Modeling Property Based Stream Templates with TTCN-3

Juergen Grossmann[1], Ina Schieferdecker[1], and Hans-Werner Wiesbrock[2]

[1] Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, D-10589 Berlin
{juergen.grossmann,ina.schieferdecker}@fokus.fraunhofer.de
[2] IT Power Consultants, Gustav-Meyer-Allee 25, D-13355 Berlin
hans-werner.wiesbrock@itpower.de

Abstract. Test evaluation and test assessment is a time consuming and resource intensive process. More than ever this holds for testing complex systems that emanate continuous or hybrid behavior. In this article we introduce an approach that eases the specification of black box tests for hybrid or continuous systems by means of signal properties applied for evaluation. A signal property allows the characterization of individual signal shapes. It is determined by the examination of the signal's value at time, the application of pre-processing functions (like first or higher order derivatives), and the analysis and detection of sequences of values that form certain shapes of a signal (e.g. local minima and maxima). Moreover we allow the combination of properties by logical connectives.

The solution provided in this paper is based on terms and concepts defined for Continuous TTCN-3 (CTTCN-3) [12, 11], an extension of the standardized test specification language TTCN-3 [4]. Thus, we treat signals as streams and integrate the notion of signal properties with the notion of stream templates like they are already defined in CTTCN-3. Moreover, we provide a formal foundation for CTTCN-3 streams, for a selected set of signal properties and for their integration in CTTCN-3.

1 Introduction

TTCN-3 [4, 6, 5] is a procedural testing language. In its current(and standardized) state TTCN-3 provides powerful means to test message-based and procedure-based system interactions. As such it is not capable of testing system that emanates continuous or hybrid behavior. To fill the gap and to transmit parts of the approved TTCN-3 methodology to continuous and hybrid systems as well, we introduced Continuous TTCN-3 (CTTCN-3) [12, 11] and enhanced the TTCN-3 core language to the requirements of continuous and hybrid behavior while introducing:

- 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.

While [12] concentrates on system stimulation and the integration of the newly introduced concepts with the existing TTCN-3 core language, the systematic evaluation of system reaction was not discussed in depth. In this article we will catch up and work out the notion of *signal properties* and *property based stream templates*. A signal property addresses a certain but abstract aspect of a signal shape (i.e. the signal's value at a certain point in time, the derivative of the signal, and certain behavioral aspects like rising edges, extremal values etc.). A property based stream template constitutes a predicate that is based on signal properties and can be used to specify the expected system behavior for a test run.

The specification of formal properties to denote the requirements on a hybrid system is well known from the theory of hybrid automata [1]. Given a set of formal system properties denoted in a temporal logic calculus, the reachability and liveness of the properties can by automatically checked if an appropriate system model exists [7, 8]. The Reactis tool environment [13] provides a similar approach to derive test cases from models that can be applied to the system under test (SUT).

In [3, 14] such predicates are used as an explicit part of a test specification to ease the assessment of a hybrid system's reaction. In [14] a systematic approach for the derivation of so called validation functions from requirements is described. The approach introduces the notion of signal properties and their respective concatenations to detect certain — sometime very complex — signal characteristics (e.g. value changes, increase and decrease of a signal as well as signal overshoots) during the execution of a black box test run. The solution is based on Matlab/Simulink. In [3] a graphical modeling tool is outlined that is dedicated to ease the specification of signal properties for the off-line evaluation of tests. Both approaches aim to systematically denote signal properties. In this article we concentrate on a proper integration of signal properties with the existing means for testing hybrid behaviour in CTTCN-3.

In Section 2 we will give a short overview of CTTCN-3. This includes the explanation of the concepts stream, stream port and stream template. Moreover, the overview includes the definition of a formal semantics for streams, which will be used later on to properly integrate the notion of signal properties. In Section 3 we will describe a guiding example to motivate our approach. In Section 4 we will introduce the term *signal property* and a suitable classification of signal properties, in Section 5 we will introduce the integration of property based stream templates with CTTCN-3, and Section 6 concludes the paper.

2   Continuous TTCN-3

CTTCN-3 is an extension of TTCN-3 that is properly specified in [12] and as yet a theoretical prototype. In the following we will provide a short introduction to the syntax and semantics of the main constructs in CTTCN-3.

2.1   Time

For CTTCN-3 we adopted the concept of a global clock and enhance it with the notion of sampling and sampled time. As in TTCN-3, all time values in CTTCN-3 are denoted as float values and represent time in seconds. For sampling, the discrete time model $t = k * \Delta$ is used. It has a fixed step size Δ with $t, \Delta \in \mathbb{R}^+$, $k \in \mathbb{N}$. Relative time, which is used for the definition of streams and templates, is considered to be completely synchronized to the global clock.

