type and lists of timer declarations;
• stand-alone receiving operations;
• stand-alone altsteps calls;
• trigger operations;
• missing return and stop statements at the end of function, configuration function and test case definitions;
• missing stop execution statements;
• interleave statements;
• select-case statements;
• break and continue statements;
• disconnect and unmap operations without parameters; and
• default values of missing actual parameters.

In addition to the handling of short forms, the operational semantics requires a special handling for module parameters, global constants, i.e. constants that are defined in the module definitions part, and pre-processing macros. All references to module parameters, global constants and pre-processing macros shall be replaced by concrete values. This means, it is assumed that the value of module parameters, global constants and pre-processing macros can be determined before the operational semantics becomes relevant.

NOTE 1: The handling of module parameters and global constants in the operational semantics will be different from their handling in a TTCN-3 compiler. The operational semantics describes the meaning of TTCN-3 behaviour and is not a guideline for the implementation of a TTCN-3 compiler.

NOTE 2: The operational semantics handles parameters of and local constants in test components, test cases, functions and module control like variables. The wrong usage of local constants or in, out and inout parameters has to be checked statically.

6.2 Order of replacement steps
The textual replacements of short forms, global constants and module parameters have to be done in the following order:

1) replacement of lists of module parameter, constant, variable and timer declarations with individual declarations;

2) replacement of global constants and module parameters by concrete values;
3) replacement of all **select-case** statements by equivalent nested **if-else** statements;
4) embedding stand-alone receiving operations into **alt** statements;
5) embedding stand-alone altstep calls into **alt** statements;
6) expansion of **interleave** statements;
7) replacement of all **trigger** operations by equivalent **receive** operations and **repeat** statements;
8) adding **return** at the end of function and configuration function definitions without **return** statement, adding **self,stop** operations at the end of test case definitions without a **stop** statement;
9) adding **stop** at the end a module control part without stop statement;
10) expansion of **break** statements;
11) expansion of **continue** statements;
12) adding default parameters to **disconnect** and **unmap** operations without parameters; and
13) adding default values of parameters.

**NOTE:** Without keeping this order of replacement steps, the result of the replacements would not represent the defined behaviour.

### 6.3 Flow graph representation of TTCN-3 behaviour

The operational semantics assumes that TTCN-3 behaviour descriptions are provided in form of a set of flow graphs, i.e. for each TTCN-3 behaviour description a separate flow graph has to be constructed.

The operational semantics interprets the following kinds of TTCN-3 definitions as behaviour descriptions:

- a) module control;
- b) test case definitions;
- c) function definitions;
- d) altstep definitions;
- e) component type definitions;
- f) configuration functions.

The module control specifies the test campaign, i.e. the execution order (possibly repetitious) of the actual test cases. Test case definitions define the behaviour of the MTC. Functions structure behaviour. They are executed by the module control or by the test components. Altsteps are used for the definition of default behaviour or in a function-like manner to structure behaviour. Component type definitions are assumed to be behaviour descriptions because they specify the creation, declaration and initialization of ports, constants, variables and timers during the creation of an instance of a component type. Configuration functions specify the creation of static test configurations.

### 6.4 Flow graph construction procedure

The flow graphs presented in the figures 18 to 22 of ETSI ES 201 873-4 [2] and the flow graph segments presented in clause 8 [2] are only templates. They include **placeholders** for information that have to be provided in order to produce a concrete flow graph or flow graph segment. The placeholders are marked with "<" and ">" parenthesis.

The construction of a flow graph representation of a TTCN-3 module is done in three steps:

1) For each TTCN-3 statement in module control, test cases, altsteps, functions and component type definitions a concrete flow graph segment is constructed.
2) For the module control and for each test case, altstep, function, component type and configuration function definition a concrete flow graph (with reference nodes) is constructed.

3) In a stepwise procedure all reference nodes in the concrete flow graphs are replaced by corresponding flow graph segment definitions until all flow graphs only include one start node, end nodes and basic flow graph nodes.

NOTE 1: Basic flow graph nodes describe basic indivisible execution units. The operational semantics for TTCN-3 behaviour is based on the interpretation of basic flow graph nodes. Clause 8.6 presents execution methods for basic flow graph nodes only.

The replacement of a reference node by the corresponding flow graph segment definition may lead to unconnected parts in a flow graph, i.e. parts which cannot be reached from the start node by traversing through the flow graph along the flow lines. The operational semantics will ignore unconnected parts of a flow graph.

NOTE 2: An unconnected part of a flow graph is a result of the mechanical replacement procedure. For the construction of an optimal flow graph representation the different combinations of TTCN-3 statements also has to be taken into consideration. However, the goal of the present document is to provide a correct and complete semantics, not an optimal flow graph representation.

6.5 Flow graph representation of configuration functions

Schematically, the syntactical structure of a TTCN-3 test case definition is:

```
configuration <identifier> (<parameter>) <testcase-interface> <statement-block>
```

The `<testcase-interface>` above refers to the (mandatory) runs on and the (optional) system clauses in the configuration function definition. The flow graph description of a configuration function describes the behaviour of the MTC when establishing a new static configuration. Variables, timers and constants defined and declared in the component type definition are made visible to the MTC behaviour by the runs on clause in the `<testcase-interface>`. The system clause is not relevant for the MTC and is therefore not represented in the flow graph representation of a configuration function.

The scheme of the flow graph representation of a configuration function is shown in figure 22a. The flow graph name `<identifier>` refers to the name of the represented configuration function. The nodes of the flow graph have associated comments describing the meaning of the different nodes. The reference node `<return-with-value>` covers the case where no explicit return operation for the MTC is specified, i.e. the operational semantics assumes that a return operation is implicitly added. After a successful termination, a configuration function always returns a handle to the newly created static test configuration.
6.6 Retrieval of start nodes of flow graphs

For the retrieval of the start node reference of a flow graph the following function is required:

The `GET-FLOW-GRAPH` function:

\[
\text{GET-FLOW-GRAPH} \ (\text{flow-graph-identifier})
\]

The function returns a reference to the start node of a flow graph with the name `flow-graph-identifier`. The `flow-graph-identifier` refers to the module name for the control, to test case names, to function names, to altstep names to component type names and configuration function names.

6.7 Module state

As shown in figure 23 a module state is structured into a `CONTROL` state and an `ALL-CONFIGURATIONS` state. The `CONTROL` state describes the state of the module control. Module control is handled like a test component, i.e. `CONTROL` is an entity state as defined in ETSI ES 201 873-4 [2], clause 8.3.2. `ALL-CONFIGURATIONS` is a list of configuration states representing test configurations that are instantiated during the execution of module control.

<table>
<thead>
<tr>
<th>CONTROL</th>
<th>ALL-CONFIGURATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>CONFIG_1</code></td>
</tr>
<tr>
<td></td>
<td><code>...</code></td>
</tr>
<tr>
<td></td>
<td><code>CONFIG_n</code></td>
</tr>
</tbody>
</table>

Figure 23 of ETSI ES 201 873-4 [2]: Structure of a module state
6.8 Accessing the module state

The CONTROL state and the ALL-CONFIGURATIONS state of the module state can be addressed by using their names, i.e. CONTROL and ALL-CONFIGURATIONS. Configurations can be accessed by using the dot notation, e.g. ALL-CONFIGURATIONS.CONFIG[1], or by using the list operations defined in clause 8.3.1.1a of ETSI ES 201 873-4 [2].

6.9 Configuration state

As shown in figure 23a the configuration state is structured into ALL-ENTITY-STATES, ALL-PORT-STATES, TC-VERDICT, DONE and KILLED. ALL-ENTITY-STATES represents the states of all instantiated test components during the execution of a test case. The first element of ALL-ENTITY-STATES is the reference to the MTC of the configuration. ALL-PORT-STATES describes the states of the different ports. TC-VERDICT stores the actual global test verdict of a test case, DONE is a list of all currently stopped test components during test case execution and KILLED is a list of all terminated test components during test case execution.

NOTE 1: The number of updates of TC-VERDICT is identical to the number of test components that have terminated.

NOTE 2: An alive-type test component is put into the DONE list each time when it is stopped and removed from the DONE list each time when it is started. It is put into the KILLED and the DONE list when it is killed.

NOTE 3: Port states may be considered to be part of the entity states. By connect and map ports are made visible for other components and therefore, this operational semantics handles ports on the top level of the configuration state.

<table>
<thead>
<tr>
<th>ALL-ENTITY-STATES</th>
<th>ALL-PORT-STATES</th>
<th>TC-VERDICT</th>
<th>DONE</th>
<th>KILLED</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTC</td>
<td>ES₁ ... ESₙ</td>
<td>P₁ ... Pₙ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 23a of ETSI ES 201 873-4 [2]: Structure of a configuration state

6.10 Accessing the configuration state

The TC-VERDICT and the lists ALL-ENTITY-STATES, ALL-PORT-STATES, DONE and KILLED can be accessed like variables by their name and the dot notation, e.g. CONFIG.TC-VERDICT for accessing the test verdict of configuration CONFIG.

For the creation of a new configuration state the function NEW-CONFIGURATION is assumed to be available:

- NEW-CONFIGURATION();

creates a new configuration state and returns its reference. The components of the new configuration state have the following values:

- ALL-ENTITY-STATES is an empty list;
- ALL-PORT-STATES is an empty list;
- TC-VERDICT is set to none;
- DONE is an empty list;
- KILLED is an empty list.

For the handling of lists, e.g. ALL-ENTITY-STATES, ALL-PORT-STATES, DONE and KILLED in module states, the list operations add, append, delete, member, first, last, length, next, random and change can be used. They have the following meaning:

- myList.add(item) adds item as first element into the list myList and myList.add(sublist) adds the list sublist to list myList, i.e. add can be used to add single elements or lists to lists;
• `myList.append(item)` appends item as last element into the list myList and `myList.append(sublist)` appends the list sublist to list myList, i.e. `append` can be used to append single elements or lists to lists;

• `myList.delete(item)` deletes item from the list myList;

• `myList.member(item)` returns `true` if item is an element of the list myList, otherwise `false`;

• `myList.first()` returns the first element of myList;

• `myList.last()` returns the last element of myList;

• `myList.length()` returns the length of myList;

• `myList.next(item)` returns the element that follows item in myList, or `NULL` if item is the last element in myList;

• `myList.random(<condition>)` returns randomly an element of myList, which fulfils the Boolean condition <condition> or `NULL`, if no element of myList fulfils <condition>;

• `myList.change(<operation>)` allows to apply <operation> on all elements of myList.

**NOTE:** The operations random and change are not common list operations. They are introduced to explain the meaning of the keywords all and any in TTCN-3 operations. Additionally, a general `copy` operation is available. The `copy` operation copies and returns an item instead of returning a reference to an item:

• `copy(item)` returns a copy of item.

### 6.11 Entity states

Entity states are used to describe the actual states of module control and test components. In the module state, `CONTROL` is an entity state and in the configuration state, the test component states are handled in the list `ALL-ENTITY-STATES`. The structure of an entity state is shown in figure 24.

```
<table>
<thead>
<tr>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL-STACK</td>
</tr>
<tr>
<td>DEFAULT-LIST</td>
</tr>
<tr>
<td>DEFAULT-POINTER</td>
</tr>
<tr>
<td>VALUE-STACK</td>
</tr>
<tr>
<td>E-VERDICT</td>
</tr>
<tr>
<td>TIMER-GUARD</td>
</tr>
<tr>
<td>DATA-STATE</td>
</tr>
<tr>
<td>TIMER-STATE</td>
</tr>
<tr>
<td>PORT-REF</td>
</tr>
<tr>
<td>SNAP-ALIVE</td>
</tr>
<tr>
<td>SNAP-DONE</td>
</tr>
<tr>
<td>SNAP-KILLED</td>
</tr>
<tr>
<td>KEEP-ALIVE</td>
</tr>
<tr>
<td>STATIC</td>
</tr>
</tbody>
</table>
```

**Figure 24 of ETSI ES 201 873-4 [2]: Structure of an entity state**

The `STATUS` describes whether the module control or a test component is `ACTIVE`, `BREAK`, `SNAPSHOT`, `REPEAT` or `BLOCKED`. Module control is blocked during the execution of a test case. Test components are blocked during the creation of other test components, i.e. when they call a `create` operation, and when they wait for being started. The status `SNAPSHOT` indicates that the component is active, but in the evaluation phase of a snapshot. The status `REPEAT` denotes that the component is active and in an `alt` statement that should be re-evaluated due to a `repeat` statement.
The **BREAK** status is set when a **break** statement is executed for leaving altstep. In this case, the **alt** statement in which the altstep was directly or indirectly (i.e. by means of the default mechanism) called is immediately left.

The **CONTROL-STACK** is a stack of flow graph node references. The top element in **CONTROL-STACK** is the flow graph node that has to be interpreted next. The stack is required to model function calls in an adequate manner.

The **DEFAULT-LIST** is a list of activated defaults, i.e. it is a list of pointers that refer to the start nodes of activated defaults. The list is in the reverse order of activation, i.e. the default that has been activated first is the last element in the list.

During the execution of the default mechanism, the **DEFAULT-POINTER** refers to the next default that has to be evaluated if the actual default terminates unsuccessfully.

The **VALUE-STACK** is a stack of values of all possible types that allows an intermediate storage of final or intermediate results of operations, functions and statements. For example, the result of the evaluation of an expression or the result of the **mtc** operation will be pushed onto the **VALUE-STACK**. In addition to the values of all data types known in a module, the special value **MARK** to be part of the stack alphabet has been defined. When leaving a scope unit, the **MARK** is used to clean **VALUE-STACK**.

The **E-VERDICT** stores the actual local verdict of a test component. The **E-VERDICT** is ignored if an entity state represents the module control.

The **TIMER-GUARD** represents the special timer, which is necessary to guard the execution time of test cases and the duration of call operations. The **TIMER-GUARD** is modelled as a timer binding (see ETSI ES 201 873-4 [2], clause 8.3.2.4 and figure 28).

The **DATA-STATE** is considered to be a list of lists of variable bindings. The list of lists structure reflects nested scope units due to nested function and altstep calls. Each list in the list of lists of variable bindings describes the variables declared in a certain scope unit and their values. Entering or leaving a scope unit corresponds to adding or deleting a list of variable bindings from the **DATA-STATE**. A description of the **DATA-STATE** part of an entity state can be found in ETSI ES 201 873-4 [2], clause 8.3.2.2.

