2009 International Conference on Software Testing Verification and Validation

Testing Embedded Real Time Systems with TTCN-3

Juergen Grossmann, Diana Serbanescu, Ina Schieferdecker

{juergen.grossmann,diana.serbanescu,ina.schieferdecker}@fokus.fraunhofer.de

Abstract

The problems of testing software based systems that, like automobiles, steadily increase in complexity are still not solved. To cope with the requirements and complexities of today's systems, adequate test solutions are needed, which at least feature a minimum of flexibility, reusability and abstraction. The Testing and Test Control Notation TTCN-3 is a test specification language, which was originally developed to meet the requirements of testing telecommunication systems. The language is modular, well-structured, standardized and supports testing of communicating systems. However, the correctness of a large number of embedded systems can not be assessed by checking functional requirements only. In addition to that non-functional requirements, especially time related input-output behavior, have to be considered. The current version of TTCN-3 has only limited capabilities for testing such non-functional properties. To overcome these limitations we will extend TTCN-3 with a small set of specific language means that are dedicated to check real-time properties of embedded system. We will explain the syntax and semantics of the new constructs, compare our solution with the expressiveness of standard TTCN-3 and, as a proof of concept, provide a small example from the automotive domain that particularly motivate the use of TTCN-3 in the context of AUTOSAR.

1 Introduction

The development of modern vehicles went through a major change regarding the realization of innovative functions and technologies in recent years. In former times, improvements were based on mechanical and electrical systems in particular. Nowadays, innovations are realized particularly by electronic systems and software-driven functions. Software systems in the automotive industry typically are embedded, distributed and—caused by the different areas of application (telematics, infotainment, power control, comfort electronics)—heterogeneous. A modern vehicle comprises between 30 and 80 different controllers, which communicate with one another over different bus systems. A comprehensive and systematic quality assurance of such complex systems is—still for the automotive industry, which is technologically and methodically very well equipped—a large challenge.

The quality assurance of software-intensive systems in the automotive industry is still characterized by high manual portions (i.e. a low level of automation) and a multiplicity of often proprietary test systems and test platforms. The latter rely on diverse description and specification techniques, that are only weakly formalized and normally not designed for the exchange between different test systems. Particularly for the automotive industry, which is characterized by strongly distributed development processes, this is not a good basis.

In contrast to this, the Automotive Open Systems Architecture (AUTOSAR) [1], a standardized platform for software architectures and software components, is currently established in the automotive industry. The standard is defined by a consortium, which is founded by a majority of European and international car manufacturers and their suppliers. In the telecommunication industry, which is confronted already for decades with the integration and interoperation of devices and systems based on telecommunication standards, a standardized test technology was established. TTCN-3 [4, 5, 6] provides a manufacturer-independent test environment consisting of modular service components (e.g. test management component, test logging component, platform adapters etc.) and a formalized test specification language. Meanwhile, the advantages of this technology were recognized by the automotive industry. Thus, AUTOSAR decided to use TTCN-3 in the context of functional tests for basic software modules.

With the next AUTOSAR release (release 4 in 2009), AUTOSAR will provide enhanced support for the definition of real-time requirements and timing (AUTOSAR timing specification). For testing these requirements, the test environment and the respective test language must support means to adequately express timing and real-time requirements. Although TTCN-3 is already used to test AUTOSAR basic software components, it provides only

978-0-7695-3601-9/09 $25.00 © 2009 IEEE
DOI 10.1109/ICST.2009.37

limited means for testing timing properties. Presenting real-time enhancements for TTCN-3, we will show how TTCN-3 can be systematically extended to support real-time testing more adequately and can as such provide a strong basis for AUTOSAR testing. The paper starts with a statement of problem (Section 2). Afterwards, we introduce the syntax and semantics of the newly introduced TTCN-3 constructs (Section 3). Section 4 shows the main differences between our approach and standard TTCN-3 and Section 5 provides a case study that addresses real time testing problems from the automotive domain [1].

2 Statement of Problem

Although there are multiple approaches in academia that address the subject of real-time testing [9, 10, 2], none of them were adopted by the industry in a larger scale. The approach described in this paper is based on the widely accepted test specification language TTCN-3. We aim to identify the weaknesses of the existing language constructs with respect to real-time testing and provide carefully selected new additions to the language that are meant to overcome the identified limitations. The additions cover aspects like measuring and verifying timing information of the tested real-time system as well as controlling the timing of the test system itself. They were developed as a consolidation of former approaches [3, 11, 12] that either provide only offline evaluation capabilities [3, 11] or introduce language constructs that are too complex to be adopted in the standard.

2.1 Real-Time Testing Requirements

Testing involves any activity aiming at evaluating attributes or capabilities of programs or systems, and finally, determining if they meet the requirements. Functional black-box testing is purely based on the requirements on the system under test (SUT) and mainly used for integration-, system- and acceptance-level testing.

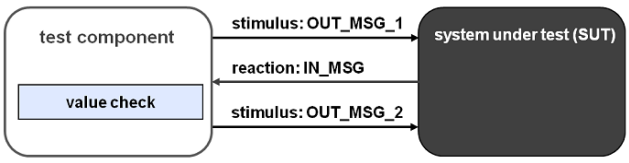

Figure 1. Simple functional black-box test

Figure 1 shows the setup for a simple functional test that does not consider timing requirements. A test component is used to stimulate the SUT. The outputs of the SUT are captured by the test component and matched against predefined message templates. We distinguish between three different situations:

1. message and template coincide, the tested requirement is considered to be fullfilled, and the test is passed,
2. message or template do not coincide and the test fails,
3. no message arrives and the test fails.

