ETSI ES 202 782 V1.2.1 (2014-06)

Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
TTCN-3 Language Extensions:
TTCN-3 Performance and Real Time Testing

Reference
RES/MTS-113ed121 T3Ext_Perf

Keywords
performance, real time, testing, TTCN-3

ETSI

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

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

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

Contents

Intellectual Property Rights

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

Foreword

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

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

ES 201 873-1: "TTCN-3 Core Language";

ES 201 873-2: "TTCN-3 Tabular presentation Format (TFT)";

ES 201 873-3: "TTCN-3 Graphical presentation Format (GFT)";

ES 201 873-4: "TTCN-3 Operational Semantics";

ES 201 873-5: "TTCN-3 Runtime Interface (TRI)";

ES 201 873-6: "TTCN-3 Control Interface (TCI)";

ES 201 873-7: "Using ASN.1 with TTCN-3";

ES 201 873-8: "The IDL to TTCN-3 Mapping";

ES 201 873-9: "Using XML schema with TTCN-3";

ES 201 873-10: "TTCN-3 Documentation Comment Specification".

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**may not**", "**need**", "**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 real time and performance testing 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 OMG CORBA based platforms, 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.

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

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 reference document (including any amendments) applies.

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

2.1 Normative references

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

[1] ETSI ES 201 873-1 (V4.6.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language".

[2] ETSI ES 201 873-4 (V4.4.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics".

[3] ETSI ES 201 873-5 (V4.6.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)".

[4] ETSI ES 201 873-6 (V4.6.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI)".

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

2.2 Informative references

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 201 873-3 (V3.2.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical presentation Format (GFT)".

[i.2] ETSI ES 201 873-7 (V4.5.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 7: Using ASN.1 with TTCN-3".

[i.3] ETSI ES 201 873-8 (V4.5.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 8: The IDL to TTCN-3 Mapping".

[i.4] ETSI ES 201 873-9 (V4.5.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 9: Using XML schema with TTCN-3".

[i.5] ETSI ES 201 873-10 (V4.5.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 10: TTCN-3 Documentation Comment Specification".

3 Definitions and abbreviations

3.1 Definitions

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

3.2 Abbreviations

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

4 Package conformance and compatibility

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

`"TTCN-3:2014 Real Time and Performance Testing"` - 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 and in ES 201 873-1 [1], ES 201 873-4 [2], ES 201 873-5 [3] and ES 201 873-6 [4].

The package presented in the present document is compatible to:

ES 201 873-1 (V4.6.1) [1]

ES 201 873-4 (V4.4.1) [2]

ES 201 873-5 (V4.6.1) [3]

ES 201 873-6 (V4.6.1) [4]

ES 201 873-7 (V4.5.1) [i.2]

ES 201 873-8 (V4.5.1) [i.3]

ES 201 873-9 (V4.5.1) [i.4]

ES 201 873-10 (V4.5.1) [i.5]

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. The present document is also compatible with the versions V.4.2.1, V.4.3.1, V.4.4.1, V.4.5.1 of the above documents.

5 Package concepts for the core language

Real-time systems have to respect special requirements for timing. Often functional requirements are directly connected to the timing of the messages and procedure calls. Thus, checking the message values and the message order is not sufficient here. A test component shall be able to check whether a message has been received in time and shall be able to control the timing for the stimulation.

Thus, a test language has to provide means to measure time, to specify time points and time spans, to control the timing of the stimulation, and to calculate and compare time values. Moreover the test execution engine has to ensure that the specified actions (time measurement, timed stimulation) are executed correctly with respect to the required precision.

To fulfil the requirements for testing real time system we define the following TTCN-3 core language extensions:

- A test system wide available test system clock, that allows the measurement of time during test case execution.
- Means to directly and precisely access the time points of the relevant interaction events between the test system and the system under test.

Real-time measurements at ports require additional resources (e.g. functionality that monitor ports and collect timestamps that describe the reception time of messages, calls, replies or exceptions) that may slow down the test execution. In order to avoid unnecessary delays at ports, such resources may only be provided when needed. An additional **`real-time`** clause for ports shall indicate the need for real-time measurement at a port.

5.1 The test system clock

In RT TTCN-3 time progress is measured with a test system clock. The clock is initialized (set to 0.0) at the beginning of each test case execution and is available during the complete test run in each component. The clock values are represented as float values. The system clock and the already available TTCN-3 timer mechanisms are synchronized with respect to time progress.

5.1.1 Accessing the current test system time

The current value of the test system clock by means of the symbol **`now`**. The **`now`** symbol is used as a TTCN-3 expression that yields the current test system clock value in seconds. The test system clock value is represented by means of a **`float`** number. The symbol **`now`** can be applied in each expression inside of testcase definitions and function definitions. It is not allowed for the TTCN-3 control part and in guard conditions of alt branches.

EXAMPLE 1:

```
// Use of now to retrieve the actual time
var float myTimePoint := now;
```

EXAMPLE 2:

```
// Use of now to retrieve the send time of a message
var float sendTimePoint;
// ...
p.send(m);
sendTimePoint:= now;
```

EXAMPLE 3:

```
// Measuring time progress
var float startTime;
startTime:= now;
p.send(m1);
// ...
p.receive(m2);
if(now-startTime >= 10.0){...};
```

Syntactical Structure

```
OpCall ::= ConfigurationOps | VerdictOps | TimerOps | TestcaseInstance |
                              FunctionInstance | TemplateOps | ActivateOp | NowOperation
NowOperation ::=  NowKeyword
NowKeyword ::=  "now"
```

5.1.2 The precision of the system time

The requirements on the overall precision of the test system clock can be specified by means of the stepsize annotation. The stepsize annotation is allowed for modules only and can be used to state the minimal necessary precision for time measurement provided by the test system clock. The precision is defined by means of a charstring value that represents a decimal number which states the smallest necessary time distance in seconds that is measureable by the test system clock. A concrete test system has to fulfil the requirements given by the stepsize annotation to be adequate for the execution of the respective test case definitions. When a test system is not adequate for the test case execution the user shall be informed, at least test run shall end with an error verdict.

EXAMPLE:

```
// specifies the requirement on a necessary precision of a millisecond
module myModule{
…
} with {stepsize “0.001”};
```

In case of module imports with different stepsize annotation the test system has to respect the stepsize annotation with the highest precision.

5.2 Communication port types for real-time measurements

This package extends the port type definition of message-based and procedure-based ports with a **`realtime`** clause. Ports facilitate communication between test components and between test components and the test system interface.

Only instances of ports with a realtime clause shall be used for real-time measurements. This means, the redirection operator **`-> timestamp`** shall only be used by receiving operations (i.e. the operations **`receive`**, **`trigger`**, **`getcall`**, **`getreply`** and **`catch`**) applied to ports with a **`realtime`** clause.

Syntactical Structure

Message-based port:

```
type port PortTypeIdentifier message [realtime] "{"
     { ( in | out | inout ) { MessageType [ "," ] }+ ";" }
"}"
```

Procedure-based port:

```
type port PortTypeIdentifier procedure [realtime] "{"
     { ( in | out | inout ) { Signature [ "," ] }+ ";" }
"}"
```

5.3 Measuring timing information for dedicated incoming communication events

Testing real time systems requires exact timing information that relates directly to the communication (reception and distribution of messages and procedure calls) between the test system and the system under test. The timing information that can be obtained by the **`now`** symbol or the TTCN-3 timer construct is related to the logical structure of the test program, thus it allows the measurement on TTCN-3 statement level. Time measurement on TTCN-3 statement level may be affected by blocked queues, decoding and matching procedures. It is not exact with respect to the real timing of the reception and disposal of messages and procedure calls at the interface between the test system and the SUT.

RT TTCN-3 introduces a mechanism to store the arrival time of messages, procedure calls at system adapter level. The time points of message reception are automatically registered by the system adapter, communicated to the test executable and stored with the message. The timing information can be retrieved directly at the communication statements by means of the redirection operator **`-> timestamp`** .

The existing redirections for getcall, getreply, receive, trigger, catch, and check operations are extended by an optional clause **`timestamp.`** A redirect specification of the form:

```
-> timestamp VariableRef
```

specifies the redirection of the time point, which has been measured at message, procedure call, reply or exception arrival to a given float variable. The redirection is processed when the respective communication statement matches.

