ETSI ES 202 786 V1.1.1 (2012-04)

Methods for Testing and Specification (MTS);
The Testing and Test Control Notation version 3;
TTCN-3 Language Extensions: Support of interfaces with continuous signals

Reference
DES/MTS-137 T3Ext_ContSig

Keywords
interface, testing, TTCN

ETSI

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

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

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

Contents

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://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 covering the Testing and Test Control Notation version 3, as identified below:

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

ES 201 873-2 [i.1]: "TTCN-3 Tabular presentation Format (TFT)";

ES 201 873-3 [i.2]: "TTCN-3 Graphical presentation Format (GFT)";

ES 201 873-4 [2]: "TTCN-3 Operational Semantics";

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

ES 201 873-6 [4]: "TTCN-3 Control Interface (TCI)";

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

ES 201 873-8 [i.4]: "The IDL to TTCN-3 Mapping";

ES 201 873-9 [i.5]: "Using XML schema with TTCN-3";

ES 201 873-10 [i.6]: "TTCN-3 Documentation Comment Specification".

1 Scope

The present document defines the "Continuous Signal 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 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.

This package defines concepts for testing systems using continuous signals as opposed to discrete messages and the characterization of the progression of such signals by use of **streams**. For both the production as well as the evaluation of continuous signals the concept of **mode** is introduced. Also, the signals can be processed as **history**-traces. Finally, basic mathematical functions that are useful for analyzing such traces are defined for TTCN-3. It is thus especially useful for testing systems which communicate with the physical world via sensors and actuators.

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 (V4.4.1): "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 (V4.4.1): "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 (V4.4.1): "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 201 873-2: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular presentation Format (TFT)".

[i.2] ETSI ES 201 873-3: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical presentation Format (GFT)".

[i.3] ETSI ES 201 873-7 (V4.4.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 7: Using ASN.1 with TTCN-3".

[i.4] ETSI ES 201 873-8 (V4.4.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 8: The IDL to TTCN-3 Mapping".

[i.5] ETSI ES 201 873-9 (V4.4.1): "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 9: Using XML schema with TTCN-3".

[i.6] ETSI ES 201 873-10 (V4.4.1): "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-3:2012 Support for Testing Continuous Signals"` - 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 (V4.4.1) [1]

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

ES 201 873-5 (V4.4.1) [3]

ES 201 873-6 (V4.4.1) [4]

ES 201 873-7 (V4.4.1) [i.3]

ES 201 873-8 (V4.4.1) [i.4]

ES 201 873-9 (V4.4.1) [i.5]

ES 201 873-10 (V4.4.1) [i.6]

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

5 Package concepts for the core language

Systems can communicate its data or signals, either in discrete form (e.g. as an integer value) or in continuous form (e.g. real values). With respect to this difference signals are classified into four categories. The categories distinguish whether the time and value domain of a signal is of discrete or continuous nature:

1) Analogue signals are continuous in the time and value domain. Analogue signals are the most 'natural' signal category, characterized by physical units (e.g. current, voltage, velocity) and measured with sensors. Typical examples of the physical quantities used in the area of embedded system development are the vehicle velocity, the field intensity of a radio station etc. Analogue signals can be described as a piecewise function over time (e.g. vx = f (t)).

2) Time quantified signals are discrete signals in the time domain. The signal values are defined only at predetermined time points (sampling points). Typical examples of time quantified signals are the time-value pairs of a recorded signal. A typical representation of a time quantified signal is a series or an array of real numbers. Even if the original signal is a synthetic function it can only be reconstructed from a time quantified signal with considerable mathematical effort.

3) Value quantified signals are time-continuous signals with discrete values. Typical examples of a value quantified signal are data that are derived from analogue signals and which are dedicated to further processing, e.g. an A/D converted sensor signal that is provided to an electrical control unit.

4) Digital signals are discrete on the time and value domain. If the set of possible signal values includes only two elements, one speaks about binary signals. Typical examples of binary signals are switching positions or flags.

Thus on a theoretical level, we distinguish between the continuous and discrete evolution of time and values. In a discrete system, the changes of states are processed at fixed and finite time steps. In a continuous system state changes occur for infinitesimally small time steps. Important mathematical models for continuous systems are ordinary differential equations. A mixed system, which shows continuous and discrete dynamics, is known as a hybrid system. Hybrid systems can be modelled with hybrid automatons. Examples for systems that show such variable dynamics are often found in the area of embedded control systems e.g. in the automotive and aircraft industry.

In the general case, a test description notation for embedded software systems shall support all of four categories of signals mentioned above. TTCN-3 currently supports the signal categories (2) and (4). The extension of the language with respect to a support of the signal categories (1) and (3) is the content of the present document.

TTCN-3 is a procedural testing language, thus test behaviour is defined by algorithms that typically send messages to ports and receive messages from ports. For the evaluation of different alternatives of expected messages, or timeout events, the port queues and the timeout queues are frozen when the evaluation starts. This kind of snapshot semantics guarantees a consistent view on the test system input during an individual evaluation step. Whereas the snapshot semantics provides means for a pseudo parallel evaluation of messages from several ports, there is no notion of simultaneous stimulation and time triggered evaluation. To enhance the core language to the requirements of continuous and hybrid behaviour we introduce:

- the notions of time and sampling;
- the notions of streams, stream ports and stream variables;
- the definition of an automaton alike control flow structure to support the specification of hybrid behaviour.

5.1 Time and Sampling

The TTCN-3 extensions defined in this package adopt the concept of a global clock and enhance it with the notion of sampling and sampled time. As in TTCN-3, all time values are denoted as float values and represent time in seconds. For sampling we intend to support simple equidistant sampling models as well as dynamic sampling models.

On technical level an equidistant sampling model of the form **t=k*bdelta,** where **t** describes the time progress, **d** specifies the number of executed sampling steps and, **bdelta** yields the minimal achievable step size for a given test system, is used as an overall basis to model equidistant samplings with larger step size or dynamic sampling.

The basic sampling with its minimal step size **bdelta** is a property of a concrete test system and not intended to be specified as part of the test case specification. However, as a consequence of this underlying model, a test system is able to execute user defined samplings if and only if all specified sampling rates at test specification level provide step sizes that are multiples of **bdelta**.

When using the TTCN-3 extension defined in this package, each reference to time, either used for the definition and evaluation of signals but as well by means of ordinary TTCN-3 timers, is considered to completely synchronized to the global clock and the base sampling.

5.1.1 The now operator

For the specification of time-dependent signal sequences, it is necessary to be able to track the passage of time. The access of time is guaranteed by a globally available clock whose current value can be accessed by means of the **now** operator. Time progress starts at the beginning of each test case execution, thus time values are related to the start of the test case execution.

Syntactical Structure

now

Semantic Description

Evaluation of the **now** operator yields the current value of the clock which is the duration of time since the start of the currently running test case.

Restrictions

The **now** operator shall only be applied from within a test case, i.e. by test cases, functions and altsteps executed on test components. The **now** operator shall neither directly nor indirectly be called by TTCN-3 control part.

Example

EXAMPLE:

```
// Use of now to retrieve the actual time since the test case has started
var float actualTime := now;
```

5.1.2 Define the default step size for sampling

For sampling, a globally valid base sampling rate defined by the test system is provided. In addition, sampling rates can be set separately and as part of the test specification by means of **stepsize** attribute.

Syntactic Structure

stepsize ***StepSizeValue***

Semantic Description

The ***StepSizeValue*** is a string-literal which must contain a decimal number. This number interpreted as seconds is used as the default rate of sampling values over the stream ports to which are affected by this **stepsize** attribute. The actual sampling rate of a specific port can be changed dynamically with the **delta** operation.

Restrictions

A **stepsize** attribute can only appear in a with-annotation. A **stepsize** attribute can be applied to individual modules, test cases, groups, component types and stream port types and effects either the statements that are contained in one of these entities or in case of component types and stream port types the respective instances.

Examples

EXAMPLE 1:

```
// sets the stepsize for a module
module myModule{
…
} with {stepsize " 0.0001" };
```

EXAMPLE 2:

```
// sets the stepsize for a testcase
testcase myTestcase() runs on myComponent{
…
} with {stepsize " 0.0001" };
```

EXAMPLE 3:

```
// sets the stepsize for all instances of the port type StreamOut
type port StreamOut stream { out float} with {stepsize " 0.0001" };
```

5.2 Data streams

In computer science the term data stream is used to describe a continuous or discrete sequence of data. Normally the length of a stream cannot be established in advance. The data rate, i.e. the number of samples per time unit, can vary. Data streams are continuously processed and are particularly suited to represent dynamically evolving variables over a course of time. Thus, streams are an ideal representation of the different discrete and continuous signals mentioned in the beginning of clause 5.

While in standard TTCN-3 interactions between the test components and the SUT are realized by sending and receiving messages through ports, the interaction between continuous systems can be represented by means of so called streams. In contrast to scalar values, a stream represents the whole allocation history applied to a port. In computer science, streams are widely used to describe finite or infinite data flows. To represent the relation to time, so called timed streams are used. Timed streams additionally provide timing information for each stream value and thus enable the traceability of timed behaviour. The TTCN-3 extension defined by this package provides timed streams. In the following we will use the term measurement (record) to denote the unity of a stream value and the related timing in timed streams. Thus, concerning the recording of continuous data, a measurement record represents an individual measurement, consisting of a stream value that represents the data side and timing information that represents the temporal perspective of such a measurement.