The **TIMER-STATE** is considered to be a list of lists of timer bindings. The list of lists structure reflects nested scope units due to nested function and altstep calls. Each list in the list of lists of timer bindings describes the known timers and their status in a certain scope unit. Entering or leaving a scope unit corresponds to adding or deleting a list of timer bindings from the **TIMER-STATE**. A description of the **TIMER-STATE** part of an entity state can be found in ETSI ES 201 873-4 [2], clause 8.3.2.4.

The **PORT-REF** is considered to be a list of lists of port bindings. The list of lists structure reflects nested scope units due to nested function and altstep calls. Nested scope units for ports are the result of port parameters in functions and altsteps. Each list in the list of lists of port bindings identifies the known ports in a certain scope unit. Entering or leaving a scope unit corresponds to adding or deleting a list of port bindings from the **PORT-REF**. A description of the **PORT-REF** part of an entity state can be found in ETSI ES 201 873-4 [2], clause 8.3.2.6.

NOTE: The TTCN-3 semantics administrates ports globally in the module state. Due to port parameterization, a test component may access a port by using different names in different scopes. The **PORT-REF** part of an entity state is used to identify port states uniquely in the module state.

The **SNAP-ALIVE** supports the snapshot semantics of test components. When a snapshot is taken, a copy of the **ALL-ENTITY-STATES** list of the module state will be assigned to **SNAP-ALIVE**, i.e. **SNAP-ALIVE** includes all entities (test components and test control) which are alive in the test system.

The **SNAP-DONE** supports the snapshot semantics of test components. When a snapshot is taken, a copy of the **DONE** list of the module state will be assigned to **SNAP-DONE**, i.e. **SNAP-DONE** is a list of component identifiers of stopped components.

The **SNAP-KILLED** supports the snapshot semantics of test components. When a snapshot is taken, a copy of the **KILLED** list of the module state will be assigned to **SNAP-KILL**, i.e. **SNAP-DONE** is a list of component identifiers of terminated components.

The **KEEP-ALIVE** field indicates whether the entity can be restarted after its termination or not. It is set to true if the entity can be restarted. Otherwise it is set to false.
The **STATIC** field indicates whether a test component is part of a static test configuration or not. It is set to **true** if the test component is created during the execution of configuration function. During the execution of a configuration function the **STATIC** field of the entity representing test control is also set to **true**. In all other cases, the **STATIC** field is set to **false**.

### 6.12 Accessing entity states

The **STATUS**, **DEFAULT-POINTER**, **E-VERDICT** and **TIMER-GUARD** parts of an entity state are handled like variables that are globally visible, i.e. the values of **STATUS**, **DEFAULT-POINTER** and **E-VERDICT** can be retrieved or changed by using the "dot" notation, e.g. `myEntity.STATUS`, `myEntity.DEFAULT-POINTER` and `myEntity.E-VERDICT`, where `myEntity` refers to an entity state.

**NOTE:** In the following, it is assumed that the "dot" notation by using references and unique identifiers can be used. For example, in `myEntity.STATUS`, `myEntityState` may be pointer to an entity state or be the value of the `<identifier>` field.

The **CONTROL-STACK**, **DEFAULT-LIST** and **VALUE-STACK** of an entity state `myEntity` can be addressed by using the "dot" notation `myEntity.CONTROL-STACK`, `myEntity.DEFAULT-LIST` and `myEntity.VALUE-STACK`.

**CONTROL-STACK** and **VALUE-STACK** can be accessed and manipulated by using the stack operations `push`, `pop`, `top`, `clear` and `clear-until`. The stack operations have the following meaning:

- `myStack.push(item)` pushes item onto `myStack`;
- `myStack.pop()` pops the top item from `myStack`;
- `myStack.top()` returns the top element of `myStack` or **NULL** if `myStack` is empty;
- `myStack.clear()` clears `myStack`, i.e. pops all items from `myStack`;
- `myStack.clear-until(item)` pops items from `myStack` until `item` is found or `myStack` is empty.

**DEFAULT-LIST** can be accessed and manipulated by using the list operations `add`, `append`, `delete`, `member`, `first`, `length`, `next`, `random` and `change`. The meaning of these list operations is defined in ETSI ES 201 873-4 [2], clause 8.3.1.1.

For the creation of a new entity state the function **NEW-ENTITY** is assumed to be available:

- **NEW-ENTITY** (`flow-graph-node-reference`, `keep-alive`, `static`)

creates a new entity state and returns its reference. The components of the new entity state have the following values:

- **STATUS** is set to **ACTIVE**;
- `flow-graph-node-reference` is the only (top) element in **CONTROL-STACK**;
- **DEFAULT-LIST** is an empty list;
- **DEFAULT-POINTER** has the value **NULL**;
- **VALUE-STACK** is an empty stack;
- **E-VERDICT** is set to **none**;
- **TIMER-GUARD** is a new timer binding with name GUARD, status **IDLE** and no default duration;
- **DATA-STATE** is an empty list;
- **TIMER-STATE** is an empty list;
- **PORT-REF** is an empty list;
- **SNAP-ALIVE** is an empty list;
- **SNAP-DONE** is an empty list;
- **SNAP-KILLED** is an empty list;
- **KEEP-ALIVE** is set to the value of the *keep-alive* parameter;
- **STATIC** is set to the value of the *static* parameter.

During the traversal of a flow graph the **CONTROL-STACK** changes its value often in the same manner: the top element is popped from and the successor node of the popped node is pushed onto **CONTROL-STACK**. This series of stack operations is encapsulated in the **NEXT-CONTROL** function:

```c
myEntity.NEXT-CONTROL(myBool) {
    successorNode := myEntity.CONTROL-STACK.NEXT(myBool).top();
    myEntity.CONTROL-STACK.pop();
    myEntity.CONTROL-STACK.push(successorNode);
}
```

### 6.13 Handling of connections among ports

A connection between two test components is made by connecting two of their ports by means of a `connect` operation. Thus, a component can afterwards use its local port name to address the remote queue. As shown in figure 30, *connection* is represented in the states of both connected queues by a pair of **REMOTE-ENTITY** and **REMOTE-PORT-NAME**. The **REMOTE-ENTITY** is the unique identifier of the test component that owns the remote port. The **REMOTE-PORT-NAME** refers to the port name that is used to declare the port in the component type definition of the test component **REMOTE-ENTITY**, **STATIC** is a Boolean which is true if connection is a static connection of a static test configuration. TTCN-3 supports one-to-many connections of ports and therefore all connections of a port are organized in a list.

**NOTE 1:** Connections made by **map** operations are also handled in the list of connections. The **map** operation: `map(PTC1:MyPort, system.PCO1)` leads to a new (non static) connection (**system**, **PCO1**, **false**) in the port state of **MyPort** owned by **PTC1**. The remote side to which **PCO1** is connected to, resides inside the SUT. Its behaviour is outside the scope of this semantics.

**NOTE 2:** The operational semantics handles the keyword **system** as a symbolic address. A connection (**system**, **myPort**, **false**) in the list of connections of a port it indicates that the port is mapped onto the port **myPort** in the test system interface. The **false** indicates that the mapping is not static.

![Figure 30 of ETSI ES 201 873-4](2): Structure of a connection

### 6.14 Handling of port states

The queue of values in a port state can be accessed and manipulated by using the known queue operations **enqueue**, **dequeue**, **first** and **clear**. Using a **GET-PORT** or a **GET-REMOTE-PORT** function references the queue that shall be accessed.

**NOTE 1:** The queue operations **enqueue**, **dequeue**, **first** and **clear** have the following meaning:

- `myQueue.enqueue(item)` puts item as last item into `myQueue`;
- `myQueue.dequeue()` deletes the first item from `myQueue`;
- `myQueue.first()` returns the first item in `myQueue` or **NULL** if `myQueue` is empty;
- `myQueue.clear()` removes all elements from `myQueue`.
The handling of port states is supported by the following functions:

a) The **NEW-PORT** function: **NEW-PORT** (myEntity, myPort)

   creates a new port and returns its reference. The OWNER entry of the new port is set to myEntity and COMP-PORT-NAME has the value myPort. The status of the new port is STARTED. The CONNECTIONS-LIST and the VALUE-QUEUE are empty. The SNAP-VALUE has the value NULL (i.e. the input queue of the new port is empty).

b) The **GET-PORT** function: **GET-PORT** (myEntity, myPort)

   returns a reference to the port identified by OWNER myEntity and COMP-PORT-NAME myPort.

c) The **GET-REMOTE-PORT** function: **GET-REMOTE-PORT** (myEntity, myPort, myRemoteEntity)

   returns the reference to the port that is owned by test component myRemoteEntity and connected to a port identified by OWNER myEntity and COMP-PORT-NAME myPort. The symbolic address SYSTEM is returned, if the remote port is mapped onto a port in the test system interface.

**NOTE 2:** **GET-REMOTE-PORT** returns NULL if there is no remote port or if the remote port cannot be identified uniquely. The special value NONE can be used as value for the myRemoteEntity parameter if the remote entity is not known or not required, i.e. there exists only a one-to-one connection for this port.

d) The STATUS of a port is handled like a variable. It can be addressed by qualifying STATUS with a **GET-PORT** call:

   **GET-PORT**(myEntity, myPort).STATUS.

e) The **ADD-CON** function: **ADD-CON** (myEntity, myPort, myRemoteEntity, myRemotePort, myStatic)

   adds a connection (myRemoteEntity, myRemotePort, myStatic) to the list of connections of the port identified by OWNER myEntity and COMP-PORT-NAME myPort.

f) The **DEL-CON** function: **DEL-CON** (myEntity, myPort, myRemoteEntity, myRemotePort)

   removes a connection (myRemoteEntity, myRemotePort, ?) with any STATIC value from the list of connections of the port identified by OWNER myEntity and COMP-PORT-NAME myPort.

g) The **GET-CON** function: **GET-CON** (myEntity, myPort, myRemoteEntity, myRemotePort)

   retrieves a connection (myRemoteEntity, myRemotePort, ?) with any STATIC value from the list of connections of the port identified by OWNER myEntity and COMP-PORT-NAME myPort.

h) The **SNAP-PORTS** function: **SNAP-PORTS** (myEntity)

   updates SNAP-VALUE for all ports owned by myEntity, i.e.

   ```
   SNAP-PORTS (myEntity) {
     for all ports p /* in the module state */ {
       if (p.OWNER == myEntity) {
         if (p.STATUS == STOPPED) {
           p.SNAP-VALUE := NULL;
         }
         else {
           if (p.STATUS == HALTED && p.first() == HALT-MARKER) {
             // Port is halted and halt marker is reached
             p.SNAP-VALUE := NULL;
             p.dequeue(); // Removal of halt marker
             p.STATUS := STOPPED;
           }
           else {
             p.SNAP-VALUE := p.first();
           }
         }
       }
     }
   }
   ```
6.15 Void

6.16 Evaluation phases, general

The evaluation procedure for a TTCN-3 module is structured into:

1) initialization phase;
2) update phase;
3) selection phase; and
4) execution phase.

The phases (2), (3) and (4) are repeated until module control terminates. The evaluation procedure is described by means of a mixture of informal text and pseudo-code.

6.17 Phase I: Initialization

The initialization phase includes the following actions:

a) Declaration and initialization of global variables:

- \texttt{INIT-FLOW-GRAPHS();} \hspace{1cm} // Initialization of flow graph handling. \texttt{INIT-FLOW-GRAPHS} is explained in ETSI ES 201 873-4 [2], clause 8.6.2.

- \texttt{Entity := NULL;} \hspace{1cm} // \texttt{Entity} will be used to refer to an entity state. An entity state either represents module control or a test component.

- \texttt{MTC := NULL;} \hspace{1cm} // \texttt{MTC} will be used to refer to the entity state of the main test component of a test case during test case execution.

NOTE 1: The global variable \texttt{CONTROL} forms the control state of a module state during the interpretation of a TTCN-3 module (see ETSI ES 201 873-4 [2], clause 8.3.1).

- \texttt{CONTROL := NULL;} \hspace{1cm} // \texttt{CONTROL} will be used to refer to the entity state of module control a;

NOTE 2: The global variable \texttt{CONFIGURATION} is used to store the reference to a configuration state in the Module state, i.e. a member of \texttt{ALL-CONFIGURATIONS} (see ETSI ES 201 873-4 [2], clause 8.3.1).

- \texttt{CONFIGURATION := NULL;};

NOTE 3: The following global variables \texttt{ALL-ENTITY-STATES}, \texttt{ALL-PORT-STATES}, \texttt{TC-VERDICT}, \texttt{DONE}, and \texttt{KILLED} are used to store references to a test configuration state of a module state during the interpretation of a TTCN-3 module (see ETSI ES 201 873-4 [2], clause 8.3.1).

- \texttt{ALL-ENTITY-STATES := NULL;};
- \texttt{ALL-PORT-STATES := NULL;};
- \texttt{TC-VERDICT := none;}
- \texttt{DONE := NULL;}
- \texttt{KILLED := NULL.}
b) Creation and initialization of module control:

- \[ \text{CONTROL} := \text{NEW-ENTITY} (\text{GET-FLOW-GRAPH} (<\text{moduleId}>), \text{false}, \text{false}); \]
  // A new entity state is created and initialized with the start node of
  // the flow graph representing the behaviour of the control of the
  // module with the name <moduleId>. The Boolean parameters
  // indicate that module control cannot be restarted after it is
  // stopped and that it is not a static component in a test configuration.

- \[ \text{CONTROL.INIT-VAR-SCOPE}(); \]  // New variable scope.

- \[ \text{CONTROL.VALUE-STACK.push(MARK);} \]  // A mark is pushed onto the value stack.

### 6.18 Phase II: Update

The update phase is related to all actions that are outside the scope of the operational semantics but influence the interpretation of a TTCN-3 module. The update phase comprises the following actions:

a) **Time progress**: All running timers are updated, i.e. the `TIME-LEFT` values of running timers are (possibly) decreased, and if due to the update a timer expires, the corresponding timer bindings are updated, i.e. `TIME-LEFT` is set to 0.0 and `STATUS` is set to `TIMEOUT`.

   **NOTE 1**: The update of timers includes the update of all running `TIMER-GUARD` timers in module states. `TIMER-GUARD` timers are used to guard the execution of test cases and call operations.

b) **Behaviour of the SUT**: Messages, remote procedure calls, replies to remote procedure calls and exceptions (possibly) received from the SUT are put into the port queues at which the corresponding receptions shall take place.