Restrictions

The redirection operator **`-> timestamp`** shall only be used by receiving operations (i.e. the operations **`receive`**, **`trigger`**, **`getcall`**, **`getreply`** and **`catch`**) applied to ports with a **`realtime`** clause.

5.3.1 Obtain the reception time for messages with the receive statement

The existing redirections for receive are extended by an optional clause "**timestamp** VariableRef". A receive statement that holds a timestamp clause and that is executed successfully (i.e. it matches a message) allocates the given variable with the reception time of the matched message.

EXAMPLE 1:

```
p.receive(t)-> timestamp myTime;
// yields the reception time of a message
if(myTime>MAX){setverdict(fail);}
```

EXAMPLE 2:

```
interleave{
    [ ] FrontOut.receive(ON) -> timestamp f_actv{
            if(f_actv>MAX){setverdict(fail);}
        };
    [ ] RearOut.receive(ON) -> timestamp r_actv{
            if(r_actv>MAX){setverdict(fail);}
        };
}
```

Syntactical Structure

```
( Port | any port ) "." receive ["(" TemplateInstance ")"] [ from AddressRef ]
[ -> [ value  VariableRef ] [ sender VariableRef ] [ timestamp VariableRef] ]
```

5.3.2 Obtain the reception time for messages with the trigger statement

The existing redirections for trigger are extended by an optional clause "**timestamp** VariableRef". A trigger statement that holds a timestamp clause and that is executed successfully (i.e. it matches a message) allocates the given variable with the reception time of the matched message.

EXAMPLE 1:

```
p.trigger(t)-> timestamp myTime;
// yields the reception time of a message
if(myTime>MAX){setverdict(fail);}
```

EXAMPLE 2:

```
interleave{
    [ ] FrontOut.trigger(ON) -> timestamp f_actv{
            if(f_actv>MAX){setverdict(fail);}
        };

    [ ] RearOut.trigger(ON) -> timestamp r_actv{
            if(r_actv>MAX){setverdict(fail);}
        };
}
```

Syntactical Structure

```
( Port | any port ) "." trigger [ "(" TemplateInstance ")" ] [ from AddressRef ]
[ -> [ value  VariableRef ] [ sender VariableRef ] ] [ timestamp VariableRef] ]
```

5.3.3 Obtain the reception time for procedure calls with getcall statement

The existing redirections for getcall are extended by an optional clause "**timestamp** VariableRef". A getcall statement that holds a timestamp clause and that is executed successfully (i.e. it matches an incoming call) allocates the given variable with the reception time of the matched message.

EXAMPLE 1:

```
p.getcall(proc: {m})-> timestamp myTime;
// yields the reception time of the message call matched by m
if(myTime>MAX){setverdict(fail);}
```

EXAMPLE 2:

```
alt{
    [ ] p.getcall(proc: {m1})-> timestamp f_actv {
            if(f_actv>MAX){setverdict(fail);}
        };
    [ ] p.getcall(proc: {m2})-> timestamp r_actv {
            if(f_actv>MAX){setverdict(fail);}
        };
}
```

Syntactical Structure

```
( Port | any port ) "." getcall  [ "(" TemplateInstance ")" ] [ from AddressRef ]
 [ "->"  [ param "(" { VariableRef ":=" ParameterIdentifier ) "," }  |
                        { VariableRef | NotUsedSymbol ) "," }
                     ")" ]
           [ sender VariableRef ]
           [ timestamp VariableRef]
]
```

5.3.4 Obtain the reception time for procedure replies with the getreply statement

The existing redirections for getreply are extended by an optional clause "**timestamp** VariableRef". A getreply statement that holds a timestamp clause and that is executed successfully (i.e. it matches an incoming procedure reply) allocates the given variable with the reception time of the matched message.

EXAMPLE 1:

```
p.getreply(proc: {m})-> timestamp myTime;
// yields the reception time of the message call matched by m
if(myTime>MAX){setverdict(fail);}
```

EXAMPLE 2:

```
p.call(proc: {_message:= m},20.0){
    [ ] p.getreply(proc: {m1})-> timestamp f_actv {
            if(f_actv>MAX){setverdict(fail);}
        };
    [ ] p.getreply(proc: {m2})-> timestamp r_actv {
            if(f_actv>MAX){setverdict(fail);}
        };
}
```

Syntactical Structure

```
( Port | any port ) "." getreply [ "(" TemplateInstance  [ value TemplateInstance ] ")" ] [ from
AddressRef ]
[ "->" [ value VariableRef ]
        [  param "(" { VariableRef ":=" ParameterIdentifier ) "," }  |
                        { VariableRef | NotUsedSymbol ) "," }
```

```
                    ")" ]
        [ sender VariableRef ]
        [ timestamp VariableRef]
]
```

5.3.5 Obtain the reception time for exceptions with the catch statement

The existing redirections for **catch** are extended by an optional clause "**timestamp** VariableRef". A catch statement that holds a timestamp clause and that is executed successfully (i.e. it matches an incoming exception) allocates the given variable with the reception time of the matched message.

EXAMPLE 1:

```
p.catch(timeout)-> timestamp myTime;
// yields the reception time of the message call matched by m
if(myTime>MAX){setverdict(fail);}
```

EXAMPLE 2:

```
p.call(proc: {_message:= m},20.0){
    [ ] p.getreply(proc: {m1})-> timestamp f_actv {
            if(f_actv>MAX){setverdict(fail);}
        };
    [ ] p.catch(*)-> timestamp r_actv {
            if(f_actv>MAX){setverdict(fail);}
        };
}
```

Syntactical Structure

```
( Port | any port ) "." catch [ "(" ( Signature "," TemplateInstance ) | TimeoutKeyword   ")" ] [
from AddressRef ]
[ "->" [ value VariableRef
         [ sender VariableRef ]
         [ timestamp VariableRef]
]
```

5.4 The wait statement

The **wait** statement suspends the execution of a component until a given point in time. The time point is specified as a float value and relates to the internal clock.

The execution of **wait** statement suspends the execution of the related component until the point in time specified by its argument. If the argument holds a value that precedes the actual clock value an error verdict shall be set.

EXAMPLE 1:

```
wait(100.0); // suspends the execution of a component
             // until 100.0 seconds after the start of the testcase
```

Syntactical Structure

```
WaitStatement::=  wait "(" Expression ")"
```

Besides the exact measurement of timing information regarding incoming communication events, a real time test system has to ensure the correct timing for message and procedure call application. Actually we consider realizing this correct scheduling of message and procedure call application by combining the wait statement directly with the send operation. In this case, the execution of a test component is suspended until the given point in time is reached and afterwards the send operation is executed.

EXAMPLE 2:

```
wait(specified_send_time);
p_out.send(OUT_MSG);
// suspends the sending of OUT_MSG until specified_send_time is reached
```

5.5 Measuring timing information for outgoing communication operations

Realtime measurements should be as exact as possible. Therefore, it has to be possible to measure the exact time when the adapter has sent a message or a call to the SUT. The point in time to be measured should be after the message or call parameters have been encoded (if necessary), right before the actual sending to the SUT is performed. By comparing this timestamp with the timestamp obtained from the responding incoming communication operation, the exact stimulus-response duration can be measured by the testcase.

To that end, all outgoing communication operations are augmented by an optional timestamp redirection assignment notation that allows the measuring of the time when the communication to the SUT is performed.

A redirect specification of the form:

```
-> timestamp VariableRef
```

specifies the redirection of the time point, which has been measured right before message, procedure call, reply or exception sending to the SUT by the adapter to a given float variable. The redirection is processed after the sending operation is successful.

Restrictions

The redirection operator **`-> timestamp`** shall only be used by sending operations (i.e. the operations **`send`**, **`call`**, **`reply`** and **`raise`**) applied to ports with a **`realtime`** clause.

NOTE: If the wait operation is used right before the operation, the variable will be initialized with the value given to the wait operation.

5.5.1 Obtain the sending time for messages with the send statement

The send statement is extended by an optional redirection clause "**-> timestamp** VariableRef". A send statement that holds a timestamp redirection clause and that is executed successfully assigns the given variable with the sending time of the sent message.