Standard TTCN-3 offers no direct support for the specification, management and modification of data streams. In this TTCN-3 extension, we introduce two different but not complementary representations of timed data streams. The term *timed* considers the fact that we are interested in the time and value domain of a signal. As a consequence we consider a stream to consist of a sequence of samples, which each provide information about the timing and the value perspective of the sample.

1) Static perspective: The static perspective provides a direct mapping between a timed stream and the TTCN-3 data structures **record** and **record of**. This kind of mapping is referred to below as the static representation of a data stream and allows random access to all elements of the data stream.

2) Dynamic perspective: To provide dynamic online access to data streams, we extend the existing concepts of TTCN-3 port type and port to provide access to data streams and their content. A so called **stream port** references exactly one data stream and provides access to the dynamically changing values of the referenced data stream.

Please note: to represent streams in the present document we use a tabular notation. The table has two rows by which the first one represents the value perspective of a stream and the second represents the temporal perspective. The temporal perspective is defined by means of timestamps that are synchronized with the overall clock. The columns represent the samples of the stream.

EXAMPLE:

value	1,2	1.4	1.5	1.7	1.7	1.5	1.2	1.0	1.1	1.4	1.5	1.2	1.0	1.1	1.4
timestamp	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4

The example shows a stream with the length of 1.4 seconds and float values that change between 1.0 and 1.5.

5.2.1 Data Streams: static perspective

A TTCN-3 data stream can be mapped directly to existing TTCN-3 data structures. The mapping considers each stream to be represented by means of a TTCN-3 **record of** data structure. This structure itself consists of individual entities, so called *samples*, each sample representing either a measurement on an incoming stream or stimulus that is dedicated to be applied to an outgoing stream.

A sample itself is represented by means of a TTCN-3 **record** data structure. The record consists of two fields. It has a generic **value** field that allows the instantiation of values for arbitrary predefined or user defined TTCN-3 data types. The value field represents what we call the value of a stream. Its data type is aligned with the data type of the corresponding stream.

The second field denotes the temporal perspective of a sample. It denotes the temporal distance to the preceding sample (the sampling step size **delta**). The second field is of type float and represents time values that have the physical unit second. Example 1 shows the exemplary definition of a data structure to specify individual samples.

EXAMPLE 1:

```
type record Sample<T>{
   T value,
   float delta
}
```

Given such a structure, a timed data stream of an arbitrary data type is modelled as a record of samples.

EXAMPLE 2:

```
type record of Sample<T> MyStreamType;
```

The static representation of data streams can be used for the online and offline evaluation of streams as well as for the piecewise in-memory definition of streams or stream templates, which are to be applied to stream ports in the subsequent test case execution. Thus, the static representation of streams can be used to assess incoming streams and to define outgoing or reference streams and template streams mostly by means of ordinary TTCN-3 operations and control structures and as such provide an ideal interface between ordinary TTCN-3 concepts and the concepts defined in this package. The following example shows a short specification of a sampled stream.

EXAMPLE 3:

```
var MyStreamType<float> myStreamVar := {
      {value:=0.0, delta:=0.1},
      {value:=0.2, delta:=0.2},
      {value:=0.1, delta:=0.1},
      {value:=0.0, delta:=0.3}
}
```

If the stream definition from above is applied to an outgoing stream port directly with the beginning of a test case, the result will look as follows.

EXAMPLE 4:

Value	0.0	0.0	0.2	0.1	0.0
Timestamp	0	0.1	0.3	0.4	0.7

Please note: each stream port is initialized with a value that defines the valuation of a stream at time 0.0. Thus the first sample in Example 4 is not defined by the specification in Example 3 but by the base initialization of the stream port. More information is provided in the following clauses.

NOTE 1: In order to create larger streams a manual specification approach is not feasible. In this case we propose to use the data processing capabilities of TTCN-3 to programmatically/algorithmically construct the dedicated record structures.

NOTE 2: The data structures presented in this section serve for illustration purposes only. They show how timed data streams can be mapped to standard TTCN-3 data structures and thus can be processed easily by using the existing TTCN-3 language features and operators. The TTCN-3 extensions provided in this package do not include type declarations from above.

5.2.2 Data Streams: dynamic perspective

In standard TTCN-3 ports are used for the communication among test components and between test components and the SUT. To be able to initiate, modify and evaluate a stream based communication between the entities of a test system, this package extends the concepts of standard TTCN-3 port types and ports with the notion of *stream-based communication* and *stream ports.* Stream ports are the endpoints of a stream based communication. Thus stream ports in TTCN-3 embedded are used to provide access to streams, their values and the respective timing information. A stream port references exactly one data stream and thus provides access to the respective stream values and timing information.

5.2.2.1 Defining stream port types

The TTCN-3 port concept of message-based and procedure-based ports is extended with stream-based ports. Stream ports support stream-based communication.

Syntactical Structure

```
type port PortTypeIdentifier stream "{"
   ( in | out | inout ) StreamValueType
"}"
```

Semantic Description

Stream port types shall be declared by using the keyword **`stream`**. Stream ports are directional. The directions are specified by the keywords **`in`** (for the in direction), **`out`** (for the out direction) and **`inout`** (for both directions).

The specified *StreamValueType* references the type of values which can be sent or received (depending on the direction of the port) over ports of the type *PortTypeIdentifier*.

Restrictions

Each stream port type definition shall have one and only one entry indicating the allowed type together with the allowed communication direction.

EXAMPLE:

```
// Stream-based port which allows stream values of type float to be received
type port StreamIn stream { in float }

//Stream-based port which allows stream values of type float to be sent
type port StreamOut stream { out float }
```

5.2.2.2 Declaration and instantiation of stream ports

The declaration of stream-based ports is similar to the declaration of message-based and procedure-based ports. The **component** type declares which ports are associated with a component. A component type can have ports with different communication characteristics (e.g. stream-based ports, message-based ports, and procedure based). All ports are instantiated together with the component that owns the port, i.e. when the component is created.

Outgoing stream ports start to emit stream values directly after the component, which contains the respective stream port, has been started. The same applies for incoming stream ports. They start receiving data directly after their component has been started. Both incoming and outgoing stream ports are updated for each sampling step. If no explicit step size is defined by means of step size annotations on module level, test case level, port type level etc. the port is initially sampled with the test systems' base sampling, which is the smallest available step size.

Outgoing stream ports may already be initialized before its first use, so that their values before the start of their component are defined. The initialization occurs in the context of their declaration.

Outgoing stream ports, when they are not explicitly initialized, are automatically initialized with implicit default values. The implicit default values for the various TTCN-3 basic data types can be found in the following table.

float	integer	boolean	charstring	bitstring	octetstring
0.0	0	**FALSE**	""	'0'B	'00'O

The initial stream port value for outgoing stream port applies to the time point 0.0 and for the following sample steps as long as no other stream value is set. The value initialization for incoming streams is in responsibility of the data provider. Hence either the system adapter or the emitting component (in case of a PTC) is responsible to initialize the streams.

Syntactical Structure

```
port StreamPortTypeReference

{ StreamPortIdentifier [ ":=" StreamDefaultValue ]  [","] }+ [";"]
```

Semantic Description

A stream port *PortInstance* named *StreamPortIdentifier* is declared inside a component type definition using a *StreamPortTypeReference* which is a type-reference expression for an existing stream port type. Optionally, a *StreamDefaultValue* can be supplied which defines the value of the stream before the first sampling over this port.

Restrictions

The *StreamDefaultValue* shall be of the type *StreamValueType* in the port type definition referenced by *StreamPortTypeReference*.

Examples

EXAMPLE 1:

```
type port StreamIn stream { in float }
type port StreamOut stream { out float }
```

```
type component SUT {
   port StreamIn A,B;
   port StreamOut C,D;
}
```

EXAMPLE 2:

```
type component SUT {
   port StreamIn A,B;
   port StreamOut C:=1.0,D:=2.0;
}
```

5.2.3 Data stream access operations

Similar to *message-based* and *procedure-based* communication incoming streams can be examined and outgoing streams can be controlled. In general, we provide access to the actual sample of a stream (i.e. the stream value, the respective timing and sampling information) by means of *stream data operations*. Moreover, we provide access to the preceding samples by means of dedicated *navigation operations*. Last but not least, we are able to extract record structured stream data as explained in clause 5.2.1 by means of *stream evaluation statements*.

In contrast to message-based and procedure-based communication, we integrate *stream data operations* and *stream navigation operations* on expression level. Thus, we are able to directly assign values to streams and read values from streams by means of ordinary TTCN-3 assignments.

5.2.3.1 The value operation

Each data stream connected to a stream port allows accessing its current value by means of the **`value`** operation. In case of incoming streams, the value operation yields the actual value that is available at a stream port.

Syntactical Structure

(***`StreamPortReference`*** | ***`StreamPortSampleReference`***) **`"." value`**

Semantic Description

The value operation can be applied to either a *StreamPortReference* expression or a *StreamPortSampleReference* expression which is yielded by the application of a navigation operation on a *StreamPortReference*. In the first case, it yields the current value of the stream port; in the second case it yields the value in the referenced sampling.

When using a StreamPortReference to an outgoing stream port, the value operation expression can also be used on the left hand side of an assignment or as an out parameter to a function.

When using a value operation expression as a value expression the type of the value is the StreamValueType of the referenced stream port.

