Int J Softw Tools Technol Transfer (2008) 10:383–400
DOI 10.1007/s10009-008-0081-2

Testing hybrid control systems with TTCN-3: an overview on continuous TTCN-3

Ina Schieferdecker · Juergen Grossmann

Published online: 30 April 2008

Abstract TTCN-3 has gained increasing significance in recent years. It was originally developed to fit the needs of testing software-based applications and systems in the telecommunication industry and has shown its applicability to a wide range of other industrial domains in the mean time. TTCN-3 provides platform-independent, universal and powerful concepts to describe tests, especially for discrete, interactive systems. However, TTCN-3 addresses systems with discrete input and output characteristics only. The lack of powerful means that reasonably allow specifying and evaluating continuous data flow makes TTCN-3 sufficient neither for the automotive industry nor for other industries that deal with highly complex software-based control systems. This paper introduces the notion of streams, stream ports and stream templates to TTCN-3. It revises the initial design of continuous TTCN-3, a TTCN-3 extension for testing continuous or hybrid systems [20,21] and demonstrates the applicability for a case study that is typical for testing embedded control systems in the automotive industry.

1 Introduction

Embedded systems play an ever increasing role in the realization of complex control functions in many industrial domains—resulting in a big variety of requirements with respect to their functionality and reliability. In particular, software-based control systems have specific characteristics, which—at least in their combination—are unique. The systems are typically embedded, interact with the environment using sensors and actuators, supervise discrete control flows, obtain and process simple and complex structured data, communicate over different bus systems and have to meet high safety and real-time requirements. While different, model-based development processes and methods for embedded systems exist, a generally recognized test technology for the analysis and evaluation of these systems—which lead to high-quality, safe and reliable systems—is missing. Such a test technology has to address the different aspects of embedded systems and it has to enable the testing of discrete behaviors for the communication sequences, continuous behaviors for the control sequences, and hybrid behaviors for a combination of both (i.e. control sequences in interaction with sensors/actuators, with other system components and the user).

TTCN-3 (the Testing and Test Control Notation [7]) provides a standardized test environment that was originally tailored to satisfy the requirements of testing systems in the telecommunication industry. Although embedded systems (e.g. in mobiles) are already covered by TTCN-3, they are not its main target. Hence, dedicated support for real-time, continuous and hybrid behavior is lacking. However, to use the full potential of TTCN-3 also to test hybrid and continuous systems, it needs to be extended.

This paper presents concepts especially dedicated to the testing of embedded systems in the automotive domain. It begins with the overall definition of requirements for an integrated test environment that satisfies the needs of the automotive domain in Sect. 2. Section 3 introduces the concepts stream, stream template and stream port, provides a computational implementation that is based on TTCN-3 and describes the integration with the already existing language

I. Schieferdecker · J. Grossmann (✉)
Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31,
10623 Berlin, Germany
e-mail: juergen.grossmann@fokus.fraunhofer.de

I. Schieferdecker
Technical University Berlin, Franklinstr. 28/29,
10589 Berlin, Germany

constructs. Section 4 presents an example application as a proof of concepts. A summary concludes the paper.

2 Testing automotive control systems

Similar to other domains relating to real-time and safety-critical systems, test processes in the automotive industry are tool-intensive and affected by technologically heterogeneous test infrastructures. Moreover the whole development process is highly distributed and fragmented. The original equipment manufacturer (OEM), i.e. the system integrator and solution provider, is responsible for the specification and integration at system level whereas software and hardware of the individual electronic control units (ECUs[1]) are normally provided by different suppliers. In recent years, the application of model-based specifications in the development and the establishment of powerful code generators have led the development process to be noticeably more effective, automated, and reaching a higher level of abstraction. Due to the availability of executable models, tests and analytical methods can be applied early and integrated into subsequent development steps. The positive effects—early error detection and early bug fixing—are obvious.

Nevertheless, model-based approaches have to be integrated into existing development processes and combined with existing methods and tools. Hence, in the industrial practice, an embedded control system has to pass several kind of tests on different levels. Tests that go along with the integration of the complete vehicle system are mainly the responsibility of the OEM. These tests address the interaction between control units, the vehicle communication infrastructure and last but not least, tests of the complete vehicle system. Tests on ECU level are mainly in the responsibility of the respective suppliers. They encompass the verification of the, in most of the cases software driven, functionality and the electronic characteristics of the ECU.

To support the test activities a wide range of simulation environments are used for test execution and to provide the necessary test environment. For tests on the model level, model-in-the-loop (MIL) environments are used. To test the software itself, software-in-the-loop (SIL) and processor-in-the-loop environments are introduced.[2] The integration between software and hardware (i.e. the complete ECU) is tested using hardware-in-the-loop (HIL) environments [13]. Besides software related tests (functionality, software integrity, and software robustness), HIL environments allow to test the electronic characteristics and may simulate a complete network of interacting ECUs. Finally, the OEM uses HIL environments to test and simulate the complete electronic infrastructure of a vehicle.

Normally, different test systems are applied for the different simulation environments and almost each test system has individual requirements for methods, languages and concepts. The established test tools from National Instruments [17], dSPACE [5], Etas [6], Vector [24], MBtech [16] for example are highly specialized. They rely on proprietary languages and technologies and are mostly closed in respect to portability, extension and integration. Efforts to address test exchange especially for the automotive domain already exists but could not solve the problem in a general manner [10,11,19]. Attempts to standardize and harmonize the existing languages and test environments are under discussion; however, they remain in the early stages.

To keep the whole development and test process efficient and manageable, the definition of an integrated and seamless approach is required. Such an approach especially would address the subjects of test exchange, autonomy of infrastructure, methods, and platforms and the reuse of tests. The respective technological basis would be constituted by a domain-specific test language, that is executable and that unifies tests of communicating, software-based systems in all of the automotive sub-domains (telematics, power train, body electronics, etc.). It would unify the test infrastructure as well as the definition and documentation of tests.

TTCN-3 has the potential to serve as such a testing middleware. It provides concepts for local and distributed, for platform- and technology-independent testing. A TTCN-3-based test solution can be adapted to concrete testing environments and to concrete systems to be tested by means of an open test execution environment with well-defined interfaces for adaptations. However, control systems in the automotive domain can be characterized as hybrid systems, which encompass discrete and continuous behavior (in time). Discrete signals are used for communication and coordination between system components. Continuous signals are used for monitoring and controlling the components and the system environment via sensors and actuators. An adequate test technology for control systems needs to be able to control, observe and analyze the timed, functional behavior of these systems. This is characterized by a set of discrete and continuous input and output signals and their relations between each other. While the testing of discrete controls is well understood and available in TTCN-3 [7], concepts for specification-based testing of continuous controls and for the relation between discrete and the continuous system parts are not. TTCN-3 especially lacks concepts for specifying tests for continuous and hybrid behavior.

[1] In the following, the term electronic control unit (ECU) is used to refer to systems that are realized as a single piece of hardware. This contrasts with the already introduced term control system that also refers to compound systems that is to systems that consist of multiple control units.

[2] The former allows the compiled software and the environment to run on standardized personal computers, the latter simulates the complete target platform to allow tests that address special target platform issues.

3 Continuous TTCN-3

TTCN-3 is a procedural testing language: test behavior is defined by algorithms that typically assign (send) messages to ports and evaluate messages (receive) from ports.[3] The evaluation of messages is done along the TTCN-3 snapshot semantics [7,12]: for the evaluation of different alternatives of expected message (sets) or timeouts, the port queues and the timeout queues are frozen. Whereas the snapshot semantics provide means for a pseudo-parallel evaluation of messages from several ports, there is no notion of simultaneous stimulation and sampled evaluation. To enhance the core language to the requirements of continuous and hybrid behavior we introduce:

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

We carefully avoided to change any of the established TTCN-3 concepts and constructs and augmented the language deliberately to support hybrid behaviors and the evaluation of timed signals. Parts of our approach were already presented in previous articles [20,21].

In the following, syntactical add-ons and newly introduced constructs are denoted using Extended Backus-Naur-Form[4] (EBNF) and exemplified by short listings. The EBNF listings are not meant as a formal integration in the existing TTCN-3 grammars but should encourage a better understanding of the introduced constructs.

We start with the introduction of time related concepts in Sect. 3.1. In Sect. 3.2 we propose new constructs for continuous signals in TTCN-3 (e.g. streams, stream types stream ports, etc.). Afterwards, we introduce a new control flow structure for the simultaneous stimulation of several ports (Sect. 3.5). Finally, we discuss the combined usage of the newly introduced concepts with standard TTCN-3 constructs.

3.1 The time concept

The notion of time is crucial for testing continuous or hybrid systems. In standard TTCN-3, there is restricted time control using timer operations only. Especially for the exact timing of stimulation events and the time quantified evaluation of system reactions, the use of timers is cumbersome and imprecise. Hence, in *Timed* TTCN-3 global time[5] and the notion of timestamps were introduced [4,18]. Access to global time is provided by an operation called *now* that returns the actual time in seconds. We adopt the concept of global time and enhance it with the notion of sampling and sampling time. As in TTCN-3, all time values in continuous TTCN-3 are denoted as non-negative float values and represent time in seconds.