EXAMPLE:

```
p.send(t)-> timestamp myTime;
// yields the sending time of a message
p.receive(t2) -> timestamp myTime2;
if(myTime2-myTime>MAX){setverdict(fail);}
```

Syntactical Structure

```
Port  "." send ["(" TemplateInstance ")"] [ to AddressRef ]
[ -> timestamp VariableRef  ]
```

5.5.2 Obtain the sending time for procedure calls with call statement

The call statement is extended by an optional redirection clause "**-> timestamp** VariableRef". A call statement that holds a timestamp redirection clause and that is executed successfully assigns the given variable with the sending time of the call.

EXAMPLE:

```
p.call(proc: {m}, nowait) -> timestamp myTime;
// yields the sending time of the message call
p.getreply(proc: ?) -> timestamp myTime2;
if(myTime2-myTime>MAX){setverdict(fail);}
```

Syntactical Structure

```
Port  "." call  [ "(" TemplateInstance, CallTimeout ")" ] [ to AddressRef ]
 [ "->"  timestamp VariableRef  ]
 [{ CallAlternatives }]
```

5.5.3 Obtain the sending time for procedure replies with the reply statement

The existing reply statement is extended by an optional redirection clause "-> **timestamp** VariableRef". A reply statement that holds a timestamp redirection clause and that is executed successfully assigns the given variable with the sending time of the reply.

EXAMPLE:

```
p.reply(proc: {m})-> timestamp myTime;
// yields the sending time of the reply
if(myTime>MAX){setverdict(fail);}
```

Syntactical Structure

```
Port "." reply [ "(" TemplateInstance  [ value TemplateInstance ] ")" ] [ to AddressRef ]
[ "->" timestamp VariableRef ]
```

5.5.4 Obtain the sending time for exceptions with the raise statement

The existing raise statement is extended by an optional redirection clause **"-> timestamp** VariableRef". A raise statement that holds a timestamp redirection clause and that is executed successfully assigns the given variable with the sending time of the exception.

EXAMPLE:

```
p.raise(proc, e) -> timestamp myTime;
// yields the sending time of the raised exception
if(myTime>MAX){setverdict(fail);}
```

Syntactical Structure

```
Port "." raise [ "(" ( Signature "," TemplateInstance )   ")" ] [ to AddressRef ]
[ "->" timestamp VariableRef ]
```

6 TRI extensions for the package

6.1 triStartClock (TE → PA)

Signature	`TriStatus triStartClock(in long ticksPerSecond)`
In Parameters	`ticksPerSecond` the precision of the clock given in ticks per second
Out Parameters	n.a
Return Value	The return status of the operation. The return status indicates the success (*`TRI_OK`*) or failure (*`TRI_Error`*) of the operation
Constraints	n.a.
Effect	The operation starts the test system clock with a given precision. The precision is defined by the in parameter *`ticksPerSecond`*. The parameter specifies the number of time units (ticks) that characterizes a second

6.2 triReadClock (TE → PA)

Signature	*TriStatus triReadClock(out long timepoint)*
In Parameters	n.a.
Out Parameters	*timepoint* current time
Return Value	The return status of the operation. The return status indicates the success (*TRI_OK*) or failure (*TRI_Error*) of the operation
Constraints	There was a preceding invocation of *triStartClock(in long ticksPerSecond)*
Effect	The operation yields the actual clock value. The clock value is given by the out parameter *timepoint*, which represents the number of time units (ticks) that has elapsed since the start of the clock (see *triStartClock*)

6.3 triBeginWait (TE → PA)

Signature	*TriStatus triBeginWait(in long timepoint, in TriComponentIDType component)*
In Parameters	*timepoint* point in time until execution of a component should be suspended *component* component whose execution should be suspended
Out Parameters	n.a.
Return Value	The return status of the operation. The return status indicates the success (*TRI_OK*) or failure (*TRI_Error*) of the operation
Constraints	There was a preceding invocation of *triStartClock(in long ticksPerSecond)*
Effect	The operation signals that the execution of component *component* should be suspended until the specified point of time *timepoint* At this point in time the PA will issue a *triEndWait(component)* operation timepoint is expressed as the number of time units (ticks) that has elapsed since the start of the clock (see *triStartClock)* A call to this operation returns immediately. The operation merely triggers the corresponding *triEndWait* operation, it does not schedule the execution of the component If *timepoint* represent a point of time in the past then the operation returns a *TRI_Error* value and has no other effect

6.4 triEndWait (PA → TE)

Signature	*void triEndWait(in TriComponentIDType component)*
In Parameters	*component* component of the corresponding *triBeginWait* operation
Out Parameters	n.a.
Return Value	n.a.
Constraints	There was a preceding invocation of *triBeginWait(timepoint, component)*
Effect	The operation signals that the point in time *timepoint* that was specified in the corresponding *triBeginWait(timepoint, component)* has been reached

6.5 triWaitUntil (SA → PA)

To be able to handle the timestamp reference passed down to the adapter by the sending operations, the adapter shall call the function triWaitUntil. This function, if given an actual timestamp will block until the given time is reached. It will call triBeginWait with the given timestamp value and then wait for the corresponding triEndWait before returning. If given a negative timestamp value, it will read the clock by usage of triReadClock and initialize the given timestamp reference with the current time before returning. If no other way of determining the correct time when to send the message is available, this function can be called directly before sending the message in the adapter.

Signature	`TriStatus triWaitUntil(inout TriTimerDuration timestamp, in TriComponentIdType componentId)`
In Parameters	`timestamp` the point in time to wait for or -1 if no waiting is required `componentId` identifier of the receiving test component
Out Parameters	**`timestamp`** If the timestamp is initialized with -1, it will be initialized with the current time.
Return Value	The return status of the `triCall` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.
Constraints	This operation is called by the SA before sending a message to the SUT.
Effect	This operation shall, if given a positive timestamp, wait until that time has arrived, and, if successful, return TRI_OK. If the time is already in the past, the function will result in Tri_Error. If the given timestamp is -1, it will initialize the timestamp with the current value of the clock before returning TRI_OK.

6.6 Communication Operations

To be able to allow the time triggered message scheduling at system adapter level, we extend the original sending operations with an additional parameter *inout TriTimerDuration timestamp.* The parameter allows the definition of a message scheduling time that has to be controlled by the adapter. Thus, it becomes possible to deliver a message to the system adapter before its intended scheduling time. The adapter is then responsible to schedule the message in time.

To be able to access the arrival time of a message, the receiving operations also get an additional parameter in TriTimerDuration timestamp. This parameter indicates the actual time of arrival so it can be accessed when the message is taken from the queue.

6.6.1 triSendRT (TE → SA)

Signature	`TriStatusType triSendRT(in TriComponentIdType componentId, in TriPortIdType tsiPortId, in TriAddressType SUTaddress, in TriMessageType sendMessage, inout TriTimerDuration timestamp)`
In Parameters	`componentId` identifier of the sending test component `tsiPortId` identifier of the test system interface port via which the message is sent to the SUT Adaptor `SUTaddress` (optional) destination address within the SUT `sendMessage` the encoded message to be sent `timestamp` the point in time when the message has to be sent or has been sent to the SUT
Out Parameters	n.a.
Return Value	The return status of the `triSendRT` 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 unicast send operation on a component port, which has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 send operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case. The encoding of `sendMessage` has to be done in the TE prior to this TRI operation call.
Effect	The SA can send the message to the SUT. The `triSendRT` operation returns ***TRI_OK*** in case it has been completed successfully and in time. Otherwise ***TRI_Error*** shall be returned. Notice that the return value ***TRI_OK*** does not imply that the SUT has received `sendMessage`.

6.6.2 triSendBCRT (TE → SA)

Signature	`TriStatusType triSendBC(in TriComponentIdType componentId,`
	`in TriPortIdType tsiPortId,`
	`in TriMessageType sendMessage,`
	`inout TriTimerDuration timestamp)`

In Parameters	`componentId` identifier of the sending test component `tsiPortId` identifier of the test system interface port via which the message is sent to the SUT Adaptor `sendMessage` the encoded message to be sent `timestamp` the point in time when the message has to be sent or has been sent to the SUT
Out Parameters	n.a.
Return Value	The return status of the `triSendBC` 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 broadcast send operation on a component port, which has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 send operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case. The encoding of `sendMessage` has to be done in the TE prior to this TRI operation call.
Effect	The SA can broadcast the message to the SUT. The `triSendBC` operation returns ***TRI_OK*** in case it has been completed successfully and in time. Otherwise ***TRI_Error*** shall be returned. Notice that the return value ***TRI_OK*** does not imply that the SUT has received `sendMessage`.