Listing 1 shows a test case representing these situations in TTCN-3.

Listing 1. Simple black-box test with TTCN-3

```
timer t;                                                    1
p_out.send(OUT_MSG_1);                                      2
t.start(TIMEOUT);                                           3
alt {                                                       4
 [] p_in.receive(IN_MSG) {p_out.send(OUT_MSG_2)}            5
 [] p_in.receive {setverdict(fail)}                         6
 [] t.timeout {setverdict(fail)}                            7
}                                                           8
setverdict(pass);                                           9
```

In addition, real-time systems have to respect special requirements for timing [7]. Moreover, functional requirements are often directly connected to the timing of the messages. Thus, checking the message values and the message order is not sufficient here. A test component must be able to check whether a message has been received in time and must be able to control the timing for the stimulation.

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

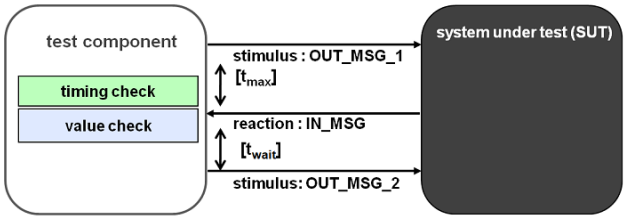

Figure 2. Black-box test with time restrictions

An example test scenario with timing requirements is given in Figure 2. With t_{max} we time constraint for the SUT that indicates the maximum limit in which the reaction to the stimulus should be received by the test component; the second is t_{wait}, which indicates the time that should elapse

[1]The results presented in this article are an outcome of the projects TEMEA (www.temea.org) and DMint, which are part-financed by the European Union.

Listing 2. Timing test with TTCN-3

```
timer t;                                                    1
p_out.send(OUT_MSG_1);                                      2
t.start(t_max);                                             3
alt{                                                        4
  [] p_in.receive(IN_MSG){                                  5
    t.start(t_wait);                                        6
    t.timeout;                                              7
    p_out.send(OUT_MSG_2);};                                8
  [] p_in.receive {                                         9
    t.stop;                                                 10
    setverdict(fail)};                                      11
  [] t.timeout(){setverdict(fail)}                          12
}                                                           13
```

at the test component side, between the receiving of the first reactions from the SUT and the sending of the second stimulus to the SUT. The test logic for this example is given by traditional TTCN-3 source code in Listing 2.

2.2 Description of TTCN-3 Weaknesses

Obviously, traditional TTCN-3 already provides means to describe test cases that respect timing. Nevertheless, today's TTCN-3 test systems lack real-time support. The solution denoted in Listing 2 is based on TTCN-3 timers and constructs (e.g. the ***alt*** statement) that follow the TTCN-3 snapshot semantics. The snapshot semantics by itself is not real-time and the concept of a timer was originally not intended to specify real-time properties, but conceived only for catching, typically mid- or long-term, timeouts. Using timers for the specification of real-time properties is often clumsy and, due to the lack of a real-time semantics for TTCN-3 snapshots, not precise enough.

- The use of a timer is directly affected by the TTCN-3 snapshot semantics and by the order in which receive and timeout statements are arranged in the alt statement. The measured time point is in fact not the time point of message reception (i.e. the time a certain message has entered the input queue of the test system), but the time point of the evaluation of the queues by the test program. This kind of time measurement is not exact.

- TTCN-3 in general makes no assumptions about the duration for taking and evaluating a snapshot which may vary, depending on the content of the input queues and different implementation solutions.That is, time is consumed for the encoding and decoding of messages as well as for the process of message matching. This time consumpotion is neither controllable nor assessable by the tester.

Listing 3. Timed black-box test

```
var datetime stamp;                                         1
p_out.send(OUT_MSG_1) -> timestamp stamp;                   2
alt{                                                        3
 [] p_in.receive(IN_MSG) -> timestamp stamp{                4
      p_out.send(OUT_MSG_2) at (stamp+t_wait)}              5
 [] p_in.receive(){setverdict(fail)}                        6
}                                                           7
break after (stamp+t_max){setverdict(fail)}                 8
```

- In TTCN-3, timer values (i.e. time) are represented by floating point numbers. Floating point numbers show however inaccuracies due to their limitations of digits. Thus, arithmetical operations may result in rounding errors.

- Last but not least, TTCN-3 is itself not real-time. The tester has no control of what happens between two instructions. For example, if a garbage collection starts between the receiving of a message and the start of a timer the measurement of time is not exact. These kinds of inaccuracies are non predictable.

The concepts that are introduced in this paper are meant to overcome these limitations of traditional TTCN-3. Listing 3 informally introduces a TTCN-3 embedded implementation for the test depicted in Figure 2. The timestamp operator (`-> timestamp`) writes the accurate point of transmission time of the message `OUT_MSG_1` into the variable `stamp`. The variable `stamp` is used for the allocation of the ***break after*** operator `break after(stamp+t_max)`, which is meant to interrupt the execution of the ***alt*** statement and sets the test verdict to ***fail*** when the time limit has passed. The ***at*** operator in line 5 specifies the exact point of message delivery for message `OUT_MSG_2`.