   **NOTE 2**: This operational semantics makes no assumptions about time progress and the behaviour of the SUT.

### 6.19 Phase III: Selection

The selection phase consists of the following two actions:

a) **Selection**: Select a non-blocked entity, i.e. an entity that has not the `STATUS` value `BLOCKED`. The entity may be `CONTROL`, i.e. module control, or a test component in a test configuration that is executing a test case.

b) **Storage**:

- Store the identifier of the selected entity in the global variable `Entity`.
- If `Entity` is `CONTROL`, set `CONFIGURATION` to `NULL`.
- If `Entity` is not `CONTROL`, store the identifier of the configuration of which `Entity` is part of in the global variable `CONFIGURATION` and do the following assignments:
  - `ALL-ENTITY-STATES := CONFIGURATION.ALL-ENTITY-STATES;`
  - `MTC := CONFIGURATION.ALL-ENTITY-STATES.first();`
  - `ALL-PORT-STATES := CONFIGURATION.ALL-PORT-STATES;`
  - `TC-VERDICT := CONFIGURATION.TC-VERDICT;`
  - `DONE := CONFIGURATION.DONE;`
  - `KILLED := CONFIGURATION.KILLED.`
6.20 Phase IV: Execution

The execution phase consists of the following three actions:

a) **Execution step of the selected entity:** Execute the top flow graph node in the CONTROL-STACK of Entity.

b) **Update of the module state:** This includes an update of the configuration state of the executed Entity.

c) **Check termination criterion:** Stop execution if module control has terminated, i.e. CONTROL is NULL. Otherwise continue with Phase II.

6.21 Global functions

The evaluation procedure uses the global function **INIT-FLOW-GRAPHS**:

a) **INIT-FLOW-GRAPHS** is assumed to be the function that initializes the flow graph handling. The handling may include the creation of the flow graphs and the handling of the pointers to the flow graphs and flow graph nodes.

The pseudo-code used the following clauses to describe execution of flow graph nodes use the functions **CONTINUE-COMPONENT, RETURN, ***DYNAMIC-ERROR***:

b) **CONTINUE-COMPONENT** the actual test component continues its execution with the node lying on top of the control stack, i.e. the control is not given back to the module evaluation procedure described in this clause.

c) **RETURN** returns the control back to the module evaluation procedure described in this clause. The **RETURN** is the last action of the "execution step of the selected entity" of the execution phase.

d) *****DYNAMIC-ERROR*** refers to the occurrence of a dynamic error. The error handling procedure itself is outside the scope of the operational semantics. If a dynamic error occurs all following behaviour of the test case is meant to be undefined. In this case resources allocated to the test case shall be cleared and the error verdict is assigned to the test case. Control is given to the statement in the control part following the execute statement in which the error occurred. This is modelled by the flow graph segment <dynamic-error> (see ETSI ES 201 873-4 [2], clause 9.18.5).

NOTE: The occurrence of a dynamic error is related to test behaviour. A dynamic error as specified by the operational semantics denotes a problem in the usage of TTCN-3, e.g. wrong usage or race condition.

e) **APPLY-OPERATOR** used as generic function for describing the evaluation of operators (e.g. +, *, / or -) in expressions (see ETSI ES 201 873-4 [2], clause 9.18.4).

6.22 Clear port operation

The syntactical structure of the **clear** port operation is:

```
<portId>.clear
```

The flow graph segment <clear-port-op> in figure 59 defines the execution of the **clear** port operation.
Figure 59 of ETSI ES 201 873-4 [2]: Flow graph segment <clear-port-op>

6.23 Configuration function call

The invocation of a configuration function starts with the creation of the MTC. In a static test configuration the MTC is modelled as a static alive component. Then the MTC is started with the behaviour defined in the configuration function. Afterwards, the module control waits until the configuration function terminates. The creation and the start of the MTC can be described by using `create` and `start` statements:

```plaintext
var mtcType MyMTC := mtcType.create alive static;
MyMTC.start(ConfigurationFunctionName(P1...Pn));
```

The flow graph segment `<config-func-call>` in figure 59a defines the execution of a configuration function by using the flow graph segments of the operations `create` and the `start`. 
6.24 Connect operation

The syntactical structure of the **connect** operation is:

```
connect(<component-expression_1>::<portId1>, <component-expression_2>::<portId2>) [static]
```

The identifiers `<portId1>` and `<portId2>` are considered to be port identifiers of the corresponding test components. The components to which the ports belong are referenced by means of the component references `<component-expression_1>` and `<component-expression_2>`. The references may be stored in variables or is returned by a function, i.e. they are expressions, which evaluate to component references. The value stack is used for storing the component references.

A present `static` clause indicates that the new connection is static, i.e. established during the execution of a configuration function. Presence and absence of the `static` clause is handled as a Boolean flag in the operational semantics (see `static` parameter of the basic flow graph node `connect-op` in figure 60).
The execution of the `connect` operation is defined by the flow graph segment `<connect-op>` shown in figure 60. In the flow graph description the first expression to be evaluated refers to `<component-expression_1>` and the second expression to `<component-expression_2>`, i.e. the `<component-expression_2>` is on top of the value stack when the `connect-op` node is executed.

```plaintext
let { // begin of a local scope
  var portOne, portTwo; // variables for ports
  var comp2 := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop();
  var compl := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop();
  if (static == true && CONTROL_STATIC != true) {
    ***DYNAMIC-ERROR***
    // Static connections have to be established
    // within a configuration function
  } else if (static != true && CONTROL_STATIC == true) {
    ***DYNAMIC-ERROR***
    // Non-static connections cannot be established
    // within a configuration function
  } else {
    portOne := compl.portId1.COMP-PORT-NAME;
    portTwo := comp2.portId2.COMP-PORT-NAME;
    ADD-CON(comp1, portOne, comp2, portTwo, static);
    ADD-CON(comp2, portTwo, comp1, portOne, static);
  }
} // end of local scope
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 60 of ETSI ES 201 873-4 [2]: Flow graph segment `<connect-op>`

### 6.25 Create operation

The syntactical structure of the `create` operation is:

```plaintext
<componentType Id>.create [alive] [static]
```

A present `alive` clause indicates that the created component can be restarted after it has been stopped. Presence and absence of the `alive` clause is handled as a Boolean flag in the operational semantics (see `alive` parameter of the basic flow graph node `create-op` in figure 62).

A present `static` clause indicates that the new component is static, i.e. part of a static test configuration and created during the execution of a configuration function. Presence and absence of the `static` clause is handled as a Boolean flag in the operational semantics (see `static` parameter of the basic flow graph node `create-op` in figure 62).

The flow graph segment `<create-op>` in figure 62 defines the execution of the `create` operation.
let { // Local scope
  var newEntity; // for storing the newly created entity

  // creation of the entity state for the new component
  if (static == true) { // creation of a static component
    if (CONTROL.STATIC != true) { // creation of a static component is only
      ***DYNAMIC-ERROR*** // allowed in a configuration function
    } else {
      newEntity := NEW-ENTITY(componentTypeID, true, true); // the alive flag is set because static
      // components behave like alive components
    }
  } else {
    newEntity := NEW-ENTITY(componentTypeID, static, false);
  }

  // The reference to the new entity is pushed onto the value stack of the
  // 'father' entity.
  Entity.VALUE-STACK.push(newEntity);

  // The identifier of the 'father' entity is pushed onto the value stack of the
  // new entity. The identifier is needed to restore the status of the 'father'
  // entity after completion of the entity creation. The 'father' entity is
  // blocked until all ports, variables, timers specified in the component type
  // definition are instantiated. This instantiation is done by executing the
  // flow graph that represents 'componentTypeID' by the new entity.
  newEntity.VALUE-STACK.push(Entity);

  // The new entity is put into the module state
  ALL-ENTITY-STATES.append(newEntity);

  } // End local scope

  // The actual status of the 'father' entity is saved and the 'father' entity goes
  // into a blocking state. Note the restoration of the status of the father entity
  // is described in flow graph segment <finalize-component-init>.
  Entity.VALUE-STACK.push(Entity.STATUS); // Saving the actual status
  Entity.STATUS := BLOCKED;

  Entity.NEXT-CONTROL(true); // Return of control
  RETURN;

Figure 62 of ETSI ES 201 873-4 [2]: Flow graph segment <create-op>
6.26 Flow graph segment <disconnect-all>

The flow graph segment <disconnect-all> defines the disconnection of all components at all connected ports. Static connections will not be disconnected. Their lifetime is bound to the lifetime of the static test configuration.

```plaintext
let { // local scope
  var port := ALL-PORT-STATES.first();
  var connection;
  while (port != NULL) {
    connection := port.CONNECTIONS.first();
    while (connection != NULL) {
      if (connection.STATIC == true) { // static connection or mapping
        connection := port.CONNECTIONS.next(connection);
      } else {
        if (connection.REMOTE-ENTITY == system) {
          connection := NULL; // mapped port
        } else {
          port.CONNECTIONS.delete(connection);
          connection := port.CONNECTIONS.first();
        }
      }
    }
    port := ALL-PORT-STATES.next(port);
  }
} // End of local scope
Entity.NEXT-CONTROL(true);
RETURN;
```

---

Figure 64b of ETSI ES 201 873-4 [2]: Flow graph segment <disconnect-all>
6.27 Flow graph segment <disconnect-comp>

The flow graph segment <disconnect-comp> defines the disconnection of all ports of a specified component. Static connections will not be disconnected. Their lifetime is bound to the lifetime of the static test configuration.

```
let { // local scope
  var comp := Entity.VALUE-STACK.top();
  var port := ALL-PORT-STATES.first();
  while (port != NULL) {
    connection := port.CONNECTIONS.first();
    while (connection != NULL) {
      if (connection.STATIC == true) { // static connection or mapping
        connection := port.CONNECTIONS.next(connection);
      } else {
        if (connection.REMOTE-ENTITY == system) {
          connection := NULL;  // mapped port
        } else if (connection.REMOTE-ENTITY == comp
          or (port.OWNER == comp) {
          port.CONNECTIONS.delete(connection);
          connection := port.CONNECTIONS.first();
        } else {
          connection := port.CONNECTIONS.next(connection);
        }
      }
    }
    port := ALL-PORT-STATES.next(port);
  }
  Entity.VALUE-STACK.pop();  // clear value stack
} // End of local scope
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 64c of ETSI ES 201 873-4 [2]: Flow graph segment <disconnect-comp>
6.28 Flow graph segment <disconnect-port>

The flow graph segment <disconnect-port> defines the disconnection of a specified port. Static connections will not be disconnected. Their lifetime is bound to the lifetime of the static test configuration.

```plaintext
let { // local scope
    var portId, rPortId;
    var comp, rComp;
    var port;

    portId := Entity.VALUE-STACK.top();
    Entity.VALUE-STACK.pop();
    comp := Entity.VALUE-STACK.top();
    Entity.VALUE-STACK.pop();
    port := GET-PORT(comp, portId);

    var connection := port.CONNECTIONS.first();
    while (connection != NULL) {
        if (connection.REMOTE-ENTITY == SYSTEM) { // port is not a connected port
            ***DYNAMIC-ERROR***
        } else if (connection.STATIC == true) { // static connection
            connection := port.CONNECTIONS.next(connection);
        } else {
            rComp := connection.REMOTE-ENTITY;
            rPortId := connection.REMOTE-PORT-NAME;
            DEL-CON(comp, portId, rComp, rPortId);
            DEL-CON(rComp, rPortId, comp, portId);
            connection := port.CONNECTIONS.first();
        }
    }
} // End of local scope

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 64d of ETSI ES 201 873-4 [2]: Flow graph segment <disconnect-port>

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

The flow graph segment <disconnect-two-par-pairs> shown in figure 64e defines the execution of the disconnect operation with two parameter pairs which disconnects specific connections. In the flow graph segment the first expression to be evaluated refers to <component-expression1> (see syntactical structure of the disconnect operation in ETSI ES 201 873-4 [2], clause 9.14) and the second expression to <component-expression2>, i.e. the <component-expression2> is on top of the value stack when the disconnect-two node is executed. Applying the disconnect operation to a static connection leads to a dynamic error.
6.30 Execute statement

The syntactical structure of the `execute` statement is:

```
execute(<testCaseId>({<act-par1>, … , <act-parn>})) [, <float-expression>] [, <config-expression>])
```

The `execute` statement describes the execution of a test case `<testCaseId>` with the (optional) actual parameters `<act-par1>, … , <act-parn>`. Optionally the execute statement may be guarded by a duration provided in form of an expression that evaluates to a `float`. If within the specified duration the test case does not return a verdict, a timeout exception occurs, the test configuration is destroyed and an `error` verdict is returned.

If a test case is executed on an existing static test configuration, the configuration shall be provided in form on an expression that evaluates to a `configuration`. If the test case does not return a verdict, a timeout exception occurs, the test configuration is destroyed and an `error` verdict is returned.

If no timeout exception occurs, the MTC is created or started, the control instance (representing the control part of the TTCN-3 module) is blocked until the test case terminates, and for the further test case execution the flow of control is given to the MTC. The flow of control is given back to the control instance when the MTC stops its execution.

The flow graph segment `<execute-stmt>` in figure 67 defines the execution of an `execute` statement. The operational semantics distinguishes the cases where a test case is executed on an existing static test configuration and where not.
6.31 **Flow graph segment `<execute-without-config>`**

The flow graph segment `<execute-without-config>` in figure 67a distinguishes between the case where the execution is guarded by a timeout and the case where the statement is not guarded.

6.32 **Flow graph segment `<execute-on-config>`**

The flow graph segment `<execute-on-config>` in figure 69a distinguishes between the case where the execution of a test case on a configuration is guarded by a timeout and the case where the execution is not guarded.

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

Executing a test case on a static configuration means to start the behaviour of the test case on the MTC of the test configuration, i.e. `MyMTC.start(TeStCaSeNaMe(P1...Pn))`.

- In addition the following parts of the configuration state have to be reset to the following values:
  - the global test case verdict and all local component verdicts are set to `none`;
  - the local default lists of all components of the test configuration are emptied;
the global lists DONE and KILLED are emptied. These lists are used for storing the test components that stopped their execution or have been killed during test execution.

The flow graph segment `<execute-on-config-without-timeout>` in figure 69b specifies the execution of a test case on a static configuration where the execution is not guarded by a timer. It makes use of the `start` component operation.

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

The flow graph segment `<execute-on-config-timeout>` in figure 69c defines the execution of a test case on a configuration that is guarded by a timeout value. The flow graph segment also models the execution of the test case by starting the behaviour of the test case on the MTC on an existing static test configuration. In addition, `TIMER-GUARD` guards the termination.
Figure 69c of ETSI ES 201 873-4 [2]: Flow graph segment <execute-on-config-timeout>
6.35 Flow graph segment <statement-block>

The syntactical structure of a statement block is:

{ <statement_1>; … ; <statement_n> }

A statement block is a scope unit. When entering a scope unit, new scopes for variables, timers and the value stack have to be initialized. When leaving a scope unit, all variables, timers and stack values of this scope have to be destroyed.

NOTE 1: A Statement block can be embedded in another statement blocks or can occur as body of functions, altsteps, test cases and module control, and within compound statements, e.g. alt if-else or do-while.

NOTE 2: Receiving operations and altstep calls cannot appear in statement blocks, they are embedded in alt statements or call operations.

NOTE 3: The operational semantics also handles operations and declarations like statements, i.e. they are allowed in statement blocks.

NOTE 4: Some TTCN-3 functions, like e.g. system or self, are considered to be expressions, which are not useful as stand-alone statements in statement blocks. Their flow graph representations are not listed in figure 78.

The flow graph segment <statement-block> in figure 78 defines the execution of a statement block.
6.36 Halt port operation

The syntactical structure of the `halt` port operation is:

```plaintext
<portId>.halt
```

The flow graph segment `<halt-port-op>` in figure 89a defines the execution of the `halt` port operation.
Figure 89a of ETSI ES 201 873-4 [2]: Flow graph segment <halt-port-op>

NOTE: The HALT–MARKER that is put by a halt operation into the port queue is removed by the SNAP–PORTS function (see ETSI ES 201 873-4 [2], clause 8.3.3.2) when the marker is reached, i.e. all messages preceding the marker have been processed. The SNAP–PORTS function is called when taking a snapshot.

6.37 Kill component operation

The syntactical structure of the kill component statement is:

```
<component-expression>.kill
```

The kill component operation stops the specified component and removes it from the test system. All test components will be stopped and removed from the test system, i.e. the test case terminates, if the MTC is killed (e.g. \texttt{mtc.kill}) or kills itself (e.g. \texttt{self.kill}). The MTC may kill all parallel test components by using the \texttt{all} keyword, i.e. \texttt{all component.kill}.

Special rules apply for using the \texttt{kill} component operation in static test configurations: Applying the \texttt{kill} component operation to a static component leads to a dynamic error. The lifetime of all static components (including the MTC) is bound to the lifetime of the test configuration. However, the MTC may kill all non-static parallel test components by using the \texttt{all} keyword, i.e. \texttt{all component.kill}.

A component to be killed is identified by a component reference provided as expression, e.g. a value or value returning function. For simplicity, the keyword "all component" is considered to be special values of \texttt{<component-expression>}. The operations \texttt{mtc} and \texttt{self} are evaluated according to ETSI ES 201 873-4 [2], clauses 9.33 and 9.43.

The flow graph segment \texttt{<kill-component-op>} in figure 90a defines the execution of the \texttt{kill} component operation.
segment <kill-component-op>

// The Expression shall evaluate
// to a component reference. The
// result is pushed onto VALUE-STACK

<expression>

decision

true

false

if (Entity.VALUE-STACK.top() == 'all component') {
  Entity.VALUE-STACK.pop(); // clean value stack
  if (Entity != MTC) {
    ***DYNAMIC-ERROR*** // 'all' not allowed
  } else {
    Entity.NEXT-CONTROL(true);
  }
} else {
  Entity.NEXT-CONTROL(false);
}
RETURN;

if (Entity.VALUE-STACK.top() == MTC) {
  ***DYNAMIC-ERROR*** // kill is applied to a
  // static component
} else if (Entity.VALUE-STACK.top() == ALL-ENTITY-STATES) {
  ***DYNAMIC-ERROR***
} else {
  Entity.NEXT-CONTROL(false);
}
RETURN;

if (ALL-ENTITY-STATES.member(Entity.VALUE-STACK.top())) {
  Entity.NEXT-CONTROL(true);
} else {
  if (KILLED.member(Entity.VALUE-STACK.top())){
    // NULL operation, component already terminated
    Entity.VALUE-STACK.pop(); // clean value stack
    Entity.NEXT-CONTROL(false);
  } else {
    // component id has not been allocated
    ***DYNAMIC-ERROR***
  }
  RETURN;
}

if (Entity.VALUE-STACK.top() == 'all') {
  Entity.VALUE-STACK.pop(); // clean value stack
  if (Entity != MTC) {
    ***DYNAMIC-ERROR*** // 'all' not allowed
  }
  Entity.NEXT-CONTROL(true);
} else {
  Entity.VALUE-STACK.pop(); // clean value stack
  Entity.NEXT-CONTROL(false);
}
RETURN;

if (Entit.Y.VALUE-STACK.top() == STATIC == true) {
  ***DYNAMIC-ERROR*** // kill is applied to a
  // static component
} else if (Entity.VALUE-STACK.top() == MTC) {
  Entity.VALUE-STACK.pop(); // clean value stack
  Entity.NEXT-CONTROL(true);
} else {
  Entity.NEXT-CONTROL(false);
}
RETURN;

<kill-mtc>

prepare-kill

false

true

<kill-component>

Figure 90a of ETSI ES 201 873-4 [2]: Flow graph segment <kill-component-op>
6.38 Flow graph segment <kill-mtc>

The <kill-mtc> flow graph segment in figure 90b describes the killing of the MTC. The effect is that the test case terminates, i.e. the final verdict is calculated and pushed onto the value stack of module control. The release of all resources are released is modelled by deleting the test configuration from the ALL-CONFIGURATIONS list.

```plaintext
let { // local scope for variables
  var myEntity := ALL-ENTITY-STATES.first();

  // Update test case verdict and deletion of components
  while (myEntity != NULL) {
    if (myEntity.E-VERDICT == fail or TC-VERDICT == fail) {
      TC-VERDICT := fail;
    } else {
      if (myEntity.E-VERDICT == inconc or TC-VERDICT == inconc) {
        TC-VERDICT := inconc;
      } else {
        if (myEntity.E-VERDICT == pass or TC-VERDICT == pass) {
          TC-VERDICT := pass;
        }
      }
      myEntity := ALL-ENTITY-STATES.next(myEntity);
    }

    // TC-VERDICT is the result of the execute operation
    CONTROL.VALUE-STACK.push(TC-VERDICT);

    // Update of test case reference parameters
    UPDATE-REMOTE-LOCATIONS(MTC, CONTROL);

    // Deletion of test configuration
    ALL-CONFIGURATIONS.delete(CONFIGURATION)

    // Resetting of global variables
    ALL-ENTITY-STATES := NULL;
    ALL-PORT-STATES := NULL;
    DONE := NULL;
    KILLED := NULL;
    MTC := NULL;
    CONTROL.STATUS := ACTIVE; // Control continues
  }
} // End of local scope
RETURN;
```

Figure 90b of ETSI ES 201 873-4 [2]: Flow graph segment <kill-mtc-op>

6.39 Flow graph segment <kill-all-comp>

The <kill-all-comp> flow graph segment in figure 90d describes the termination of all parallel test components of a test case.
```javascript
let { // local scope for variable myEntity
    var myEntity := ALL-ENTITY-STATES.next(MTC);
    var port;
    var connection;

    // Update test case verdict
    while (myEntity != NULL) {
        if (myEntity.STATIC != true) { // not a static test component
            if (myEntity.E-VERDICT == fail or TC-VERDICT == fail) {
                TC-VERDICT := fail;
            } else {
                if (myEntity.E-VERDICT == inconc or TC-VERDICT == inconc) {
                    TC-VERDICT := inconc;
                } else {
                    if (myEntity.E-VERDICT == pass or TC-VERDICT == pass) {
                        TC-VERDICT := pass;
                    }
                }
            }
            myEntity := ALL-ENTITY-STATES.next(myEntity);
        } else { // a static test component
            myEntity := ALL-ENTITY-STATES.next(myEntity);
        }
    }

    // Deletion of test components
    myEntity := ALL-ENTITY-STATES.next(MTC);
    while (myEntity != NULL) {
        if (myEntity.STATIC == true) { // a static test component
            myEntity := ALL-ENTITY-STATES.next(myEntity);
        } else { // not a static test component
            // disconnect and unmap component
            port := ALL-PORT-STATES.first();
            while (port != NULL) {
                connection := port.CONNECTIONS.first();
                while (connection != NULL) {
                    if (connection.REMOTE-ENTITY == comp
                        or (port.OWNER == comp) {
                        port.CONNECTIONS.delete(connection);
                        connection := port.CONNECTIONS.first();
                    } else {
                        connection := port.CONNECTIONS.next(connection);
                    }
                }
                port := ALL-PORT-STATES.next(port);
            }
            DONE.append(myEntity); // Update of DONE
            KILLED.append(myEntity); // Update of KILLED
            DEL-ENTITY(myEntity); // Deletion of entity
            myEntity := ALL-ENTITY-STATES.next(MTC); // Next component to delete
        }
    }
}
// End of local scope
Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 90d of ETSI ES 201 873-4 [2]: Flow graph segment <stop-all-comp>
6.40 Kill execution statement

The syntactical structure of the **kill** execution statement is:

```
kill
```

The effect of the **kill** execution statement depends on the entity that executes the **kill** execution statement:

a) If **kill** is performed by the module control, the test campaign ends, i.e. all test components and the module control disappear from the module state.

b) If the **kill** is executed by the MTC, all parallel test components and the MTC stop execution. The global test case verdict is updated and pushed onto the value stack of the module control. Finally, control is given back to the module control and the MTC terminates.

c) If the **kill** is executed by a test component, the global test case verdict TC-VERDICT and the global DONE and KILLED lists are updated. Then the component disappears from the module.

The execution of the **kill** execution statement by any static test component (including the MTC of a static test configuration) is not allowed. It leads to a dynamic error.

The flow graph segment `<kill-exec-stmt>` in figure 90e describes the execution of the kill statement.

```plaintext
segment <kill-exec-stmt>

if (Entity == CONTROL {
    Entity.NEXT-CONTROL(true);
})
else {
    Entity.NEXT-CONTROL(false);
}
RETURN;

if (Entity.STATIC == true) {
    ***DYNAMIC-ERROR*** // static Entity
} else {
    if (Entity == MTC) {
        Entity.NEXT-CONTROL(true);
    } else {
        Entity.VALUE-STACK.push(Entity);
        Entity.NEXT-CONTROL(false);
    }
    RETURN;

return <kill-mtc> <kill-component>
```

Figure 90e of ETSI ES 201 873-4 [2]: Flow graph segment `<kill-exec-stmt>`
6.41 Kill configuration operation

The syntactical structure of the **kill** configuration operation is:

```
<configuration-expression>.kill
```

The **kill** configuration operation destructs the specified test configuration and removes it from the test system. The kill configuration operation shall only be executed by module control. The configuration to be killed is identified by means of a `<configuration-expression>`, i.e. an expression that evaluates to a reference to a configuration.

The flow graph segment `<kill-config-op>` in figure 90f defines the execution of the **kill** configuration operation.

```
let { // begin of a local scope
    var config := Entity.VALUE-STACK.top();
    Entity.VALUE-STACK.pop();
    if (Entity != CONTROL) {
        ***DYNAMIC-ERROR*** // kill config operation is not
        // invoked by module control
    } else if (ALL-CONFIGURATIONS.member(config) != true) {
        ***DYNAMIC-ERROR*** // configuration to be killed
        // does not exist
    } else {
        ALL-CONFIGURATIONS.delete(config)
    }
} // end of local scope

Entity.NEXT-CONTROL(true);
RETURN;
```

**Figure 90f of ETSI ES 201 873-4 [2]: Flow graph segment <kill-config-op>**

6.42 Map operation

The syntactical structure of the **map** operation is:

```
map(<component-expression>::<portId1>, system:<portId2>) [static]
```

The identifiers `<portId1>` and `<portId2>` are considered to be port identifiers of the corresponding test component and test system interface. The component to which the `<portId1>` belongs is referenced by means of the component reference `<component-expression>`. The reference may be stored in variables or is returned by a function, i.e. it is an expression, which evaluates to a component reference. The value stack is used for storing the component reference.

A present `static` clause indicates that the new mapping is static, i.e. established during the execution of a configuration function. Presence and absence of the `static` clause is handled as a Boolean flag in the operational semantics (see `static` parameter of the basic flow graph node `map-op` in figure 93).

**NOTE:** The **map** operation does not care whether the `system:<portId>` statement appears as first or as second parameter. For simplicity, it is assumed that it is always the second parameter.

The execution of the **map** operation is defined by the flow graph segment `<map-op>` shown in figure 93.
Figure 93 of ETSI ES 201 873-4 [2]: Flow graph segment <map-op>

6.43 Start port operation

The syntactical structure of the start port operation is:

```
start
```

The flow graph segment <start-port-op> in figure 121 defines the execution of the start port operation.
6.44 Stop component operation

The syntactical structure of the `stop` component statement is:

```
<component-expression>.stop
```

The `stop` component operation stops the specified component. All test components will be stopped, i.e. the test case terminates, if the MTC is stopped (e.g. `mtc.stop`) or stops itself (e.g. `self.stop`). The MTC may stop all parallel test components by using the `all` keyword, i.e. `all component.stop`.

Stopped components created with an `alive` clause in the `create` operation are not removed from the test system. They can be restarted by using a `start` statement. Variables, ports, constants and timers owned by such a component, i.e. declared and defined in the corresponding component type definition, keep their status. A `stop` operation for a component created without an `alive` clause is semantically equivalent to a `kill` operation. The component is removed from the test system.

A component to be stopped is identified by a component reference provided as expression, e.g. a value or value returning function. For simplicity, the keyword "all component" is considered to be special values of `<component-expression>`. The operations `mtc` and `self` are evaluated according to ETSI ES 201 873-4 [2], clauses 9.33 and 9.43.

The flow graph segment `<stop-component-op>` in figure 125 defines the execution of the `stop` component operation.