6.6.3 triSendMCRT (TE → SA)

Signature	`TriStatusType triSendMC(in TriComponentIdType componentId,`
	`in TriPortIdType tsiPortId,`
	`in TriAddressListType SUTaddresses,`
	`in TriMessageType sendMessage,`
	`inout TriTimerDuration timestamp)`

In Parameters	`componentId` identifier of the sending test component `tsiPortId` identifier of the test system interface port via which the message is sent to the SUT Adaptor `SUTaddresses` destination addresses within the SUT `sendMessage` the encoded message to be sent `timestamp` the point in time when the message has to be sent or has been sent to the SUT
Out Parameters	n.a.
Return Value	The return status of the `triSendMC` 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 multicast send operation on a component port, which has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 send operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case. The encoding of `sendMessage` has to be done in the TE prior to this TRI operation call.
Effect	The SA can multicast the message to the SUT. The `triSendMC` operation returns ***TRI_OK*** in case it has been completed successfully and in time. Otherwise ***TRI_Error*** shall be returned. Notice that the return value ***TRI_OK*** does not imply that the SUT has received `sendMessage`.

6.6.4 triEnqueueMsgRT (SA → TE)

Signature	`void triEnqueueMsg(in TriPortIdType tsiPortId,`
	`in TriAddressType SUTaddress,`
	`in TriComponentIdType componentId,`
	`in TriMessageType receivedMessage,`
	`in TriTimerDuration timestamp)`	

In Parameters	`tsiPortId`	identifier of the test system interface port via which the message is enqueued by the SUT Adaptor
	`SUTaddress`	(optional) source address within the SUT
	`componentId`	identifier of the receiving test component
	`receivedMessage`	the encoded received message
	`timestamp`	the point in time when the message has been received from the SUT
Out Parameters	n.a.	
Return Value	Void	
Constraints	This operation is called by the SA after it has received a message from the SUT. It can only be used when tsiPortId has been either previously mapped to a port of componentId or has been referenced in the previous triExecuteTestCase statement. In the invocation of a `triEnqueueMsg` operation `receivedMessage` shall contain an encoded value.	
Effect	This operation shall pass the message to the TE indicating the component componentId to which the TSI port tsiPortId is mapped. The decoding of receivedMessage has to be done in the TE.	

6.6.5 triCallRT (TE → SA)

Signature	`TriStatusType triCall(in TriComponentIdType componentId,`
	`in TriPortIdType tsiPortId,`
	`in TriAddressType SUTaddress,`
	`in TriSignatureIdType signatureId,`
	`in TriParameterListType parameterList,`
	`inout TriTimerDuration timestamp)`	

In Parameters	`componentId`	identifier of the test component issuing the procedure call
	`tsiPortId`	identifier of the test system interface port via which the procedure call is sent to the SUT Adaptor
	`SUTaddress`	(optional) destination address within the SUT
	`signatureId`	identifier of the signature of the procedure call
	`parameterList`	a list of encoded parameters which are part of the indicated signature. The parameters in `parameterList` are ordered as they appear in the TTCN-3 signature declaration
	`timestamp`	the point in time when the call has to be sent or has been sent to the SUT
Out Parameters	n.a.	
Return Value	The return status of the `triCall` 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 unicast call operation on a component port, which has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 call operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case. All *in* and *inout* procedure parameters contain encoded values. The procedure parameters are the parameters specified in the TTCN-3 signature template. Their encoding has to be done in the TE prior to this TRI operation call.	
Effect	On invocation of this operation the SA can initiate the procedure call corresponding to the signature identifier `signatureId` and the TSI port `tsiPortId`. The `triCall` operation shall return without waiting for the return of the issued procedure call (see note). This TRI operation returns *TRI_OK* on successful initiation of the procedure call in time, *TRI_Error* otherwise. No error shall be indicated by the SA in case the value of any *out* parameter is non-null. Notice that the return value of this TRI operation does not make any statement about the success or failure of the procedure call. Note that an optional timeout value, which can be specified in the TTCN-3 ATS for a call operation, is *not* included in the `triCall` operation signature. The TE is responsible to address this issue by starting a timer for the TTCN-3 call operation in the PA with a separate TRI operation call, i.e. `triStartTimer`.	
NOTE: This might be achieved for example by spawning a new thread or process. This handling of this procedure call is, however, dependent on implementation of the TE.		

6.6.6 triCallBCRT (TE → SA)

Signature	`TriStatusType triCallBC(in TriComponentIdType componentId,`
	`in TriPortIdType tsiPortId,`
	`in TriSignatureIdType signatureId,`
	`in TriParameterListType parameterList,`
	`inout TriTimerDuration timestamp)`

In Parameters	`componentId` identifier of the test component issuing the procedure call `tsiPortId` identifier of the test system interface port via which the procedure call is sent to the SUT Adaptor `signatureId` identifier of the signature of the procedure call `parameterList` a list of encoded parameters which are part of the indicated signature. The parameters in `parameterList` are ordered as they appear in the TTCN-3 signature declaration. `timestamp` the point in time when the call has to be sent or has been sent to the SUT
Out Parameters	n.a.
Return Value	The return status of the `triCallBC` 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 broadcast call operation on a component port, which has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 call operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case. All *in* and *inout* procedure parameters contain encoded values. The procedure parameters are the parameters specified in the TTCN-3 signature template. Their encoding has to be done in the TE prior to this TRI operation call.
Effect	On invocation of this operation the SA can initiate and broadcast the procedure call corresponding to the signature identifier `signatureId` and the TSI port `tsiPortId`. The `triCallBC` operation shall return without waiting for the return of the issued procedure call (see note). This TRI operation returns *TRI_OK* on successful initiation of the procedure call in time, *TRI_Error* otherwise. No error shall be indicated by the SA in case the value of any *out* parameter is non-null. Notice that the return value of this TRI operation does not make any statement about the success or failure of the procedure call. Note that an optional timeout value, which can be specified in the TTCN-3 ATS for a call operation, is *not* included in the `triCallBC` operation signature. The TE is responsible to address this issue by starting a timer for the TTCN-3 call operation in the PA with a separate TRI operation call, i.e. `triStartTimer`.
NOTE: This might be achieved for example by spawning a new thread or process. This handling of this procedure call is, however, dependent on implementation of the TE.	

6.6.7 triCallMCRT (TE → SA)

Signature	`TriStatusType triCallMC(in TriComponentIdType componentId,`
	`in TriPortIdType tsiPortId,`
	`in TriAddressListType SUTaddresses,`
	`in TriSignatureIdType signatureId,`
	`in TriParameterListType parameterList,`
	`inout TriTimerDuration timestamp)`

In Parameters	`componentId` identifier of the test component issuing the procedure call `tsiPortId` identifier of the test system interface port via which the procedure call is sent to the SUT Adaptor `SUTaddresses` destination addresses within the SUT `signatureId` identifier of the signature of the procedure call `parameterList` a list of encoded parameters which are part of the indicated signature. The parameters in `parameterList` are ordered as they appear in the TTCN-3 signature declaration. `timestamp` the point in time when the call has to be sent or has been sent to the SUT
Out Parameters	n.a.
Return Value	The return status of the `triCallMC` 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 multicast call operation on a component port, which has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 call operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case. All *in* and *inout* procedure parameters contain encoded values. The procedure parameters are the parameters specified in the TTCN-3 signature template. Their encoding has to be done in the TE prior to this TRI operation call.
Effect	On invocation of this operation the SA can initiate and multicast the procedure call corresponding to the signature identifier `signatureId` and the TSI port `tsiPortId`. The `triCallMC` operation shall return without waiting for the return of the issued procedure call (see note). This TRI operation returns *TRI_OK* on successful initiation of the procedure call in time, *TRI_Error* otherwise. No error shall be indicated by the SA in case the value of any *out* parameter is non-null. Notice that the return value of this TRI operation does not make any statement about the success or failure of the procedure call. Note that an optional timeout value, which can be specified in the TTCN-3 ATS for a call operation, is *not* included in the `triCallMC` operation signature. The TE is responsible to address this issue by starting a timer for the TTCN-3 call operation in the PA with a separate TRI operation call, i.e. `triStartTimer`.
NOTE: This might be achieved for example by spawning a new thread or process. This handling of this procedure call is, however, dependent on implementation of the TE.	