3 Real-time Extensions for TTCN-3

To concisely describe the capabilities of our approach we rely on a simple time model $t \in \mathbb{R}^+$ based on positive real numbers and introduce a simplistic model $\mathcal{TS} = \{P, Q, C, M, TP, OP\}$ of a TTCN-3 test system. The latter consists of:

- a set P of ports to communicate with the environment,

- a set Q of input queues to organize the order of incomming messages,

- a set C of clocks that can be used to measure time and to simulate TTCN-3 timers,

- a set M of messages,

- a set TP of predicates that are used to characterize the properties of incoming messages, and

- a set $OP = \{snap, check, enqueue, dequeue, encode, decode, match\}$ of time-consuming operations that are necessary to organize the handling of messages at ports.

In general, a TTCN-3 test system communicates with its environment via ports $p_i \in P$. In the following we are mainly interested in estimations on timings of the test system's basic operations OP. As already mentioned, these operations consume time. The sending of a message is affected mainly by the duration of the message encoding. The reception and evaluation of messages is more complex. Each incoming port is associated with an input queue $q_{p_i} \in Q$. When a message $m \in M$ arrives at a port p_i, it will be added to the queue (i.e. the operation $enqueue : M \rightarrow Q$ is triggered).

The evaluation of ports (i.e. the evaluation of the associated queues with TTCN-3 receiving statements) and clocks (i.e. TTCN-3 timers) is organized with respect to the so called TTCN-3 snapshot semantics. Let S be the set of system snapshots. A snapshot $s_j \in S$ associates a distinct point in time $t \in \mathbb{R}^+$ with an immutable view on the test system's state (i.e. the contents of the input queues and the state of the clocks). It is taken by the operation $snap : \mathbb{R}^+ \times Q^{|Q|} \times C^{|C|} \rightarrow S$. Formerly, we refer to the snapshot's point of time by the function $t : S \rightarrow \mathbb{R}$ and to the system's state, which is stored in a snapshot, by indexing the respective queues and clocks with the snapshot (e.g. $q_{p_i}^{s_j} \in Q$ or $c^{s_j} \in C$). Moreover, the execution of TTCN-3 receiving statements refer to a consecutive series of snapshots and triggers the execution of the queue related operations (i.e. $check : Q \rightarrow \mathbb{B}$ and $dequeue : Q \rightarrow M$), the decoding operation (i.e. $decode : M \rightarrow M$) as well as of the evaluations of the message properties (i.e. $match : M \times TP \rightarrow \mathbb{B}$). We refer to a message, which is taken from the top position of a queue $q_{p_i}^{s_j}$ for the purpose of evaluation, with $m^{q_{p_i}^{s_j}}$.

Last but not least, our approach is based on a simple but effective scheduling assumption. The algorithm provides an interrupt, that enables the execution of $enqueue : M \rightarrow Q$ with highest priority whenever a message arrives at the test system. Moreover $enqueue$ itself is an atomic operation that can not be interrupted.

In the following we will introduce the syntax and semantics of the newly introduced TTCN-3 constructs. The syntax is defined by examples and a set of rules in Backus-Naur-Form (BNF) [8]. Semantics is given by referring to the model from above.

Listing 4. Datetime and timespan types

```
// declare datetime variables                    1
var datetime starttime, actualtime;              2
// declare timespan variables                    3
var timespan distance, maximum;                  4
// using predefined symbols                      5
starttime := startTimeTestcase;                  6
// using the now operator                        7
actualtime := now;                               8
```

3.1 Representation of Time

We refer to time points as positive real numbers $\mathbb{R}^+$. We consider our test system to have a base clock $c_0 \in C$ that is initialized and started with the start of the test system. The function $t : C \rightarrow \mathbb{R}^+$ returns the actual time value of the clock. Each additional clock is considered to be synchronized with the base clock c_0 and can be used to realize TTCN-3 timers. In order to ease the manipulation of time values we introduce two new abstract data types to TTCN-3.

- ***datetime*** designates time points that uniquely specify the timing of events. For ***datetime*** values we use a time model based on positive real numbers $t \in \mathbb{R}^+$ that are either measured directly or denoted using symbols that represent the time point of predefined test system events (e.g. start of the test case, see Listing 4).

- ***timespan*** designates time distances between different points in time (i.e. datetime values). It is used to represent the amount of time that passed between events. Timespan values can be formerly denoted by real numbers $t \in \mathbb{R}$. In TTCN-3, ***timespan*** literals are constructed by integer values multiplied with time constants that represent the basic time units (e.g. microseconds, milliseconds, seconds, hours, etc., see Listing 5).

In order to give the tester access to test system related timing events, some predefined symbols are introduced. The ***now*** symbol returns the current time value (i.e. $t \in t(c_0)$), the ***startTimeTestcase*** symbol returns the time point when the test case execution has started, and the ***startTimeComponent*** symbol returns the point in time when the test component execution has started.

Time values can be used in arithmetic and boolean expressions (see Listing 5) respecting a few simple rules: the subtraction or addition of two ***timespan*** values results in a new ***timespan*** value. Timespans can be multiplied or divided by floats or integers and the result should be evaluated to timespan. The difference between two ***datetime*** values results in a ***timespan*** value. To enable a seamless integration of time types, either with each other as well

Listing 5. Time releted expressions

```
// calculating with time values                                  1
distance:= now-startTimeTestcase;                                2
// literal of a timespan value                                   3
maximum:= 200*millisec;                                          4
// comparision of timespan values                                5
if(distance>maximum) {log("limit hurt") }                        6
```

Listing 6. Measurement of time

```
var datetime sendtime, receivetime;                              1
p_out.send(MSG_OUT) -> timestamp sendtime;                       2
p_in.receive(MSG_IN) -> timestamp receivetime;                   3
```

as with existing TTCN-3 types, we introduce conversion functions that provide conversions between ***timespan*** values, ***datetime***, ***string*** values and/or ***float*** values.