Restrictions

If the value operation expression is used as the target of an assignment, the type of the assigned value shall be compatible with the StreamValueType of the referenced stream port.

Examples

EXAMPLE 1:

```
// accessing the actual input value of a stream
var float myVar:=streamInPort.value;
```

EXAMPLE 2:

```
// accessing the actual input value of a stream
// and compare it with a given expectation
if (streamInPort.value>= 100.0) {…};
```

NOTE 1: The value, which is provided by means of the **value** operation, is the value that has been measured at the beginning of the actual sampling period.

EXAMPLE 3:

```
// setting the actual output value of a stream
streamOutPort.value:= 100.0;
```

NOTE 2: The use of the **value** operation can be combined in such a way that the specification of complex equations and equation systems is supported.

EXAMPLE 4:

```
// calculating the Ohms' law
voltage.value:= amperage.value * resistance.value;
```

NOTE 3: The effectiveness of the stream port evaluation is delayed. A value, which has been assigned to a stream port value handle, becomes effective inside and outside the component at the beginning of the next sampling step.

5.2.3.2 The timestamp operation

Similar to the **value** operation the **timestamp** operation allows to access the time related information of the actual sample.

Syntactical Structure

(StreamPortReference | StreamPortSampleReference) **"." timestamp**

Semantic Description

The **timestamp** operation can be applied to a stream port referenced by a *StreamPortReference* expression or a *StreamPortSampleReference* referring to a specific sample of a stream port.

The application of the **timestamp** operation on a *StreamPortReference* yields the exact time point at which the actual stream port value has been measured. The application of the **timestamp** operation on a *StreamPortSampleReference* yields the exact time point at which the referenced sample has been measured. The exact sample time denotes the moment when a stream value has been made available at the test system's input and thus strongly dependent on the sampling rate.

The time point is provided as a floating-point number (**float**) and has the physical unit seconds. The time information is completely synchronized with the test system clock described in clause 5.1.

Restrictions

The **timestamp** operation always yields a non-negative **float** value.

Example

EXAMPLE:

```
// access of the sample time
// for the current sample
var float measurementTime1:=streamport.timestamp;
```

NOTE: Data streams are used to represent samples in a dynamic measurement process. A sample that is taken from a data stream is usually historical information, i.e. the result of a **timestamp** operation refer to the state of the system (i.e. the SUT) at a time in the past.

5.2.3.3 The delta operation

The step size of a data stream can dynamically change during a test execution. The change can be initiated either by the test specification or by means of the measurement system (i.e. the system adapter). The **delta** operation provides access to the actual step size of a port.

In addition to the timestamp operator TTCN-3 embedded allows to obtain the step size that has been used to measure a certain value. This information is provided by the **delta** operation. The delta operation can be used in a similar way than the value and the timestamp operation. It returns the size of the last sampling step (in seconds).

Syntactical Structure

(***StreamPortReference*** | ***StreamPortSampleReference***) **"." delta**

Semantic Description

When used on a *StreamPortReference*, the **delta** operation allows read and write access to the actual step size of a port. When the **delta** operation is used for reading on a *StreamPortReference*, it yields the actual step size for a given port. When the **delta** operation is used for writing on a *StreamPortReference* it sets the length of the step size for future writing and reading at the given port. The step size is defined as a **float** number and has the unit seconds.

When used on a *StreamPortSampleReference* it yields the actual step size active at the time of the referenced sample measurement.

NOTE: A value, which has been assigned to a stream port delta handle, affects the length of the next sampling period, not the actual one. Thus, it cannot be used to shorten or lengthen the actual sampling step.

Restrictions

When used on a *StreamPortSampleReference*, the **delta** operation only allows read access.

Examples

EXAMPLE 1:

```
  var float actualStepSize;
// reads the actual stream size from a port
actualStepSize: = streamport.delta;
```

EXAMPLE 2:

```
// sets the actual step size for a port
streamport.delta:= 0.001;
```

5.2.4 Data stream navigation operations

Beside access to the actual values of a stream, additional access to the history of streams by means of so called *stream navigation operations* is provided. The result of a navigation operation is a handle, which allows the application of the ***value***, ***timestamp*** or ***delta*** operation for preceding stream states. Such a state is identified by means of two different operations. The **at** operation demands a time index of type **float** that denotes the time that has passed since the beginning of the test case. The **prev** operation backtracks the sample steps beginning with the actual step and demands an **integer** index value to define the number of sampling steps to step back.

5.2.4.1 The prev operation

The **prev** operation returns a handle to obtain stream related information for previous states of a stream.

Syntactic Structure

StreamPortReference **"." prev ["(" *PrevIndex* ")"]**

Semantic Description

The **prev** operation can be applied to a stream port *StreamPortReference*. It can optionally be parameterized with an **integer** index parameter *PrevIndex* and returns a *StreamPortSampleReference* handle to retrieve values, timestamps and sampling step sizes for preceding stream states. The index parameter denotes the number of samples to step back in stream history. If no parameter list is given, this is equivalent with the index 1.

Restrictions

The **prev** operation can only appear as an operand to a **value**, **timestamp** or **delta** read operation.

NOTE 1: The application of the **`prev`** operation needs the combination with the **`value`** operation, the **`timestamp`** operation or the **`delta`** operation to provide meaningful results.

Examples

EXAMPLE 1:

```
port.prev(0).value; // provides access to the actual stream value
port.prev.value;    // provides access to the previous stream value
port.prev(1).value; // provides access to the previous stream value
port.prev(2).value; // provides access to the stream value 2 steps ago
```

NOTE 2: The expressions `port.prev` and `port.prev(1)` yield identical results.

EXAMPLE 2:

```
port.prev(0).timestamp; // provides access to the timestamp
                        // that denotes the beginning the actual sampling step
port.prev(0).delta;        // provides access to the length of the last sampling step
port.prev(1).timestamp; // provides access to the timestamp
                        // that denotes the beginning the preceding sampling step
port.prev(1).delta;     // provides access to the length of the sampling step 2 steps ago
```

EXAMPLE 3:

Value	1.2	1.4	1.5	1.7	1.7	1.5	1.2	1.0	1.1	1.4	1.5	1.2	1.0	1.1	1.4
Timestamp	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4

```
port.prev(0).value;     // yields 1.4
port.prev.value;        // yields 1.1
port.prev(1).value;     // yields 1.1
port.prev(2).value;     // yields 1.0

port.prev(0).timestamp; // yields 1.4
port.prev(0).delta;     // yields 0.1
port.prev(1).timestamp; // yields 1.3
port.prev(1).delta;     // yields 0.1
```

5.2.4.2 The at operation

The **at** operation returns a handle to obtain stream related information for previous states of a stream, which are identified by means of a timestamp value.

Syntactical Structure

`StreamPortReference` **`"."`** **`at [ "("`** ***`Timepoint`*** **`")" ]`**

Semantic Description

The **at** operation can be applied to a stream port *StreamPortReference.* The **at** operation can optionally be parameterized with a **float** parameter *Timepoint* and returns a *StreamPortSampleReference* handle to retrieve values, timestamps and sampling step sizes for preceding stream states. The *Timepoint* parameter represents a time stamp that identifies a sample at a certain place in time. The time stamp denotes the time that has passed since the start of the test case (see clause 5.1). It references the sample that has either the same time stamp or, if such a sample does not exist, the sample with the next smaller time stamp.

Restrictions

The **at** operation can only appear as an operand to a **value**, **timestamp** or **delta** read operation.

NOTE: The application of the **at** operation has to be done in combination with a **value** operation, a **timestamp** operation or a **delta** operation to provide meaningful results.

Examples

EXAMPLE 1:

```
port.at(now).value;  // provides access to the actual stream value
port.at(0).value;    // provides access to the initial stream value
                     // (i.e. the stream value at beginning of the test case)
port.at(10.0).value; // provides the stream value at the time point 10.0
                     // (i.e. 10. Seconds after the beginning of the test case)
```

EXAMPLE 2:

```
port.at(now).timstamp;    // provides access to the beginning of the actual sampling step
port.at(0).timstamp;      // provides access to the beginning of the initial sampling
                          // step (i.e. always 0.0)

port.at(10.0).timestamp;  // provides access to the beginning of the sampling step
                          // at time point 10.0
```

EXAMPLE 3:

Value	1.2	1.4	1.5	1.7	1.7	1.5	1.2	1.0	1.1	1.4	1.5	1.2	1.0	1.1	1.4
Timestamp	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4

```
port.at(now).value;        // yields 1.4
port.at(0).value;          // yields 1.2
port.at(1.0).value;        // yields 1.5
port.at(1.09).value;       // yields 1.5


port.at(now).timstamp;     // yields 1.4
port.at(0).timstamp;       // yields 0.0
port.at(1.09).timestamp;   // yields 1.0
```

5.2.5 Data stream extraction and application operations

Beside access to individual values of a stream, this package supports the extraction and application of stream segments that are represented by means of the **record of** data structure (data perspective) described in clause 5.2.1. The **history** operation allows to extract arbitrary stream segments. The **apply** operation is used to apply extracted or manually or programmatically defined stream segments to stream ports and the **find** and **violates** operations are used to identify stream segments at ports by means of stream template expressions.

5.2.5.1 The history operation

The **history** operation allows obtaining the complete or partial history of a stream as a TTCN-3 **record of** structure (see clause 5.1, data representation). The **history** operation has two parameters that denote the start time and end time of the desired stream segment.