2.2   The Test System

The SUT is represented in terms of its interface — the so called test interface. A test interface is defined by a set of input and output ports. Each port can be characterized by its direction of communication (i.e. unidirectional input or output, or bidirectional), the data types being transported (e.g. boolean, integer, float, etc.), and its communication characteristics (i.e. message-based, procedure-based, or stream-based). In this article, we denote the input ports of a SUT as an n-tuple $\boldsymbol{x} = (x_1, x_2, .., x_n)$ and the output ports as an m-tuple $\boldsymbol{y} = (y_1, y_2, .., y_m)$ with $m, n \in \mathbb{N}$ (see Figure 1)[1]. Moreover, we define a set of data types called $\daleth$ to specify the information structure transferrable via ports. For each port x_n, y_m there is a set $X_n, Y_m \in \daleth$ defining the domain of the port.
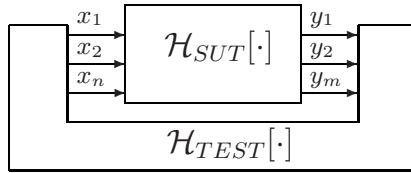

Fig. 1. An Black-box test system enclosing a SUT

System behavior is defined in terms of the given allocation of ports. Reactive system behavior can be denoted as an operator $\mathcal{T}[\cdot]$ that continuously operates on the inputs of the system [10]. The allocation of individual ports are defined by a function $f_{x_i}(t)$ over time. The complete System inputs over time — reflecting our definition of an SUT — are defined as an n-tuple,

$$\boldsymbol{x}(t) := (f_{x_1}(t), f_{x_2}(t), ..., f_{x_n}(t)) \;\; with \;\; \boldsymbol{x}(t) \in X_1^\epsilon \times X_2^\epsilon \times ... \times X_n^\epsilon.$$

The output of a system is defined by an equation system using the behavior operator $\mathcal{T}_{SUT}[\cdot]$.

$$\boldsymbol{y}(t) := \mathcal{T}_{SUT}[\boldsymbol{x}(t)] \;\; with \;\; \boldsymbol{y}(t) \in Y_1^\epsilon \times Y_2^\epsilon \times ... \times Y_m^\epsilon.$$

---

[1] A bidirectional port contributes both to the input and the output tuple of ports.

2.3   Streams

In contrast to scalar values, a stream [2, 9] represents the whole allocation history applied to a port. In $CTTCN$-3 the term stream is used to denote the data structure $s \in (STRM)_T$ that describes the complete history of data that yield as allocation of a certain port x_n, y_m. The index T denotes the type of a stream. It is defined as a cross product between a value domain $d \in \daleth$ and the step size $\Delta \in \mathbb{R}^+$ with $T \in \daleth \times \mathbb{R}^+$. In the following we only consider discrete (i.e. sampled) streams $s \in (DSTRM)_T \subset (STRM)_T$. A discrete stream s is represented by a structure $s := (\Delta, \langle m_k \rangle)$ where Δ represents the sample time, $\langle m_k \rangle$ a sequence of values (messages), and $s \in (DSTRM)_T$. The sequence of values is defined as follows:

$$\langle m_k \rangle := \{f_{x_i}(0), f_{x_i}(1 * \Delta), ..., f_{x_i}((k-1) * \Delta)\}$$

To obtain basic information on streams and their content we provide simple access operations. We distinguish between time-related and non-time-related access operations.

- For non-time-related access operations we use $\#s$ for the number of values and with $s[i]$, $i \in \mathbb{N}$ we denote the ith value in a stream s.
- Time-related access operations rely on a timing function $\tau_s(i) := (i-1) * \Delta$ with $i \in [1..(\#s)[$ that returns a time value for an arbitrary index value of a stream s. The operation $dur(s)$ returns the length of time for a stream s and is defined as $dur(s) = \tau_s(\#s)$. Further on we provide the operation $s@t$ to obtain the value m associated with an arbitrary point in time with $t \in \mathbb{R}^+$. The operation $s@t$ is defined as $s@(t) = s.i$ when $t \in [\tau_s(i), \tau_s(i+1)[$.

In $CTTCN$-3 we are able to explicitly declare *stream ports* and *stream variables* by the notion of *stream types T*. The step size Δ is defined using the keyword **sample**. Listing 1 shows the declaration of a sample and the declarations of a stream type, of two stream ports, as well as of a stream variable. Moreover the stream variable is initialized with a stream of infinite length.