---

**Figure 121 of ETSI ES 201 873-4 [2]: Flow graph segment <start-port-op>**

```javascript
let { // Begin of local scope
  var portRef := NULL
  var portState := NULL;

  if (Entity.STATIC == true) {
    ***DYNAMIC-ERROR*** // port operation on a
    // static test component
  } else if (portId == "all port") {
    portState := ALL-PORT-STATES.first();
    while (portState != NULL) {
      if (portState.OWNER == Entity) {
        portState.VALUE-QUEUE.clear();
        portState.STATUS := STARTED
      }
      portState := ALL-PORT-STATES.next(portState);
    }
  } else {
    portRef := Entity.portId.COMP-PORT-NAME;
    GET-PORT(Entity, portRef).clear();
    GET-PORT(Entity, portRef).STATUS := STARTED;
  } // End of scope

Entity.NEXT-CONTROL(true);
RETURN;
```
// The Expression shall evaluate
// to a component reference. The
// result is pushed onto VALUE-STACK

if (Entity.VALUE-STACK.top() == 'all component') {
    Entity.VALUE-STACK.pop(); // clean value stack
    if (Entity != MTC) {
        ***DYNAMIC-ERROR*** // 'all' not allowed
    } else {
        Entity.NEXT-CONTROL(true);
    }
} else {
    Entity.NEXT-CONTROL(false);
}
RETURN;

if (ALL-ENTITY-STATES.member(Entity.VALUE-STACK.top())) {
    Entity.NEXT-CONTROL(true);
} else {
    if (DONE.member(Entity.VALUE-STACK.top())) {
        // NULL operation, component already stopped
        // or killed.
        Entity.VALUE-STACK.pop(); // clean value stack
        Entity.NEXT-CONTROL(false);
    } else {
        // component id has not been allocated
        ***DYNAMIC-ERROR***
    }
    RETURN;

if (Entity.VALUE-STACK.top().KEEP-ALIVE == true)) {
    Entity.NEXT-CONTROL(true); // Component is not
    // removed from the
    // test system
} else {
    Entity.NEXT-CONTROL(false); // Component is killed
}
RETURN;

Figure 125 of ETSI ES 201 873-4 [2]: Flow graph segment <stop-component-op>
6.45 Flow graph segment <stop-mtc>

The flow graph segment <stop-mtc> in figure 125a describes the stopping of an MTC. The effect of stopping an MTC is that a test case or a configuration function terminates. Depending on where and how an MTC has been executed, three cases have to be distinguished:

1) The MTC stops the behaviour of a test case that has not been executed on a static test configuration.
2) The MTC stops the behaviour of a test case that has been executed on a static test configuration.
3) The MTC stops the execution of a configuration function.

```
if (MT.CSTATIC == false) {
    // stopping a test case that is not executed
    // on static test configuration
    Entity.NEXT-CONTROL(false);
} else {
    // stopping either a test case that has been
    // executed on a static test configuration or
    // a configuration function terminates
    Entity.NEXT-CONTROL(true);
}
RETURN;
```

```
if (CONFIGURATION.STATIC == true) {
    // termination of a configuration function
    Entity.NEXT-CONTROL(true);
} else {
    // stopping a test case executed
    // on a static configuration
    Entity.NEXT-CONTROL(false);
}
```

Figure 125a of ETSI ES 201 873-4 [2]: Flow graph segment <stop-mtc>
6.46 Flow graph segment <stop-config>

The <stop-config> flow graph segment in figure 127a describes the stopping of an MTC that has executed a configuration function.

```javascript
let { // local scope
    var compVarScope := copy(MTC.DATA-STATE.first());
    var compTimerScope := copy(MTC.TIMER-STATE.first());
    var compPortScope := copy(MTC.PORT-REF.first());

    // Update of component state. This is necessary, if the behaviour of the configuration function is structured into further function.
    MTC.STATUS := BLOCKED;
    MTC.CONTROL-STACK := NULL;
    MTC.DEFAULT-LIST := NULL;
    MTC.VALUE-STACK := NULL;
    MTC.VALUE-STACK.push(MARK); // for component scope
    MTC.TIMER-GUARD.STATUS := IDLE;
    MTC.DATA-STATE := NULL
    MTC.DATA-STATE.add(compVarScope);
    MTC.TIMER-STATE := NULL;
    MTC.TIMER-STATE.add(compTimerScope);
    MTC.PORT-REF := NULL
    MTC.PORT-REF.add(compPortScope);
    MTC.SNAP-ALIVE := NULL;
    MTC.SNAP-DONE := NULL;
    MTC.SNAP-KILLED := NULL;

    // Update of test case reference parameters
    UPDATE-REMOTE-LOCATIONS(MTC, CONTROL);

    CONTROL.STATUS := false; // Reset of STATIC flag in module control
    CONTROL.STATUS := ACTIVE; // Control continues execution
}

} // End of local scope
RETURN;
```

Figure 127a of ETSI ES 201 873-4 [2]: Flow graph segment <stop-config>
6.47 Flow graph segment <stop-tc-config>

The <stop-tc-config> flow graph segment in figure 127b describes the termination of a test case that is executed on a static test configuration.

```plaintext
let { // local scope
    var compVarScope := copy(MTC.DATA-STATE.first());
    var compTimerScope := copy(MTC.TIMER-STATE.first());
    var compPortScope := copy(MTC.PORT-REF.first());
    // Update test case verdict
    if (MTC.E-VERDICT == fail or TC-VERDICT == fail) {
        TC-VERDICT := fail;
    } else if (MTC.E-VERDICT == inconc or TC-VERDICT == inconc) {
        TC-VERDICT := inconc;
    } else if (MTC.E-VERDICT == pass or TC-VERDICT == pass) {
        TC-VERDICT := pass;
    }
    // TC-VERDICT is the result of the execute operation
    CONTROL.VALUE-STACK.push(TC-VERDICT);
    // Update of test case reference parameters
    UPDATE-REMOTE-LOCATIONS(MTC, CONTROL);
    // Update of component state, if the behaviour of the configuration function is structured into functions.
    MTC.STATUS := BLOCKED;
    MTC.CONTROL-STACK := NULL;
    MTC.DEFAULT-LIST := NULL;
    MTC.VALUE-STACK := NULL;
    MTC.VALUE-STACK.push(MARK); // for component scope
    MTC.TIMER-GUARD.STATUS := IDLE;
    MTC.DATA-STATE := NULL
    MTC.DATA-STATE.add(compVarScope);
    MTC.TIMER-STATE := NULL;
    MTC.TIMER-STATE.add(compTimerScope);
    MTC.PORT-REF := NULL
    MTC.PORT-REF.add(compPortScope);
    MTC.SNAP-ALIVE := NULL;
    MTC.SNAP-DONE := NULL;
    MTC.SNAP-KILLED := NULL;
    CONTROL.STATUS := ACTIVE; // Control continues execution
}
// End of local scope
RETURN;
```

Figure 127b of ETSI ES 201 873-4 [2]: Flow graph segment <stop-tc-config>
6.48 Stop port operation

The syntactical structure of the `stop` port operation is:

```
<portId>.stop
```

The flow graph segment `<stop-port-op>` in figure 129 defines the execution of the `stop` port operation.

```plaintext
let { // Begin of local scope
    var portRef := NULL;
    var portState := NULL;

    if (Entity.STATIC == true) {
        ***DYNAMIC-ERROR*** // port operation on a // static test component
    }
    else if (portId == "all port") {
        portState := ALL-PORT-STATES.first();
        while (portState != NULL) {
            if (portState.OWNER == Entity) {
                portState.STATUS := STOPPED
            }
            portState :=
                ALL-PORT-STATES.next(portState);
        }
    }
    else {
        portRef := Entity.portId.COMP-PORT-NAME;
        GET-PORT(Entity, portRef).STATUS := STOPPED;
    } // End of scope

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 129 of ETSI ES 201 873-4 [2]: Flow graph segment `<stop-port-op>`
6.49 Flow graph segment <unmap-all>

The flow graph segment <unmap-all> defines the unmapping of all components at all mapped ports. Static mappings will not be unmapped. Their lifetime is bound to the lifetime of the static test configuration.

```plaintext
let { // local scope
    var port := ALL-PORT-STATES.first();
    var connection;
    while (port != NULL) {
        connection := port.CONNECTIONS.first();
        while (connection != NULL) {
            if (connection.REMOTE-ENTITY == system) { // mapped port
                if (connection.STATIC == true) { // static mapping
                    connection := port.CONNECTIONS.next(connection);
                } else {
                    port.CONNECTIONS.delete(connection);
                    connection := port.CONNECTIONS.first();
                }
            } else {
                connection := NULL; // connected port
            }
        }
        port := ALL-PORT-STATES.next(port)
    } // End of local scope
Entity.NEXT-CONTROL(true);
RETURN;
}
```

Figure 136a of ETSI ES 201 873-4 [2]: Flow graph segment <unmap-all>
6.50 Flow graph segment <unmap-comp>

The flow graph segment <unmap-comp> defines the unmapping of all mapped ports of a specified component. Static mappings will not be unmapped. Their lifetime is bound to the lifetime of the static test configuration.

```plaintext
let { // local scope
    var comp := Entity.VALUE-STACK.top();
    var connection;
    var port := ALL-PORT-STATES.first();

    while (port != NULL) {
        if (port.OWNER == comp) { // port of comp
            connection := port.CONNECTIONS.first();
            if (connection.REMOTE-ENTITY == system) { // mapped port of comp
                if (connection.STATIC != true) { // not a static mapping
                    port.CONNECTIONS.delete(connection);
                }
            }
        }
        port := ALL-PORT-STATES.next(port);
    }
    Entity.VALUE-STACK.pop(); // clear value stack
} // End of local scope

Entity.NEXT-CONTROL(true);
RETURN;
```

Figure 136b of ETSI ES 201 873-4 [2]: Flow graph segment <unmap-comp>
6.51 Flow graph segment <unmap-port>

The flow segment <unmap-port> defines the unmap operation for a specific mapped port.

```plaintext
segment <unmap-port>

let { // local scope
  var portId;
  var comp;
  var port;
  var connection;

  portId := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop();
  comp := Entity.VALUE-STACK.top();
  Entity.VALUE-STACK.pop();
  port := GET-PORT(comp, portId);

  connection := port.CONNECTIONS.first();
  if (connection.REMOTE-ENTITY != SYSTEM) {
    ***DYNAMIC-ERROR*** // port is not a mapped port
  }
  else if (connection != NULL) { // mapped port
    if (connection.STATIC == true) { // static mapping
      ***DYNAMIC-ERROR*** // static mappings cannot be unmapped
    }
    else {
      port.CONNECTIONS.delete(connection);
    }
  }
  else { // do nothing, port is neither connected nor mapped
    Entity.NEXT-CONTROL(true);
  }

RETURN;
```

Figure 136c of ETSI ES 201 873-4 [2]: Flow graph segment <unmap-port>

7 TRI Extensions for the Package

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

If this package is being used, the TriExecuteTestCase operation shall be used only for initialization purposes of the SA, but not for the establishment of static connections. In order to establish static connections, the TriStaticMap operation shall be used instead. The TriUnmap can be used for closing dynamic and static connections.
Clause 5.5.2.1  triExecuteTestCase (TE → SA)

This clause is changed as follows.

| Signature       | TriStatusType triExecuteTestCase( |
| In Parameters   | in TriTestIdType testId, |
|                | in TriPortIdListType tsiPortList) |

**In Parameters**
- `testCaseId`: identifier of the test case that is going to be executed
- `tsiPortList`: a list of test system interface ports defined for the test system

**Out Parameters**
- n.a.

**Return Value**
The return status of the `triExecuteTestCase` operation. The return status indicates the local success (`TRI_OK`) or failure (`TRI_Error`) of the operation.

**Constraints**
This operation is called by the TE immediately before the execution of any test case. The test case that is going to be executed is indicated by the `testCaseId`. If a system component has not been explicitly defined for the test case in the TTCN-3 ATS then the `tsiPortList` contains all communication ports of the MTC test component. The ports in `tsiPortList` are ordered as they appear in the respective TTCN-3 component declaration.

**Effect**
The SA can set up any static connections to the SUT and can initialize any communication means for TSI ports. The `triExecuteTestCase` operation returns `TRI_OK` in case the operation has been successfully performed, `TRI_Error` otherwise.

Clause 5.5.2.3  triUnmap (TE → SA)

This clause is changed as follows.

| Signature       | TriStatusType triUnmap(in TriPortIdType compPortId, |
| In Parameters   | in TriPortIdType tsiPortId) |

**In Parameters**
- `compPortId`: identifier of the test component port to be unmapped
- `tsiPortId`: identifier of the test system interface port to be unmapped

**Out Parameters**
- n.a.

**Return Value**
The return status of the `triUnmap` operation. The return status indicates the local success (`TRI_OK`) or failure (`TRI_Error`) of the operation.

**Constraints**
This operation is called by the TE when it executes any TTCN-3 unmap operation.

**Effect**
The SA shall close a dynamic or static connection to the SUT for the referenced TSI port. The `triUnmap` operation returns `TRI_Error` in case a connection could not be closed successfully or no such connection has been established previously, `TRI_OK` otherwise. The operation should return `TRI_OK` in case no connections have to be closed by the test system.

Clause 5.5.2.3  triStaticMapParam (TE → SA)

| Signature       | TriStatusType triStaticMapParam(in TriPortIdType compPortId, |
| In Parameters   | in TriPortIdType tsiPortId, |
|                | in TriParameterListType paramList) |

**In Parameters**
- `compPortId`: identifier of the test component port to be mapped
- `tsiPortId`: identifier of the test system interface port to be mapped
- `paramList`: configuration parameter list

**Out Parameters**
- n.a.

**Return Value**
The return status of the `triStaticMapParam` operation. The return status indicates the local success (`TRI_OK`) or failure (`TRI_Error`) of the operation.

**Constraints**
This operation is called by the TE when it executes a TTCN-3 map operation including parameters.

**Effect**
The SA can establish a static connection to the SUT for the referenced TSI port. The `triStaticMapParam` operation returns `TRI_Error` in case a connection could not be established successfully, `TRI_OK` otherwise. The operation should return `TRI_OK` in case no static connection needs to be established by the test system. The configuration parameter `paramList` can be used for setting connection establishment specific parameters.
Clause 5.5.2.5  
**triStaticMap (TE \(\rightarrow\) SA)**

This clause is to be added.

<table>
<thead>
<tr>
<th>Signature</th>
<th>TriStatusType triStaticMap(in TriPortIdType compPortId, in TriPortIdType tsiPortId)</th>
</tr>
</thead>
</table>
| In Parameters | compPortId identifier of the test component port to be mapped in a static connection  
  tsiPortId identifier of the test system interface port to be mapped in a static connection |
| Out Parameters | n.a. |
| Return Value | The return status of the triStaticMap operation. The return status indicates the local success (TRI_OK) or failure (TRI_Error) of the operation. |
| Constraints | This operation is called by the TE when it executes a TTCN-3 static map operation. |
| Effect | The SA can establish a static connection to the SUT for the referenced TSI port. The triStaticMap operation returns TRI_OK in case a connection could not be established successfully, TRI_OK otherwise. The operation should return TRI_OK in case no static connection needs to be established by the test system. |

7.2  
**Extensions to clause 6 of ETSI ES 201 873-5 [3] Java™ language mapping**

Clause 6.5.2.1  
**triCommunicationSA**

The triCommunicationSA interface mapping is to be extended with the definition for TriStaticMap:

```java
// TriCommunication  
// TE \(\rightarrow\) SA
package org.etsi.ttcn.tri;
public interface TriCommunicationSA {  
  // Connection handling operations  
  // Ref: TRI-Definition 5.5.2.5
  public TriStatus triStaticMap(TriPortId compPortId, TriPortId tsiPortId);
};
```

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

Clause 7.2.4  
**TRI operation mapping**

The table is to be extended with the definition for TriStaticMap:

<table>
<thead>
<tr>
<th>IDL Representation</th>
<th>ANSI C Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TriStatusType triStaticMap (in TriPortIdType compPortId, in TriPortIdType tsiPortId)</td>
<td>TriStatus triStaticMap (const TriPortId* compPortId, const TriPortId* tsiPortId)</td>
</tr>
</tbody>
</table>

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

Clause 8.6.1  
**TriCommunicationSA**

The triCommunicationSA interface mapping is to be extended with the definition for TriStaticMap. In addition, the description of TriUnmap has to be changed to handle also the closing of static connections:

```cpp
class TriCommunicationSA {  
public:  
  //To establish a static connection between two ports.
  virtual TriStatus triStaticMap (const TriPortId *compPortId, const TriPortId *tsiPortId)=0;
};
```
virtual TriStatus triUnmap (const TriPortId *comPortId, const TriPortId *tsiPortId)=0;

7.5 Extensions to clause 9 of ETSI ES 201 873-5 [3]

C# language mapping

Clause 9.5.2.1 ITriCommunicationSA

The ITriCommunicationSA interface is to be extended with the definition for TriStaticMap. In addition, the description of TriUnmap has to be changed to handle also the closing of static connections:

```c
public interface ITriCommunicationSA {
    //To establish a static connection between two ports.
    TriStatus TriStaticMap (ITriPortId comPortId, ITriPortId tsiPortId);
    
    //To close a dynamic or static connection to the SUT for the referenced TSI port.
    virtual TriStatus triUnmap (const TriPortId *comPortId, const TriPortId *tsiPortId)=0;
};
```

8 TCI Extensions for the Package

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

Management

The management type TciTestComponentKindType has to be extended with constants for static test components:

```c
TciTestComponentKindType
```

A value of type TciTestComponentKindType is a literal of the set of kinds of TTCN-3 test components, i.e. CONTROL, MTC, PTC, SYSTEM, PTC_ALIVE, MTC_STATIC, PTC_STATIC, and SYSTEM_STATIC. This abstract type is used for component handling.

8.2 Extensions to clause 7.3.1.1 of ETSI ES 201 873-6 [4]

TCI-TM required

In order to handle static configurations via TCI-TM, the operations tciStartConfig and tciKillConfig are defined as follows.

Clause 7.3.1.1.7 tciStartTestCase

This clause is to be extended.

<table>
<thead>
<tr>
<th>Signature</th>
<th>void tciStartTestCase(in TciTestCaseIdType testCaseId, in TciParameterListType parameterList, in TciConfigurationIdType ref)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Parameters</td>
<td>testCaseId</td>
</tr>
<tr>
<td></td>
<td>parameterList</td>
</tr>
<tr>
<td></td>
<td>ref</td>
</tr>
<tr>
<td>Return Value</td>
<td>void</td>
</tr>
<tr>
<td>Constraint</td>
<td>Shall be called only if a module has been selected before. Only testCaseIds for test cases that are declared in the currently selected TTCN-3 module shall be passed. Only if the test case requires a static configuration, a ref to a started static configuration that is of the configuration type in the test case definition shall be given. If no static configuration has been defined for the test case in the TTCN-3 ATS, the distinct value null shall be passed in for ref. Test cases that are imported in a referenced module cannot be started. To start imported test cases the referenced (imported) module shall be selected first using the tciRootModule operation.</td>
</tr>
<tr>
<td>Effect</td>
<td>tciStartTestCase starts a test case in the currently selected module with the given parameters. A tciError will be issued by the TE if no such test case exists or if the static configuration has not been started or has been killed but is required by the test case. All in and inout test case parameters in parameterList contain Value. All out test case parameters in parameterList shall contain the distinct value of null since they are only of relevance when the test case terminates.</td>
</tr>
</tbody>
</table>

**Clause 7.3.1.11  tciStartConfig**

This clause is to be added.

| Signature | TciConfigurationIdType tciStartConfig (in TciBehaviourIdType configId, in TciParameterListType parameterList) |
| In Parameters | configId | A configuration function identifier as defined in the TTCN-3 module. |
| | parameterList | A list of Values where each value defines a parameter from the parameter list as defined in the TTCN-3 configuration function definition. The parameters in parameterList are ordered as they appear in the TTCN-3 signature of the configuration function. If no parameters have to be passed either the null value or an empty parameterList, i.e. a list of length zero shall be passed. |
| Return Value | TciConfigurationIdType |
| Constraint | Shall be called only if a module has been selected before. Only configId for test cases with static configurations that are declared in the currently selected TTCN-3 module shall be passed - see tciStartTestCase. |
| Effect | Starts a static configuration of the selected module as described in the TTCN-3 configuration function and returns an identifier to this configuration. A static configuration started from TCI-TM will be used by test cases that reference the static configuration and are executed from TCI-TM. |

**Clause 7.3.1.12  tciKillConfig**

This clause is to be added.

| Signature | void tciKillConfig(in TciConfigurationIdType ref) |
| In Parameters | ref | The reference to the static configuration. |
| Return Value | void |
| Constraint | Shall be called only if a module has been selected before. |
| Effect | tciKillConfig causes the destruction of the static test configuration ref. If ref is currently not started, the operation will be ignored. |

**Clause 7.3.3.1.18  tciExecuteTestCase**

This clause is to be extended.

| Signature | void tciExecuteTestCase (in TciTestCaseIdType testCaseId, in TriPortIdListType tsiPortList, in TciConfigurationIdType ref) |
| In Parameters | testCaseId | A test case identifier as defined in the TTCN-3 module. |
| | tsiPortList | Contains all ports that have been declared in the definition of the system component for the test case or in the configuration type of the test case, i.e. the TSI ports. If a system component has not been explicitly defined for the test case, then the tsiPortList contains all communication ports of the MTC. The ports in tsiPortList are ordered as they appear in the TTCN-3 component type declaration or in the configuration type declaration respectively. If no ports have to be passed either the null value or an empty tsiPortList, i.e. a list of length zero shall be passed. |
| | ref | An optional reference to a static configuration for the test case, which is to be used if a static configuration has been defined for the test case. |
Return Value  | void  
---|---
Constraint  | This operation shall be called by the CH at the appropriate local TE when at a remote TE a provided 
tciExecuteTestCaseReq has been called. Only if the test case requires a static configuration, a ref to a started static configuration that is of 
the configuration type in the test case definition shall be given. If no static configuration has been 
defined for the test case in the TTCN-3 ATS, the distinct value null shall be passed in for ref. 

Effect  | The local TE determines whether static connections to the SUT and the initialization of communication means for TSI ports should be done. This is for example not the case if the static configuration has been started already.

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

In order to enable the indication of static configuration start and destruction at TCI-TM, the operations tciConfigStarted and tciConfigKilled are defined as follows.

**Clause 7.3.1.2.7 tciConfigStarted**

This clause is to be added.

<table>
<thead>
<tr>
<th>Signature</th>
<th>void tciConfigStarted(in TciConfigurationIdType ref)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Parameters</td>
<td>ref</td>
</tr>
<tr>
<td>Return Value</td>
<td>TciConfigurationIdType</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Shall only be called after the static configuration has been started either using the required operations tciStartConfig or internally by the TE.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effect</td>
<td>tciConfigStarted indicates to the TM that static configuration ref has been started. It will not be distinguished whether the static configuration has been started explicitly using the required operation tciStartConfig or implicitly while executing the control part.</td>
</tr>
</tbody>
</table>

**Clause 7.3.1.2.8 tciConfigKilled**

This clause is to be added.

<table>
<thead>
<tr>
<th>Signature</th>
<th>void tciConfigKilled(in TciConfigurationIdType ref)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Parameters</td>
<td>ref</td>
</tr>
<tr>
<td>Return Value</td>
<td>TciConfigurationIdType</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Shall only be called after the static configuration has been killed either using the required operations tciKillConfig or internally by the TE.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effect</td>
<td>tciConfigKilled indicates to the TM that static configuration ref has been destructed. It will not be distinguished whether the static configuration has been started explicitly using the required operation tciKillConfig or implicitly while executing the control part.</td>
</tr>
</tbody>
</table>

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

In order to establish static connections, the tciStaticConnect and tciStaticMap operations shall be used at TCI-CH. The tciDisconnect and TciUnmap can be used for closing static connections.
Clause 7.3.3.1.21  tciStaticConnect

This clause is to be added.

<table>
<thead>
<tr>
<th>Signature</th>
<th>void tciStaticConnect(in TriPortIdType fromPort, in TriPortIdType toPort)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Parameters</td>
<td>fromPort Identifier of the test component port to be connected from.</td>
</tr>
<tr>
<td></td>
<td>toPort Identifier of the test component port to be connected to.</td>
</tr>
<tr>
<td>Return Value</td>
<td>void</td>
</tr>
<tr>
<td>Constraint</td>
<td>This operation shall be called by the CH at the local TE when at a remote TE a provided tciStaticConnectReq has been called.</td>
</tr>
<tr>
<td>Effect</td>
<td>The TE shall statically connect the indicated ports to one another.</td>
</tr>
</tbody>
</table>

Clause 7.3.3.1.21  tciStaticMap

This clause is to be added.

<table>
<thead>
<tr>
<th>Signature</th>
<th>void tciStaticMap(in TriPortIdType fromPort, in TriPortIdType toPort)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Parameters</td>
<td>fromPort Identifier of the test component port to be mapped from.</td>
</tr>
<tr>
<td></td>
<td>toPort Identifier of the test component port to be mapped to.</td>
</tr>
<tr>
<td>Return Value</td>
<td>void</td>
</tr>
<tr>
<td>Constraint</td>
<td>This operation shall be called by the CH at the local TE when at a remote TE a provided tciStaticMapReq has been called.</td>
</tr>
<tr>
<td>Effect</td>
<td>The TE shall statically map the indicated ports to one another.</td>
</tr>
</tbody>
</table>

8.5    Extensions to clause 7.3.3.2 of ETSI ES 201 873-6 [4]

TCI CH provided

In order to establish static connections, the tciStaticConnectReq and tciStaticMapReq operations shall be used at TCI-CH. The tciDisconnectReq and TciUnmapReq can be used for closing static connections.

Clause 7.3.3.2.26  tciExecuteTestCaseReq

This clause is to be extended.

<table>
<thead>
<tr>
<th>Signature</th>
<th>void tciExecuteTestCaseReq (in TciTestIdType testCaseId, in TriPortIdListType tsiPortList, in TciConfigurationIdType ref)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Parameters</td>
<td>testCaseId A test case identifier as defined in the TTCN-3 module.</td>
</tr>
<tr>
<td></td>
<td>tsiPortList contains all ports that have been declared in the definition of the system component for the test case or in the configuration type of the test case, i.e. the TSI ports. If a system component has not been explicitly defined for the test case, then the tsiPortList contains all communication ports of the MTC. The ports in tsiPortList are ordered as they appear in the TTCN-3 component type declaration or in the configuration type declaration respectively. If no ports have to be passed either the null value or an empty tsiPortList, i.e. a list of length zero shall be passed.</td>
</tr>
<tr>
<td></td>
<td>ref An optional reference to a static configuration for the test case, which is to be used if a static configuration has been defined for the test case.</td>
</tr>
<tr>
<td>Return Value</td>
<td>void</td>
</tr>
<tr>
<td>Constraint</td>
<td>This operation can be called by the TE immediately before it starts the test case behaviour on the MTC (in course of a TTCN-3 execute operation). Only if the test case requires a static configuration, a ref to a started static configuration that is of the configuration type in the test case definition shall be given. If no static configuration has been defined for the test case in the TTCN-3 ATS, the distinct value null shall be passed in for ref.</td>
</tr>
<tr>
<td>Effect</td>
<td>CH transmits the execute test case request to the remote TEs having system ports of the indicated test case. Static connections to the SUT and the initialization of communication means for TSI ports can be set up if needed. This is for example not the case if the static configuration has been started.</td>
</tr>
</tbody>
</table>
Clause 7.3.3.29  tciStaticConnectReq

This clause is to be added.

<table>
<thead>
<tr>
<th>Signature</th>
<th>void tciStaticConnectReq(in TriPortIdType fromPort, in TriPortIdType toPort)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Parameters</td>
<td>fromPort</td>
</tr>
<tr>
<td></td>
<td>toPort</td>
</tr>
<tr>
<td>Return Value</td>
<td>void</td>
</tr>
<tr>
<td>Constraint</td>
<td>This operation shall be called by the TE when it executes a TTCN-3 static connect operation.</td>
</tr>
<tr>
<td>Effect</td>
<td>CH transmits the static connection request to the remote TE where it calls the tciStaticConnect operation to establish a logical static connection between the two indicated ports. Note that both ports can be on remote TEs. In this case, the operation returns only after calling the tciStaticConnect operation on both remote TEs.</td>
</tr>
</tbody>
</table>

Clause 7.3.3.1.30  tciStaticMapReq

This clause is to be added.

<table>
<thead>
<tr>
<th>Signature</th>
<th>void tciStaticMapReq(in TriPortIdType fromPort, in TriPortIdType toPort)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Parameters</td>
<td>fromPort</td>
</tr>
<tr>
<td></td>
<td>toPort</td>
</tr>
<tr>
<td>Return Value</td>
<td>void</td>
</tr>
<tr>
<td>Constraint</td>
<td>This operation shall be called by the TE when it executes a TTCN-3 static map operation.</td>
</tr>
<tr>
<td>Effect</td>
<td>CH transmits the static map request to the remote TE where it calls the tciStaticMap operation to establish a logical static connection between the two indicated ports.</td>
</tr>
</tbody>
</table>

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

In order to log the handling of static connections and of static components, the operations are tliCStaticCreate, tliPStaticConnect, and tliPStaticMap are defined. For the logging of the starting and destruction of static configurations, the operations tliConfigStarted and tliConfigKilled are defined.

Clause 7.3.4.1.106  tliCStaticCreate

This clause is to be added.

<table>
<thead>
<tr>
<th>Signature</th>
<th>void tliCStaticCreate(in TString am, in TInteger ts, in TString src, in TInteger line, in TriComponentIdType c, in TriComponentIdType comp, in TString name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In Parameters</td>
<td>am</td>
</tr>
<tr>
<td></td>
<td>ts</td>
</tr>
<tr>
<td></td>
<td>src</td>
</tr>
<tr>
<td></td>
<td>line</td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>comp</td>
</tr>
<tr>
<td></td>
<td>name</td>
</tr>
<tr>
<td>Return Value</td>
<td>void</td>
</tr>
<tr>
<td>Constraint</td>
<td>Shall be called by TE to log the create component operation. This event occurs after component creation.</td>
</tr>
<tr>
<td>Effect</td>
<td>The TL presents all the information provided in the parameters of this operation to the user. The kind of the created component (see TcITestIdComponentKindType) can be logged in am. How this is done is not within the scope of the present document.</td>
</tr>
</tbody>
</table>
Clause 7.3.4.1.107  tliPStaticConnect

This clause is to be added.

| Signature | void tliPStaticConnect(in TString am, in TInteger ts, in TString src, |
|           | in TInteger line, in TriComponentIdType c, |
|           | in TriPortIdType port1, in TriPortIdType port2) |

| In Parameters | am | An additional message. |
|               | ts | The time when the event is produced. |
|               | src | The source file of the test specification. |
|               | line | The line number where the request is performed. |
|               | c | The component which produces this event. |
|               | port1 | The first port to be connected. |
|               | port2 | The second port to be connected. |

| Return Value | void |

| Constraint | Shall be called by CH or TE to log the connect operation. This event occurs after the connect operation. |

| Effect | The TL presents all the information provided in the parameters of this operation to the user. The kind of the connection (i.e. dynamic or static) can be logged in am. How this is done is not within the scope of the present document. |

Clause 7.3.4.1.108  tliPStaticMap

This clause is to be added.

| Signature | void tliPStaticMap(in TString am, in TInteger ts, in TString src, |
|           | in TInteger line, in TriComponentIdType c, |
|           | in TriPortIdType port1, in TriPortIdType port2) |

| In Parameters | am | An additional message. |
|               | ts | The time when the event is produced. |
|               | src | The source file of the test specification. |
|               | line | The line number where the request is performed. |
|               | c | The component which produces this event. |
|               | port1 | The first port to be mapped. |
|               | port2 | The second port to be mapped. |

| Return Value | void |

| Constraint | Shall be called by SA or TE to log the map operation. This event occurs after the map operation. |

| Effect | The TL presents all the information provided in the parameters of this operation to the user. The kind of the connection (i.e. dynamic or static) can be logged in am. How this is done is not within the scope of the present document. |

Clause 7.3.4.1.109  tliConfigStarted

This clause is to be added.

| Signature | void tliConfigStarted (in TString am, in TInteger ts, in TString src, |
|           | in TInteger line, in TriComponentIdType c, |
|           | in TciBehaviourIdType configId, in TciParameterListType tciPars, |
|           | in TciConfigurationIdType ref) |

| In Parameters | am | An additional message. |
|               | ts | The time when the event is produced. |
|               | src | The source file of the test specification. |
|               | line | The line number where the request is performed. |
|               | c | The component which produces this event. |
|               | configId | The static configuration function being started. |
|               | tciPars | The parameters of the started configuration function. |
|               | ref | The resulting static configuration reference. |

| Return Value | void |

| Constraint | Shall be called by TE to log the starting of a static test configuration. This event occurs after static configuration start. |

| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document. |
Clause 7.3.4.1.110  tliConfigKilled
This clause is to be added.

<table>
<thead>
<tr>
<th>Signature</th>
<th>void tliConfigKilled (in TString am, in TInteger ts, in TString src, in TInteger line, in TriComponentIdType c, in TciConfigurationIdType ref)</th>
</tr>
</thead>
</table>
| In Parameters | am  An additional message.  
|             | ts  The time when the event is produced.  
|             | src  The source file of the test specification.  
|             | line  The line number where the request is performed.  
|             | c  The component which produces this event.  
|             | ref  The static configuration reference that has been destructed.  |
| Return Value | void |
| Constraint | Shall be called by TE to log the kill configuration operation. This event occurs after configuration kill.  |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document.  |

Clause 7.3.4.1.111  tliPSetState
This clause is to be added.

<table>
<thead>
<tr>
<th>Signature</th>
<th>void tliPSetState (in TString am, in TInteger ts, in TString src, in TInteger line, in TriComponentIdType c, in TInteger state, in TString reason)</th>
</tr>
</thead>
</table>
| In Parameters | am  An additional message.  
|             | ts  The time when the event is produced.  
|             | src  The source file of the test specification.  
|             | line  The line number where the request is performed.  
|             | c  The component which produces this event.  
|             | state  The new translation state.  
|             | reason  The optional reason of the port.setstate statement.  |
| Return Value | void |
| Constraint | Shall be called by TE to log the port.setstate operation. This event occurs after the port state is set.  |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document.  |

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

Clause 8.2.2.5  TciTestComponentKindType
This clause is to be extended.

