### 5.2.1 Translation capability in port type declaration

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

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

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

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

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

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

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

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

***Restrictions***

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

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

EXAMPLE:

 **type port** TransportPort

 {

 **inout** TransportMessage;

 }

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

 {

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

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

 }

### 5.2.2 Mapping and connecting ports

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

The translation mode is activated in these cases:

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

In all other cases, normal mode is activated.

EXAMPLE:

 **type port** TransportPort {

 ...

 }

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

 ...

 }

 **type component** SystemComponent{

 **port** DataPort dataPort;

 **port** TransportPort transportPort;

 }

 **type component** TestComponent{

 **port** DataPort dataPort;

 }

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

 {

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

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

 // in the map operation)

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

 }

 **else**{

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

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

 }

 }

### 5.2.3 Translation functions

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

***Syntactical Structure***

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

 [**port** PortTypeId]

 StatementBlock

***Semantic Description***

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

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

***Restrictions***

1. Translation functions shall never return a value.

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

1. Translation functions shall not contain a runs on clause.
2. Translation function containing a **port** clause can be referenced only in the port type referenced in this port clause.
3. The type of the **in** parameter of a translation function referenced as an *InFunction* in an **in** clause shall be the *OuterInType* immediately preceding the *InFunction* reference and the type of its **out** parameter shall be the *InnerInType*.
4. The type of the **in** parameter of a translation function referenced as an *OutFunction* in an **out** clause shall be the *InnerOutType* and the type of its **out** parameter shall be the *OuterOutType* immediately preceding the *OutFunction* reference.
5. The type of the **in** parameter of a translation function referenced as an *AddrOutFunction* in a port **address** declaration shall be the *AddrType* and the type of its **out** parameter shall be the *OuterAddrType* that immediately precedes the *AddrFunction* reference.
6. The type of the **in** parameter of a translation function referenced as an *AddrInFunction* in a port **address** declaration shall be the *OuterAddrType* that immediately precedes the *AddrFunction* reference and the type of its **out** parameter shall be the *AddrType*.
7. Translation functions shall not contain any blocking operations.
8. Invoking a function with a **port** clause explicitly shall cause an error.

EXAMPLE:

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

 {

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

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

 **var octetstring** vp\_remainings

 }

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

 ...

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

 }

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

 ...

 **port**.**setstate**("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 didn't finish translation, because the input data didn't 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 hasn't been translated yet (i.e. the input data contained more than one message).

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

***Syntactical Structure***

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

***Semantic Description***

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

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

***Restrictions***

1. The value passed to the **setstate** operation in the first parameter shall be of the **integer** type and shall have one of the following values:
* 0 (meaning *translated*)
* 1 (meaning *not translated*)
* 2 (meaning *fragmented*)
* 3 (meaning *partially translated*)

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.

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

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

### 5.2.5 Sending

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

* If no *OutFunction* is specified for the given *InnerOutType,* it is simply sent over the port transparently.
* If an *OutFunction* is specified for the *InnerOutType*, the translation procedure first sets the translation state to *Unset*. Then the *OutFunction* is automatically invoked to translate the *InnerOutType* to the *OuterOutType.* When the function execution is finished, then depending on the current translation state one of the following actions is taken:
* The *unset* state shall cause an error (i.e. if there is no **setstate** operation is invoked in the translation function).
* If the state is *not translated*, the translation procedure tries to translate the message using the next *OutFunction* specified for the given *InnerOutType*. *OutFunction*-s are tried according to their textual order in the port type definition. If there is no such a function, an error is generated.
* If the state is *fragmented*, the translation procedure ends but no data is sent to the connected or mapped port (the port will wait for the next fragment to complete translation). The **to** clause of the following send operation shall be the same as the **to** clause of the current send operation or missing if the current send operation doesn't 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.

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

### 5.2.6 Receiving

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

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

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

TE

**Port in translation mode**

**System**

**adaptor**

TRI message

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

Co**dec**

**decoded value**

**Outer queue**

**Inner** **queue**

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

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

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

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

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

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

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

### 5.2.7 Address

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

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

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

EXAMPLE:

 **type port** TransportPort

 {

 ...

 **address** TransportAddress;

 }

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

 {

 ...

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

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

 }

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

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

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

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

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

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

## 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>

## 6.24 Connect operation

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

 **connect**(<component-expression1>:<portId1>, <component-expression2>:<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-expression1> and <component-expression2>. 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-expression1> and the second expression to <component-expression2>, i.e. the <component-expression2> is on top of the value stack when the connect-op node is executed.



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

## 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.



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

## 6.36 Halt port operation

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

 <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>

## 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. **mtc**.**kill**) or kills itself (e.g. **self**.**kill**). The MTC may kill all parallel test components by using the **all** keyword, i.e. **all** **component**.**kill**.

Special rules apply for using the **kill** component operation in static test configurations: Applying the **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 **all** keyword, i.e. **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 <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 <kill-component-op> in figure 90a defines the execution of the **kill** component operation.



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

 <portId>**.start**

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



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

## 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.



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