3.2 Measurement of Time

In our approach, the observation of time is directly connected to the reception and provisioning of messages $m \in M$ at communication ports $p \in P$. The time points of message reception and provisioning are automatically registered and stored as a property of the message. In our formal description we access these time values by $t : M \rightarrow \mathbb{R}^+$. In TTCN-3 a similar access is provided by so called redirections `-> timestamp datetimevar`. Timestamp redirections trigger the test engine to store either the arrival time of a message (i.e. the time when the message is enqueued) or the sending time (i.e. when the message is encoded and transmitted to the output port) to the given ***datetime*** variable. Redirections can be applied to all kind of TTCN-3 communication statements.

⟨**ReceiveStatment**⟩**::=**
⟨PortOrAny⟩ `"."` `"receive"`[`"("` ⟨ReceiveParameter⟩ `")"`]
[⟨FromClause⟩] [⟨PortRedirect⟩]

⟨**PortRedirect**⟩**::=**
`"->"`
((⟨ValueSpec⟩ [SenderSpec] | ⟨SenderSpec⟩) [TimeStampSpec])
| ⟨TimeStampSpec⟩

⟨**TimeStampSpec**⟩**::=**
`"timestamp"` ⟨Identifier⟩

3.3 Temporal Predicates

To control the timing of incoming and outgoing messages $m \in M$, we introduce temporal predicates. In general we distinguish between simple temporal predicates and complex temporal predicates. A simple predicate consists of one of the temporal operators ***at***, ***after***, ***before***, and ***within*** and their respective negation (e.g. ***not before***). A complex predicate is a combination of predicates that are connected by the logical operators ***or*** and ***and***.

⟨**TemporalPredicate**⟩**::=**
⟨SimplePredicate⟩[(`"and"` | `"or"`)⟨SimplePredicate⟩]

⟨**SimplePredicate**⟩**::=**
([`"not"`]⟨TemporalPredicate⟩) | ⟨Within⟩ | ⟨BeforeAfterAt⟩

⟨**Within**⟩**::=**
`"within""("`
(⟨DateTimeExp⟩ | ⟨TimeSpanExp⟩)
`"..."`
(⟨DateTimeExp⟩ | ⟨TimeSpanExp⟩)`")"`

⟨**BeforeAfterAt**⟩**::=**
(`"before"` | `"after"` | `"at"`) (⟨DateTimeExp⟩ | ⟨TimeSpanExp⟩)

A temporal predicate $tpr \in TPR \subseteq TP$ is an expression that specifies a set of time points $DT_{tpr} \subseteq \mathbb{R}^+$. Such a predicate matches a time point $t \in \mathbb{R}^+$ if and only if the time point is included in the set DT_{tpr}. However, using temporal operators we are able to specify temporal predicates and thus ranges of date time values, which form the set DT_{tpr} and restrict the availability of TTCN-3 statements with respect to time.

- The ***at*** operator takes a ***datetime*** value t and specifies a range of time points consisting of a single value $[t, t]$. The respective inversion ***not at*** defines the complement thus $[0, t)$ and $(t, \infty]$.
- The ***before*** and ***after*** operators are each parametrized with a ***datetime*** value t. They specify a range of time points consisting of the values $[0, t]$ in the case of before and $[t, \infty]$ in the case of after. The respective inversions are defined by $(t, \infty]$ and $[0, t)$.
- The ***within*** operator is parametrized with two ***datetime*** parameters (e.g. t_1 and t_2) that directly denote the boundary of a time interval, thus $[t_1, t_2]$. The negation of the ***within*** operator is defined by $[0, t_1)$ and $(t_2, \infty]$.
- The ***and*** and ***or*** operators are directly mapped to equivalent operations in set arithmetics $\cap$ and $\cup$.

Please note, in the context of simple temporal predicates we allow—besides ***datetime*** expressions—to use ***timespan*** expressions (e.g. `timespanval`) for the definition of time points (e.g. **`startTimeTestcase`**`+timespanval`). In this case, and only in this case, these ***timespan*** expressions are interpreted to denote the distance to the start of the test case and are automatically converted to the respective time points. Temporal predicates can be applied to receiving and sending TTCN-3 statements as well as to the ***alt***, the ***interleave*** and the ***call*** statements. However, temporal predicates yield different behaviour dependent on the statements they are applied to.

Listing 7. Control of application

```
p_out.send(MSG_OUT) at                                    1
  startTimeTestcase+300*millisec;                         2
```

3.3.1 Controlling Outgoing Communication

Applied to TTCN-3 sending statements (e.g. ***send***, ***call***, ***reply*** etc.), a temporal predicate ensures that the sending statement will be executed with respect to the predicate $tpr \in TPR$. Thus, the test system suspends the execution of the component until the first possible temporal match occurs, i.e. the actual time ($t_{now} = t(c_0)$) is included in the set defined by the predicate tpr (i.e. $t_{now} \in DT_{tpr}$). Afterwards, the component continuous its execution by dispatching the respective message or procedure call, etc. Please note, if the system is delayed, i.e. no temporal match is possible any more (i.e. $\forall_t : t \geq t_{now} \implies t \notin DT_{tpr}$), the test system indicates its delay by setting an error verdict and thus informs the tester that it hurts the real-time requirements that are necessary to accomplish the test.

The syntactical extension introduced for adding temporal predicates to TTCN-3 sending statement are exemplified by the send statement. The extensions for ***call***, ***reply***, and ***raise*** are defined in a similar way.