3.1.1 Sampling

In digital, software-based test systems, we have to deal with digital representations of continuous real-world signals. The mapping between continuous signals and suitable digital representations is known as sampling. Sampling is the process of reducing continuous signals to discrete representations using fixed or variable sampling rates. In this paper, we restrict ourselves to simple sampling methods with a fixed sampling rate. The step size is defined along the definition of a sample:

⟨SampleDef⟩ ::=
"sample" ⟨SampleName⟩ **"("** ⟨Expression⟩ **")"**

A sample represents a scoped access to the global time. Access to a sample yields the current sampled time. Time is updated for each sampling step, which is defined by the float parameter ⟨Expression⟩. ⟨SampleName⟩ represents the currently sampled time. Its value is updated every sample step, i.e.

$\langle \text{SampleName} \rangle_{current} =$
$\langle \text{SampleName} \rangle_{preceding} + \langle \text{Expression} \rangle$

The sampling rate ⟨Expression⟩ of a sample depends on its current scope as it may be changed to multiples of its own. This is called down sampling and may be used to adapt to certain environmental constraints or to simply reduce the complexity of calculation. Down sampling can be declared locally in behavioural scopes (e.g. in testcases, functions, altsteps, and statement blocks) by use of the *scale* operation.

⟨SampleName⟩ **"("** ⟨Expression⟩ **")"**

This operation can be used in the definition of stream types defined in Sect. 3.2. The positive integer parameter ⟨Expression⟩ represents the respective downsampling factor.

3.2 Streams

A *stream* is a flow of data over time, which has a data value being defined for every point in time—the *stream elements*.

[3] TTCN-3 does not only support asynchronous message-based communication via ports but also synchronous signature-based communication, which however is not considered in this paper.

[4] Terminals are denoted in bold face, non terminals are enclosed by ⟨ ⟩. We denote with | alternatives, with [ ] optional structures and with { } repetitions.

[5] In *Timed* TTCN-3, all test components being in the same timezone share the same understanding (and progress) of time. We assume that all test components are in the same time zone.

⟨StreamType⟩ ::=
"type" "stream" ⟨ScalarType⟩ ⟨StreamName⟩
"(" ⟨SampleName⟩ ")"
[":=" ⟨StreamValue⟩]

We define streams relative to a sample that represents global time. This is in most cases more intuitive than a definition based on absolute time. It also enables the reuse of stream definitions. As scalar types, all TTCN-3 base types and user-defined structured types (and their restrictions) are allowed. Stream values can be finite or infinite. Every stream element has a value and a timestamp. The timestamp of the ith stream element is evaluated to $(i-1)*$⟨Expression⟩, where ⟨Expression⟩ is the sampling rate of the sample associated to the stream type.

Streams can be referenced/accessed via *stream variables* or *stream constants* as it is done in standard TTCN-3. Listing 1 shows a declaration of a sample, a stream type and a variable of that type. We use these definitions for the following listings consecutively.

```
sample t(1.0);                                    1
type stream float FloatStrm(t);
var FloatStrm myStrm:=
        {1.0,2.0,45.0,66.0,77.0};                 4
```

Listing 1 Stream types and variables

Generally, we distinguish between finite streams and infinite streams. Infinite streams are defined over the complete time domain (i.e. for each point in time there exists a well-defined stream value). Finite streams or—generally speaking—partially defined streams may exhibit definition gaps. If so there are points in time where no well-defined stream value exists. The use of finite streams or partially defined streams is allowed in principle but may lead to runtime errors when the non-defined parts are addressed directly. We will come back to this issue later on. Finite streams can be defined element-wise with value lists. Listing 1 contains such an element-wise definition. The stream value `{1.0,2.0,45.0, 66.0,77.0}` defines a finite stream value for `myStrm`.

Furthermore, we propose to extend TTCN-3 with the ability to define default values for user-defined types as shown in Listing 2. A stream value (in the example `{1.0,2.0,45.0, 66.0,77.0}`) can also be used as a default value for any stream (instance) of that type.

```
type stream float FloatStrm(t):=
      {1.0,2.0,45.0,66.0,77.0,88.0};
```

Listing 2 Default values for stream types

Concerning the large amount of data normally necessary to represent a sampled signal, this approach is not feasible for signals with significant length. Therefore, we also use expressions over time. The expression may contain the module sample, e.g. t, that act as a control variable and expresses local time progress. These expressions may use the full TTCN-3 expressiveness for expressions including the use of variables, functions and the newly introduced stream access operations defined in Sect. 3.2.2. Listing 3 presents the definition of a constant float stream and of two time-dependent float streams (referring to the sample `t` defined in the sample type definition of `FloatStrm`).

```
const FloatStrm myFirstStrm@t:= 4.0;
const FloatStrm mySecondStrm@t:= sin(t)+100.0;
const FloatStrm myThirdStrm@t:=                   3
      mySecondStrm@(t+10.0)+4.0;
```

Listing 3 Stream constants

Note the difference between `myFirstStrm@t:= 4.0`, which yields an infinite constant stream of `4.0`, and `myFirstStrm:= {4.0}`, which yields a finite stream containing one element `4.0`. Please also note that although the stream elements of `mySecondStrm` and `myThirdStrm` vary, the stream itself is a constant: the stream elements are defined and cannot be changed. This however is true for any constant in TTCN-3.

Besides implicitly stating the stream duration by denoting the number of individual stream element values (see Listing 2), we may express durations using an extension of the TTCN-3 range operator. Whereas the TTCN range operator can only expresses closed ranges we introduce an extended and more powerful range construct that additionally can express open and half-open ranges.

⟨RangeDef⟩ ::=
("[" | "(")
⟨Expression⟩ ".." ⟨Expression⟩
(")" | "]")

The left outer bound of the range is denoted by ("[" | "(") ⟨Expression⟩. With "(" we define a left side open range, that is the value of ⟨Expression⟩ itself is not included in the range. With "[" we define a left side closed range that includes the value of ⟨Expression⟩. Accordingly, the definition of the right outer bound and the meaning for the symbols ")" and "]" can be derived easily from the definitions above.

For the definition of streams we combine the range operator with the `@`-operator. To be able to define infinite streams, the range operator is to be combined with the TTCN-3 keyword `infinity`.

⟨StreamValue⟩ ::=
⟨StreamTimedValue⟩ | ⟨StreamSampledValue⟩

⟨StreamTimedValue⟩ ::=
⟨StreamName⟩ "@" ⟨RangeDef⟩ ":=" ⟨Expression⟩

For example, `myStrm@[0.0..100.0):=4.0` denotes the assignment of the value `4.0` to the first `100.0` seconds of

stream `myStrm`. Moreover, the range operator in an array-alike notation can be used to specify values for stream elements.

⟨StreamSampledValue⟩ ::=
⟨StreamName⟩ ⟨RangeDef⟩ `":="` ⟨Expression⟩

Furthermore, we allow the segment-wise definition of streams where each segment is a finite stream by itself, i.e. a stream defined over a fixed period of time or by a fixed number of elements. We provide a condensed notation that allows to define a stream consisting of several individually defined stream segments as shown in Listing 4.

```
const FloatStrm myFourthStrm:={
  @[0.0..100.0)       :=sin(t)*100.0,
  @[100.0..2000.0]    :=4.0,                    3
  @(2000.0..infinity):=4.0+sin(t/20.0)}

const FloatStrm myFifthStrm:={                  6
  [0..99]           :=sin(t)*100.0,
  [120..1999]       :=4.0,
  [2000..infinity] :=4.0+sin(t/20.0)}           9
```

Listing 4 Segment-wise definition of streams

Note that definition gaps (see Lines 7, 8 in Listing 4) may lead to partially undefined streams.

3.2.1 Stream definition with equation systems

For more complex stream characterizations, one can use equation systems which enable the definition of stream elements from elements of other streams as shown in Listing 5.

```
var FloatStrm f1@t:= sin(t);
var FloatStrm f2@t:= cos(t);
var FloatStrm g@t:= (t < 10.0) ? f1@t : f2@t; 3
```

Listing 5 Streams defined by equation system

The conditional expression operator "`(_)?_:_`" has the following meaning: if the condition in "`()`" holds then the statements right after "`?`" are performed, otherwise the statements right after "`:`" are performed. This operator is added to TTCN-3 to ease the definition of streams.