Listing 1. Stream Types and Variables

```
sample( t )=1;
type stream float  FloatStrm ( t );
type port  FloatOut stream {out FloatStrm}            3
type port  FloatIn stream {in FloatStrm}

type component MyComponent {                          6
  port  FloatOut  p1;
  port  FloatIn  p2;}
                                                      9
var  FloatStrm  myStrm:=  sin ( t );
```

2.4   Stream Templates

In TTCN-3, especially for the definition of the expected system reaction, the use of templates is encouraged. In [12] we advanced the notion of templates to be applicable to streams. We confined ourselves to templates for numerical streams and to the definition of upper and lower bounds only.

Similar to streams, stream templates $tp \in (TP)_T$ are classified by stream types T. Templates are generally applicable to streams of the same type or of compatible type[2]. In CTTCN-3 the application of a template to a stream or a stream port is carried out by either a *sense* statement (for the on-line evaluation of ports) or a *match* statement (for the offline evaluation of the data structure stream). The result of the application is dependent on the execution context. Inside the $carry-until$ statement, the template evaluation is carried out sample-wise, that is, it is defined as a function $\chi_{tp} : (DSTRM)_T \rightarrow (DSTRM)_{\mathbb{B}}$ where $r \in (DSTRM)_{\mathbb{B}}$ is a **stream** of boolean values *true* or *false*.

Outside the carry-until construct the evaluation of a stream template is calculated as a whole, that is the complete stream is evaluated and the evaluation is defined as a function $\chi_{tp} : (DSTRM)_T \rightarrow \mathbb{B}$ where $r \in \mathbb{B}$ is one of the boolean values *true* or *false*. In both cases the function $\chi_{tp}(s)$ is determined by the template definition TP. For more details concerning the meaning of stream template please refer to [12].

3   Guiding Example

The main objective of this article is to ease the specification of expected system behavior through the notion of signal properties (i.e. predicates) that, on the one hand can be used to closely describe the shape of individual signals, but also provide means to flexible address abstract characteristics of a signal. In order to motivate the concepts and constructs we present in this article we will start with a typical scenario that emanates from ECU testing in the automotive domain. In a drive case the tester activates the gas pedal, releases the pedal, and after a while he activates the break pedal. In the context of this example we are interested in the velocity control. Concerning the velocity, we expect a nearly linear increase at the beginning. That followed, the velocity remains constant for a short while to start slightly decreasing, and in the end it slows down to 0. Figure 2 shows a simple outline of our expectations.

In the following, we are looking for a feasible, that is abstract but exact, way to formerly describe our expectations. The crucial property we are interested in, is the signal's slope. Being more precisely, we expect the slope being nearly lets say 2.0 $[m/s^2]$ or nearly 0 or roughly -2.0 $[m/s^2]$. Moreover, we would like to address the respective durations and sequencing.

---

[2] Compatibility between stream types is dependent of the value domain $d \in \daleth$ and the sampling domain $\varDelta \in \mathbb{R}^+$. For the value domain we consider the given TTCN-3 compatibility rules. For the time domain we consider two types compatible when they obey the same sampling or one is a down sample of the other one.
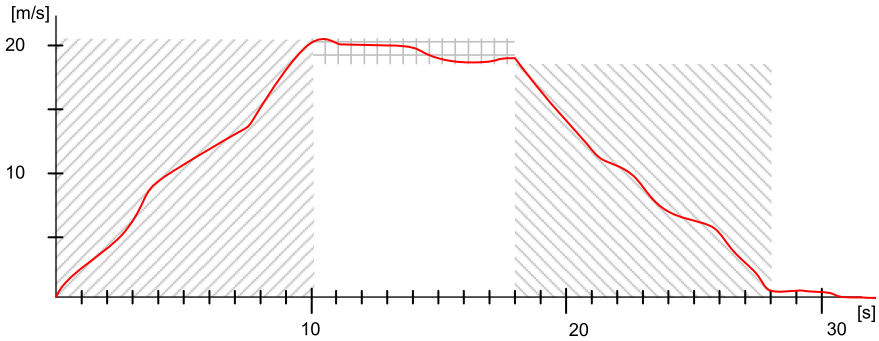
Fig. 2. Simple Shape of a Signal