⟨**SendStatement**⟩**::=**
⟨Port⟩ `"."` `"send"` `"("` ⟨SendParameter⟩ `")"`
[ToClause] [TimeStampClause] [⟨TemporalPredicate⟩]

Listing 7 shows the application of the ***at*** operator to a send statement. The message will be dispatched exactly 300 milliseconds after the test case has started.

3.3.2 Verifying Incoming Communication

Applied to TTCN-3 receiving statements (e.g. ***receive***, ***getcall***, ***getreply***, ***trigger***, ***catch*** etc.), the temporal predicates serve as verification instruments. They are integrated in the TTCN-3 matching mechanism and directly influence the evaluation of the input queues. That is, a receiving statement with a temporal predicate attached (e.g. **`receive`**`(tpl)` **`before`**`(datetimeval)`), is only processed successfully when, like in ordinary TTCN-3, the message value conforms to the value template and, introduced in our approach, the reception time associated with the message, conforms to the temporal predicate. A message conforms to a temporal predicate if, and only if, its time of reception $t(m)$ is included in the time interval (i.e. $t(m) \in DT_{tpr}$) defined by the temporal predicate $tpr \in TPR$.

The syntactic extension for the receive statement is given below. The temporal operators can also be applied to

Listing 8. Verification of incoming message

```
// the message is valid when its value conforms to      1
// MSG_IN and is received in 300 milliseconds           2
// after the start of the test case                     3
p_in.receive(MSG_IN)                                    4
    within (startTimeTestcase                           5
            ..startTimeTestcase+300*millisec);          6
```

trigger, ***getcall***, ***getreply***, and ***catch*** statements.

⟨**ReceiveStatement**⟩**::=**
⟨PortOrAny⟩ `"."` `"receive"` [`"("` ⟨ReceiveParameter⟩ `")"`]
[⟨FromClause⟩] [⟨TemporalPredicate⟩] [⟨PortRedirect⟩]

Listing 8 illustrates a small piece of code where the expected message should conform to `MSG_IN` and as an additional restriction, it should be received between the indicated points in time. These points refer to the start of the test case and are calculated by adding timespan values.

3.3.3 Break Clause for Alt, Interleave and Call Blocks

Using the temporal predicates we are able to define the timing for outgoing message and verification of timings for incoming messages, but no timeouts for the absence of messages (see Section 2.1). Thus, we additionally allow the usage of temporal predicates as a temporal restriction for the snapshot related statements ***alt***, ***interleave*** and ***call***. The syntactical structure is called ***break*** clause and enables the execution of an alternative statement block when the associated temporal predicate matches. The syntax of the ***break*** clause is exemplified for the ***alt*** statement in the following.

⟨**AltStatement**⟩**::=**
`"alt"`⟨AltGuardList⟩
[`"break"`⟨TemporalPredicate⟩ [⟨StmtBlock⟩]]

The ***break*** clause restricts the evaluation of the input queues and thus the creation of snapshots $s_1, s_2, .., s_j$ with regard to timing. To be more precise, the predicate $tpr_{break} \in TPR$ defines the temporal exit condition for the evaluation of the set of alt-branches (i.e. on alternative expectations on the system under test) that are grouped by an ***alt*** (***interleave***, ***call*** etc.) statement and thus evaluated pseudo-concurrently. With respect to time control, each alt-branch can be modelled as an element $a_k = (q_{p_i,a_k}, tpr_{a_k}) \in ALTB \subseteq Q_{alt} \times TPR_{alt}$, where $Q_{alt} \subseteq Q$ are the queues that are addressed from inside the ***alt*** statement and $TPR_{alt} \subseteq TP$ are the temporal predicates applied to the queues. Let s_{cur} be the actual snapshot. A break clause associated with an ***alt*** statement is triggered either:

- when the message timing $t(m^{q^{s_{cur}}_{p_i,a_k}})$ is matched by an alternative a_k (i.e. $t(m^{q^{s_{cur}}_{p_i,a_k}}) \in DT_{tpr_{a_k}}$ holds) and

Listing 9. Additional timeout

```
p_out.send(MSG_OUT);                                          1
alt{                                                          2
  [] p_in.receive(MSG_IN)                                     3
     within (startTimeTestcase                                4
     ..startTimeTestcase+300*millisec){}                      5
}break after(startTimeTestcase+300*millisec){                 6
     setverdict(fail);                                        7
}                                                             8
```

as well matched by the temporal predicate of the break clause (i.e $t(m^{q^{s_{cur}}_{p_i,a_k}}) \in DT_{tpr_{break}}$), or

- when every possible future match of one of the alternatives will match the predicate of the break clause in either case (i.e. $\forall_{s_j \in S}, \forall_{a \in ALTB} : t(s_j) \geq t(s_{cur}) \implies match(m^{q^{s_j}_{p_i,a}}) \wedge t(s_j) \in DT_{tpr_{break}}$.

When the ***break*** clause is triggered the statement block of the ***break*** clause is executed and the afterwards the execution of the respective ***alt*** (***interleave***, ***call*** etc.) statement is suspended. The main difference between the ***break*** clause and ordinary TTCN-3 timeouts is the larger expressiveness of the temporal predicates and their direct reference to the timestamp mechanism.

4 Time Measurement by Comparison

In TTCN-3, the evaluation of the system's reaction is performed during test execution and is organized by the alt statement. In the following we analyze the runtime behaviour of the ***alt*** statement and we show some differences between standard TTCN-3 and our approach. The operational semantics of TTCN-3 [5] defines the semantics of the ***alt*** statement by means of flow graphs (see Figures 3).