3.2.2 Accessing stream elements

Current stream elements and preceding stream elements can be accessed by using the **@**-operator for a time-based access and "`[]`" for an index-based access. These operators are applicable to stream variables and stream constants and allow the selective writing and reading of individual stream elements. They are also applicable to stream ports as defined in Sect. 3.4. Note that any read operation applied to segments of a stream that are not defined will result in an error. Examples for the use of stream access operations are given in Listing 6.

```
var FloatStrm f:= {1.0,1.1,1.2};

var float f_elem:= f[1];                        3
//yields 1.1 as the indexing starts at 0

var float f_elem:= f@2.0;                       6
//yields 1.2 assuming a sampling rate of 1.0

var float f_elem:= f@1.5;                       9
//yields 1.1 as a stream element holds up
//until the next sample
                                                12
var float f_elem:= f@5.0;
//yields error as the stream f is not defined
//for the point in time with value 5            15
```

Listing 6 Accessing stream elements

3.2.3 Calculating with streams

Besides the basic access operators we additionally provide arithmetic operations on streams as well as a concatenation operation for streams. We allow to calculate with streams (e.g. `myStrm1+myStrm2`), to directly compare streams (e.g. `myStrm1>referenceStrm`) and to assign streams to stream constants or variables (e.g. `var FloatStrm f:= myStrm1`). Concerning type compatibility between streams we adopt the existing compatibility rules that are defined in standard TTCN-3. Hence, we declare a stream type compatible to another stream type, when the respective scalar value types are compatible in standard TTCN-3. Otherwise, explicit conversion functions have to be used.

Arithmetic and comparison operations on streams are defined by the element-wise application of the respective operations for the scalar type of the stream. This requires that the operations are defined for the scalar types and that the scalar types of the streams are compatible. It also requires that the sampling rates of the streams are compatible: we assume here that they are the same or that one is a valid downsampling of the other. The result is a stream again. If the stream operands have different lengths, the result is calculated for the shorter length of the two. If the streams have differently scaled sampling rates, the comparable elements are calculated only, e.g. if one stream has a double sampling rate, every second element is calculated only. The resulting stream has the lower sampling rate of the two streams. All the other values are ignored. This may result in streams with a shorter duration or a lower sampling rate. For arithmetic operations on infinite streams we propose lazy evaluation, that is, the scalar values, which represent the result of the arithmetical operation, are calculated only when they are addressed. Note that the calculation with partially defined streams lead to partially defined result streams. That is to say the property of being undefined is invariable in calculations and propa-

gates into the result of the calculation. Refer to Listing 7 for an example of arithmetic operations on streams and for arithmetic operations on partially defined streams.

```
var FloatStrm a:= {1.0,1.1,1.2,1.3,1.4};
var FloatStrm b:= {
  [0..1]          := 4;                    3
  [3..infinity]   := 2*t;
}
                                           6
var FloatStrm f_add:= a+b;
// yields {5.0,5.1,-,7.3,9.4}
// with '-' denoting an undefined value here  9
```

Listing 7 Calculating with streams

For concatenation of streams we reuse the concatenation operator "&" that is so far used for strings in standard TTCN-3. This operator let us seamlessly join streams. For streams with different types and/or sampling rates we use the same compatibility rules that we have already defined for the arithmetic operations (see above). If the left operand of a concatenation expression represents an infinite stream the result of the concatenation operation is equal to the left operand (that is the right concatenation operand is ignored). An example is given in Listing 8.

```
var FloatStrm f:= {1.0,1.1,1.2};

var FloatStrm f_conc:= f&f;                3
//yields {1.0,1.1,1.2,1.0,1.1,1.2}
```

Listing 8 Concatenating streams

3.2.4 Stream templates

In TTCN-3, especially for the definition of reference values and value sets, the use of templates is encouraged. A template defines a pattern for the characterization of values of a given type. A concrete value can either be matched by a template or not. We extend the notion of streams to include also *streams of template matches*, where every match or mismatch is represented by a Boolean true or false. We also apply the notion of templates to stream types, but consider equality, upper and lower bounds for basic (scalar) stream elements, and complements for bounds only. More complex stream templates and stream matches will be subject to further research.

A stream template defines a pattern for the stream elements. The syntactical structure to define stream templates are taken from standard TTCN-3. In ⟨ParList⟩ we allow the definition of parameters of types, the stream is constructed of, or of scalar types.

"template" ⟨StreamType⟩ ⟨Identifier⟩
["(" ⟨ParList⟩ ")"] "@" ⟨SampleName⟩ ":="
⟨StreamTemplateBody⟩

An elementary ⟨StreamTemplateBody⟩ is an individual stream itself (see `t1` in Listing 9). Such a template matches only if the stream that the stream template is compared to is exactly the same.

```
sample t(1.0);
type stream float FloatStrm(t);
template FloatStrm t1@t:=100.0*t;          3
template FloatStrm t2@t:=(100.0..200.0);
template FloatStrm t3@t:=
(sin(t)+100.0 .. sin(t)-100.0);            6
template FloatStrm t4@t:=complement(t3@t)
```

Listing 9 Stream templates

Similar to scalar value templates for numerical types (e.g. `float` or `integer`), we allow template definitions that denote the upper and lower bounds for numerical stream elements. For that, we propose to reuse the range operator that we already defined in Sect. 3.2. To be able to define ranges with infinite upper borders and lower borders, the range operator is to be combined with the TTCN-3 keywords `infinity` and `-infinity`.

("[" | "(")
⟨StreamLowerBound⟩ ".." ⟨StreamUpperBound⟩
(")" | "]")

Furthermore, we support the complement of stream templates by extending the *complement* construct of TTCN-3.

"complement" "(" ⟨StreamTmpl⟩ ")"

A complement of a given ⟨StreamTmpl⟩ matches if and only if the original stream template would not match for every single point in time it is defined for. Note that the complement does not simply negate the result of the template evaluation in general but complements the meaning of the stream template expressions for every single point in time. Hence, a stream template and its complement may both fail when the examined stream hurts both definitions over time.

Listing 9 presents a simple stream template, two range stream templates and a stream template that is defined by a complement of an other template. Template `t1` is defined by an individual stream, `t2` is defined by a range with the constant lower bound `100.0` and the constant upper bound `200.0`, template `t_3` uses dynamic evolving boundaries defined by stream expressions, and `t4` is defined as a complement of stream `t3`.

Moreover, ranges—especially in the field of signal processing—are often used to indicate tolerances, either as a fixed value tolerance or a relative (percentage) value tolerance. To express tolerances explicitly, we introduce an additional syntactical construct for values and ranges of stream elements.

⟨Value⟩ ["|" ⟨Expression⟩ ["%"]]

The ⟨Expression⟩ after | denotes a fixed value tolerance for the range. If `%` is used in addition, the tolerance is inter-

preted as a percentage value tolerance. Example template definitions are shown in Listing 10.

```
template FloatStrm t5@t:=(100.0|5.0);
// accepts stream elements in
// between [95.0 .. 105.0]

template FloatStrm t6@t:=(20.0|5.0%);
// accepts stream elements in
// between [19.0 .. 21.0]

template FloatStrm t7@t:=(20.0|5.0% .. 100.0);
// accepts stream elements in
// between [19.0 .. 100.0]
```

Listing 10 Tolerance templates

The TTCN-3 *match* operator can also be applied to streams and finite stream templates, e.g. `match(t2, myStrm)`. It can also be applied to single stream elements, e.g., `match(t2@10.0,myStrm@10.0)`. The stream elements of different time stamps can also be matched with the match operator.

3.3 Predefined functions for streams

To ease the handling of streams and to enable access to certain stream properties we provide a number of predefined functions on streams. To be able to denote the functions' signatures we introduce some informal super types that either address scalar values of arbitrary type with *any_type* or stream values of any stream type with *any_stream_type*.

3.3.1 Obtaining the length and duration of streams

We distinguish between the length of a stream and the duration of a stream. Whereas the length of a stream is defined as the number of countable elements of a stream, the duration is defined as the length in time.

lengthof(any_stream_type value) **return integer**

The function *lengthof* returns the length of a stream, that is, the number of its elements. The elements are counted from the beginning of the stream, that is, from the element with the index 0 up to the last well-defined element. The function returns actually the number of the last element, i.e. the index of the last element plus one. The length of an infinite stream is infinite—the function returns in this case `infinity`.[6]

durof(any_stream_type value) **return float**

[6] Note that one cannot calculate with `infinity`, but check for (in)equality to `infinity`

The duration of a stream *durof(strm)* denotes its length in time. It is defined as the timestamp of the element that follows the last well-defined element of a stream. The duration of an infinite stream is infinite—the function returns in this case `infinity`. Examples for stream length and stream duration are given in Listing 11.

```
var FloatStrm f:= {1.0,1.1,1.2};

var integer f_length:= lengthof(f);
//yields 3 as the indexing starts at 0

var float f_duration:= durof(f);
//yields 3.0 as the actual sample is 1.0
```

Listing 11 Accessing stream length and duration

3.3.2 Inspecting streams

To check whether a stream, a stream segment, or a stream element is well defined, we introduce two functions for sanity checks on streams.

isdefined(any_stream_type value) **return boolean**

The function *isdefined* returns the value `true` if and only if the referenced stream is completely defined from the beginning until its end. The function is mainly used to detect definition gaps in finite and infinite streams.

ispresent(any_type value) **return boolean**

We also extend the function *ispresent* to streams. The function ispresent returns the value `true` if and only if the value of the referenced stream element is defined. The argument to *ispresent* shall be a reference to a stream element that is expressed either by a stream index operator or the timed operator `@`. The function is mainly used to detect undefined stream elements. Examples for stream sanity checks are given in Listing 12.

Note that the last part of the example highlights an interesting aspect regarding the accuracy of sampled streams. The stream specification states that the stream is defined for the time values `[0.0..5.0)` and `(5.0..100.0]`. Due to the definition gap for the time value 5 and a sampling rate of 1.0 s the stream is in fact not defined between `[5.0..6.0)`.