Syntactical Structure

StreamPortReference **"." history "("** *StartTime* **","** *EndTime* **")"**

Semantic Description

The **history** operation provides a **record of** based sample representation of a stream. The operation has two parameters StartTime and EndTime that denote the start time and end time of the stream segment that is designated for export. The parameters are each of type **float** and represent the time that has passed since the beginning of the respective test case. Time values are given in units of seconds. The first parameter describes the measurement time of the first stream entry to be considered for history export. The second parameter denotes the time of the last record. If the specified start time value is greater than the specified end time value the **history** operation results in an empty **record of** structure.

Restrictions

The *EndTime* parameter shall not have a value greater than **now**.

Examples

EXAMPLE 1:

```
myStreamRec:= myPort.history(0.0, now);
```

EXAMPLE 2:

```
type record BoolSample {boolean v,float t}
type port BoolStreamType stream {in boolean}
type component MyStreamComponent {port myPort BoolStreamType}
var record of BoolSample myStreamRec;
...
myStreamRec:= myPort.history(0.0, now);
```

EXAMPLE 3:

Value	1.2	1.4	1.5	1.7	1.7	1.5	1.2	1.0	1.1	1.4	1.5	1.2	1.0	1.1	1.4
Timestamp	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4

```
myStreamRec:= port.history(0.0, now);
// yields
// {{1.2,0.0}, {1.4,0.1},{1.5,0.1},{1.7,0.1},{1.7,0.1},{1.5,0.1},{1.2,0.1},{1.0,0.1},
// {1.1,0.1},{1.4,0.1},{1.5,0.1},{1.2,0.1},{1.0,0.1},{1.1,0.1},{1.4,0.1}}
```

5.2.5.2 The values operation

The **values** function allows obtaining the complete or partial history of a stream as a TTCN-3 **record of** structure without any timing information.

Syntactical Structure

***StreamPortReference* "." values "(" *StartTime* "," *EndTime* ")"**

Semantic Description

The value operation has two parameters *StartTime* and *EndTime* that denote the start time and end time of the desired stream segment.

The **history** function provides a **record of** based value representation of a stream. The parameters are each of type float and represent the time that has passed since the beginning of the respective test case. Time values are given in units of seconds. The first parameter describes the measurement time of the first stream entry to be considered for history export. The second parameter denotes the time of the last record. If the specified start time value is greater than end time value the **history** operation results in an empty **record of** structure.

The result of the value operation applied to a stream port of type T is a value of **record of T**.

Restrictions

The *EndTime* parameter shall not have a value greater than **now**.

Examples

EXAMPLE 1:

```
myStreamRec:= port.values(0.0, now);
```

EXAMPLE 2:

```
type port BoolStreamType {in boolean}
type component{ port myPort BoolStreamType}
var record of boolean myStreamRec;
…
myStreamRec:= myPort.values(0.0, now);
```

EXAMPLE 3:

Value	1.2	1.4	1.5	1.7	1.7	1.5	1.2	1.0	1.1	1.4	1.5	1.2	1.0	1.1	1.4
Timestamp	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4

```
myStreamRec:= port.history(0.0, now);
// yields
// {1.2, 1.4, 1.5, 1.7, 1.7, 1.5, 1.2, 1.0, 1.1, 1.4, 1.5, 1.2, 1.0, 1.1, 1.4}
```

5.2.5.3 The apply operation

The apply operation is used to apply stream data to a stream port that are represented by means of a TTCN-3 **record of** structure. The apply operation applies the sample records contained in the **record of** data structure one after the other in time to the given port.

Syntactical Structure

StreamPortReference **"." apply "("** ***Samples*** **")"**

Semantic Description

Application of the **apply** operation to the stream port *StreamPortReference*, it will consecutively write the values of the given **record of** *Samples to the port, using the sampling deltas from Samples as deltas for writing the values, as well.*

The application of an apply operation **p.apply(v)** is equivalent to the following construction:

```
cont{
   var integer i := 0;
      p.value := v[i].value;
   if (i+1 < lengthof(v)) {
     p.delta := v[i+1].delta;
   }
}
until{
   [true]{if(lengthof(v)>count){i:= i+1; continue}}
}
```

Examples

EXAMPLE 1:

```
type port FloatIn {in float}
type port FloatOut {out float}

type component{ port myInPort FloatIn;
                port myOutPort FloatOut }

type record Sample {boolean value, float delta};
var record of Sample myStreamRec;
testcase myTestcase () runs on tester{
  // measure on all incoming ports for 100 seconds
  wait(100.0);
  // get the all sampless at myInport until now
  myStreamRec:= myInPort.history(0.0, now);
  // and apply the measured data to myOutPort.
  myOutPort.apply(myStreamRec); // lasts 100 seconds
}
```

EXAMPLE 2:

```
var MyStreamType<float> myStream := {
      {0.0, 0.1},
      {0.2, 0.2},
      {0.1, 0.1},
      {0.0, 0.3}
}
port.apply(myStream);
// yield -> see table below
```

Value	0.0	0.0	0.2	0.1	0.0
Timestamp	0	0.1	0.3	0.4	0.7

5.3 The assert statement

The **assert** statement is used as a short hand for the specification of expected system behaviour.

Syntactical Structure

assert "(" *Predicate* { "," *Predicate* } ")"

Semantic Description

The **assert** statement specifies one or a list of predicates that express the expectation on the SUT. A predicate consist of an arbitrary TTCN-3 boolean expression. If one of the predicates fail, the **assert** statement automatically sets the verdict to **fail**. The assert statement is allowed at any place in the TTCN-3 source code that allows the application of the **setverdict** statement. To assess continuous data it will be used in particular within the hybrid machine alike control flow structures described in clause 5.4.

NOTE: The semantics of the assert statement can be mapped to existing TTCN-3 statements in the following way:

```
assert (pred1, pred2,...,predn);
```

is fully equivalent to

```
if(! pred1) setverdict(fail);
if(! Pred2) setverdict(fail);
...
if(! predn) setverdict(fail);
```

Examples

EXAMPLE 1:

```
assert(a.value==4.0);
```

EXAMPLE 2:

```
assert(a.value==4.0, b.value ==5.0, d.value ==445.0);
```

5.4 Control structures for continuous and hybrid behaviour

This clause introduces control flow structures that allow the parallel and sampled application and assessment of stream values at ports. The concepts defined in clauses 5.1, 5.2 and 5.3 allow the construction, application and assessment of individual streams. For more advanced test behaviour, such as concurrent application and assessment of multiple streams and the detection of complex events (e.g. zero crossing or flag changes at multiple ports), we need stronger concepts. For this purpose, we combine the concepts defined in the last clauses with state-machine-like specification concepts, so called *modes*.

A *mode* expresses a certain runtime mode of a system or an SUT. This kind of runtime mode is characterized by a defined behaviour at ports and a set of predicates that limit the applicability of the behaviour. Unlike ordinary behavioural TTCN-3 statements, a mode applies its behaviour over time (at least for one sampling step).

5.4.1 Modes

The term *mode* is used to specify the discrete and countable macro states of a dynamic hybrid system. It mainly serves to distinguish the macro states of a hybrid system from the theoretically infinite number of micro-states. By means of modes, this package provides a layer of abstraction that helps distinguishing between the discrete changes of a hybrid system (or test system) that are relevant from the users (and testers) perspective and the discrete changes that are introduced by the underlying test execution environment in order to map continuous behaviour to a computational environment (which is naturally discrete). The interpretation and calculation of micro steps depend on the underlying technical environment, i.e. the sampling. Thus, a micro step is calculated by the combination of the active macro-states with the sampled evaluation of data at the stream ports.

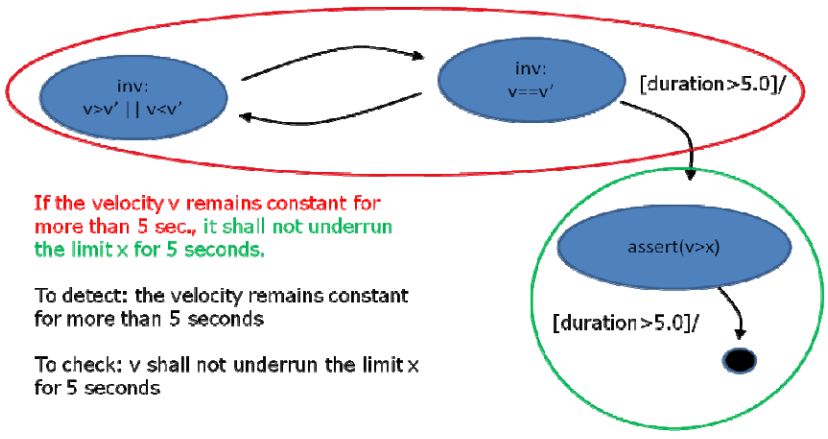

Figure 1: Abstract test specification for a continuous system that show the values v and x

Modes and the transitions between modes can be written down in a state-machine-like structure, which is closely defined in the theory of hybrid automatons. Figure 1 shows an abstract test specification that consists of three atomic modes, transitions, invariants and assertions.

For realizing such hybrid automatons, three new block statements are introduced, the **cont** statement, the **seq** statement and the **par** statement. While the **cont** statement is used for the specification of atomic modes, the **par** and **seq** statement are used to aggregate modes to larger constructs by means of parallel and sequential composition.