In order to asses a concrete test result w.r.t. to our expectations, we have to denote our expectation in form of predicates, that closely characterizes the possible outcomes. Using the formalism sketched in the former paragraph, we end up with the following situation. The possible test outcome for some ECU is described by a real valued stream, i.e. the velocity measured at the sample times of our test run. The expectation that at the beginning the increase will be nearly linearly and we must check for it. We can formulate such predicates in terms of templates[3]. Heuristically we define:

Listing 2. Heuristic: Linear Increase

```
// in the time interval [0 s, 10 s],
// the derivative of the velocity is in [1.75..2.25]                    2

template FloatStrm linearSlope_0_135@t:={
   @[0.0..10.0] := (differentiate(current) in [1.75..2.25])}; 5
```

That is, the derivative in the time interval [0 s,10 s] is nearly constant. Regarding this example, one can obviously distinguish two different parts, the proper *predicate* and the *time scope*, that is the time interval the predicate is applied for. We could intuitively split this up and rewrite the expressions to:

Listing 3. Heuristic: Linear Increase, Time Scope and Predicate

```
template FloatStrm linearSlope@t:={                                      1
   (differentiate(current) in [1.75..2.25])}

template FloatStrm linearSlope_0_10@t:={                                 4
   @[0.0..10.0]:= linearSlope@t };
```

Furthermore, we would like to address the temporal segmentation of the signal. That is, after the part of linear increase the signal will remain nearly constant. We

---

[3] The introduced syntax is in fact an anticipation of means we will systematically introduce later.

may revert to the heuristic from above in order to characterize the signal's shape but we are not able to address the sequential split up, which is determined by the validity of properties. To address the activation and deactivation of time scopes w.r.t. to the evaluation results of templates applied before, we need to refer to the begin of the phase a signal, a property is valid for, and the respecting end of the phase. During on-line analysis the *carry until* construct in CTTCN-3 already provides means to realize the detection of such phases. In this article we concentrate on templates and provide a declarative approach. That is, we introduce the function *start of* and *end of* to address the points in time that represent the beginning and the end of the valid phase of a predicate (i.e. template).

Listing 4. Start and End Marks

```
sof(linearSlope)                                                      1
// start time, from where the property holds for the stream
eof(linearSlope)
// end time of this property, after which the property            4
// doesn't hold anymore
```

By use of such functions we can now define how properties depend and evolve:

Listing 5. Heuristic Example

```
template FloatStrm   constantSlope@t:={                             1
  (differentiate(current) in [−0.1..0.1])}

template FloatStrm linear_and_ConstSlope@t:={                       4
  @[0.0..10.0]  := linearSlope@t ,
  @[eof(linearSlope)+1..eof(linearSlope)+5]:= constantSlope};
```

It is obvious, that the provided means could be extended to achieve a proper description of the expected velocity curve in our automotive example. The crucial ingredients could be identified as

- templates on streams, which mimick properties resp. predicates of signal outcomes,
- time scope restrictions of such templates, and
- start and end markings of the durations limits.

In this article we will mainly focus on the necessary extensions of templates and show how they naturally integrate these with CTTCN-3.

4   Evaluation of Signals

A signal property is a formal description of certain defined attributes of a signal. This subsumes the signal value at a certain point in time, the increase and decrease of a signal, or the occurrence of extrema. Table 1 shows a selection of *basic properties* adopted from [3].

Table 1. Signal Properties

Property Name	Characteristic	Description	Locality
Signal Value	value = *exp*	the signal value equals *exp*	local
	value in [*range exp*]	the signal value is in [*range exp*]	local
Value Change	no	a constant signal	frame-local
	increase	an increasing signal	frame-local
	decrease	an decreasing signal	frame-local
Extremal Value	minimum	the signal has a local minimum	frame-local
	maximum	the signal has a local maximum	frame-local
Signal Type	step-wise	a step function	global
	linear	a partially linear signal	global
	flat	a partially flat signal	global

While the actual signal value is a property that is completely *local* (i.e. it is quantifiable without the history of the signal) the other properties are only allocatable when the predecessor values are considered. The latter is named *frame-local*, when the history can be limited to a certain frame and *global* when not. Local properties are adequate for on-line analysis in any case, frame-local properties are adequate, but only in consideration of the frame size. Large frames may constrain the real time capabilities of the test environment. Global properties are normally not meaningful applicable for on-line analysis because they depend on the complete signal. In this article we confine ourselves to local and frame-local properties.

To address frequencies, monotony and the exact amount of decrease or increase a signal has, we introduce the notion of preprocessing functions. A preprocessing function obtain a signal as input and provides a transformed signal as output. In systems engineering multiple meaningful preprocessing functions (e.g. derivation, high-pass filters, fourier transformation etc.) exist. In this article we only consider the derivation of a signal.