3.4 Stream ports

To interact with the environment, TTCN-3 uses the notion of ports and differentiates between two communication paradigms for ports: asynchronous and synchronous communication. A port serves as an access point for incoming

```
var FloatStrm f:= {
  @[0.0..5.0):=4
  @(5.0..100.0]:=2                                    3
};

var boolean defined:= isdefined(f);                   6
// yields false because f is not defined
// for the time value 5.0
                                                      9
var boolean defined:= ispresent(f@4.5);
// yields true as the stream f is defined
// for the time value 4.5                             12

var boolean defined:= ispresent(f[4]);
// yields false as the actual sample is 1.0           15
// and the stream is not defined
// for the time value 5
                                                      18
var boolean defined:= ispresent(f@5.5);
// yields false !!! as the actual sample
// is 1.0 and the stream is not defined               21
// for the time value 5
```

Listing 12 Sanity checks on streams and stream elements

and outgoing data. To support continuous data flows we supplement TTCN-3 with the so-called *stream ports*. A stream port is a named variable with a history that—unlike the discrete (message- or procedure-based) ports—receives a value at every sampling step. And unlike the discrete ports, access to the data present at the port is not only possible via the top element but via the current time and via any other time in the past. Stream ports[7] are initialized when they are created (hence, when the test component owning that stream port is created) and start to emit or sense data immediately. The stream elements accessed through a port are related to the sample on which the stream type is based. A stream port emits and senses `omit` whenever it is not connected to a stream source. In the case of the outgoing direction, this is either because nothing is currently sent or the finite stream has been completed already.

A stream port is specified by its type definition.

"type" "port" ⟨StreamPortType⟩ *"stream"*
"{" ⟨Direction⟩ ⟨StreamType⟩ *"}"*

For ⟨Direction⟩ the TTCN-3 directions `in`, `out` and `inout` can be used. With ⟨StreamType⟩ we denote the data type that characterizes the stream. A stream port inherits the sampling rate from the stream type it is transporting.

The instantiation of stream ports and their assignment to components are equivalent to standard TTCN-3. Examples of stream port definitions are shown in Listing 13.

[7] To clarify our terms: A stream port emits or senses a stream but it is not the stream itself. Nevertheless, the stream port name can be used to address the stream pipelined through the respective port. Hence, it can be used to access individual stream elements as well as global information on the stream (like length of the stream, its duration, etc.).

```
type port FloatOut stream {out FloatStrm}
type port FloatIn stream {in FloatStrm}
type port FloatInOut stream {inout FloatStrm}  3

type component MyComponent {
  port FloatOut p1;                                   6
  port FloatIn p2;
  port FloatInOut p3;
}                                                     9
```

Listing 13 TTCN-3 stream port definition

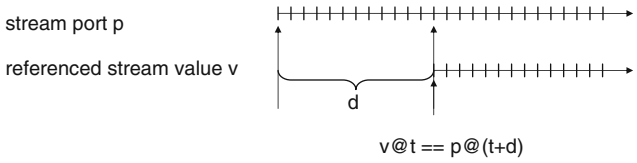

Fig. 1 The @ operator on stream ports

3.4.1 Accessing stream elements at ports

In conjunction with stream ports, the **@**-operator has a particular meaning. Besides accessing stream elements, it serves as a user friendly synchronization and mapping mechanism between global time—that is the basis for ports—and local time—that is used inside stream definitions and other constructs.

To explain the underlying problem: streams are defined from a local perspective, i.e. independent of the history of a port before a stream is being sent or received. On the other hand, the emission or sensing of stream elements at ports start with the creation of a port. The timestamps for ports and the timestamps for local stream definitions are synchronized along the sampling rate. However, they are not comparable with respect to the timestamps of the individual stream elements (as the stream port has typically a number of stream elements transported before referencing a given stream value). This especially conflicts, when using expressions such as `port.sense(strm@t)` or `port.emit(strm@t)` which require synchronized timestamps.

The solution is given as a built in feature of the **@**-operator on stream ports. It takes the time shift into account as shown in Fig. 1.

3.4.2 Stream generation on ports

In Sect. 3.2 we presented different forms for the definition of streams. Whenever a stream is to be emitted via a port, we use basically the assignment of a stream to a port. For that we introduce the *emit* statement on stream ports.

⟨StreamPort⟩ *"."* *"emit"* *"("* ⟨StreamTmpl⟩ *")"*

An example for the use of emit is given in Listing 14.

```
sample t(1.0);
type stream float FloatStrm(t);
type port FloatIn stream {in FloatStrm}
type port FloatOut stream {out FloatStrm}
template FloatStrm myTmpl@t:= 0.1*t;
// an infinite stream value of
// increasing stream elements

type component MyComponent {
  port FloatOut myOutPort;
  port FloatIn myInPort
}

function MyFunc() runs on MyComponent {
        myOutPort.emit(myTmpl)
}
```

Listing 14 Generating a stream on a port

By use of structured types, these constructs allow already to define the simultaneous generation of several single streams as shown in Listing 15.

```
sample t(1.0);
type record R { float a,b,c }
type stream R RStrm(t);
type port ROut stream {out RStrm}
template RStrm myRTmpl@t:= { 0.1*t,2*t,2.0+t }
// an infinite stream value of
// structured stream elements
```

Listing 15 Structured streams

Note that starting to emit streams with a definition gap is possible in principle but will typically lead to a runtime error, if the emission is not terminated before reaching this gap. When the gap is reached, the emit process is stopped and returns with an error.

3.4.3 Stream evalution on ports

Besides the emission of streams via stream ports, the sensing and matching of streams need to be supported. This is done by use of stream templates as introduced in Sect. 3.2.4. We use the *sense* statement on stream ports for that:

⟨StreamPort⟩ *"."* *"sense"* *"("* ⟨StreamTmpl⟩ *")"*

A potential use of sense is depicted in Listing 16—it is an extension of Listing 14.

```
function MyFunc() runs on MyComponent {
        myOutPort.emit(myTmpl);
        myInPort.sense(myTmpl)
}
```

Listing 16 Sensing and matching a stream on a port

Here, an issue related to the duration of stream emission or sensing becomes apparent. The template `myTmpl` is defined by an infinite stream making the emit operation unlimited. No matter however if the stream is finite or infinite: we need a way to emit and sense streams in parallel. That has lead us to the introduction of the *carry-until* statement presented in the following section.

3.5 The carry-until statement

Although structured types can be used to emit and sense several scalar streams in parallel and although finite streams can be used instead of infinite ones, it is not always applicable and practical to do that. Therefore, we introduce a new control flow structure that enables the generation of streams (and of messages and procedures) on parallel ports. Like the alt statement, it also supports the parallel evaluation.

The proposed control flow structure is called *carry-until* statement. It is related to the alt statement already used in TTCN-3 and has been introduced in previous articles (see [20,21]).

"carry" *"{"*
{ *"["* [⟨BoolExpr⟩] *"]"* ⟨OutEventStatement⟩ }
"}" [*"until"* *"{"* [⟨AltGuardList⟩] *"}"*]

The *carry block* is used to define the stimulation of the system under test or of other test components. It may contain multiple TTCN-3 statements for outEvents, i.e. the emission of streams or message sends. It is however not allowed to use receiving statements, alt statements, etc. that require a snapshot. Input events are to be contained in the *until block* only. The until-block contains a list of alternatives. An alternative in ⟨AltGuardList⟩ has an optional guard followed by an evaluation statement, which is optionally followed by a TTCN-3 statement block.

"[" [⟨BoolExpr⟩] *"]"* ⟨GuardOp⟩ [⟨StatementBlock⟩]

The carry block is completed whenever one until alternative matches. Then, the subsequent statement block is executed. When a *continue* statement is used in the statement block, the remaining until-block is continued to be evaluated without resetting the sampled time. When *repeat* is used instead, the carry block is restarted with a reset of the sampled time. Listing 17 shows a pseudo-code representation of the carry-until statement. The carry-block defines with `outEvent*` the stream elements to be emitted or the messages to be sent. With `inEvent*` we denote evaluation statements such as receive statements, timeout statements, or stream sensing statements that trigger the subsequent statements if they match. With `statement*` we denote arbitrary TTCN-3 statements.[8] For stream ports, the *emit* statement

[8] In particular, also the carry-until statement is a TTCN-3 statement by itself. Hence, as depicted, it is allowed to place one or more carry-until statements inside a statement block that is directly connected to an until-alternative.

```
label restart;
// CU1
carry {                                          3
    [] emit outEvent1
    [] emit outEvent2
}                                                6
until {
    [] sense inEvent1 {}
    [] sense inEvent2 {                          9
        // CU3
        carry { [] emit outEvent3 }
        until { [] sense inEvent4 {} }           12
        }
    [] sense inEvent3 { statement1; continue}
}                                                15
// CU2
carry {
    [] emit outEvent4                            18
}
until {
    [] sense inEvent5 {}                         21
    [] sense inEvent6 {statement2; repeat}
    [] sense inEvent7 {statement3; goto restart}
}                                                24
```