Modes in general are characterized by their duration and their internal behaviour (i.e. the assignment and assessment of values at stream ports). The duration, or better the duration of the mode's activity, is defined by a set of predicates, which relates to time or the valuation of (stream) ports, variables etc.

Syntactical Structure

```
( cont | par | seq ) "{"
   {Declaration}
   [OnEntryBlock]
   [InvariantBlock]
   Body
   [OnExitBlock]
"}"
 [UntilBlock]
```

A mode specification consists of several syntactical compartments:

- local declarations to be used inside the mode;
- an optional *onentry* block, that defines behaviour that has to be executed once at the activation of the mode;
- an optional *invariant* block that defines predicates that must not be hurt while the mode is active;
- an obligatory *body* to specify the mode's internal behaviour;
- an optional onexit block that defines the behaviour that has to be executed once at the deactivation of the mode;
- and an optional *transition* block (*UntilBlock*) that defines the exit conditions to end the mode's activity.

Atomic modes may be composed to composite modes. Composite modes show nearly the same structural setup as atomic modes. The only differences refer to their behavioural descriptions. While atomic modes contain assignments, assert statements and the *inv*, *onexit*, *onentry* blocks described above, composite modes contain other modes instead of statements. As far as invariants, *onentry* and *onexit* blocks and transitions are concerned, the structural setup and the behaviour of composite modes both are identical to atomic modes.

Semantic Description

While a mode is active, each invariant of a composite mode has to hold. Additionally, each transition of a composite mode ends the activity of the mode when it applies.

When a mode is entered, its onentry-block is executed. When a mode is exited, its onexit-block is executed.

For every step of an active mode, the contents (either modes or statements) of the mode are executed.

Examples

Example 1 shows the definition of an atomic mode consisting of two assignments to stream ports, an invariant that checks the state of an outgoing stream port, an *onentry* block that initializes the variable `x`, and an *onexit* block that resets the stream port `to_Set_Point` to the value of `0.0`, and transitions that check the valuation of an incoming stream port.

EXAMPLE 1:

```
cont{//body

   onentry{x:=10.0;}

   inv{//invariant
         to_Set_Point.value>20000.0;
   }

   to_Set_Point.value:=3.0*now;
   to_Engine_Perturbation.value:=0.0+x;

   onexit{to_Set_Point.value:=0.0}

}
until{//transition
  [ti_Engine_Speed.value>2000.0]{to_Engine_Perturbation.value:=2.0;}
  [ti_Engine_Speed.value>3000.0]{to_Engine_Perturbation.value:=1.0;}
}
```

Example 2 shows the setup of a parallel mode that contains two sequential modes, which each of them containing further atomic modes.

EXAMPLE 2:

```
par { // overall perturbation and assessment

   inv{//invariant

   }

   seq{// perturbation sequence

    cont{// stimulation action 1}

    cont{// stimulation action 2}

    ...

   }

   seq{// assessment sequence

    cont{// assessment action 1}

    cont{// assessment action 1}

    ...

   }

}

until{//transition
}
```

5.4.1.1 Definition of the until block

The ***until*** block allows the specification of exit conditions for modes and additionally the specification of explicit transitions between modes. The entries of the until block are called *transitions*. Each transition specifies conditions for their activation (i.e. guards and trigger events) and may provide an explicit definition of the mode that has to be activated next (target mode). An until block can contain several alternative transitions that each specify different exit conditions and target modes.

5.4.1.1.1 Definition of transition guards and events

The ***until*** block defines a number of transitions between modes. A transition contains either a guard or a trigger event specification or both. The guard and the trigger event specification are both used to determine whether a transition can fire or not. A guard is modelled as a boolean TTCN-3 expression. A trigger event is modelled by means of TTCN-3 receiving operations (***receive*** statement, ***trigger*** statement, ***getcall*** statement etc.). The predicate or the TTCN-3 receiving operations may be followed by an optional statement block, which contain instructions to be executed upon activation of the transition. At the end of the transition there may be a goto clause which specifies the follow-up mode.

Syntactical Structure

```
until "{"
  { "[" [Guard] "]" [TriggerEvent] [StatementBlock] [goto Target] }
"}"
```

Semantic Description

A transition is considered to be activated if the guard expression is satisfied and a valid receiving event occurred at the specified TTCN-3 receiving operation and the invariant of the target mode holds. Transitions are checked for each active mode at each sampling step. If a transition becomes active then the optional statement block is executed once. Afterwards the enclosing mode and all his child modes are deactivated. The control flow is continued with the activation of the follow-up mode. The transitions in an until block are checked in the given order. If multiple transitions exist, the first transition that fulfils the activation conditions is activated.

Restrictions

In the optional statement block of a transition any TTCN-3 statement is principally allowed, except:

- Blocking instructions (i.e. receive statements, alt-statements) shall not be used. Such constructs may block the execution of the statement block with the consequence, that the next sampling step is missed.
- Each type of control flow related statement that leads to the leaving of the enclosing mode (e.g. ***goto***, ***return***).

EXAMPLE:

```
cont{ //mode
A.value:=3;
}
until { // transitions
 [C.value > 4.0] MPort1.receive(TemplExp) { log(" statement block 1" ); }
 [C.value > 4.0 and D.value > E.value]{ log(" statement block 1" ); }
 [] Port2.receive(TemplExp) { log(" statement block 1" ); }
}
```

5.4.1.1.2 Definition of follow up modes

The explicit definition of follow up modes by means of a **goto** clause is possible. Each mode specification can have a preceding label that defines the target for a **goto** clause. Moreover each transition can have an optional ***goto*** clause that refers to an mode label.

Semantic Description

If a transition with a **goto** clause is activated, the optional statement block is executed and afterwards the execution is continued at the label position with the activation of the following mode.

Restrictions

Besides the restrictions that already exist for the use of the **goto** statement, this package defines additional restrictions for the use of the **goto** clause in the context of modes. Goto jumps are only allowed in a sequential environment, either inside **seq** modes or on the top level of a composition, i.e. directly on **testcase** level. Moreover, goto jumps are not allowed to violate the composition hierarchy, thus it is not possible to jump to a parent mode or into a child mode. Jumps are only allowed between modes on the same hierarchy level.

However, if no follow up mode is explicitly defined by means of a ***goto*** statement the sequential ordering of mode specification implicitly defines the follow up mode. Thus, when two atomic modes follow each other in the specification, the second mode is the follow up mode for all active transition transitions of the preceding mode that do not have an explicit **goto** clause.

Examples

Example 1 shows the application of labels and **goto** statements in the context of modes.

EXAMPLE 1:

```
label state1;
cont{ //mode
  A.value:=3;
}
until{[C.value > 2.0]}
label state2;
cont{ //mode
  A.value:=4;
} until { // transitions
 [C.value > 4.0] { log(" statement block 1" ); } goto state1
 [D.value > E.value]{ log(" statement block 2" ); } goto state2
 [] Port2.receive(TemplExp) { log(" statement block 1" ); }
}
```

EXAMPLE 2:

```
cont{ A.value:=3;} until {[B.value >3]}
cont{ A.value:=5;} until {[C.value >=3*D.value]}
cont{ A.value:=7;} until {[C.value >=3]}
```

5.4.1.1.3 The repeat statement

The control flow of a mode's transition's statement block may end in a **repeat** statement.

Semantic Description

The **repeat** statement causes the re-execution of a **par** statement, **seq** statement or **cont** statement, i.e. the execution of the **par** statement, **seq** statement or **cont** is activated again and executed with the next sampling step.

NOTE 1: In case of the execution of the **repeat** statement the local time of the respective mode (see **duration** symbol in clause 5.4.1.3) is reset, in case of composite modes the child modes are first deactivated and then again activated according to the kind (parallel or sequential) of the mode. Moreover, the respective **onentry** and **onexit** blocks are executed.

Example

EXAMPLE:

```
cont{ //mode
  A.value:=4;
} until { // transitions
```

```
 [C.value > 4.0] { log(" statement block 1" ); goto state1};
 [D.value > E.value]{ log(" repeat the execution" ); repeat};
 [] Port2.receive(TemplExp) { log(" statement block 1" ); }
}
```

NOTE 2: The **repeat** statement is functional equivalent to the use of a **goto** clause that addresses a label directly above the current mode.

5.4.1.1.4 The continue statement

The control flow of a mode's transition's statement block may end with a **continue** statement.

Semantic Description

The **continue** statement causes the further execution of a **par** statement, **seq** statement or **cont** statement, i.e. the execution of the **par** statement, **seq** statement or **cont** is continued with the next sampling step without a reset to the local time (see duration symbol in clause 5.4.1.3). The **onentry** and **onexit** blocks are not executed.

Example

```
cont{ //mode
  A.value:=4;
} until { // transitions
 [C.value > 4.0] { log(" statement block 1" ); goto state1};
 [D.value > E.value]{ log(" continue the execution" ); continue};
 [] Port2.receive(TemplExp) { log(" statement block 1" ); }
}
```

5.4.1.2 Definition of invariant blocks

Syntactical Structure

inv "{" *Predicate* {"," *Predicate*} "}"

Semantic Description

An invariant block contains **boolean** predicates (expressions) which characterize the applicability of a mode. Thus, an invariant block is always related to its containing mode specification and it specifies the conditions that shall be valid for a mode during runtime.