```java
// TCI IDL TciTestComponentKindType
public interface TciTestComponentKind {
    public final static int TCI_MTC_STATIC_COMP = 5;
    public final static int TCI_PTC_STATIC_COMP = 6;
    public final static int TCI_SYSTEM_STATIC_COMP = 7;
}
```

Clause 8.3.2.4  TciTypeClassType
This clause is to be extended.

```java
public interface TciTypeClass {
    public final static int CONFIGURATION = 25;
}
```
Clause 8.3.2.16 TciConfigurationIdType

This clause is to be added.

TciConfigurationIdType is mapped to the following interface:

```java
package org.etsi.ttcn.tci;
public interface TciConfigurationId {
    public String getConfigurationId();
    public String getConfigurationName();
    public String getConfigurationTypeName();
    public boolean equals(TciConfigurationId conf);
}
```

Methods

- `getConfigurationId`
  - Returns a representation of this unique configuration identifier.

- `getConfigurationName`
  - Returns the configuration name as defined in the TTCN-3 specification. If no name is provided, an empty string is returned.

- `getConfigurationTypeName`
  - Returns the configuration type name as defined in the TTCN-3 specification.

- `equals`
  - Compares `conf` with this `TciConfigurationId` for equality. Returns `true` if and only if both configurations have the same representation of this unique configuration identifier, `false` otherwise.

Clause 8.5.1.1 TCI TM provided

This clause is to be extended.

```java
package org.etsi.ttcn.tci;
public interface TciTMProvided {
    public void tciConfigStarted(TciConfigurationId ref);
    public void tciConfigKilled(TciConfigurationId ref);
}
```

Clause 8.5.1.2 TCI TM required

This clause is to be extended.

```java
package org.etsi.ttcn.tci;
public interface TciTMRequired {
    public void tciStartTestCase(String testCaseId, TciParameterList parameterList, TciConfigurationId ref);
    public TciConfigurationId tciStartConfig(TciBehaviourId configId, TciParameterList parameterList);
    public void tciKillConfig(TciConfigurationId ref);
}
```
Clause 8.5.3.1  TCI CH provided

This clause is to be extended.

```java
// TciCHProvided
// TE -> CH
package org.etsi.ttcn.tci;
public interface TciCHProvided {
    public void tciStaticConnectReq(TriPortId fromPort, TriPortId toPort);
    public void tciStaticMapReq(TriPortId fromPort, TriPortId toPort);
}
```

Clause 8.5.3.2  TCI CH required

This clause is to be extended.

```java
// TciCHRequired
// CH -> TE
package org.etsi.ttcn.tci;
public interface TciCHRequired extends TciCDRequired {
    public void tciStaticConnect(TriPortId fromPort, TriPortId toPort);
    public void tciStaticMap(TriPortId fromPort, TriPortId toPort);
}
```

Clause 8.5.4.1  TCI TL provided

This clause is to be extended.

```java
// TCI-TL
// TE, TM, CH, CD, SA, PA -> TL
package org.etsi.ttcn.tci;
public interface TciTLProvided {
    public void tliCStaticCreate(String am, int ts, String src, int line, TriComponentId c, TriComponentId comp, String name);
    public void tliPStaticConnect(String am, int ts, String src, int line, TriComponentId c, TriPortId port1, TriPortId port2);
    public void tliPStaticMap(String am, int ts, String src, int line, TriComponentId c, TriPortId port1, TriPortId port2);
    public void tliConfigStarted (String am, int ts, String src, int line, TriComponentId c, TciBehaviourId configId, TciParameterList tciPars, TciConfigurationId ref);
    public void tliConfigKilled (String am, int ts, String src, int line, TriComponentId c, TciConfigurationId ref);
    public void tliPSetState (String am, int ts, String src, int line, TriComponentId c, int state, String reason);
}
```

8.8  Extensions to clause 9 of ETSI ES 201 873-6 [4]

ANSI C language mapping

Clause 9.5  Data

The table is to be extended.

<table>
<thead>
<tr>
<th>TCI IDL ADT</th>
<th>ANSI C representation (Type definition)</th>
<th>Notes and comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TciTypeClassType</td>
<td>typedef enum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TCI_CONFIGURATION_TYPE = 25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>} TciTypeClassType;</td>
<td></td>
</tr>
</tbody>
</table>
### Clause 9.4.1.1 TCI TM provided

This clause is to be extended.

```c
void tciConfigStarted(TciConfigurationId ref);
void tciConfigKilled(TciConfigurationId ref)
```

### Clause 9.4.1.2 TCI TM required

This clause is to be extended.

```c
void tciStartTestCase(TciTestCaseIdType testCaseId, TciParameterListType parameterlist, TciConfigurationIdType ref)
void tciStartConfig(TciBehaviourIdType configId, TciParameterListType parameterList)
void tciKillConfig(TciConfigurationId ref)
```

### Clause 9.4.3.1 TCI CH provided

This clause is to be extended.

```c
void tciExecuteTestCaseReq(TciTestCaseIdType testCaseId, TriPortIdList tsiPortList, TciConfigurationIdType ref);
void tciStaticConnectReq(TriPortId fromPort, TriPortId toPort);
void tciStaticMapReq(TriPortId fromPort, TriPortId toPort);
```

### Clause 9.4.3.2 TCI CH required

This clause is to be extended.

```c
void tciExecuteTestCase(TciTestCaseIdType testCaseId, TriPortIdList tsiPortList, TciConfigurationIdType ref);
void tciStaticConnect(TriPortId fromPort, TriPortId toPort);
void tciStaticMap(TriPortId fromPort, TriPortId toPort);
```

### Clause 9.4.4.1 TCI TL provided

This clause is to be extended.

```c
void tliCStaticCreate (String am, int ts, String src, int line, TriComponentId c, TriComponentId comp, String name)
void tliPStaticConnect (String am, int ts, String src, int line, TriComponentId c, TriPortId port1, TriPortId port2)
void tliPStaticMap (String am, int ts, String src, int line, TriComponentId c, TriPortId port1, TriPortId port2)
```
void tliConfigStarted (String am, int ts, String src, int line, TriComponentId c, TciBehaviourIdType configId, TciParameterListType tciPars, Value ref)
void tliConfigKilled (String am, int ts, String src, int line, TriComponentId c, Value ref)
void tliPSetState (String am, int ts, String src, int line, TriComponentId c, int state, String reason)

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

Clause 10.5.2.13 TciTestComponentKind

This clause is to be extended.

class TciTestComponentKind {
public:
    static const TciTestComponentKind MTC_STATIC_COMP;
    static const TciTestComponentKind PTC_STATIC_COMP;
    static const TciTestComponentKind SYSTEM_STATIC_COMP;
};

Clause 10.5.2.14 TciTypeClassType

This clause is to be extended.

typedef enum {
    TCI_CONFIGURATION = 25
} TciTypeClass;

Clause 10.5.2.16 TciConfigurationId

This clause is to be added.

Identifies a static configuration. It is mapped to the pure virtual class:

class TciConfigurationId {
public:
    virtual ~TciConfigurationId ()
    virtual const QualifiedName & getConfigurationTypeName () const =0;
    virtual void setConfigurationTypeName (const QualifiedName &tName)=0;
    virtual const TString & getConfigurationName () const =0;
    virtual void setConfigurationName (const TString &sName)=0;
    virtual const Tinteger & getConfigurationId () const =0;
    virtual void setConfigurationId (const Tinteger &id)=0;
    virtual Tboolean operator== (const TciConfigurationId &cmp) const =0;
    virtual TciComponentId * cloneConfigurationId () const =0;
    virtual Tboolean operator< (const TciConfigurationId &cmp) const =0;
};

Methods

- ~TciConfigurationId
  Destructor.

- getConfigurationTypeName
  Returns a const reference to the configuration type name.

- setConfigurationTypeNamem
  Set the configuration type name.

- getConfigurationName
  Gets the configuration name.

- setConfigurationName
  Set the configuration name.
• **getConfigurationId**
  Returns the configuration identifier.

• **setConfigurationId**
  Set the configuration identifier.

• **operator==**
  Returns true if both TciConfigurationId objects are equal.

• **cloneConfigurationId**
  Returns a copy of the TciConfigurationId.

• **operator<**
  Operator < overload.

**Clause 10.6.1.1 TciTmRequired**

This clause is to be extended.