Listing 17 The carry-until construct

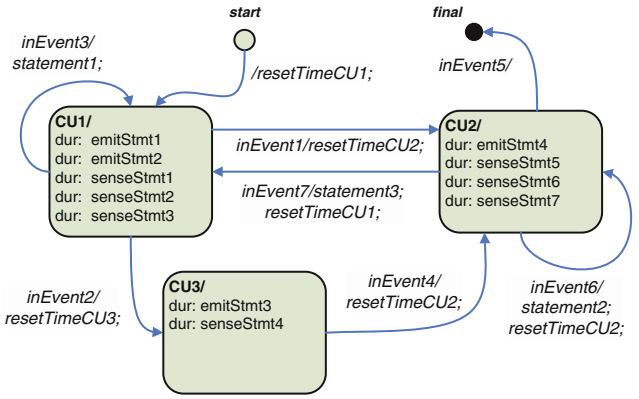

Fig. 2 Carry-until mapped to a hybrid automaton

introduced in Sect. 3.4.2 is used as `outEvent` statement. The *sense* statement introduced in Sect. 3.5 is used as ⟨GuardOp⟩. In addition, we allow to use Boolean expression on samples as ⟨GuardOp⟩. For example, `t>10.0` can be used to check the passage of `10s` after entering the carry-until statement. The statements within a carry-until are evaluated sample-wise—both in the carry-block and in the until-block.

The semantics of the carry-until is defined by a hybrid automaton [1,2,14,15]. Each carry block represents a certain state and the respective until block covers all outgoing transitions that are associated with the state. Hence, each single TTCN-3 alternative that is contained in the until block can be seen as a transition that refers to a target state. The target state itself is a representation of a carry-until statement that is either directly contained in the alternative's statement block or referenced from there (i.e. by use of `continue`, `repeat`, or **`goto`** statements).

The continuous stream processing statements used in carry-until are executed continuously, once for every sampling step. The execution of the stream emission statements that are defined inside the carry block lasts as long as no event within the until block is matched. When events are matched, the execution of the carry block is stopped immediately and the statement block belonging to this event is executed. For the evaluation of inputs at ports we consider delayed effectiveness, i.e. each emission to an output port is available for input not before the next iteration. This condition holds if for every time t, the values of the output ports are defined by use of the values of the input ports for $t' \leq t$ and by use of the values of the output ports for $t' < t$ only.

To demonstrate the operational semantics of the given concepts we map the carry-until construct given in Listing 17 to an automaton depicted in Fig. 2. The automaton is denoted in a graphical notation that is derived from Stateflow [23] and is annotated with statements similar to Stateflow Action Semantics [22].

In Fig. 2 each state is represented by a circle. The *entry state* is marked by a an incoming transition coming from a small circle that is annotated with "`start`". A *state* is identified by a name that is denoted in the upper left corner. Moreover, it may contain actions. With "`dur:<action>`" we denote *during actions* that are processed once for each sample step as long as the state is active. *Transitions* are depicted as annotated arrows. Transitions can be annotated with events and actions. *Events* are separated from actions by "`/`" (e.g. `inEvent1/statement1`). Multiple actions can be separated by "`;`". *Transition actions* represent ordinary TTCN-3 statements whereas during actions represent emit or sense statements.

Once the automaton is started, it remains in its current state until a transition fires. A transition fires when it is associated with the actual state and when the event is triggered that is connected to the transition it is labeled with. Following that, the automaton executes all transition actions in the given order and passes control to the associated target state. The duration of the transition execution is considered to be timeless, time passes along sampling steps. For further details regarding the automaton semantics, refer to the Stateflow semantics [22].

To come back to the given example, the carry-until statements in Listing 17 translate into the three states depicted in Fig. 2. The names of the states are derived from the comments in the listing and serve as a point of reference to identify the individual carry-until constructs. The alternatives, which are given in the until part of a carry-until construct, translate into the transitions of the automaton. To denote the given statements and events we use pseudo-code symbols like we already did in Listing 17. To be able to distinguish different kind of statements, we denote "`statement*`" for ordinary TTCN-3 statements, "`emitStmt*`" for emit statements, and

"senseStmt*" for sense statements.[9] To be able to show the difference between repeat and continue statements inside carry-until, we explicitly introduce the internally used "resetTimeCU*" action that resets the local sample time of the addressed carry-until construct (e.g. resetTimeCU1 resets the time for carry-until CU1).

Figure 2 shows the complete translation of the carry-until constructs from Listing 17. The state in the upper left corner represents the first carry-until construct (CU1). The lines dur:emitStmt1 or dur:senseStmt1 denote stream processing statements (e.g. p1.emit(sin(t) or q1.sense(y1@t)). These statements are executed continuously until the state is left, that is when an event connected to one of the transitions inEvent3 is triggered. Then, the transition actions (e.g. statement1) are executed. Finally, the target state turns to be active and the respective during actions are executed.

Let us now consider the differences between repeat and continue statements. The transition labeled inEvent3/statement1 in Fig. 2 represents the alternative [] sense inEvent3 {statement1; continue;} from the listing. It will be triggered when the sense statement senseStmt3 matches. As mentioned earlier, when a transition is fired all transition actions (i.e. statement1) are executed. Afterward the state execution (i.e. the execution of CU1) is continued. Note that the continue statement does not reset time nor does it discontinue the evaluation of alternatives. By contrast the execution of repeat (i.e. [] inEvent6 {statement2; repeat;} causes an immediate interruption of port evaluation and initiates a restart of the carry-until execution. That is, the local time is reset (see resetTimeCU1) and all statements inside the carry-until statement are executed from their beginning.

The latter holds also for the goto statement given in the example. The goto label restart is placed at the beginning of the carry-until construct CU1. Hence, the execution of goto restart causes a complete restart of the carry-until. In contrast to repeat, the application of goto may manipulate the control flow to bypass several statements (ordinary TTCN-3 statements as well as carry-until statements).

Listing 18 presents another concrete carry-until example. We imply a sample with 0.1 seconds for the example to achieve the required accuracy.

The behavior lasts as long as none of the input guards in the until block is matched. In the given example, the carry block stops when the local sampling time exceeds 6.0 s. Otherwise, it is continued or repeated when y1 exceeds the value of 1.0. Figure 3 shows a plot of possible resulting signals that exhibit non-continuous changes for the moment where the carry-until block is restarted.

[9] To indicate the interdependence between events and statements (e.g. senseStmt1 throws inEvent1 or outEvent1 triggers emitStmt1) we annotate symbols that belong together with a similar postfix (e.g. outEvent1 belongs to senseStmt1). Moreover, we rely on the implicit direction of events (i.e. an outEvent* refers to an emitStmt* but not to a senseStmt*.

```
template FloatStrm y1@t:= (-infinity..1.0];

carry {                                                    3
  [] p1.emit(sin(t))
  [] p2.emit((t<0.2) ? 0.0 : 2.0*p1@(t-0.2))
}                                                          6
until {
  [] t >= 6.0 {}
  [] q1.sense(y1@t) { repeat; }                            9
}
```

Listing 18 Realizing equation systems using the carry-until construct

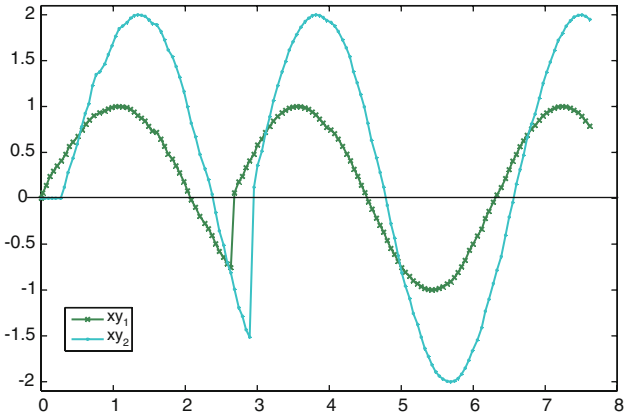

Fig. 3 Plot of signals

3.5.1 Standalone emit and sense statements

Listing 19 shows a standalone emit statement. The listing starts with the declaration of a stream variable and its initialization with a finite stream, i.e. a sine wave with the duration of 100.0 s. Then, the stream is emitted via port streamOutPort.

```
var FloatStrm myOutStrm :=
 {@[0.0..100.0):=sin(t)};
streamOutPort.emit(myOutStrm);                             3
```

Listing 19 Standalone emit statement

Like a stand-alone receive statement is a shorthand for an alt statement, a standalone emit statement is a shorthand for a carry-until statement (Listing 20).

The execution of the following constructs is delayed for the duration of the stream emission. The termination of the carry-until is guided by the duration of the stream being emitted. Whenever the stream is infinite, statements following a standalone emit will never be performed.