For each mode, all invariants are checked for each sampling step when the mode is active. While a mode is active the invariants of a mode shall not be violated. If an invariant of an active mode is violated, the mode must be able to switch to another mode that has valid invariants during the respective sampling step. If this is not possible, the test system must set an **error** verdict. The invariants block is always checked at the beginning of each sampling step, even before the body of each mode is executed.

Examples

Example 1 below shows the definition of an atomic mode that sets the out port A continuously with the value of 3.0. Moreover, the invariant prescribes conditions on the incoming ports B, C and D. When one of the invariants is violated by the actual value at ports, mode execution is stopped.

EXAMPLE 1:

```
type port StreamIn stream { in float }
type port StreamOut stream { out float }

type component SUT {
port StreamIn A,B;
port StreamOut C,D;
```

```
}

cont{
   A.value:=3;
   inv {B.value > 3,C.value >=3*D.value}
}
```

The specification of invariants allows the easy definition of ending conditions for the execution of modes. Based on a simple sequential control flow paradigm, this supports the specification of sequences of modes, that are executed one after the other whenever the invariant state of the active mode changes.

EXAMPLE 2:

```
cont{
   A.value:=3;
   inv {B.value >3,C.value >=3*D.value}
}
cont{
   A.value:=5;
   inv {B.value <=3,C.value >=3*D.value}
}
```

5.4.1.3 Definition of the onentry block

The **onentry** block contains a statement list that is to be executed once and only once during the activation of a mode.

Syntactical Structure

```
onentry StatementBlock
```

Semantic Description

The **onentry** block is executed as part of the activation procedure of a mode. To successfully start the **onentry** block all invariants must satisfy their conditions. The **onentry** blocks of hierarchically ordered modes are executed sequentially, beginning with the **onentry** block of the outer-most mode to the inner modes.

Restrictions

In an **onentry** block of a mode any TTCN-3 statement is principally allowed, except:

- Blocking instructions (i.e. receive statements, alt-statements) shall not be used. Such constructs would potentially block the execution of the statement block with the consequence, that the next sampling step is missed.
- Each type of control flow related statement that leads to the leaving of the mode (e.g. **goto**, **return**).

Example

The example below shows the definition of an atomic mode that sets the sampling of a port during its activation time.

```
cont{
   onentry {A.delta:=0.001;}
   A.value:=3;
}
```

5.4.1.4 Definition of the onexit block

The **onexit** block contains a statement list that is to be executed once and only once during the deactivation of a mode.

Syntactical Structure

```
onexit StatementBlock
```

Semantic Description

The **onexit** block is executed as part of the deactivation procedure of a mode. The execution of the **onexit** block is triggered either by an activated transition or the violation of an invariant that lead to the leaving of the mode. In case of an active transition the **onexit** block is executed directly after the execution of the transition's optional action block. The **onexit** blocks of hierarchically ordered modes are executed sequentially, beginning with the **onexit** blocks of the inner-most modes towards the outer modes.

Restrictions

In an **onexit** block of a mode any TTCN-3 statement is principally allowed, except:

- Blocking instructions (i.e. receive statements, alt-statements) shall not be used. Such constructs would potentially block the execution of the statement block with the consequence, that the next sampling step is missed.
- Each type of control flow related statement that leads to the leaving of the mode (e.g. **goto**, **return**).

Example

The following example shows the definition of an atomic mode that sets the sampling of a port during its deactivation time.

```
cont{
   A.value:=3.0;
   onexit {A.value:=1.0;}
} until {[B.value> 3.0]}
```

5.4.1.5 Local predicate symbols in the context of modes

To enable an explicit treatment of some exceptional situations, we introduce the keywords **notinv** and **finshed** that represent special predicates with a mode local evaluation.

Semantic Description

The keyword **notinv** can be used as a predicate that indicates the violation of any local mode invariant. Thus, if one of the invariants of a mode is violated and the mode is active, the evaluation of the **notinv** symbol yields true for all expressions in the contained until block. Otherwise it yields false. Thus, the **notinv** symbol allows the explicit handling of occurring invariant violation by means of transitions.

The **finished** keyword can be used as a predicate to handle the proper termination of a composite mode. A proper termination is given when the termination is triggered by the status of the child elements of a composite mode and not by its transitions or invariants. If and only if a mode is terminated by the status of its child elements the term finished yields true. Thus, the **finished** predicate allows the explicit handling of proper mode terminations by means of transitions.

Examples

EXAMPLE 1:

```
cont{ //mode
      A.value:=3;
}
until { // transitions
  [notinv] { log(" Invariant violated" ); }
  [] Port2.receive(TemplExp) { log(" Invariant not violated" ); }
}
```

EXAMPLE 2:

```
par { //mode
   cont { //inner mode 1
   A.value:=3.0;
   } until {[C.value>3.0]}
   cont { //inner mode 2
   B.value:=3.0;

   } until {[D.value>3.0]}
} until { // transitions
  [finished] { log(" finished by childs' state" ); }
  [D.value > 4.0] { log(" not finished by childs' state" ); }
}
```

5.4.1.6 The duration operator

Within a mode there is continuous access to the time that has elapsed since the beginning of the test case by using the **now** operator. It is also possible to access the time that has elapsed since the activation of the enclosing mode construct. The access is provided by means of the **duration operator**, which is applicable in expressions in all mode related substructures like the body block, the invariant block and the until block.

NOTE: The evaluation of the **duration** operator depends on its context. Thus, it may differ dependent on its place of application.

Examples

EXAMPLE 1:

```
   cont{ A.value:=3.0;} until {[now > 4.0]}
   // executes the content of the body block until
// the overall test case time has reached 4.0 seconds
```

EXAMPLE 2:

```
   cont{ A.value:=3.0;} until {[duration > 4.0]}
   // executes the content of the body block for 4.0 seconds
```

The following example shows the application of the **duration** operator in two different modes. Both modes are activated at different times and thus the application of the duration symbol in the second cont mode yields different results than the application of the **duration** operator in the enclosing **par** mode.

EXAMPLE 3:

```
par{
  cont{ A.value:=2.0;} until (duration > 4.0)
  cont{ A.value:=3.0;} until (duration > 4.0)
} until{[duration > 6.0]}
```

5.4.2 Atomic modes: the cont statement

Syntactical Structure

The syntactical structure and context for the **cont** statement is part of the syntactical structure provided in clause 5.4.1.

Semantic Description

The **cont** statement is used to define atomic modes. Atomic modes directly define the test behaviour at stream ports by means of value allocation and value assessments. A **cont** mode may contain assignments and assert statements and forms the leaves of a hierarchical mode structure.

When a **cont** statement is activated, all contained elements are executed repetitively for each sample step. The execution ends when a transition fires or an invariant is violated.

Restrictions

a) A **cont** mode shall not invoke any potentially blocking behaviour.

b) A **cont** mode cannot contain other modes.

Examples

EXAMPLE 1:

```
// executes the assignments at each sample step
cont { // Mode 1
   Port1.value := 10.0;
   Port2.value := 2.0 * duration;
}
until (duration > 5.0)
```

NOTE: Assignment and evaluation of the **cont** mode is, in a theoretical sense, continuous, i.e. executed at each step, provided for sampling. The **cont** mode allows the organization of periodic assignments and periodic revisions of values or variables of stream and stream ports.

EXAMPLE 2:

```
cont { // mode 1
   outport1.value := inport1.prev.value *2;
   streamvar.value := inport1.prev(5).value;
}
```

EXAMPLE 3:

```
cont { // mode 1
   outport1.value := inport1.prev.value *2;
   streamvar.value := inport1.prev(5).value;
   inv {
      streamvar.value > 200.0
   }
}
until { // Transition
   [streamvar.value >150] { streamvar.value =0; }
   [streamvar.value >180] {}
}
```

5.4.3 Parallel mode composition: the par statement

The parallel composition of modes is specified by means of the **par** statement. A parallel composition may contain sequential modes, parallel modes and atomic modes.

Syntactical Structure

The syntactical structure and context for the **cont** statement is part of the syntactical structure provided in clause 5.4.1.

The general structure of the **par** statement is similar to the **cont** statement and the **seq** statement. It consists of a body part, which defines the overall behaviour of the mode. In case of the **par** statement the body part contains the mode definitions that are to be composed in parallel. The mode can define an optional invariant and a transition part, as well as onentry and onexit blocks.

Semantic Description

In case of its activation, a parallel composition leads to a parallel execution of all composed (i.e. contained) modes.

While being active, each invariant of a composite mode has to hold. Additionally, each transition of a composite mode ends the activity of the mode when it fires. Furthermore, each mode provides access to an individual local clock that returns the time that has passed since the mode has been activated. The value of the local clock can be obtained by means of the duration keyword.

The activation of a parallel mode leads to the parallel activation of all child modes. During execution, the parallel mode is responsible to check the status of all contained modes. The execution of a parallel mode ends, either when a transition in the transition block has fired or when the execution of at least one child mode has been completed. The second situation is called a proper termination of a parallel mode and forces the local symbol finished to yield ***true*** (see clause 5.4.1.3).

Examples

EXAMPLE 1:

```
var integer count := 0;
par{
   cont{
      x.value:=1;
      y.value:=2;
   }
   until { // Transition
      [z.value> 3.0] { }
      [] Port2.receive {}
   }
   cont{
      x.value:=2;
      y.value:=1;
   }
   until { // Transition
      [z.value> 10.0] { }
      [] Port1.receive {}
   }
}
until { // Transition
[finished] {if(count > 1) {count++; continue}}
}
```

NOTE 1: The predicate **finished** yields **`true`** only during the distinct sample step when a child of a parallel mode has finished. Moreover, it yields true for every child element that has finished. Thus, it serves as a notification event, which can be used to model complex termination conditions for parallel modes.