```
virtual void tciStartTestCase
  (const TciTestCaseId *testCaseId, const TciParameterList *parameterList,
   const TciConfigurationId *ref)=0;
```

```
virtual const TciConfigurationId *tciStartConfig
  (const TciBehaviourId *configId, TciParameterList *parameterList)=0;
```

```
virtual void tciKillConfig(const TciConfigurationId *ref)=0;
```

**Clause 10.6.1.2 TciTmProvided**

This clause is to be extended.

```
//Indicates the start of a static configuration
virtual void tciConfigStarted(const TciConfigurationId *ref) =0;
```

```
virtual void tciConfigKilled(const TciConfigurationId *ref)=0;
```

**Clause 10.6.3.1 TciChRequired**

This clause is to be extended.

```
virtual void tciExecuteTestCase
  (const TciTestCaseId *testCaseId, const TriPortIdList *tsiPortList,
   const TciConfigurationId *ref)=0;
```

```
virtual void tciStaticConnect(const TriPortId *fromPort, const TriPortId *toPort)=0;
```

```
virtual void tciStaticMap(const TriPortId *fromPort, const TriPortId *toPort)=0;
```

**Clause 10.6.3.2 TciChProvided**

This clause is to be extended.

```
virtual void tciExecuteTestCaseReq
  (const TciTestCaseId *testCaseId, const TriPortIdList *tsiPortList,
   const TciConfigurationId *ref)=0;
```

```
virtual void tciStaticConnectReq(const TriPortId *fromPort, const TriPortId *toPort)=0;
```

```
virtual void tciStaticMapReq(const TriPortId *fromPort, const TriPortId *toPort)=0;
```

**Clause 10.6.4.1 TciTlProvided**

This clause is to be extended.

```
virtual void tliCStaticCreate (const Tstring &am, const timeval ts, const Tstring src,
  const Tinteger line, const TriComponentId *c, const TriComponentId *comp,
  const Tstring &name)=0;
```

```
virtual void tliPStaticConnect (const Tstring &am, const timeval ts, const Tstring src,
  const Tinteger line, const TriComponentId *c, const TriPortId *port1, const TriPortId *port2)=0;
```
virtual void tliPStaticMap (const Tstring &am, const timeval ts, const Tstring src, const Tinteger line, const TriComponentId *c, const TriPortId *port1, const TriPortId *port2)=0;
virtual void tliConfigStarted (const Tstring &am, const timeval ts, const Tstring src, const Tinteger line, const TriComponentId *c, const TciBehaviourId *configId, const TciParameterList *tciPars, const TciValue *ref)=0;
virtual void tliConfigKilled (const Tstring &am, const timeval ts, const Tstring src, const Tinteger line, const TriComponentId *c, const TciValue *ref)=0;
virtual void tliPSetState (const Tstring &am, const timeval ts, const Tstring src, const Tinteger line, const TriComponentId *c, const Tinteger status, const TString &reason)=0;

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

Clause 11.3.2.21 TciConfigurationIdType

This clause is to be added.

TciConfigurationIdType is mapped to the following complex type:

```xml
<xsd:complexType name="TciConfigurationIdType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:element name="null" type="Templates:null"/>
      <xsd:element name="id" type="Types:Id"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

Elements:
- **id** The identifier of the static configuration.
- **null** The null identifier. To be used if there is no static configuration identifier.

Attributes:
- none.

Clause 11.4.2.1 TCI TL provided

This clause is to be extended.

```xml
<xsd:complexType name="tliTcExecute">
  <xsd:complexContent mixed="true">
    <xsd:extension base="Events:Event">
      <xsd:sequence>
        <xsd:element name="tcId" type="Types:TciTestCaseIdType"/>
        <xsd:element name="tciPars" type="Types:TciParameterListType" minOccurs="0"/>
        <xsd:element name="dur" type="SimpleTypes:TriTimerDurationType" minOccurs="0"/>
        <xsd:element name="ref" type="Types:TciConfigurationIdType" minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="tliTcStart">
  <xsd:complexContent mixed="true">
    <xsd:extension base="Events:Event">
      <xsd:sequence>
        <xsd:element name="tcId" type="Types:TciTestCaseIdType"/>
        <xsd:element name="tciPars" type="Types:TciParameterListType" minOccurs="0"/>
        <xsd:element name="dur" type="SimpleTypes:TriTimerDurationType" minOccurs="0"/>
        <xsd:element name="ref" type="Types:TciConfigurationIdType" minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```


Clause B.5  TCI TL XML Schema for Events

The five additional events defined for clause "11.4.2.1 TCI TL provided" have to be added to the events schema definition given in clause B.5.
Clause B.6 TCI TL XML Schema for a Log

```xml
<xs:complexType name="Body">
  <xs:choice maxOccurs="unbounded">
    <!-- static configurations -->
    <xs:element name="tliCStaticCreate" type="Events:tliCStaticCreate"/>
    <xs:element name="tliPStaticConnect" type="Events:tliPStaticConnect"/>
    <xs:element name="tliPStaticMap" type="Events:tliPStaticMap"/>
    <xs:element name="tliConfigStarted" type="Events:tliConfigStarted"/>
    <xs:element name="tliConfigKilled" type="Events:tliConfigKilled"/>
    <xs:element name="tliPSetState" type="Events:tliPSetState"/>
  </xs:choice>
</xs:complexType>
```
8.11 Extensions to clause 12 of ETSI ES 201 873-6 [4] C# language mapping

**Clause 12.4.2.5 TciTestComponentKindType**

This clause is to be extended.

```csharp
public enum TciTestComponentKind {
    TciMtcStaticComp = 5;
    TciPtcStaticComp = 6;
    TciSystemComp = 7;
}
```

**Clause 12.4.2.4 TciTypeClassType**

This clause is to be extended.

```csharp
public enum TciTypeClass {
    Configuration = 25;
}
```

**Clause 12.4.2.16 TciConfigurationIdType**

This clause is to be added.

TciConfigurationIdType is mapped to the following interface:

```csharp
public interface ITciConfigurationId {
    string ConfigurationId { get; }
    string ConfigurationName { get; }
    IQualifiedName ConfigurationTypeName { get; }
    bool Equals(ITciConfigurationId conf);
}
```

**Methods**

- **ConfigurationId**
  Returns a representation of this unique configuration identifier.

- **ConfigurationName**
  Returns the configuration name as defined in the TTCN-3 specification. If no name is provided, an empty string is returned.

- **ConfigurationTypeName**
  Returns the configuration type name as defined in the TTCN-3 specification.

- **Equals**
  Compares `conf` with this `TciConfigurationId` for equality. Returns `true` if and only if both configurations have the same representation of this unique configuration identifier, `false` otherwise.

**Clause 12.5.1.1 TCI-TM provided**

This clause is to be extended.

```csharp
public interface ITciTMProvided {
    void tciConfigStarted(ITciConfigurationId ref);
    void tciConfigKilled(ITciConfigurationId ref);
}
```
Clause 12.5.1.2 TCI-TM required

This clause is to be extended.

```java
public interface ITciTMRequired {
    void TciStartTestCase(ITciTestCaseId testCaseId, ITciParameterList parameterList, ITciConfigurationId ref);
    ITciConfigurationId tciStartConfig (ITciBehaviourId configId, ITciParameterList parameterList)
    void tciKillConfig(ITciConfigurationId ref)
}
```

Clause 12.5.3.1 TCI-CH provided

This clause is to be extended.

```java
public interface ITciCHProvided {
    void TciExecuteTestCaseReq(ITriComponentId component, ITriPortIdList tsiPortList, ITciConfigurationId ref);
    void tciStaticConnectReq(ITriPortId fromPort, ITriPortId toPort);
    void tciStaticMapReq(ITriPortId fromPort, ITriPortId toPort);
}
```

Clause 12.5.3.2 TCI-CH required

This clause is to be extended.

```java
public interface ITciCHRequired {
    void TciExecuteTestCase (ITciTestCaseId testCaseId, ITriPortIdList tsiPortList, ITciConfigurationId ref);
    void tciStaticConnect(ITriPortId fromPort, ITriPortId toPort);
    void tciStaticMap(ITriPortId fromPort, ITriPortId toPort);
}
```

Clause 12.5.4.1 TCI-TL provided

This clause is to be extended.

```java
public void tliCStaticCreate(string am, System.DateTime ts, string src, int line, ITriComponentId c, ITriComponentId comp, sString name);
void tliPStaticConnect(string am, System.DateTime ts, string src, int line, ITriComponentId c, ITriPortId port1, ITriPortId port2);
void tliPStaticMap(string am, System.DateTime ts, string src, int line, ITriComponentId c, ITriPortId port1, ITriPortId port2);
void tliConfigStarted (string am, System.DateTime ts, string src, int line, ITriComponentId c, ITciBehaviourId configId, ITciParameterList tciPars, ITciValue ref);
void tliConfigKilled (string am, System.DateTime ts, string src, int line, ITriComponentId c, ITciValue ref);
void tliPSetState (string am, System.DateTime ts, string src, int line, ITriComponentId c, int state, string reason);
```
Annex A (normative):
BNF and static semantics

A.1 Additional TTCN-3 terminals

Table A.1 presents all additional TTCN-3 terminals which are reserved words when using this package. Like the reserved words defined in the TTCN-3 core language, the TTCN-3 terminals listed in table A.1 shall not be used as identifiers in a TTCN-3 module. These terminals shall be written in all lowercase letters.

<table>
<thead>
<tr>
<th>configuration</th>
<th>static</th>
<th>setstate</th>
</tr>
</thead>
</table>

A.2 Modified TTCN-3 syntax BNF productions

This clause includes all BNF productions that are modifications of BNF rules defined in the TTCN-3 core language document ETSI ES 201 873-1 [1]. When using this package the BNF rules below replace the corresponding BNF rules in the TTCN-3 core language document. The rule numbers define the correspondence of BNF rules.
A.3 Additional TTCN-3 syntax BNF productions

This clause includes all additional BNF productions that needed to define the syntax introduced by this package. All rules start with the digits "781". Additional BNF rules that have a relation to modified BNF rules defined in clause A.2, will have the rule number of the modified rule followed by a lower case letter, e.g. number of modified rule 316, number of related additional rule 781316a. The numbering of other new rules start with number 781001.

781186a. ExecuteOnSpec ::= ExecuteKeyword OnKeyword ConfigurationRef

781264a. StaticKeyword ::= "static"

781001. ConfigurationDef ::= ConfigurationKeyword ConfigurationIdentifier "(" [TemplateOrValueFormalParList ] ")" ConfigSpec StatementBlock

781002. ConfigurationKeyword ::= "configuration"

781003. ConfigurationIdentifier ::= Identifier

781004. ConfigurationInstance ::= ConfigurationRef "(" [ActualParList ] ")"

781005. ConfigurationRef ::= [ ModuleId Dot ] ConfigurationIdentifier
KillConfigStatement ::= ConfigurationReference Dot KillKeyword

ConfigurationReference ::= VariableRef | FunctionInstance

TranslationPortAttribs ::= MessageKeyword OuterPortTypeSpec "{" {
  TranslationAddrDecl | TranslationMessageList | ConfigParamDef 
}[SemiColon]+ "}"*

OuterPortTypeSpec ::= OuterPortTypeMapSpec | OuterPortTypeConnectSpec

OuterPortTypeMapSpec ::= MapKeyword ToKeyword Type {""," Type } [ OuterPortTypeConnectSpec ]

OuterPortTypeConnectSpec ::= ConnectKeyword ToKeyword Type {""," Type }

TranslationAddrDecl ::= AddressKeyword Type | TranslationAddrSpec {"","TranslationAddrSpec } |

TranslationAddrSpec ::= ( ToKeyword | FromKeyword ) Type WithKeyword FunctionRef "(""")"

TranslationMessageList ::= InParKeyword TranslationInTypeList | OutParKeyword TranslationOutTypeList | InOutParKeyword TypeList

TranslationInTypeList ::=TranslationInType {""," TranslationInType }

TranslationInType ::= Type [ TranslationInSpec {""," TranslationInSpec } ]

TranslationInSpec ::= FromKeyword Type WithKeyword FunctionRef "(""")"

TranslationOutTypeList ::= TranslationOutType {""," TranslationOutType }

TranslationOutType ::= Type [ TranslationOutSpec {"","TranslationOutSpec } ]

TranslationOutSpec ::= ToKeyword Type WithKeyword FunctionRef "(""")"

FuncPortSpec ::= PortKeyword Identifier

SetPortState ::= PortKeyword "." SetStateKeyword "(" SingleExpression {""," LogItem } ")"

SetStateKeyword ::= "setstate"

ExceptConfigurationSpec ::= ConfigurationKeyword IdentifierListOrAll

ImportConfigurationSpec ::= ConfigurationKeyword AllKeyword

PortOp ::= PortKeyword
Annex B (informative):
Library of useful types

B.1 Limitations

The types and constants described in this annex use the same rule as specified in clause E.1 of ETSI ES 201 873-1 [1].

B.2 Useful TTCN-3 types

B.2.1 Status values for port states

Type and constants defined in this clause support the secure usage of the setstate port operation defined in clause 5.10.4 of ETSI ES 201 873-1 [1].

The type definition for this type is:

```plaintext
type integer translationState(0..4);
```

Useful constant definitions for working with object states are:

```plaintext
const translationState TRANSLATED := 0;
const translationState NOT_TRANSLATED := 1;
const translationState FRAGMENTED := 2;
const translationState PARTIALLY_TRANSLATED := 3;
const translationState DISCARDED := 4;
```
## History

### Document history

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1.1.1</td>
<td>August 2010</td>
<td>Publication</td>
</tr>
<tr>
<td>V1.2.1</td>
<td>June 2013</td>
<td>Publication</td>
</tr>
<tr>
<td>V1.3.1</td>
<td>June 2014</td>
<td>Publication</td>
</tr>
<tr>
<td>V1.4.1</td>
<td>June 2015</td>
<td>Publication</td>
</tr>
<tr>
<td>V1.5.1</td>
<td>May 2017</td>
<td>Publication</td>
</tr>
<tr>
<td>V1.6.1</td>
<td>May 2018</td>
<td>Publication</td>
</tr>
<tr>
<td>V1.7.1</td>
<td>February 2019</td>
<td>Membership Approval Procedure  MV 20190415: 2019-02-14 to 2019-04-15</td>
</tr>
<tr>
<td>V1.7.1</td>
<td>April 2019</td>
<td>Publication</td>
</tr>
</tbody>
</table>