Likewise, a standalone sense statement is also a shorthand for a carry-until statement. The equivalent for the sense statement in Listing 21 is shown in Listing 22.

```
var FloatStrm myOutStrm :=
 {@[0.0..100.0):=sin(t)};
carry {                                    3
[] streamOutPort.emit(myOutStrm)
}
until {                                    6
[] t>=durof(myOutStrm) {}
}
```

Listing 20 Equivalent for the standalone emit statement

```
var FloatStrm myInStrm :=                  1
 {@[0.0..100.0):=sin(t)};
streamInPort.sense(myInStrm);
```

Listing 21 Standalone sense statement

```
var FloatStrm myInStrm :=
 {@[0.0..100.0):=sin(t)};
var boolean matchStrm:= true;              3
carry {}
until {
[matchStrm] t>=100.0                       6
   {}            //successful match
[] streamInPort.sense(complement(myInStrm))
   { matchStrm:= false;                    9
     continue } //blocking
}
```

Listing 22 Equivalent for the Standalone Sense Statement

The port is sensed against the stream template. As long as the stream template matches the sensed stream elements, the sensing is continued up until the end of the stream template is reached. If however something different is received, the carry-until enters an infinite loop. This is the equivalent to the blocking nature of receive on message ports: as a stream port evaluation repeats every sampling step (evaluating stream elements at the next time point), the mismatch case needs to be treated as a livelock. In other words, the sensing will take at least as long as the duration of the stream template. If the stream template is infinite or if the stream template does not match at only one sampling time, subsequent statements will not be executed. Nevertheless, standalone sense statements and carry-until statements may be influenced by activated defaults. We will come back to default statements for stream operations later on.

3.6 The delay operator

Another often needed concept when testing real-time systems is that of delaying the start and respective execution of statements:

" (" ⟨expression⟩ ") " ⟨statement-block⟩

The *delay operator* provokes a delay for the `statement-block` with respect to the (functional) enabling of the statement block, for example in a sequence of statements such as given in Listing 23.

```
messageOutPort1.send(myMsgTmpl1);
(4.0){ messageOutPort2.send(myMsgTmpl2); }
messageOutPort3.send(myMsgTmpl3)           3
```

Listing 23 The delay operator

The second send statement is delayed by `4.0 s` after the first send has been completed. The third send statement is however executed immediately after the second send. Note that the delay is determined with respect to the enabling of the statement block. The send sequence in Listing 23 behaves therefore differently within a carry-until. The carry-until defines the parallel stimulation whereas it has been a sequential stimulation before (Listing 24).

```
carry {
[] messageOutPort1.send(myMsgTmpl1)
[] (4.0){messageOutPort2.send(myMsgTmpl2)}   3
[] messageOutPort3.send(myMsgTmpl3)
}
until {...}                                  6
```

Listing 24 Delays within carry-until

This means that the first and third send statement are performed immediately when entering the carry-until statement, whereas the second send is delayed by `4.0 s`. This is depicted in Fig. 4.

3.7 Integration with standard TTCN-3

This section focuses on the operational integration of the newly introduced constructs in standard TTCN-3. We examine the combined use of emit, sense and carry-until together with send, receive, alt, and altsteps. Particular emphasis is given to the snapshot semantics of TTCN-3.

In continuous TTCN-3, we provide means to define finite and infinite streams. These streams can be emitted or sensed via stream ports. The emission or sensing consumes at most one sampling step, so that subsequent statements are delayed.

Standard TTCN-3 is not explicit about the way time is progressing. Timers are used to define timed test control. The timer expiry is handled like an event that can be matched—a timeout queue is used to keep track of timeouts. The expi-

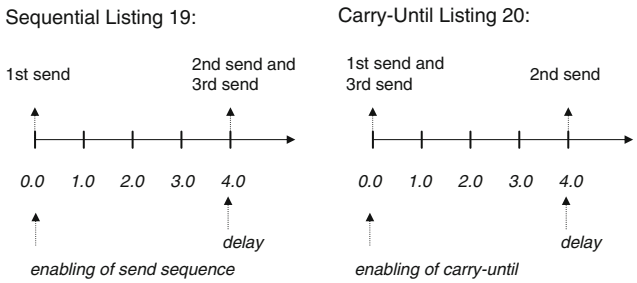

Fig. 4 Delay in sequences and in carry-until

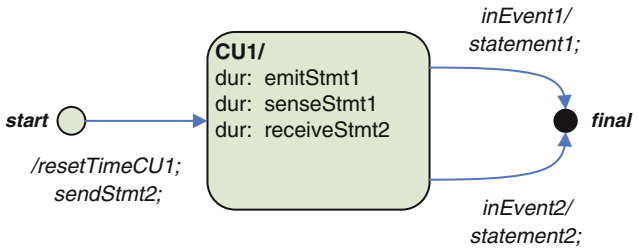

Fig. 5 Carry-until with send/receive mapped to a hybrid automaton

ration of a timer itself is however left open (and realized within the platform adapter of TRI [9]). Conceptually, any TTCN-3 statement is considered to be executed in zero time. Whenever a snapshot is taken for the evaluation of ports (i.e. port, timer, and test component states are frozen), time may progress as defined in the TTCN-3 operational semantics with the *SNAP_TIMER* function [8].

The issue in intermixing standard TTCN-3 statements with stream related statements is to respect the sampling rate of stream ports, which imposes various requirements on the semantics for continuous TTCN-3:

Within any carry-until statements, the outEvents of the carry-block together with the matching against inEvents and the execution of triggered statements in the until-block need to be performed in one sampling step. This implies that a real-time semantics for TTCN-3 is needed, which enables upper limits for the execution of any TTCN-3 statement. However, there are principle issues in TTCN-3 which prevent such an upper limit: an alt statement may take several snapshots before a matching alternative is being identified or a structured type may have an unlimited number of nested fields that require several nested matches, etc. We follow two approaches: one is to limit the potentially unlimited aspects in TTCN-3 to gain a real-time semantics by e.g. allowing n snapshots for a given n only—if then no match can be identified the test case returns with an `error` verdict. The other is to keep the established semantics but to trace if the sampling rate is violated. If so, the test execution continuous but the test results are marked with an `OutOfSync` attribute. The analysis for this loss of synchronization can be done offline and online. Whenever the synchronization has been lost; however, the test runs are marked and need further analysis Fig. 5.

Hence, from an operational perspective it is possible to combine stream processing statements with standard TTCN-3 statements. Under certain conditions this is possible without loss of the required timed continuity for the stimulation and evaluation of stream ports. That is, we can provide a good portion of the expressiveness of standard TTCN-3 in conjunction with the stream processing constructs. The exceptions we made are not serious and may be avoided if necessary—in particular when a real-time execution engine for TTCN-3 is combined with a careful use of the TTCN-3 expressiveness.

3.7.1 Combination of carry-until with message-based statements

In the following, we consider the embedding of message send and receive statements in a carry-until statement. For that, let us consider the example given in Listing 25. We extend the pseudo-code used in Sect. 3.5 to denote with "`sendStmt*`" send statements and with "`receiveStmt*`" receive statements.

```
carry {
    [] emit outEvent1
    [] send outEvent2                          3
}
until {
    [] sense inEvent1 {statement1;}            6
    [] receive inEvent2 {statement2;}
}
```

Listing 25 Combine message send/receive with carry-until

Send TTCN-3 statements are executed once when entering a carry-until, i.e. the message `myMsgTmpl` is sent at the beginning of the carry-until-execution only. The stream emission statements are processed synchronously to the actual sampling rate. They start at the entry phase and their execution is limited by the until events or by the duration of the individual streams. For receive statements in the until-block, we can distinguish three different situations:

1. If no message is captured by the actual snapshot the evaluation is postponed until the next snapshot occurs.
2. Once a message is captured by a snapshot, it can be evaluated by the receive statement. If we obtain a matching event, the execution of statements inside the carry block is terminated, the message is taken from the message port queue and the subsequent statement block is executed.
3. If the captured message does not match, the evaluation is blocked and will be skipped for all further snapshots unless alternative `inEvent` statements on that port overrule this.

The corresponding automaton is depicted in Listing 5.

3.7.2 Combination with alt statements

The carry-until statement is a conceptual extension of the alt statement: The alt statement provides the simultaneous evaluation of input ports by its underlying snapshot mechanism. The carry-until statement provides additionally the simultaneous stimulation of output ports. While the alt statement takes a snapshot for every cycle of identifying a matching alternative, the carry-until statement provides a snapshot

for every iteration induced by the sampling step. Listing 26 shows a combination of stream processing statements with an alt statement.

```
var integer x:= 1;
alt {
[] messageInPort.receive(MsgTmpl) {}
[] streamInPort.sense(myInStrm)
   { x:= x+1; repeat }
}
```

Listing 26 Combination with an alt statement

As discussed earlier, the snapshot for the alt statement is synchronized with the sampled snapshot of the stream ports. In fact, the alt statement translates into a carry-until statement, which results in Listing 27.

```
var boolean matchStrm:= true;
var integer x:= 1;
carry {}
until {
[] messageInPort.receive(MsgTmpl) {}
[matchStrm] t>=durof(myInStrm)
            { x:= x+1; repeat }
[] streamInPort.sense(complement(myInStrm))
            { matchStrm:= false; continue}
}
```

Listing 27 Equivalent for the alt statement combination

The resulting carry-until statement uses the equivalent for a standalone sense given in Sect. 3.5.1 extended with a flag to denote a mismatch to the stream template. In the given alt statement, the blocking may result both from a mismatch on the message template or a mismatch on the stream template. Hence, not only the stream mismatch should run into a livelock. Incoming messages and incoming stream elements are matched against the message and stream template. Whenever a message template matches, the carry-until terminates. The matching against the incoming stream is continued for the duration of the stream template. If there is a mismatch, it is noted and disallows the further match against the stream template. Note that the continue in the equivalent is needed to sense the complete stream template, while the repeat denotes the repeat that has been defined in the original: once the stream template has been sensed completely, the variable `x` is increased and the alt statement is to be repeated—and hence the carry-until representing that alt statement is to be repeated.