NOTE 2: For parallel execution, it is always possible that several children modes terminate at the same time. Thus, counting the finished child modes to determine if all child modes have finished is not reliable. Instead, the child modes should set conditions that can be queried in the finished.

EXAMPLE 2:

```
par{
   cont{
      x.value:=1;
      y.value:=2;
   }
   until { // Transition
      [z.value> 1.0] { }
      [] Port1.receive(msg1) {}
   }
   cont{
      x.value:=2;
      y.value:=1;
   }
   until { // Transition
      [z.value> 10.0] { }
      [] Port1.receive() {}
   }
}
until { // Transition
      [z.value > 11.0] { }
      [] Port1.receive(msg) {}
}
```

5.4.4 Sequential mode composition: the seq statement

The sequential composition of modes is specified by means of the ***seq*** statement. A sequential composition may contain sequential modes, parallel modes and atomic modes.

Syntactical Structure

The syntactical structure and context for the **`cont`** statement is part of the syntactical structure provided in clause 5.4.1.

The general structure of the **`seq`** statement is similar to the cont statement and the par statement. It consists of a body part, which defines the overall behaviour of the mode. In case of the ***seq*** statement the body part contains the mode definitions that are to be composed.

Semantic Description

In case of its activation, a sequential composition leads to a sequential execution of the composed (i.e. contained) modes.

While being active, each invariant of a composite mode has to hold. Additionally, each transition of a composite mode ends the activity of the mode when it fires. Furthermore, each mode provides access to an individual local clock that returns the time that has passed since the mode has been activated. The value of the local clock can be obtained by means of the **duration** keyword.

The activation of a sequential mode leads to the activation of its first child mode. During execution, the sequential mode is responsible to schedule the contained modes in their sequential order. Thus, when a child mode has finished, the target mode of the exit transition is activated. Per default, the target mode is the next mode is the sequence\. The execution of a sequential mode ends either when a transition in the transition block is fired or when the execution of the last child mode has been completed. The second situation is called a proper termination of a sequential mode and forces the local symbol ***finished*** to yield ***true*** (see clause 5.4.1.3).

Example

The following example defines the sequential execution of two atomic modes, which are composed sequentially by means of a sequential mode.

EXAMPLE:

```
seq{
   cont{
      x.value:=1;
      y.value:=2;
   }
   until { // Transition
      [z.value> 2.0] { }
      [] Port1.receive() {}
   }
   cont{
      x.value:=2;
      y.value:=1;
   }
   until { // Transition
      [z.value> 1.0] { }
      [] Port1.receive() {}
   }
}
until { // Transition
      [z.value> 12.0] { }
      [] Port1.receive(msg) {}
}
```

5.4.5 Parameterizable modes

To provide a higher degree of flexibility, it is possible to specify parameterizable modes. Values, templates, ports, and modes can be used as mode parameters. The definition of parameterizable modes is similar to the definition of TTCN-3 functions.

NOTE: Unlike functions parameterizable modes are not called in the sense of a function call but inserted by means of a substitution mechanism at compile time. Thus, the recursive application of parameterizable modes is not possible.

5.4.5.1 Mode types

Mode types are the set of identifiers of mode definitions with a specific parameter list and runs on clauses. They denote those modes defined in the test suite that have a compatible parameter list and compatible **`runs on`** clauses.

Syntactical Structure

```
type mode BehaviorTypeIdentifier
"(" { ( FormalValuePar | FormalTimerPar | FormalTemplatePar | FormalPortPar | FormalModePar )
[","] } ")"
[ runs on ( ComponentType | self ]
```

Example

```
type mode ModeType assert_mode() runs on Tester;
```

5.4.5.2 Parameterizable mode definitions

A parameterizable mode definition allows the definition of reusable and parameterizable modes. A parametrizable mode may be defined within a module.

Syntactical Structure

mode *ModeName*

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

[**runs on** *ComponentType*]

(*ContMode* | *ParMode* | *SeqMode*)

Semantic Description

In a module, the behaviour of a mode can be defined by using the statements and operations described in clauses 5.4.1 to 5.4.4.

Restrictions

a) If a mode uses variables, constants, timers and ports that are declared in a component type definition, the component type shall be referenced using the **`runs on`** keywords in the mode header. The one exception to this rule is if all component-wide information used within the mode is passed in as parameters.

b) A mode without **`runs on`** clause shall never invoke functions or modes with a **`runs on`** clause locally.

Examples

EXAMPLE 1:

```
mode myMode runs on Tester
cont{assert(engine_speed >= 500.0)}
```

EXAMPLE 2:

```
type mode assert_mode() runs on Tester;

mode assertion1()  runs on Tester cont{assert(engine_speed >= 500.0)}
mode assertion2() runs on Tester cont{assert(engine_speed >= 0.0)}

mode pert_seq_2(in float startVal,
                in float increase,
                in assert_mode assertion)
runs on Tester
par{
  seq{// perturbation sequence
    cont{to_Set_Point:=startVal} until {[duration>=2.0]}
    cont{to_Set_Point:=startVal + duration/to_Set_Point.delta*increase}
    until {[duration>=5.0]}
  }
  assertion();
}

testcase myTestcase runs on EngineTester {
  // reusable mode application
  pert_seq_2(1000.0, 10.0, assertion1);
  pert_seq_2(5000.0, 1.0, assertion2);
```

```
}
```

5.5 The wait statement

Syntactical Structure

wait "(" *Expression* ")"

Semantic Description

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 the **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:

```
streamoutport.value = 10.0;
wait(100.0 + now);              // suspends the execution of a component
                                // until 100.0 seconds after the start of the testcase
streamoutport.value = 12.0;
```

NOTE: The **wait** statement has no impact on sampling. All stream ports of the given component are still sampled with respect to their sampling rate.

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(int lont 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 triNextSampling (TE → PA, SA → PA)

Signature	*TriStatus triNextSampling(in long timepoint,* *in TriPortIDType port)*
In Parameters	*timepoint* point in time when the execution of the next sample step for a given stream *port* shall be started *port* the stream port the sample step is requested for
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(int lont ticksPerSecond).*
Effect	The operation signals that the next sample step for a given port shall start at the specified point of time *timepoint*. At this point in time the PA will issue a *triProcessStep(in TriPortIDListType ports)* operation to inform the TE which ports shall be sampled next. The paremeter *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 *triProcessStep* operation. 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 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(int lont 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.

6.5 triProcessStep (PA → TE)

Signature	*void triProcessStep(in TriPortIDListType ports)*
In Parameters	*Ports* a list of ports that shall be sampled at the operation call
Out Parameters	n.a.
Return Value	n.a.
Constraints	There was a preceding invocation of *triNextSampling(timepoint, port*
Effect	The operation signals that the point in time *timepoin*t that was specified in the corresponding *triNextSampling(timepoint, port)* has been reached.

6.6 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 *timepoin*t that was specified in the corresponding *triBeginWait(timepoint, component)* has been reached.

6.7 triSetStreamValue (TE → SA)

Signature	*TriStatusType triSetStreamValue(* *in TriComponentIdType componentId,* *in TriPortIdType tsiPortId,* *in TriAddressType SUTaddress,* *in TriMessageType streamValue)*
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 Adapter `SUTaddress` (optional) destination address within the SUT `streamValue` the encoded stream value (message) to be sent
Out Parameters	n.a.
Return Value	The return status of the `triSetStreamValue` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.
Constraints	The TE calls this operation when it executes a new sampling step on a sampled output stream port, which has been mapped to a TSI port. The TE calls the operation for all sampling steps of all outgoing stream ports 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 `streamValue` has to be done in the TE prior to this TRI operation call.
Effect	The SA can update the message to the SUT. The `triSetStreamValue` operation returns ***TRI_OK*** in case it has been completed successfully. Otherwise ***TRI_Error*** shall be returned. Notice that the return value ***TRI_OK*** does not imply that the SUT has received `streamValue`.

6.8 triGetStreamValue (TE → SA)

Signature	*TriStatusType triGetStreamValue(* *in TriComponentIdType componentId,* *in TriPortIdType tsiPortId,* *in TriAddressType SUTaddress,* *out TriMessageType streamValue)*
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 Adapter `SUTaddress` (optional) destination address within the SUT
Out Parameters	`streamValue` the encoded stream value (message) that has been received from the SUT.
Return Value	The return status of the `triGetStreamValue` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.
Constraints	The TE calls this operation when it executes a new sampling step on a sampled input stream port, which has been mapped to a TSI port. The TE calls the operation for all sampling steps of all outgoing stream ports 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 decoding of `streamValue` has to be done in the TE after to this TRI operation call.
Effect	The SA can update the stream value at the input port. The `triGetStreamValue` operation returns ***TRI_OK*** in case it has been completed successfully. Otherwise ***TRI_Error*** shall be returned.