Finally, we distinguish basic properties that address one and only one of the characteristic from Table 1 and complex properties that address a combination of characteristics. Complex properties are constructed by use of logical connectives (e.g. negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), and implication ($\rightarrow$)). Moreover we aim to address the temporal evolution of a signal along the time axis. Hence, the specification of temporal order and temporal dependencies are necessary.

5   The *C*TTCN-3 Solution

*C*TTCN-3 already provides a limited set of means to check the properties of a signal. Signals are represented as streams and predicates that check properties, which are specified by use of so called stream templates. The respective concepts

are introduced in [12] in detail and summarized in 2.4. To systematically meet the requirements from above these means have to be enhanced.

– To provide transformations that are needed for pre-processing of streams (see Section 5.2) we introduce so called predefined transformation functions on streams and show how they integrate in the definition of stream templates.
– To model templates that address the properties of streams (as well as for their pre-processed derivation) we propose to enhance the syntax of template definitions. We introduce the notion of a *predicate expressions* to closely specify values and *time scopes* to restrict the application of a predicate in time. The original form of a stream template definition is short form of the one we introduce in this paper.
– For the logical and temporal combination of predicates we will introduce the construction of complex templates by use of logical connectives and the ability to trigger on the activation and deactivation of templates.

We start with the definition of predefined transformation function to realize the pre-processing of streams to be analyzed. Afterwards, we emphasize on the construction of complex templates (i.e. on time scopes, predicate expressions and the syntactical integration of transformation functions in the definition), and on the specification of temporal dependencies between templates.

5.1   Pre-processing Functions

We propose to specify the basic pre-processing functions as so called predefined functions. Predefined functions are defined as part of the core language [4] and are meant to be provided by the CTTCN-3 runtime environment.

The function $differentiate$ returns the first order derivative of a signal. In CTTCN-3 signals are represented as streams. The derivative $s'{=}differentiate(s)$ of a given source stream s is — similar to the source stream — defined by a structure $s' := (\Delta, \langle m'_k \rangle)$ where Δ represents the sample time, $\langle m'_k \rangle$ a sequence of values, and $s' \in (DSTRM)_T$. The sequence of values is defined as a left side derivative:

$$\langle m'_i \rangle := \{m'_1, m'_2, m'_3, ..., m'_k\}$$
$$and$$

$$m'_i = \begin{cases} 0 & when\ i = 1 \\ \frac{m_i - m_{(i-1)}}{\Delta} & when\ i > 1 \end{cases}$$

Please note that $m'_1 = 0$ due to the fact that m_i is not defined for $i = 0$. In CTTCN-3 the function $differentiate$ is specified with the following signature.

differentiate(numeric_stream_type value**)** numeric_stream_type

5.2   The General Setup of Stream Templates

While Section 2.4 provides a short overview over the notion of stream templates like they are already defined in [12], this section revises the initial design and provides a much more detailed insight in the underlying concepts. We start with the description of the general setup of a stream templates. On basis of this we will systematically introduce new features to enhance the expressiveness of stream templates to become a powerful instrument for the evaluation of system response in hybrid system testing.

Nevertheless, as mentioned before, a template generally consists of a *time scope* and a *predicate expression*. The time scope constitutes the validity of the predicate in respect to timing. The predicate expression constraints the value side of a stream.

Listing 6. Stream Templates

```
template FloatStrm t4@:={
  @[0.0..30.0]    := [0.0..55.0]};                           3

template FloatStrm t5@t:={
  @[0.0..10.0)    := [2*t],
  @[10.0..18.0)   := [19.0..21.0],                           6
  @[18.0..28.0)   := [20-(0.2*t)]};
```

The templates in Listing 6 consist of time scopes (at the left hand side e.g. `[0.0..30.0]`) and predicates expressed by values or value ranges (at the right hand side e.g. `[0.0..55.0]`). The predicates address the evolution of signal values only, which is obviously not enough to properly meet the requirements from Section 3.

Moreover a template may be defined segment-wise, that is, it may have different predicates for different segments of time, each defined by time scopes that precede the respective predicate (see *t4* in Listing 6). A segment definition may override a precedent segment definition when the respective time scopes overlap.

Predicates: A predicate is dedicated to characterizes the values of a stream on different levels of abstraction. In [12] we confined the notion of predicates to be simple relational expressions that are expressed by values or value ranges (e.g. `[0.0..55.0]` or `[20-(0.2*t)]` in Listing 6). In this article we enhance the notion of predicates to be more efficient in terms of signal properties and introduce the notion of *predicate expressions* (i.e. more complex relational expressions, templates itself and the logical composition of templates and relational expressions).