The static structure of the ***alt*** statement consists of a finite number of alt-branches. An alt-branch itself is an element $a_k = (q_{p_i,a_k}, tp_{a_k})$. The evaluation of queues is realized by the snapshot semantics. Let S_{alt} define the set of snapshots that is taken during the execution of an alt statement and E_{alt} a set of events that denotes the exact occurrence of messages or the exact expiration of timers. Moreover, $t : E \rightarrow \mathbb{R}^+$ is a function that returns the exact time point of the event occurrence and the function $dur : CALL_{alt} \rightarrow \mathbb{R}$ returns the time consumed by the execution of one of the operations from OP.

In general, the creation of snapshots is event-triggered, that is snapshots are taken only when a message has received or a timeout has occurred. The distance between two snapshots is at least the duration that is needed to process the evaluation of all the messages that were received before the snapshot was taken. Let l be the number of alternatives

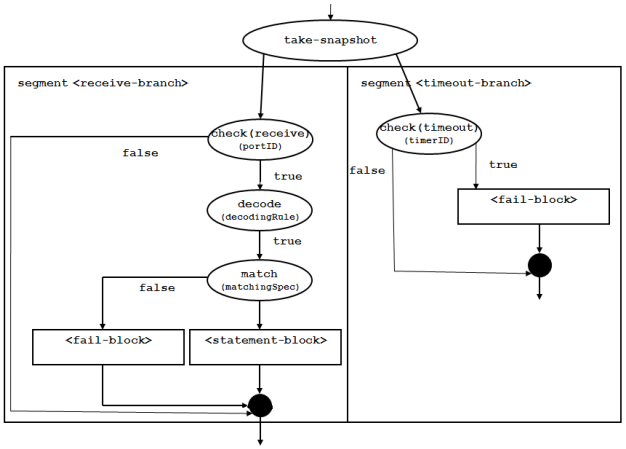

Figure 3. Flow graph of the TTCN-3 alt statement

in an ***alt*** statement. The worst case is given when each alternative addresses different queues and new messages have arrived at the top position of each queue:

$$worst(t(s_{n+1}) - t(s_n)) = \sum_{x=1}^{l} (dur(check(q^{s_n}_{p_i,a_x})) + dur(decode(m^{q^{s_n}_{p_i,a_x}})) + dur(match(m^{q^{s_n}_{p_i,a_x}}), tp_{a_x})) + dur(snap)$$

Figure 4 presents an exemplary scenario that shows the occurrences of events over time. With $s_n, s_{n+1} \in S_{alt}$ we denote two consecutive snapshots and with $e_0, e_1, e_2, ..., e_{k+1} \in E_{alt}$ we denote a number of consecutive events, where e_0 triggers s_n and $e_1, e_2, ..., e_k$ occur between s_n and s_{n+1}.

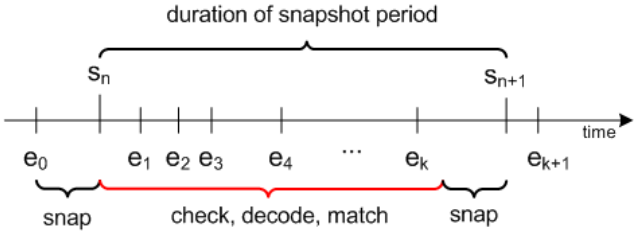

Figure 4. Snapshots over time

Please note, standard TTCN-3 does not provide dedicated timing information for events. Hence, the timing of events can only be estimated in relation to the occurrence of other events (e.g. $t(e_0) \leq t(e_1), t(e_2), ..$ etc.). Moreover, only events that occur in different snapshots are distinguishable with respect to time. In Figure 4 the event e_0 is iso-

lated and can be distinguished from all the following events by its presence in s_n. Effectively, we can at best estimate the timing of e_0 to be smaller than $t(e_1)$. The next snapshot s_1 is triggered by the next event e_1. In fact, it can not be taken until the evaluation of event e_0, that is the execution of $check, decode, match, snap$, has finished. Thus, it is possible that between two snapshots a number of additional events $e_1..e_k$ may occur. To assess the real-time properties of a system, we have to introduce a temporal order for these events $t(e_1) < t(e_2) < ... < t(e_k) < t(s_{n+1})$, which is not possible with the means of standard TTCN-3. In the end, we can only compare the occurrence of the events $e_1..e_k$ with an arbitrary event occurrence after $t(s_{n+1})$, thus we assess $t(e_{k+1}) \geq t(e_1)..t(e_k)$.

Unlike standard TTCN-3, our approach provides a runtime model where each event is time-stamped at its occurrence. The respective processing of receiving events and timeouts is an atomic operation and has the highest priority. That is, the processing is able to interrupt ongoing tasks like $decode$ and $match$ to provide a maximum accuracy for the time stamping of events. Although, the evaluation and assessment of ports in our approach is carried out with the same frequency and timing than in standard TTCN-3 (i.e. at the time points $t(s_n)$, $t(s_{n+1})$ etc.), we can provide a much more precise data basis to refer to the systems state. By means of the time stamping mechanism we able to reconstruct the state of queues and clocks at any time. Thus, instead of relying on the estimation $t(e_1) = t(e_2) = ... \leq t(s_{n+1})$, we can provide direct access to $t(e_k)$ for each relevant event. Moreover, we are able to directly refer to the timing of events by means of temporal predicates and thus provide a user friendly interface to assess timing properties of a message based communication.