7 TCI extensions for the package

7.1 tciSetStreamValueReq (TE → CH)

Signature	*TriStatusType tciSetStreamValueReq(in TriPortIdType sender,* *in TriComponentIdType receiver,* *in Value streamValue)*
In Parameters	`sender` identifier of the port via which the message is sent to the receiving component. `receiver` identifier of the receiving component `SUTaddress (optional) destination address within the SUT` `streamValue` the encoded stream value (message) to be set
Out Parameters	n.a.
Return Value	The return status of the `tciSetStreamValueReq` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.
Constraints	The TE calls this operation at the CH when it executes a new sampling step on a sampled output stream port, which has been connected with a test component port.
Effect	The CH can call tciSetStreamValue in the TE on the node where the receiver component is deployed. The `tciSetStreamValueReq` operation returns ***TRI_OK*** in case it has been completed successfully. Otherwise ***TRI_Error*** shall be returned. Notice that the return value ***TRI_OK*** does not imply that the connected component has received `streamValue`.

7.2 tciSetStreamValue (CH → TE)

Signature	*TriStatusType tciSetStreamValue(* *in TriPortIdType sender, in TriComponentIdType receiver* *in Value streamValue)*
In Parameters	`sender` identifier of the port from which the message is sampled. `receiver` identifier of the receiving component `streamValue` the stream value that has been sampled
Out Parameters	n.a.
Return Value	The return status of the `tciSetStreamValue` operation. The return status indicates the local success (***TRI_OK***) or failure (***TRI_Error***) of the operation.
Constraints	The CH calls this operation in the local TE where receiver is deployed when tciSetStreamValueReq has been called.
Effect	The CH can update the respective stream input port on the test component. The `tciGetStreamValue` operation returns ***TRI_OK*** in case it has been completed successfully. Otherwise ***TRI_Error*** shall be returned.

Annex A (normative): BNF and static semantics

A.1 Changed BNF Rules

```
1.OpCall ::= ConfigurationOps |
             GetLocalVerdict |
             TimerOps |
             TestcaseInstance |
             (FunctionInstance [ExtendedFieldReference]) |
             (TemplateOps [ExtendedFieldReference]) |
             ActivateOp |
             NowOp |
             StreamDataOps |
             StreamNavigationOps |
             ModeLocalOps
2.AttribKeyword ::= EncodeKeyword |
                    VariantKeyword |
                    DisplayKeyword |
                    ExtensionKeyword |
                    OptionalKeyword |
                    StepsizeKeyword
3.PortDefAttribs ::= MessageAttribs |
                     ProcedureAttribs |
                     MixedAttribs |
                     StreamAttribs
4.PortElement ::= Identifier [ArrayDef] [AssignmentChar PortInitialValue]
5.CommunicationStatements ::= SendStatement |
                              CallStatement |
                              ReplyStatement |
                              RaiseStatement |
                              ReceiveStatement |
                              TriggerStatement |
                              GetCallStatement |
                              GetReplyStatement |
                              CatchStatement |
                              CheckStatement |
                              ClearStatement |
                              StartStatement |
                              StopStatement |
                              HaltStatement |
                              CheckStateStatement |
                              StreamEvalStatements

6.FunctionStatement ::= ConfigurationStatements |
                        TimerStatements |
                        CommunicationStatements |
                        BasicStatements |
                        BehaviourStatements |
                        SetLocalVerdict |
                        SUTStatements |
                        TestcaseOperation |
                        AssertStatement |
                        WaitStatement
```

A.2 New BNF Rules

```
7.NowOp ::= " now"
8.StepsizeKeyword ::= " stepsize"
9.StreamAttribs ::= StreamKeyword " {"  StreamDirection Type " }"
10.StreamKeyword ::= " stream"
11.StreamDirection ::= InParKeyword | OutParKeyword
12.PortInitialValue ::= Expression

13.StreamDataOps ::= Port Dot PortDataOp
14.PortDataOp ::= PortValueOp |
                  PortTimestampOp |
                  PortDeltaOp |
                  PortHistoryOp |
                  PortValuesOp
15.PortValueOp ::= " value"
16.PortTimestampOp ::= " timestamp"
17.PortDeltaOp ::= " delta"
18.PortHistoryOp ::= HistoryOpKeyword [ " ("  StartValue [, EndValue] " )"  ]
19.HistoryOpKeyword ::= " history"
20.StartValue ::= Expression
21.EndValue ::= Expression
22.PortValuesOp ::= ValuesOpKeyword [ " ("  StartValue [, EndValue] " )"  ]
23.ValuesOpKeyword ::= " values"
24.StreamNavigationOps ::= Port Dot ( PortPrevOp | PortAtOp ) [ Dot PortDataOp ]
25.PortPrevOp ::= PrevOpKeyword [ " ("  IndexValue " )"  ]
26.PrevOpKeyword ::= " prev"
27.IndexValue ::= Expression
28.PortAtOp ::= AtOpKeyword [ " ("  TimeIndexValue " )"  ]
29.AtOpKeyword ::= " at"
30.TimeIndexValue ::= Expression
31.ModeLocalOps ::= DurationOp | FinishedOp | NotinvOp
32.DurationOp ::= " duration"
33.FinishedOp ::= " finished"
34.NotinvOp ::= " notinv"
35.StreamEvalStatements ::= Port Dot (PortApplyOp | PortFindOp | PortViolatesOp)
36.PortApplyOp ::= ApplyKeyword [ " ("  ApplyParameter " )"  ]
37.ApplyKeyword ::= " apply"
38.ApplyParameter ::= TemplateInstance
39.PortFindOp ::= FindKeyword [ " ("  FindParameter " )"  ]
40.FindeKeyword ::= " find"
41.FindParameter ::= TemplateInstance
42.PortViolatesOp ::= ViolatesKeyword [ " ("  ViolatesParameter " )"  ]
43.ViolatesKeyword ::= " violates"
44.ViolatesParameter ::= TemplateInstance
45.AssertStatement ::= AssertKeyword [ " ("  AssertionList " )"  ]
46.AssertKeyword ::= " Assert"
47.AssertionList ::= Expression {, Expression }
48.WaitStatement ::=  WaitKeyword " ("  WaitDuration " )"
49.WaitKeyword ::= " wait"
50.WaitDuration ::= Expression


51.ModeSpecification ::= ( BasicMode | ComplexMode ) [ UntilBlock ]

52.BasicMode ::= ContKeyword " {"  {VarInstance}
                                   [OnEntryBlock]
                                   [InvariantBlock]
                                   {BasicModeOp}
                                   [OnExitBlock]
                                " }"

53.ContKeyword ::= " cont"
54.OnEntryBlock ::= OnEntryKeyword " {"  StatementBlock " }"
55.OnEntryKeyword ::= " onentry"
56.InvariantBlock ::= InvKeyword " {"  InvariantList " }"
57.InvKeyword ::= " inv"
58.InvariantList ::= [BooleanExpression {" ,"  BooleanExpression }]
59.BasicModeOp ::= Assignment | AssertStatement
60.OnExitBlock ::= OnExitKeyword "{" StatementBlock "}"
61.OnExitKeyword ::= " onexit"
```

```
62.ComplexMode ::= (SeqKeyword | ParKeyword) " {"  {VarInstance}
                                  [OnEntryBlock]
                                  [InvariantBlock]
                                  ModeList
                                  [OnExitBlock]
                               " }"
63.SeqKeyword ::= " seq"
64.ParKeyword ::= " par"
65.ModeList ::= { [LabelStatement] ModeSpecification }

66.UntilBlock ::= UntilKeyword " {"  UntilGuardList " }"
67.UntilKeyword ::= " until"

68.UntilGuardList ::= {UntilGuardStatement}
69.UntilGuardStatement ::= " ["  [BooleanExpression] " ]"  GuardOp StatementBlock [GotoStatement]


70.ModeTypeDef ::= TypeKeyword ModeKeyword Identifier
                      [ " ("  TemplateOrValueFormalParList " )"  ]
                      [RunsOnSpec]
71.ModeKeyword ::= " mode"

72.ModeInstance ::= ModeKeyword Identifier
                      [ " ("  TemplateOrValueFormalParList " )"  ]
                      [RunsOnSpec]
                     ModeSpecification
```

Annex B (informative): Bibliography

ALUR, Rajeev; COURCOUBETIS, Costas; HENZINGER, Thomas A.; HO, Pei-Hsin: "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems". In: Hybrid Systems, 1992, S. 209-229.

ALUR, Rajeev (Hrsg.); HENZINGER, Thomas A. (Hrsg.); SONTAG, Eduardo D. (Hrsg.): Hybrid Systems III: "Verification and Control, Proceedings of the DIMACS/SYCONWorkshop", October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA. Bd. 1066. Springer, 1996 (Lecture Notes in Computer Science). - ISBN 3-540-61155-X.

BROY, Manfred: "Refinement of Time". In: BERTRAN, M. (Hrsg.); RUS, Th. (Hrsg.): Transformation-Based Reactive System Development, ARTS'97, TCS, 44 - 63.

CONRAD, M.: "Modell-basierter Test eingebetteter Software im Automobil": Auswahl und Beschreibung von Testszenarien. Dissertation, Deutscher Universitätsverlag, Wiesbaden (D), 2004.

CONRAD, M.; SAX, E.: "Mixed Signals". In: E. Broekman, E. Notenboom: "Testing Embedded Software". Addison-Wesley, London (GB), 2003, S. 229-249.

DIN 40146 "Begriffe der Nachrichtenübertragung".

History

Document history		
V1.1.0	February 2012	Membership Approval Procedure MV 20120415: 2012-02-15 to 2012-04-16
V1.1.1	April 2012	Publication