Time Scopes: The application of a time scope restricts the evaluation of predicates in time. It consists of a start event $\phi_{start} \in \Sigma$, that activates the evaluation of a predicate and an end event $\phi_{end} \in \Sigma$ that deactivates the evaluation.

Moreover we consider a timing function $\tau_\phi : \Sigma \to \mathbb{R}$ that returns the point in time when an event has occurred. Please note, for events the accuracy of the timing function is restricted by sampling. Hence, events are considered time-consuming and lasts for exactly one step size.

Concerning time scopes we distinguish between templates having a *global time scope* and templates having a *local time scope*. A template has a global time scope, when the time scope specification is omitted or when $\tau_\Sigma(\phi_{start}) = 0.0$ and ϕ_{end} does not occur (e.g. `@[0.0..infinity]`). A time scope is identified local when the time scope defines a finite time period or when $\tau_\Sigma(\phi_{start}) > 0.0$ (e.g. `@[0.2..10.0]`). The syntactical structure to denote time scopes is similar to the structure of value ranges already defined in Continuous TTCN-3[4].

5.3   Evaluation of Templates

The evaluation of streams is carried out by the application of a template to a stream or a stream port. The result of a stream evaluation is affected by the time scope and the predicate of the applied template as well as by the application statement. While *match* initiates a global evaluation of a stream, the *sense* operator allows the sample-wise evaluation (see Section 2.4).

Concerning the calculation of match and sense results, we propose a *tolerant evaluation* of templates. A tolerant evaluation only checks the defined time scope of a template. Hence, a template with a local time scope is evaluated to boolean values *true* as long the analysis affects samples that are outside the template's time scope. Regarding samples that are covered by the time scope, the result of the evaluation is determined by predicate. That is, it yields *true* when the predicate matches and *false* when the predicate does not match. Let us consider $r \in \mathbb{B}$ to be the result of a template application to a stream $s \in (DSTRM)_T$. Moreover we define $\chi_p : T \to \mathbb{B}$ the evaluation of a stream value at a certain point in time t by a predicate p. We define tolerant evaluation with:

$$r@t = \begin{cases} \chi_p(s@t) & when\ t \in [\tau_\Sigma(\phi_{start}), \tau_\Sigma(\phi_{end})] \\ true & else \end{cases}$$

Please note, the tolerant evaluation of templates is caused by the match or by the sense operation and is not a property of the templates itself. Thus, complex predicates that themselves may consists of multiple embedded templates are internally calculated in a strict mode, that is the undefined segments remain undefined. Tolerant mode is only used when the outermost template is applied to a stream.

---

[4] That is, the outer bound of a time scope is denoted by "[" or "(" and ")" or "]". With "(" we define a left side open time scope, that is the occurrence time of an *event* itself is not included in the time scope. With "[" we define a left side closed time scope that includes the occurrence time of *event*. The meaning of ")" or "]" is analogue.

5.4   Complex Predicates

Unlike the original version of CTTCN-3 the revised version provides predicate expressions. A complex predicate may consists of:

- relational expressions,
- templates or template references, and
- templates or template references connected by logical connectives.

We start with the presentation of how relational expression integrate in our conception of predicates and continue with an explanation on how already defined and named templates can be used and combined to form more complex predicate.

Relational Expressions: The original form of a stream template comprise a predicate that consists of a simple relational expression (i.e. a stream value equals a template value or is in a range of values). The subject of predication is naturally the (current) stream to which the template is applied (i.e. by means of a CTTCN-3 match or sense statement). If we intend to use pre-processing functions inside the definition of templates, the subject of predication may not be the current stream under analysis but one of its pre-processed derivation. To be able to distinguish between different subjects we propose to explicitly denote a subject and to provide means to relate a given subject to a value predicate (e.g. a value expressions or a range expression). Precisely because a subject is always defined as a transformation on the current stream, we need a symbol that represents the access to the current stream and that can be used inside a template definition.

- Hence, we introduce the keyword *current* to represent the stream the template is currently applied to, and
- we introduce the operators "in" and "$=$", that relate a subject (e.g. current or pre-processed derivations of current) to concrete value expression.

The operator "$=$" relates a given subject (Listing 6 the subject *current*) to a concrete value or a stream definition. The operator "in" does the same for ranges.