6.6.8 triReplyRT (TE → SA)

Signature	`TriStatusType triReply(in TriComponentIdType componentId,`
	`in TriPortIdType tsiPortId,`
	`in TriAddressType SUTaddress,`
	`in TriSignatureIdType signatureId,`
	`in TriParameterListType parameterList,`
	`in TriParameterType returnValue,`
	`inout TriTimerDuration timestamp)`

In Parameters	`componentId` identifier of the replying test component `tsiPortId` identifier of the test system interface port via which the reply is sent to the SUT Adaptor `SUTaddress` (optional) destination address within the SUT `signatureId` identifier of the signature of the procedure call `parameterList` a list of encoded parameters which are part of the indicated signature. The parameters in `parameterList` are ordered as they appear in the TTCN-3 signature declaration `returnValue` (optional) encoded return value of the procedure call `timestamp` the point in time when the reply has to be sent or has been sent to the SUT
Out Parameters	n.a.
Return Value	The return status of the `triReply` 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 unicast reply operation on a component port that has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 reply operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case. All *out* and *inout* procedure parameters and the return value contain encoded values. The `parameterList` contains procedure call parameters. These parameters are the parameters specified in the TTCN-3 signature template. Their encoding has to be done in the TE prior to this TRI operation call. If no return type has been defined for the procedure signature in the TTCN-3 ATS, the distinct value `null` shall be passed for the return value.
Effect	On invocation of this operation the SA can issue the reply to a procedure call corresponding to the signature identifier `signatureId` and the TSI port `tsiPortId`. The `triReply` operation will return ***TRI_OK*** on successful execution of this operation in time, ***TRI_Error*** otherwise. The SA shall indicate no error in case the value of any *in* parameter or an undefined return value is different from null.

6.6.9 triReplyBCRT (TE → SA)

Signature	`TriStatusType triReplyBC(in TriComponentIdType componentId,`
	`in TriPortIdType tsiPortId,`
	`in TriSignatureIdType signatureId,`
	`in TriParameterListType parameterList,`
	`in TriParameterType returnValue,`
	`inout TriTimerDuration timestamp)`

In Parameters	`componentId` identifier of the replying test component `tsiPortId` identifier of the test system interface port via which the reply is sent to the SUT Adaptor `signatureId` identifier of the signature of the procedure call `parameterList` a list of encoded parameters which are part of the indicated signature. The parameters in `parameterList` are ordered as they appear in the TTCN-3 signature declaration `returnValue` (optional) encoded return value of the procedure call `timestamp` the point in time when the reply has to be sent or has been sent to the SUT
Out Parameters	n.a.
Return Value	The return status of the `triReplyBC` 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 broadcast reply operation on a component port that has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 reply operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case. All *out* and *inout* procedure parameters and the return value contain encoded values. The `parameterList` contains procedure call parameters. These parameters are the parameters specified in the TTCN-3 signature template. Their encoding has to be done in the TE prior to this TRI operation call. If no return type has been defined for the procedure signature in the TTCN-3 ATS, the distinct value `null` shall be passed for the return value.
Effect	On invocation of this operation the SA can broadcast the reply to procedure calls corresponding to the signature identifier `signatureId` and the TSI port `tsiPortId`. The `triReplyBC` operation will return ***TRI_OK*** on successful execution of this operation in time, ***TRI_Error*** otherwise. The SA shall indicate no error in case the value of any *in* parameter or an undefined return value is different from null.

6.6.10 triReplyMCRT (TE → SA)

Signature	`TriStatusType triReplyMC(in TriComponentIdType componentId,`
	`in TriPortIdType tsiPortId,`
	`in TriAddressListType SUTaddresses,`
	`in TriSignatureIdType signatureId,`
	`in TriParameterListType parameterList,`
	`in TriParameterType returnValue,`
	`inout TriTimerDuration timestamp)`

In Parameters	`componentId` identifier of the replying test component `tsiPortId` identifier of the test system interface port via which the reply is sent to the SUT Adaptor `SUTaddresses` destination addresses within the SUT `signatureId` identifier of the signature of the procedure call `parameterList` a list of encoded parameters which are part of the indicated signature. The parameters in `parameterList` are ordered as they appear in the TTCN-3 signature declaration `returnValue` (optional) encoded return value of the procedure call `timestamp` the point in time when the reply has to be sent or has been sent to the SUT
Out Parameters	n.a.
Return Value	The return status of the `triReplyMC` 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 multicast reply operation on a component port that has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 reply operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case. All *out* and *inout* procedure parameters and the return value contain encoded values. The `parameterList` contains procedure call parameters. These parameters are the parameters specified in the TTCN-3 signature template. Their encoding has to be done in the TE prior to this TRI operation call. If no return type has been defined for the procedure signature in the TTCN-3 ATS, the distinct value `null` shall be passed for the return value.
Effect	On invocation of this operation the SA can multicast the reply to procedure calls corresponding to the signature identifier `signatureId` and the TSI port `tsiPortId`. The `triReplyMC` operation will return ***TRI_OK*** on successful execution of this operation in time, ***TRI_Error*** otherwise. The SA shall indicate no error in case the value of any *in* parameter or an undefined return value is different from null.

6.6.11 triRaiseRT (TE → SA)

Signature	`TriStatusType triRaise(in TriComponentIdType componentId,`
	`in TriPortIdType tsiPortId,`
	`in TriAddressType SUTaddress,`
	`in TriSignatureIdType signatureId,`
	`in TriExceptionType exc,`
	`inout TriTimerDuration timestamp)`

In Parameters	`componentId` identifier of the test component raising the exception `tsiPortId` identifier of the test system interface port via which the exception is sent to the SUT Adaptor `SUTaddress` (optional) destination address within the SUT `signatureId` identifier of the signature of the procedure call which the exception is associated with `Exc` the encoded exception `timestamp` the point in time when the exception has to be sent or has been sent to the SUT
Out Parameters	n.a.
Return Value	The return status of the `triRaise` 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 unicast raise operation on a component port that has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 raise operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case. The encoding of the exception has to be done in the TE prior to this TRI operation call.
Effect	On invocation of this operation the SA can raise an exception to a procedure call corresponding to the signature identifier `signatureId` and the TSI port `tsiPortId`. The `triRaise` operation returns *TRI_OK* on successful execution of the operation in time, *TRI_Error* otherwise.

6.6.12 triRaiseBCRT (TE → SA)

Signature	`TriStatusType triRaiseBC(in TriComponentIdType componentId,`
	`in TriPortIdType tsiPortId,`
	`in TriSignatureIdType signatureId,`
	`in TriExceptionType exc,`
	`inout TriTimerDuration timestamp)`

In Parameters	`componentId` identifier of the test component raising the exception `tsiPortId` identifier of the test system interface port via which the exception is sent to the SUT Adaptor `signatureId` identifier of the signature of the procedure call which the exception is associated with `exc` the encoded exception `timestamp` the point in time when the exception has to be sent or has been sent to the SUT
Out Parameters	n.a.
Return Value	The return status of the `triRaiseBC` 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 broadcast raise operation on a component port that has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 raise operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case. The encoding of the exception has to be done in the TE prior to this TRI operation call.
Effect	On invocation of this operation the SA can raise and broadcast an exception to procedure calls corresponding to the signature identifier `signatureId` and the TSI port `tsiPortId`. The `triRaiseBC` operation returns *TRI_OK* on successful execution of the operation in time, *TRI_Error* otherwise.