5 Case Study

As a proof of concept, we provide an example that is typical for the automotive industry. Figure 5 shows the setup for a blinker system that is composed of a central controller unit (master) and several actuator components that serve the indicator lights on the front and the backside of the car.

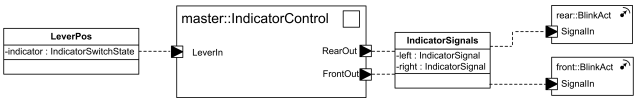

Figure 5. Model of the blinker system

Our simplified controller model has three ports: an input port (`LeverIn`) that receives the actual position of the lever, and two output ports (`RearOut, FrontOut`) that are used to communicate the indicator signals to different actuator components. Such an indicator system is usually realized as a distributed system. The central software component (master) is placed on a central ECU (e.g. SAM in the case of Mercedes or BCM in the case of VW) that usually hosts multiple applications. The actuator components can either be located on the central ECU (usually the ones for the front indicators) or on distributed controllers (e.g. rear controller, door controllers etc.). The functionality of our controlling component is supposed to be simple. It can be activated from the outside through the lever signal and—in response to that—it generates two output signals, one for the rear indicators and one for the front indicators.

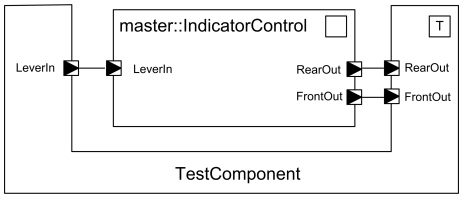

Figure 6. Test setup for the blinker example

The subject of testing is the timing of the signals as depicted in Figure 7. The timing aspects can be decomposed into the following three timing requirements:

- The outgoing signals should react on their activation at most `60ms` after the respective input signal has received. This requirement is tested by test case `tc_1` given in Listing 11.
- Both outgoing signals should behave synchronously with a tolerance of `30ms`. This requirement is tested by test case `tc_2`, which is given in Listing 12.
- The length of the indicator period should last `600ms` (`300ms ON` and `300ms OFF`) with a tolerance of `20ms`. This requirement is tested by test case `tc_3` given in Listing 13.

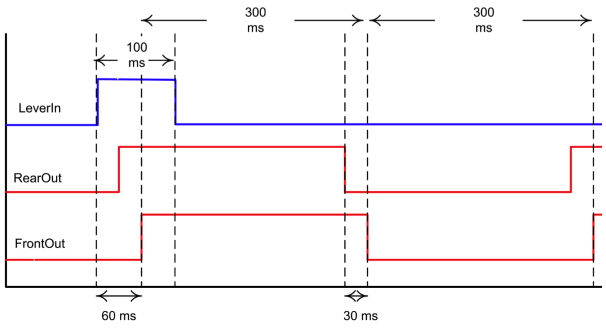

Figure 7. Signals and timing

For testing, we replace the actuator components and the input emitting component by a single test component. The

Listing 10. Test interface definition

```
module MainModule {
  type enumerated IndicatorSwitchState {
    OFF, LEFT, RIGHT}
  type enumerated IndicatorSignal {OFF, ON}
  type port LeverPos message {
    out IndicatorSwitchState}
  type port IndicatorSignals message {
    in IndicatorSignal}

  type component IndicatorTestComponent {
    port IndicatorSignals RearOut, FrontOut;
    port LeverPos LeverIn}
}
```

Listing 11. Initial activation of the indicator

```
testcase tc_1() runs on IndicatorTestComponent{
 var datetime t_sig;
 const timespan T_MAX = 60*millisec;
 // saving the time of sending the signal
 LeverIn.send(LEFT) -> timestamp t_sig;
 interleave{
   [] FrontOut.receive(ON){}
   [] RearOut.receive(ON){}
 } break after (t_sig+T_MAX){
   setverdict(fail); stop}
 setverdict(pass);
}
```

proposed configuration of the test system is visualized in Figure 6.

The TTCN-3 test interface is directly derived from the AUTOSAR models (see Figure 5 and 6). The respective TTCN-3 code that specifies the ports and messages is depicted in Listing 10. The values of the messages are specified by enumeration types. The switch signal (`LeverIn`) is specified as an outgoing signal and the indicator signals (`RearOut, FrontOut`) as incoming signals. We expect the system to produce messages and restrict ourselves to the verification of timing. A not responding system is handled by the break clause.

Test case `tc_1`, given in Listing 11, tests the activation of the indicator signals. After the `LeverIn` signal has switched to `LEFT`, we expect to receive an activation (`ON`) signal from `FrontOut` and `RearOut` on the associated ports. Both signals should reach the test system within 60 milliseconds after the activation signal (`LeverIn`) was sent. To check the respective timing we save the time point, when we sent the activation signal, in the datetime variable `t_sig`, using the timestamp mechanism (line 5). This point in time is used as reference to check the incoming data in the interleave construct. For each incoming message we check whether it has the correct value (e.g.

Listing 12. Synchronisation between signals

```
testcase tc_2() runs on IndicatorTestComponent{
  var datetime t_sig_r, t_sig_f, t_start;
  const timespan T_MAX = 30*millisec;
  const timespan T_FAIL = 90*millisec;

  LeverIn.send(LEFT)-> timestamp t_start;
  interleave{
   [] FrontOut.receive(ON) ->timestamp t_sig_f{}
   [] RearOut.receive(ON) ->timestamp t_sig_r{}
  }break after (t_start+T_FAIL) {
   setverdict(fail); stop}
  if (abs(t_sig_r-t_sig_f)> T_MAX){
   setverdict(fail)}
  else {setverdict(pass)};
}
```

`FrontOut.receive(ON)`) and whether the timing is correct (`break after (t_sig+T_MAX)`). Please note, the interleave construct ensures that the test system waits for both messages independent of their order. However, if the timing does not match the limits defined for the receive statements the ***break*** clause sets the test verdict to fail and finalizes the test case.