The significance of the new statements become clear regarding the templates $t6$ and $t7$ from Listing 7. Both integrate the application of the pre-processing function $differentiate$. While template $t4$ or $t5$ in Listing 6 can only be used to check whether the values of a stream are in a certain range, template $t6$ can be used to check whether the values of the derivation of a stream are between 1.75 and 2.25 and template $t7$ can be used to do similar for the second order derivation of a stream[5].

---

[5] The brackets that clasp around the predicate expressions are optional and are only used to provide readability. Thus, the syntax of the original CTTCN-3 stream template constructs can be considered as a short form of the new constructs exemplified in Listing 7.

Listing 7. Application of Pre-processing Functions

```
template FloatStrm t6@t:={
  @[0.0..10.0] := differentiate(current) in [1.75..2.25]};
                                                                    3
template FloatStrm t7@t:={
  @[0.0..300.0] :=
    differentiate(differentiate(current)) in [-1.0..1.0]};    6
```

The Composition of Predicates: Besides the specification of relational expressions we allow the construction of templates by means of already defined templates and by logical expressions, which itself consist of logical connectives (i.e. and, or, not, and implies), templates, and relational expressions. Listing 8 presents the composition of templates to ensure a certain increase of a signal and also checks the allowed value domain.

Listing 8. Defining Complex Templates by Applying Logical Connectives

```
template FloatStrm t8@t := {t5 and t6};
```

Please note, time scopes of enclosed template definitions remain valid. This holds for references to individual templates as well as for logical expressions. Nevertheless, we allow the restriction of enclosed time scopes by the application of a new time scope for the enclosing template. Lets take a simple example[6]. Template $t10$ in Listing 9 addresses the already time scoped template $t9$ and restricts the resulting time scope to @[0.0..1.0]. In contrast to that, the enlargement of time scopes is not possible, thus the absolute time scope of $t11$ is not @[0.0..10.0] but @[0.0...6.0].

Listing 9. Reusing Time Scopes

```
template FloatStrm t9@t:= {@[0.0..6.0]:=0}

template FloatStrm t10@t:= {@[0.0..1.0]:= t9};            3

template FloatStrm t11@t:= {@[0.0..10.0]:= t9};

                                                          6
template FloatStrm t12@t:= {@[2.0..10.0]:= t9};
```

Please also note, that the time expressions that are defined inside the embedded template will be synchronized with the activation of the enclosing template. That is, the absolute time scope of template $t12$ lasts from 2.0 to 8.0.

Due to the fact that time scopes are preserved, templates and logical expressions, which contain time scoped templates, already provide the ability to specify the temporal evolution of complex properties. Nevertheless, we propose a carefully reuse of time scoped templates to not get lost in complexity.

---

[6] As from now we leave the automotive example, we will come back to it later.

5.5   Complex Time Scopes

So far, time scope definitions rely on time events that are local to the template definition only. To become more flexible in respect to the definition of time scopes, we introduce flexible time scopes and the definition of time scopes that are bounded by the result of template evaluations, that is the activation and deactivation of a templates may rely on the evaluation of other templates.

Flexible Time Scopes: Introducing flexible time scopes we provide the ability to formulate a more flexible beginning and ending of a time scope. The bounds of a flexible time scope are denoted as a range of possible time values. That is, for both, the beginning and ending of a scope, two events are denoted. The lower bound event denotes the first time point that is allowed for beginning or ending and the second event denotes the last possible time point that is allowed for beginning or ending. Listing 10 presents two examples.

Listing 10. Flexible Time Scopes

```
template FloatStrm t13@t:={
  @[[0..4]..8] := differentiate(current) = 0};                2

template FloatStrm t14@t:={
  @[[0..4]..[8..10]] := t9};                                  5
```

The template $t13$ is activated between `[0..4]` and deactivated at 8. Applied to a stream it evaluates to *true* when the predicate (`differentiate(current) = 0`) at least is valid between 4 and 8. Please note, the time scope of embedded templates affects the evaluation of their enclosing templates, when they obey flexible time scopes, in a certain manner. Template $t14$ is activated between 0 and 4. The enclosed template (see Listing 9) exhibit a time scope with a length of 6 seconds (`@[0..6]`). The absolute time scope of the enclosing template depends on its actual activation. When it is activated between 0 and 2 the length of the absolute time scope is completely determined by the enclosed template. When the outer time scope is activated later, the enclosing time scope weakens the condition for deactivation (by `[8..10]`) and hence the possible duration of stream evaluation.