6.6.13 triRaiseMCRT (TE → SA)

Signature	`TriStatusType triRaiseMC(in TriComponentIdType componentId,`
	`in TriPortIdType tsiPortId,`
	`in TriAddressListType SUTaddresses,`
	`in TriSignatureIdType signatureId,`
	`in TriExceptionType exc,`
	`inout TriTimerDuration timestamp)`

In Parameters	`componentId` identifier of the test component raising the exception `tsiPortId` identifier of the test system interface port via which the exception is sent to the SUT Adaptor `SUTaddresses` destination addresses within the SUT `signatureId` identifier of the signature of the procedure call which the exception is associated with `exc` the encoded exception `timestamp` the point in time when the exception has to be sent or has been sent to the SUT
Out Parameters	n.a.
Return Value	The return status of the `triRaiseMC` 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 multicast raise operation on a component port that has been mapped to a TSI port. This operation is called by the TE for all TTCN-3 raise operations if no system component has been specified for a test case, i.e. only a MTC test component is created for a test case. The encoding of the exception has to be done in the TE prior to this TRI operation call.
Effect	On invocation of this operation the SA can raise and multicast an exception to a procedure calls corresponding to the signature identifier `signatureId` and the TSI port `tsiPortId`. The `triRaiseMC` operation returns *TRI_OK* on successful execution of the operation in time, *TRI_Error* otherwise.

6.6.14 triEnqueueCallRT (SA → TE)

Signature	`void triEnqueueCall(in TriPortIdType tsiPortId,`
	`in TriAddressType SUTaddress,`
	`in TriComponentIdType componentId,`
	`in TriSignatureIdType signatureId,`
	`in TriParameterListType parameterList,`
	`in TriTimerDuration timestamp)`

In Parameters	`tsiPortId` identifier of the test system interface port via which the procedure call is enqueued by the SUT Adaptor `SUTaddress` (optional) source address within the SUT `componentId` identifier of the receiving test component `signatureId` identifier of the signature of the procedure call `parameterList` a list of encoded parameters which are part of the indicated signature. The parameters in `parameterList` are ordered as they appear in the TTCN-3 signature declaration. Description of data passed as parameters to the operation from the calling entity to the called entity `timestamp` the point in time when the call has been received from the SUT
Out Parameters	n.a.
Return Value	Void
Constraints	This operation can be called by the SA after it has received a procedure call from the SUT. It can only be used when `tsiPortId` has been either previously mapped to a port of `componentId` or referenced in the previous `triExecuteTestCase` statement. In the invocation of a `triEnqueueCall` operation all *in* and *inout* procedure parameters contain encoded values.
Effect	The TE can enqueue this procedure call with the signature identifier `signatureId` at the port of the component `componentId` to which the TSI port `tsiPortId` is mapped. The decoding of procedure parameters has to be done in the TE. The TE shall indicate no error in case the value of any *out* parameter is different from null.

6.6.15 triEnqueueReplyRT (SA → TE)

Signature	`void triEnqueueReply(in TriPortIdType tsiPortId,`
	`in TriAddressType SUTaddress,`
	`in TriComponentIdType componentId,`
	`in TriSignatureIdType signatureId,`
	`in TriParameterListType parameterList,`
	`in TriParameterType returnValue,`
	`in TriTimerDuration timestamp)`	

In Parameters	`tsiPortId`	identifier of the test system interface port via which the reply is enqueued by the SUT Adaptor
	`SUTaddress`	(optional) source address within the SUT
	`componentId`	identifier of the receiving test component
	`signatureId`	identifier of the signature of the procedure call
	`parameterList`	a list of encoded parameters which are part of the indicated signature. The parameters in `parameterList` are ordered as they appear in the TTCN-3 signature declaration
	`returnValue`	(optional) encoded return value of the procedure call
	`timestamp`	the point in time when the reply has been received from the SUT
Out Parameters	n.a.	
Return Value	Void	
Constraints	This operation can be called by the SA after it has received a reply from the SUT. It can only be used when `tsiPortId` has been either previously mapped to a port of `componentId` or referenced in the previous `triExecuteTestCase` statement. In the invocation of a `triEnqueueReply` operation all *out* and *inout* procedure parameters and the return value contain encoded values. If no return type has been defined for the procedure signature in the TTCN-3 ATS, the distinct value `null` shall be used for the return value.	
Effect	The TE can enqueue this reply to the procedure call with the signature identifier `signatureId` at the port of the component `componentId` to which the TSI port `tsiPortId` is mapped. The decoding of the procedure parameters has to be done within the TE. The TE shall indicate no error in case the value of any *in* parameter or an undefined return value is different from null.	

6.6.16 triEnqueueExceptionRT (SA → TE)

Signature	`void triEnqueueException(in TriPortIdType tsiPortId,`
	`in TriAddressType SUTaddress,`
	`in TriComponentIdType componentId,`
	`in TriSignatureIdType signatureId,`
	`in TriExceptionType exc,`
	`in TriTimerDuration timestamp)`	

In Parameters	`tsiPortId`	identifier for the test system interface port via which the exception is enqueued by the SUT Adaptor
	`SUTaddress`	(optional) source address within the SUT
	`componentId`	identifier of the receiving test component
	`signatureId`	identifier of the signature of the procedure call which the exception is associated with
	`exc`	the encoded exception
	`timestamp`	the point in time when the exception has been received from the SUT
Out Parameters	n.a.	
Return Value	Void	
Constraints	This operation can be called by the SA after it has received a reply from the SUT. It can only be used when `tsiPortId` has been either previously mapped to a port of `componentId` or referenced in the previous `triExecuteTestCase` statement. In the invocation of a `triEnqueueException` operation `exception` shall contain an encoded value.	
Effect	The TE can enqueue this exception for the procedure call with the signature identifier `signatureId` at the port of the component `componentId` to which the TSI port `tsiPortId` is mapped. The decoding of the exception has to be done within the TE.	

6.7 Definition of Interfaces

Instead of changing the existing interfaces, we define new additional interfaces containing the newly introduced declarations:

- triCommunicationSART – TE → SA
- triCommunicationTERT – SA → RT
- triPlatformPART – TE → SA
- triPlatformTERT – TE → SA

6.8 Changes for Java™ Language Mapping

For all methods, the timestamp parameter is mapped to a parameter of type `TriTimerDuration`.

6.8.1 Mapping of interface triCommunicationSART

```
package org.etsi.ttcn.tri.rt;

import org.etsi.ttcn.tri.*;

public interface TriCommunicationSART {
    // Message based communication operations
    // Ref: TRI-Definition 5.5.3.1
    public TriStatus triSend(TriComponentId componentId, TriPortId tsiPortId,
            TriAddress sutAddress, TriMessage sendMessage, inout TriTimerDuration timestamp);
    // Ref: TRI-Definition 5.5.3.2
    public TriStatus triSendBC(TriComponentId componentId, TriPortId tsiPortId,
            TriMessage sendMessage, inout TriTimerDuration timestamp);
    // Ref: TRI-Definition 5.5.3.3
    public TriStatus triSendMC(TriComponentId componentId, TriPortId tsiPortId,
            TriAddressList sutAddresses, TriMessage sendMessage, inout TriTimerDuration timestamp);

    // Procedure based communication operations
    // Ref: TRI-Definition 5.5.4.1
    public TriStatus triCall(TriComponentId componentId,
            TriPortId tsiPortId, TriAddress sutAddress,
            TriSignatureId signatureId, TriParameterList parameterList, inout TriTimerDuration
timestamp);
    // Ref: TRI-Definition 5.5.4.2
    public TriStatus triCallBC(TriComponentId componentId,
            TriPortId tsiPortId,
            TriSignatureId signatureId, TriParameterList parameterList, inout TriTimerDuration
timestamp);
    // Ref: TRI-Definition 5.5.4.3
    public TriStatus triCallMC(TriComponentId componentId,
            TriPortId tsiPortId, TriAddressList sutAddresses,
            TriSignatureId signatureId, TriParameterList parameterList, inout TriTimerDuration
timestamp);

    // Ref: TRI-Definition 5.5.4.4
    public TriStatus triReply(TriComponentId componentId,
            TriPortId tsiPortId, TriAddress sutAddress,
            TriSignatureId signatureId, TriParameterList parameterList,
            TriParameter returnValue, inout TriTimerDuration timestamp);
    // Ref: TRI-Definition 5.5.4.5
    public TriStatus triReplyBC(TriComponentId componentId,
            TriPortId tsiPortId,
            TriSignatureId signatureId, TriParameterList parameterList,
            TriParameter returnValue, inout TriTimerDuration timestamp);
    // Ref: TRI-Definition 5.5.4.6
    public TriStatus triReplyMC(TriComponentId componentId,
            TriPortId tsiPortId, TriAddressList sutAddresses,
            TriSignatureId signatureId, TriParameterList parameterList,
            TriParameter returnValue, inout TriTimerDuration timestamp);

    // Ref: TRI-Definition 5.5.4.7
    public TriStatus triRaise(TriComponentId componentId, TriPortId tsitPortId,
            TriAddress sutAddress,
            TriSignatureId signatureId,
```

```
            TriException exc, inout TriTimerDuration timestamp);
    // Ref: TRI-Definition 5.5.4.8
    public TriStatus triRaiseBC(TriComponentId componentId,
            TriPortId tsitPortId,
            TriSignatureId signatureId,
            TriException exc, inout TriTimerDuration timestamp);
    // Ref: TRI-Definition 5.5.4.9
    public TriStatus triRaiseMC(TriComponentId componentId, TriPortId tsitPortId,
            TriAddresses sutAddresses,
            TriSignatureId signatureId,
            TriException exc, inout TriTimerDuration timestamp);
}
```