The sampling `t` of `myInStrm` is taken relative to the start of the carry-until statement. Moreover, the snapshot in every sampling step freezes the port queues (of message and of stream ports) as well as the timeout queue. While the message port freeze gives access to the top element of the port queue only, the stream port freeze allows also accessing preceding stream elements.

3.7.3 Combination with altsteps and defaults

In Sects. 3.7.1 and 3.7.2 we examined the interaction between stream processing statements specified for Continuous TTCN-3 and message statements of standard TTCN-3. We discussed in particular the seamless integration of evaluation statements (matching alternatives for streams and for messages), so that they are applicable to until blocks as well as to alt statements. We showed that the sampling based snapshot from continuous TTCN-3 can be integrated with the functional snapshot of TTCN-3.

Accordingly, we enhance *altsteps* with the ability to contain sensing statements for streams. Hence, we support the definition of altsteps using standard TTCN-3 statements (i.e. timeout or receive statements), sense statements (i.e. the comparison against scalar values or against ranges for stream elements), and a mixture of both. Listing 28 shows an altstep definition of that kind.

```
altstep myAltstep() runs on ACCTester{
[] messageInPort.receive(myMsgTmpl)
   { setverdict(inconc) }
[] streamInPort.sense(myStrmTmpl)
   { setverdict(fail) }
}
```

Listing 28 Altstep definition

The instantiation of such an altstep within a carry-until or an alt statement follows the TTCN-3 rules for altsteps and the additional rules given in Sects. 3.7.1 and in 3.7.2. For example, the equivalent for Listing 29 is given in Listing 30.

```
alt{
[] messageInPort.receive(myMsgTmpl2)
   { setverdict(pass) }
[] myAltstep() {}
}
```

Listing 29 Integration with altsteps

```
var boolean matchStrm:= true;
carry {}
until {
[] messageInPort.receive(myMsgTmpl2)
   { setverdict(pass) }
[] messageInPort.receive(myMsgTmpl)
   { setverdict(inconc) }
[matchStrm] t>=durof(myStrmTmpl)
   { setverdict(fail) }
[] streamInPort.sense(complement(myStrmTmpl))
   { matchStrm:=false; continue }
}
```

Listing 30 Integration with altsteps

Furthermore, the handling of defaults along the activation and deactivation of altsteps follows the rules of standard TTCN-3 and need not to be explained further.

3.7.4 Relation of carry-until and timer

Finally, let us consider the relation of the carry-until statement and TTCN-3 timers. In fact, the carry-until provides an alternative mean to express timer. A typical timer usage which guards a send-receive pair is shown in Listing 31.

```
timer T(4.0);
messageOutPort.send(myReq);
T.start;                                    3
alt{
[] messageInPort.receive(myAck)
   {setverdict(pass);T.stop}                6
[] T.timeout {setverdict(fail)}
}
```

Listing 31 Timer usage

A corresponding carry-until representation is given in Listing 32. Here, no explicit timer is needed. Instead, the expiration of the waiting period is checked directly via the sample `t`.

```
carry {
[] messageOutPort.send(myReq)
}                                           3
until {
[] messageInPort.receive(myAck)
   {setverdict(pass)}                       6
[] t>4.0 {setverdict(fail)}
}
```

Listing 32 Timeout representation with carry-until

However, note the difference between the two representations. In the first case, the usage of timer and the checking of the timer expiry is based on the traditional snapshot of TTCN-3 meaning that the timeout detection may be delayed some (unknown) time. In the second case, the expiration of the (timer) duration is checked every sample time. Hence, the tolerance of detecting the timeout is at most a sampling time (up until cases, where the sampling rate is not kept).

Also note that we do not argue in favor for one or the other representation. The timing on message ports is well guarded with timers, while for stream ports an extra timer is not needed. The sample and conditions on the sample can be used instead. It is however important that the two forms do not contradict and that they even can be used in combination. This follows an analogous argumentation as the combination of carry-until with send and receive statements given in Sect. 3.7.1.

4 Case study

To demonstrate the usage of the introduced concepts and to show their applicability to automotive test tasks, we provide a small case study that represents a TTCN-3 realization for testing an *Adaptive Cruise Control* (ACC) [3].

4.1 The system under test

An ACC is a cruise control that automatically detects vehicles running ahead and—in the case of a slow vehicle ahead (the target)—it adjusts the actual speed (`v_acc`) so that a safe distance to the detected vehicle is guaranteed (the distance control mode). When there is no vehicle ahead, an ACC works like any other cruise control (the velocity control mode). Today there are different variants of ACCs available. We use a simplified example here (see Fig. 6) that does not match the requirements of a real product but that is sufficient to demonstrate the new features that we have introduced for TTCN-3.

The ACC consists of three major parts. The ACC-control unit provides the main functional behavior. It is responsible to control the actual velocity (`v_acc`), the destination distance to the vehicle ahead (`d_des`), and it provides a warning signal to inform the driver when the safe distance is violated. The ACC-control is supplemented by the target detection unit that preprocesses sensor information on the velocity of the vehicle ahead (`v_target`) and the pedal interpretation unit that preprocesses the driver's input (`phi_brake, phi_gas`). For testing purposes we use the test interface specified in Table 1. Note that TTCN-3 uses a test system centric perspective, i.e. system inputs are declared as outputs here and system outputs as inputs.

4.2 The Informal Test Specification

In this example we test the changeover between distance control mode—when there is a slow vehicle ahead—and velocity control mode—without any vehicle ahead. This forms one of the more complex tasks of an ACC and can be tested by the following test behavior (for a similar test specification see [3]).

1. **Init:** Accelerate the vehicle `phi_gas:=100` until velocity `v_acc` rises to more than 40 m/s. Then switch on the cruise control `leverpos:=HOLD_ACC`.
2. **Activate distance control mode:** Now, we introduce a vehicle ahead by setting the target velocity to `v_target:= 35+sin(t)` m/s. The initial distance `d_init:=90` m was set before the test execution using the parameter interface in Matlab/Simulink. The ACC should switch to distance control mode after a few seconds. The safe distance to the vehicle ahead can be calculated with `v_acc/2*df`. The symbol `df` represents a distance factor that is set to the value of `2.0` here.
3. **Accelerate target:** To test whether the ACC switches back to velocity control mode, we accelerate the tar-

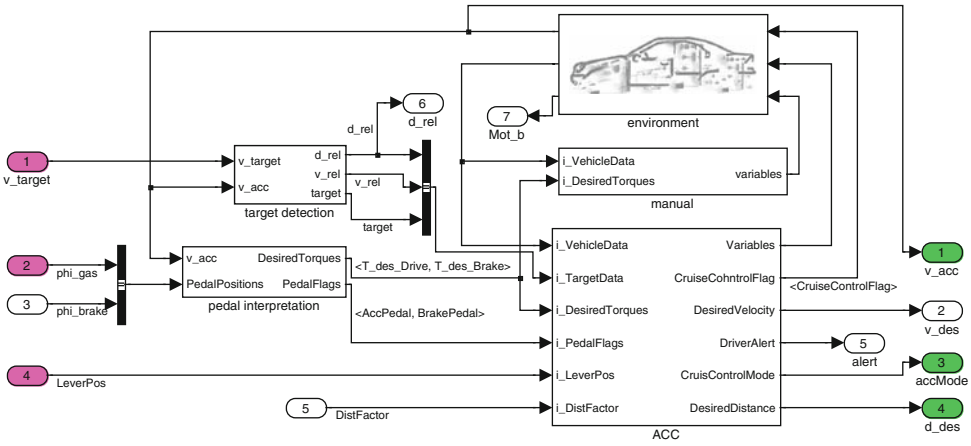

Fig. 6 Simulink model of an adaptive cruise control

Table 1 ACC test interface

Symbol	Dir	Unit	Datatype
`v_target`	Out	m/s	Double
`phi_gas`	Out	%	Double
`le-verpos`	Out	–	Enumeration
`d_des`	In	m/s	Double
`acc_mode`	In	–	Boolean
`v_acc`	In	m/s	Double

get vehicle using a smooth ramp. The ACC should now adjust the actual velocity according to the vehicle ahead as long as the velocity exceeds `40` m/s. We allow a tolerance of `10%` here. Afterward, the ACC should switch back to velocity control mode.