Dependent Time Scopes: With the means provided in the last two subsection we are already capable to activate and deactivate templates by means of the time scope of other templates. Nevertheless, the activation is directly connected to time. This subsection provides means to relates the definition of a time scope to the validation of templates. For this purpose we introduce the functions:

- *start of* or short `sof(template)` that fires an event when a template is evaluated to true for the first time and
- *end of* or short `eof(template)` that fires an event when a template was already true and either is evaluated to false or is deactivated.

Hence, we can define the activation of a template in dependence on other templates. Listing 11 shows such an example. Template $t17$ is activated when $t15$ switches becomes invalid and is deactivated when $t16$ becomes invalid. To determine the sof() and eof() events the strict evaluation of the templates $t15$ and $t16$ is necessary.

Listing 11. Temporal Order

```
template FloatStrm t15@t:=0;
template FloatStrm t16@t:=10;

                                                                        3
template FloatStrm t17@t{
  @[eof(t15)..sof(t16)]:=  sin(t)};
```

Moreover, we can combine the notion of flexible time scopes with the notion of dependent time scopes and pick up the example from Listing 5 and provide a more flexible version in Listing 12. Template $t20$ first checks for the phase with linear slope and expects the constant phase to start at least one second and at most 2 seconds after the first phase has ended.

Listing 12. Heuristic Example II

```
template FloatStrm    constantSlope@t:={                                 1
  (differentiate(current) in [−0.1..0.1])}

template FloatStrm t20@t:={                                              4
  @[0.0..10.0] := linearSlope@t ,
  @[[(eof(linearSlope)+1.0)..(eof(linearSlope)+2.0)]..
    (eof(linearSlope)+6.0)] := constantSlope@t};                         7
```

6 Summary and Outlook

In this article we have discussed the application of predicates to characterize the properties of a signal. By means of a simple scenario from the automotive domain, we have illustrated the concepts that are needed to properly define such predicates. Moreover, we provided a list of properties to be checked and examined their adequacy for the on-line analysis of system reaction, that is the analysis during test runtime. The second part of this article provides a simple integration of the introduced concepts to CTTCN-3. We could show how signal predicates can be realized by means of CTTCN-3 stream templates. Moreover we have provided the necessary syntactical add-ons to denote complex templates, that is templates that are build upon other templates, and to model the temporal dependencies between template invocation.

More effective means like the introduction temporal logic operators (e.g. globally, exists, until, release etc.) and the specification of dependencies of templates that relates the properties of different signals to each other will be subject of further research.

References

[1] Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In: Hybrid Systems, pp. 209–229 (1992)

[2] Broy, M.: Refinement of Time. In: Bertran, M., Rus, T. (eds.) Transformation-Based Reactive System Development, ARTS 1997, TCS, pp. 44–63 (1997)

[3] Gips, C., Hans-Werner, W.: Proposal for an Automatic Evaluation of ECU Output. In: Dagstuhl-Workshop MBEES 2007. Modellbasierte Entwicklung eingebetteter Systeme, Braunschweig, GER (2007)

[4] 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)

[5] 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)

[6] 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)

[7] Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HYTECH: A Model Checker for Hybrid Systems. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 460–463. Springer, Heidelberg (1997)

[8] Joshi, A., Heimdahl, M.P.E.: Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP Bd. 2005. LNCS, vol. 3688, pp. 122–135. Springer, Heidelberg (2005)

[9] Lehmann, E.: Time Partition Testing Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen. Berlin, TU-Berlin, Diss. (2004)

[10] Lim, J.S., Oppenheim, A.V.: Advanced topics in signal processing. Prentice-Hall, Inc., Upper Saddle River (1987)

[11] Schieferdecker, I., Bringmann, E., Grossmann, J.: Continuous TTCN-3: testing of embedded control systems. In: SEAS 2006. Proceedings of the 2006 international workshop on Software engineering for automotive systems, pp. 29–36. ACM Press, New York (2006)

[12] Schieferdecker, I., Grossmann, J.: Testing Hybrid Control Systems with TTCN-3, An Overview on Continuous TTCN-3. In: TTCN-3 STTT (2008)

[13] Sims, S., DuVarney, D.C.: Experience report: the reactis validation tool. In: Hinze, R., Ramsey, N. (eds.) ICFP, pp. 137–140. ACM, New York (2007)

[14] Zander-Nowicka, J., Schieferdecker, I., Pérez, A.M.: Automotive Validation Functions for On-line Test Evaluation of Hybrid Real-time Systems. In: IEEE 41st Anniversary of the Systems Readiness Technology Conference (AutoTestCon 2006) (2006); IEEE Catalog Number: 06CH37750C, ISBN 1-4244-0052-X