6.8.2 Mapping of interface triCommunicationTERT

```
package org.etsi.ttcn.tri.rt;

import org.etsi.ttcn.tri.*;

public interface TriCommunicationTERT {
    // Message based communication operations
    // Ref: TRI-Definition 5.5.3.4
    public void triEnqueueMsg(TriPortId tsiPortId,
            TriAddress sutAddress, TriComponentId componentId,
            TriMessage receivedMessage, TriTimerDuration timestamp);

    // Procedure based communication operations
    // Ref: TRI-Definition 5.5.4.10
    public void triEnqueueCall(TriPortId tsiPortId,
            TriAddress sutAddress, TriComponentId componentId,
            TriSignatureId signatureId, TriParameterList parameterList, TriTimerDuration timestamp);

    // Ref: TRI-Definition 5.5.4.11
    public void triEnqueueReply(TriPortId tsiPortId, TriAddress sutAddress,
            TriComponentId componentId, TriSignatureId signatureId,
            TriParameterList parameterList, TriParameter returnValue, TriTimerDuration timestamp);

    // Ref: TRI-Definition 5.5.4.12
    public void triEnqueueException(TriPortId tsiPortId,
            TriAddress sutAddress, TriComponentId componentId,
            TriSignatureId signatureId, TriException exc, TriTimerDuration timestamp);
}
```

6.8.3 Mapping of interface triPlatformPART

```
package org.etsi.ttcn.tri.rt;

import org.etsi.ttcn.tri.*;

public interface TriPlatformPART {

    // Timer handling operations
    // Ref: TRI-Definition 6.1
    public TriStatus triStartClock(long ticksPerSecond);

    // Ref: TRI-Definition 6.2
    public TriStatus triReadClock(TriTimerDuration timestamp);

    // Ref: TRI-Definition 6.3
    public TriStatus triBeginWait(TriTimerDuration timestamp, TriComponentId componentId);

    // Ref: TRI-Definition 6.5
    public TriStatus triWaitUntil(TriTimerDuration timestamp, TriComponentId componentId);

}
```

6.8.4 Mapping of interface triPlatformTE

The following declarations have to be added to the interface **triPlatformTE**:

```
package org.etsi.ttcn.tri.rt;

import org.etsi.ttcn.tri.*;
```

```
public interface TriPlatformTERT {

    // Ref: TRI-Definition 6.4
    public TriStatus triEndWait(TriComponentId componentId);
}
```

6.9 Changes for ANSI C Language Mapping

The following declarations have to be added:

```
TriStatus triStartClock
(long ticksPerSecond)
TriStatus triReadClock
(TriTimerDuration* timepoint)
TriStatus triBeginWait
(TriTimerDuration timepoint,
 TriComponentId* componentId)
TriStatus triWaitUntil
(TriTimerDuration* timepoint,
 TriComponentId* componentId)
TriStatus triEndWait
(TriComponentId* componentId)
TriStatus triSendRTRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriAddress* sutAddress,
 const TriMessage* sendMessage,
 TriTimerDuration* timepoint)
TriStatus triSendBCRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriMessage* sendMessage,
 TriTimerDuration* timepoint)
TriStatus triSendMCRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriAddressList* sutAddresses,
 const TriMessage* sendMessage,
 TriTimerDuration* timepoint)
void triEnqueueMsgRT
(const TriPortId* tsiPortId,
 const TriAddress* sutAddress,
 const TriComponentId* componentId,
 const TriMessage* receivedMessage,
 TriTimerDuration timepoint)
TriStatus triCallRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriAddress* sutAddress,
 const TriSignatureId* signatureId,
 const TriParameterList* parameterList,
 TriTimerDuration* timepoint)
TriStatus triCallRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriAddress* sutAddress,
 const TriSignatureId* signatureId,
 const TriParameterList* parameterList,
 TriTimerDuration* timepoint)
TriStatus triCallBCRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriSignatureId* signatureId,
 const TriParameterList* parameterList,
 TriTimerDuration* timepoint)
TriStatus triCallMCRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriAddressList* sutAddresses,
 const TriSignatureId* signatureId,
 const TriParameterList* parameterList,
 TriTimerDuration* timepoint)
TriStatus triReplyRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriAddress* sutAddress,
```

```
 const TriSignatureId* signatureId,
 const TriParameterList* parameterList,
 const TriParameter* returnValue,
 TriTimerDuration* timepoint)
TriStatus triReplyBCRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriSignatureId* signatureId,
 const TriParameterList* parameterList,
 const TriParameter* returnValue,
 TriTimerDuration* timepoint)
TriStatus triReplyMCRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriAddressList* sutAddresses,
 const TriSignatureId* signatureId,
 const TriParameterList* parameterList,
 const TriParameter* returnValue,
 TriTimerDuration* timepoint)
TriStatus triRaiseRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriAddress* sutAddress,
 const TriSignatureId* signatureId,
 const TriException* exception,
 TriTimerDuration* timepoint)
TriStatus triRaiseBCRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriSignatureId* signatureId,
 const TriException* exception,
 TriTimerDuration* timepoint)
TriStatus triRaiseMCRT
(const TriComponentId* componentId,
 const TriPortId* tsiPortId,
 const TriAddressList* sutAddresses,
 const TriSignatureId* signatureId,
 const TriException* exception,
 TriTimerDuration* timepoint)
void triEnqueueCallRT
(const TriPortId* tsiPortId,
 const TriAddress* sutAddress,
 const TriComponentId* componentId,
 const TriSignatureId* signatureId,
 const TriParameterList* parameterList,
 TriTimerDuration timepoint)
void triEnqueueReplyRT
(const TriPortId* tsiPortId,
 const TriAddress* sutAddress,
 const TriComponentId* componentId,
 const TriSignatureId* signatureId,
 const TriParameterList* parameterList,
 const TriParameter* returnValue,
 TriTimerDuration timepoint)
void triEnqueueExceptionRT
(const TriPortId* tsiPortId,
 const TriAddress* sutAddress,
 const TriComponentId* componentId,
 const TriSignatureId* signatureId,
 const TriException* exception,
 TriTimerDuration timepoint)
```