4.3 The continuous TTCN-3 test specification

We use TTCN-3 and the concepts specified in Sect. 3 to implement the test case. We start with the specification of the test system architecture and the test interface. We declare continuous stream ports to cover the velocity (`v_des, d_des, v_target`), the pedal output (`phi_gas`), and the input of the actual acc status (`acc_mode`). To set the lever position we choose a port with a message based communication characteristics.

After the definition of the structural setup, we define constant values and streams that are used for the stimulation of the system later on.

For the evaluation of system behavior, we define stream templates. The template `s_dist_fail` covers an essential safety requirement. The desired distance should never under-run the minimal safety distance. The template `v_acc_fail` will be used later on to monitor the actual velocity in proportion to the velocity of the target. It has two scalar value parameters that will be bound to the distance factor and the actual velocity during the test run.

```
sample t(1.0);
type stream float FS(t);
type stream boolean BS(t);                    3
type port FloatOut stream {out FS};
type port FloatIn stream {in FS};
type port BoolIn stream {in BS};              6

type enumerated Lever { MIDDLE, HOLD_ACC,
                        HOLD_DEC, OFF };      9
type port LeverOut message
  { out Lever };
                                              12

type component ACCTester {
  port FloatOut v_target, phi_gas;            15
  port FloatIn d_des, v_acc;
  port BoolIn acc_mode;
  port LeverOut leverpos;                     18
}
```

Listing 33 Test architecture

```
const float INIT_SPEED:= 40.0;
const float INIT_T_SPEED:= 35.0;              2
const float TIMEOUT:= 20000.0;

const FS kickdown@t:= 100.0;                  5
const FS target_speed@t:= INIT_T_SPEED+sin(t);
const FS accelerate_slow@t:= t/1000.0;
```

Listing 34 Constants used in test specification

```
template FS s_dist_fail
(float df, float vel)@t:=
    complement([vel/2.0*df .. infinity]);     3

template FS v_acc_fail(float vel)@t:=
    complement( vel|10.0% );                  6
```

Listing 35 Template definitions

As proposed in Section 3.7.3 we use the altstep-construct to define reusable evaluation statements that can be activated

as default and so applied to multiple carry-until statements in the following.

```
altstep tout_and_safety(float df)
runs on ACCTester{
[] d_des.sense(s_dist_fail(df,v_acc@t))
   { setverdict(fail) }
[] t>TIMEOUT
   { setverdict(fail) }
}
```

Listing 36 Altstep definition

Now, we are able to define the test case itself. We start with the init phase and activate the ACC when `INIT_SPEED` is reached.

```
testcase ACC_Mode_Test() runs on ACCTester {
  setverdict(pass);
  var float df:= 2.0;

  carry {
  [] phi_gas.emit(kickdown)
  }
  until{
    [] v_acc.sense([INIT_SPEED .. infinity])
       { leverpos.send(HOLD_ACC) }
  }
  ...
```

Listing 37 The test case definition

In the following, we activate `tout_and_safety` as default. This guarantees the detection of safe distance violation and timeouts. For the test behavior, we use carry-until to introduce the vehicle ahead (`v_target.emit(target_speed)`). We also check that the distance control mode starts (`acc_mode.sense(true)`).

```
  var default safety_default :=
      activate(tout_and_safety(df));

  carry {
  [] v_target.emit(target_speed)
  }
  until {
  [] acc_mode.sense(true) {};
  }
  ...
```

Listing 38 Test ACC mode activation

Finally, we check whether the ACC holds the correct velocity during the acceleration of the target and whether it switches back to velocity control mode, when the destination velocity is reached again.

Occurring errors are detected during test execution and logged using the `setverdict(fail)` statement. After test execution we are able to obtain the test results and analyze them by examining the verdict and test logs provided by the test case.

```
  carry {
  [] v_target.emit(v_target@(t-1.0)
                   +accelerate_slow@t)
  }
  until{
  [] acc_mode.sense(false) {}
  [] v_acc.sense(v_acc_fail(v_target@t))
     { setverdict(fail) }
  }
  deactivate(safety_default);
}//testcase
```

Listing 39 Test ACC mode deactivation

5 Summary

This paper reviewed the general requirements for a test technology for embedded systems, which use both discrete signals (asynchronous message-based or synchronous procedure-based ones) and continuous flows (stream-based). It compared the requirements with the only standardized test specification and implementation language TTCN-3 (the testing and test control notation [7]). While TTCN-3 offers the majority of test concepts, this paper showed that it has limitations for testing systems with continuous aspects.

Hence, the paper introduced basic concepts and means to handle continuous real-world data in digital environments. We introduced streams that can be created, calculated and examined by means of continuous and discretized data. Moreover, we extended TTCN-3 with the concepts of stream-based ports, sampling, equation systems, and additional control flow structures to be able to express continuous behavior. This paper demonstrated the feasibility of the approach by providing a small example. In future work, these concepts need to be implemented completely and applied to real case studies in the field of automotive software engineering and the development of ECUs.

References

1. Alur, R., Henzinger T.A., Sontag E.D.: Hybrid systems III: verification and control. Lecture Notes in Computer Science. In: Proceedings of the DIMACS/SYCON Workshop, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA. Bd. 1066. Springer, Berlin (1996). ISBN 3-540-61155-X
2. Bringmann E., Kraemer A.: Systematic testing of the continuous behavior of automotive systems. In: SEAS '06: Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems. pp. 13–20. ACM Press, New York (2006). ISBN 1-59593-402-2
3. Conrad M.: Modell-basierter Test Eingebetteter Software im Automobil, TU-Berlin, Discussion (2004)
4. Dai, Z., Grabowski, J., Neukirchen, H.: Timed TTCN-3—A real-time extension for TTCN. In: Testing of Communicating Systems, vol. 14, Kluwer, Berlin (March 2002)
5. dSpace AG: dSPACE—AutomationDesk. http://www.dspace.com/ww/en/pub/home/products/sw/expsoft/automdesk.cfm. Version 2007

6. Etas Group: ETAS—Hardware in the loop (HiL)—ECU Testing—Applications & Solutions—ETAS Products. http://www.etas.com/en/products/applications_hardware_in_the_loop.php. Version 2007
7. ETSI: ES 201 873-1 V3.2.1: Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 1: TTCN-3 Core Language. Sophia Antipolis, France, (February 2007)
8. ETSI: ES 201 873-4 V3.2.1: Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 4: TTCN-3 Operational Semantics. Sophia Antipolis, France, (February 2007)
9. ETSI: ES 201 873-5 V3.2.1: Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 5: TTCN-3 Runtime Interfaces. Sophia Antipolis, France, (February 2007)
10. Grossmann, J.; Conrad M., Fey I., Wewetzer C., Lamberg, K., Krupp A.: TestML a language for exchange of Tests. http://www.immos-project.de/. Version 2006
11. Grossmann, J. ; Mueller, W.: A formal behavioral semantics for TestML. In: Proceedings of IEEE ISoLA 06, Paphos Cyprus, pp. 453–460 (2006)
12. International Organization for Standardization: Information technology—Open systems interconnection—Conformance testing methodology and framework—Part 3: The Tree and Tabular combined Notation (TTCN), ISO/IEC 9646-3, 2nd edn. Geneva, November 1998
13. Schäuffele, J.T.Z.: Automotive Software Engineering. Vieweg & Sohn, Wiesbaden (2006) ISBN 978–3528010409
14. Lehmann, E.: Time Partition Testing Systematischer Test des Kontinuierlichen Verhaltens Von Eingebetteten Systemen. Berlin, TU-Berlin, Disscussion (2004)
15. Lynch, N.A., Segala R., Vaandrager F.W., Weinberg, H.B.: Hybrid I/O Automata. In: Alur, R., Henzinger T.A., Sontag E.D. pp. 496–510. ISBN 3-540-61155-X
16. MBtech Group: http://www.mbtech-group.com—PROVEtech:TA—Überblick. http://www.mbtech-group.com/eu-de/electronics_solutions/test_engineering/provetechta_ueberblick.html. Version 2007
17. National Instruments: NI TestStand—Products and Services—National Instruments. http://www.ni.com/teststand/. Version 2007
18. Neukirchen, H.: Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests, Georg-August-Universität Göttingen, Discussion (2004)
19. SCC20 ATML Group: IEEE ATML Specification Drafts and IEEE ATML Status Reports. http://grouper.ieee.org/groups/scc20/tii/. Version 2006
20. Schieferdecker, I., Bringmann, E., Grossmann, J.: Continuous TTCN-3: testing of embedded control systems. In: SEAS '06: Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems. ACM Press, New York pp. 29–36 (2006). ISBN 1-59593-402-2
21. Schieferdecker, I., Grossmann, J.: Testing of Embedded Control Systems with Continous Signals. In: Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme II, TU Braunschweig, pp. 113–122 (2006)
22. The MathWorks: The MathWorks Helpdesk—Stateflow Semantics. http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/ug/f26-1032049.html. Version 2007
23. The MathWorks: Web Pages of Stateflow - Design and Simulate State Machines and Control Logic. http://www.mathworks.com/products/stateflow/. Version 2007
24. Vector Informatik GmbH: Vector [Portfolio-Übersicht "Steuergeräte-Test"]. http://www.vector-worldwide.com/vi_ecutest_de.html. Version 2007