With the second test case (see Listing 12), we check the synchrony of the indicator signals. The timing of the edge change of the two indicator signals should not vary more than 30 milliseconds from each other. We verify this requirement at the moment when the signals switch from `OFF` to `ON`. Afterwards, they are supposed to be transmitted normally, just keeping the delay.

At first, the triggering signal from the test system side is transmitted to the SUT through the `LeverIn` port (line 6). After that, the two ports for the incoming signals are checked. If the signal on `FrontOut` is sensed, the timing of its reception is saved in the datetime variable `t_sig_f` using the timestamp instruction. Analogue actions are taken if the signal expected on the `RearOut` port is received. If the SUT does not send the expected messages, the evaluation is stopped after `t_start+T_FAIL` and the test case ends with the verdict fail (line 10,11). The respective value is stored in `t_sig_r`. The time difference between the two signals is calculated and checked based on these values (line 12).

In the third test case, we test the phase length of the indicator signals (see Listing 13). We expect that the output signals remain `ON` for 300 milliseconds with a tolerance of 20 milliseconds although the activating switch signal has turned to `OFF` again. The test case begins with activating the trigger signal on port `LeverIn`. As a reaction to this command, the SUT should activate signals on `FrontOut` and `RearOut`. When the signals are received, their time of reception is stored in `t_on_s1` and `t_on_s2`, respectively (lines 9,10). This is used to calculate time margins for the

Listing 13. Duration of the indicator phase

```
testcase tc_3() runs on IndicatorTestComponent{
  var datetime t_on, t_on_s1, t_on_s2;
  const timespan T_MAX := 300*millisec;
  const timespan TOL := 20*millisec;
  const timespan T_WAIT := 100*millisec;
  var timespan upper, lower;
  LeverIn.send(LEFT)-> timestamp t_on;
  interleave{
   []FrontOut.receive(ON) ->timestamp t_on_s1{}
   []RearOut.receive(ON) ->timestamp t_on_s2{}
  } break after (t_on+T_MAX) {
    setverdict(inconclusive), stop}

  lower:=T_MAX-TOL;
  upper:=T_MAX+TOL;

  LeverIn.Send(OFF) at t_on+T_WAIT;
  interleave{
   []FrontOut.receive(OFF)
     within(t_on_s1+lower..t_on_s1+upper){}
   []RearOut.receive(OFF)
     within(t_on_s2+lower..t_on_s2+upper){}
  } break after (t_on+upper) {
    setverdict(fail); stop}
  setverdict(pass);
}
```

deactivation of both signals. After 100 milliseconds, another control signal is sent to the SUT, indicating that the output signals should be turned off when the signal period expires. In lines 18-25 we check the respective system reaction with the predefined tolerances.

6 Summary

In this paper, we introduced a real-time test technology and methodology for the automotive domain. Based on a standardized test technology, these enable the exchange of test requirements, test cases, and test procedures between different parties involved in automotive systems development. Both methodology and technology are based on TTCN-3 and will contain several enhancements that adapt the already existing TTCN-3 standard to the needs of the automotive domain and especially to the emerging AUTOSAR standard. The paper started with the specification of real-time enhancements to TTCN-3 that provide systematic support for testing timing requirements of automotive software applications under real-time conditions. We presented the real-time language features that were introduced to TTCN-3. We described their syntax and semantics and discussed their application by means of an example that reflects typical automotive timing requirements. In future work, we will provide a deeper integration with the AUTOSAR methodology and show how timing requirements can be derived directly from AUTOSAR software specifications.

References

[1] AUTOSAR. www.autosar.org - web site of the AUTOSAR consortium, 2008.

[2] R. Cardell-oliver and T. Glover. A practical and complete algorithm for testing real-time systems. In *In Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 1486*, pages 251–261. Springer-Verlag, 1998.

[3] Z. Dai, J. Grabowski, and H. Neukirchen. Timed TTCN-3 – a real-time extension for TTCN-3. In *Testing of Communicating Systems, volume 14, Berlin, March 2002. Kluwer.*, 2002.

[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, Febr. 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, Febr. 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, Febr. 2007.

[7] W. A. Halang and A. D. Stoyenko. *Constructing Predictable Real Time Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[8] D. E. Knuth. backus normal form vs. backus naur form. *Commun. ACM*, 7(12):735–736, 1964.

[9] M. Krichen and S. Tripakis. Real-time testing with timed automata testers and coverage criteria. Technical Report TR-2004-15, Verimag Technical Report, 2004.

[10] M. Mikucionis, K. G. Larsen, and B. Nielsen. T-UPPAAL: Online model-based testing of real-time systems. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 396–397, Washington, DC, USA, 2004. IEEE Computer Society.

[11] H. Neukirchen. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*. PhD thesis, Georg-August-Universitt Gttingen, 2004.

[12] D. A. Serbanescu, V. Molovata, G. Din, I. Schieferdecker, and I. Radusch. Real-time testing with TTCN-3. In K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, editors, *TestCom/FATES*, volume 5047 of *Lecture Notes in Computer Science*, pages 283–301. Springer, 2008.