6.10 Changes for C++ Language Mapping

6.10.1 Mapping of interface triCommunicationSART

```
class TriCommunicationSART {
public:

  //Destructor.
  virtual ~TriCommunicationSART ();

  //To reset the System Adaptor
  virtual TriStatus triSAReset ()=0;
```

```
  //Send operation on a component which has been mapped to a TSI port.
  virtual TriStatus triSend (const TriComponentId *componentId, const TriPortId *tsiPortId,
  const TriAddress *SUTaddress, const TriMessage *sendMessage, TriTimerDuration* timepoint)=0;

  //Send (broadcast) operation on a component which has been mapped to a TSI port.
  virtual TriStatus triSendBC (const TriComponentId *componentId, const TriPortId *tsiPortId,
  const TriMessage *sendMessage, TriTimerDuration* timepoint)=0;

  //Send (multicast) operation on a component which has been mapped to a TSI port.
  virtual TriStatus triSendMC (const TriComponentId *componentId, const TriPortId *tsiPortId,
  const TriAddressList *SUTaddresses, const TriMessage *sendMessage, TriTimerDuration*
  timepoint)=0;

  //Initiate the procedure call.
  virtual TriStatus triCall (const TriComponentId *componentId, const TriPortId *tsiPortId,
  const TriAddress *sutAddress, const TriSignatureId *signatureId, TriParameterList
  *parameterList, const TriTimerDuration* timepoint)=0;

  //Initiate and broadcast the procedure call.
  virtual TriStatus triCallBC (const TriComponentId *componentId, const TriPortId *tsiPortId,
  const TriSignatureId *signatureId, const TriParameterList *parameterList, TriTimerDuration*
  timepoint)=0;

  //Initiate and multicast the procedure call.
  virtual TriStatus triCallMC (const TriComponentId *componentId, const TriPortId *tsiPortId,
  const TriAddressList *sutAddresses, const TriSignatureId *signatureId, const TriParameterList
  *parameterList, TriTimerDuration* timepoint)=0;

  //Issue the reply to a procedure call.
  virtual TriStatus triReply (const TriComponentId *componentId, const TriPortId *tsiPortId,
  const TriAddress *sutAddress, const TriSignatureId *signatureId, const TriParameterList
  *parameterList, const TriParameter *returnValue, TriTimerDuration* timepoint)=0;

  //Broadcast the reply to a procedure call.
  virtual TriStatus triReplyBC (const TriComponentId *componentId, const TriPortId *tsiPortId,
  const TriSignatureId *signatureId, const TriParameterList *parameterList, const TriParameter
  *returnValue, TriTimerDuration* timepoint)=0;

  //Multicast the reply to a procedure call.
  virtual TriStatus triReplyMC (const TriComponentId *componentId, const TriPortId *tsiPortId,
  const TriAddressList *sutAddresses, const TriSignatureId *signatureId, const TriParameterList
  *parameterList, const TriParameter *returnValue, TriTimerDuration* timepoint)=0;

  //Raise an exception to a procedure call.
  virtual TriStatus triRaise (const TriComponentId *componentId, const TriPortId *tsiPortId,
  const TriAddress *sutAddress, const TriSignatureId *signatureId, const TriException *exc,
  TriTimerDuration* timepoint)=0;

  //Raise an broadcast an exception to a procedure call.
  virtual TriStatus triRaiseBC (const TriComponentId *componentId, const TriPortId *tsiPortId,
  const TriSignatureId *signatureId, const TriException *exc, TriTimerDuration* timepoint)=0;

  //Raise an multicast an exception to a procedure call.
  virtual TriStatus triRaiseMC (const TriComponentId *componentId, const TriPortId *tsiPortId,
  const TriAddressList *sutAddresses, const TriSignatureId *signatureId, const TriException
  *exc, TriTimerDuration* timepoint)=0;
}
```

6.10.2 Mapping of interface triCommunicationTERT

```
class TriCommunicationTERT {
public:

  //Destructor.
  virtual ~TriCommunicationTERT ();

  //Called by SA after it has received a message from the SUT.
  virtual void triEnqueueMsg (const TriPortId *tsiPortId, const TriAddress *SUTaddress, const
  TriComponentId *componentId, const TriMessage *receivedMessage, const TriTimerDuration*
  timepoint)=0;

  //Called by SA after it has received a procedure call from the SUT.
  virtual void triEnqueueCall (const TriPortId *tsiPortId, const TriAddress *SUTaddress, const
  TriComponentId *componentId, const TriSignatureId *signatureId, const TriParameterList
  *parameterList, const TriTimerDuration* timepoint)=0;

  //Called by SA after it has received a reply from the SUT.
```

```
  virtual void triEnqueueReply (const TriPortId *tsiPortId, const TriAddress *SUTaddress, const
  TriComponentId *componentId, const TriSignatureId *signatureId, const TriParameterList
  *parameterList, const TriParameter *returnValue, const TriTimerDuration* timepoint)=0;

  //Called by SA after it has received an exception from the SUT.
  virtual void triEnqueueException (const TriPortId *tsiPortId, const TriAddress *SUTaddress,
  const TriComponentId *componentId, const TriSignatureId *signatureId, const TriException *exc,
  const TriTimerDuration* timepoint)=0;

}
```

6.10.3 Mapping of interface triPlatformPART

```
class TriPlatformPART {
public:

  //Destructor.
  virtual ~TriPlatformPART ();

  //Reset all realtime activities which it is currently performing.
  virtual TriStatus triPAReset ()=0;

  //Start the global clock for the testcase with the given time progress.
  virtual TriStatus triStartClock (const long ticksPerSecond)=0;

  //Access the time that elapsed since the testcase was started.
  virtual TriStatus triReadClock (TriTimerDuration *elapsedTime)=0;

  //Begin waiting before the indicated component is notified that given timepoint is reached.
  virtual TriStatus triBeginWait (const TriTimerDuration *timepoint, const TriComponentId*
  componentId)=0;

//Wait until the given timepoint is reached or initialize timepoint with the time that
  //that elapsed since the testcase was started
  virtual TriStatus triBeginWait (TriTimerDuration *timepoint, const TriComponentId*
  componentId)=0;


}
```

6.10.4 Mapping of interface triPlatformTERT

```
class TriPlatformTERT {
public:

  //Destructor.
  virtual ~TriPlatformTERT ();

  //Notify the TE that the indicated component should stop waiting.
  virtual void triEndWAit(const TriComponentId *componentId);

}
```

7 TCI extensions for the package

No changes in TCI necessary.

Annex A (normative): BNF and static semantics

A.1 Changed BNF Rules

```
OpCall ::= ConfigurationOps |
                GetLocalVerdict |
                TimerOps |
                TestcaseInstance |
                ( FunctionInstance [ ExtendedFieldReference ] ) |
                ( TemplateOps [ ExtendedFieldReference ] ) |
                ActivateOp |
                NowOp
PortRedirect ::= PortRedirectSymbol
                    (ValueSpec [SenderSpec] [TimestampSpec] |
                     SenderSpec [TimestampSpec] |
                     TimestampSpec)
MessageAttribs ::= MessageKeyword [RealtimeSpec]
                      "{" {(AddressDecl | MessageList | ConfigParamDef) [SemiColon]}+ "}"
ProcedureAttribs ::= ProcedureKeyword [RealtimeSpec]
                        "{" {(AddressDecl | ProcedureList | ConfigParamDef) [SemiColon]}+ "}"
MixedAttribs ::= MixedKeyword [RealtimeSpec]
                    "{" {AddressDecl | MixedList | ConfigParamDef) [SemiColon]}+ "}"
FunctionStatement ::= ConfigurationStatements |
                          TimerStatements |
                          CommunicationStatements |
                          BasicStatements |
                          BehaviourStatements |
                          SetLocalVerdict |
                          SUTStatements |
                          TestcaseOperation |
                          RealtimeStatement
```

A.2 New BNF Rules

```
NowOp ::= "now"
TimestampSpec ::= "timestamp" VariableRef
RealtimeSpec ::= "realtime"
RealtimeStatement ::= WaitStatement
WaitStatement ::= "wait" "(" SingleExpression ")"
/* STATIC SEMANTICS - the SingleExpression operand shall be of type float or derivatives of this
type. */
```

Annex B (informative): Bibliography

Recommendation ITU-T T.50 (1992): "International Reference Alphabet (IRA) (Formerly International Alphabet No. 5 or IA5) - Information technology - 7-bit coded character set for information interchange".

ISO/IEC 8859-1: "Information technology - 8-bit single-byte coded graphic character sets - Part 1: Latin alphabet No. 1", 1998.

Object Management Group (OMG): "The Common Object Request Broker: Architecture and Specification - IDL Syntax and Semantics". Version 2.6, FORMAL/01-12-01, December 2001.

History

Document history		
V1.1.1	July 2010	Publication
V1.2.1	April 2014	Membership Approval Procedure MV 20140614: 2014-04-15 to 2014-06-16
V1.2.1	June 2014	Publication