*TIMED*TTCN-3 – A REAL-TIME EXTENSION FOR TTCN-3[*]

Zhen Ru Dai, Jens Grabowski, Helmut Neukirchen

Institute for Telematics, University of Lübeck

Ratzeburger Allee 160, D-23538 Lübeck, Germany

Tel.: +49 451 500 3721 Fax: +49 451 500 3722

{dai,grabowsk,neukirchen}@itm.mu-luebeck.de

Abstract   The *Testing and Test Control Notation* (TTCN-3) was originally developed as successor of the second edition of the *Tree and Tabular Combined Notation*. TTCN-3 is a standardized test specification and implementation language to test functional behaviour of distributed systems. *TIMED*TTCN-3 is a real-time extension for TTCN-3 that supports the test and measurement of real-time requirements. *TIMED*TTCN-3 introduces absolute time, allows the definition of synchronization requirements for test components and provides possibilities to specify online and offline evaluation procedures for real-time requirements.

Keywords: Distributed systems testing, Test specification, TTCN-3, Real-time testing, Test evaluation, Non-functional requirements

1.   Introduction

One of the most challenging research areas in testing is the testing of distributed real-time systems. Such systems are becoming ever more importance in daily life, such as for business and administration (e.g., E-Commerce), for home use (e.g., home brokerage), teaching (e.g., teleteaching and -tutoring) and process control (e.g., air traffic control). Testing is the most important means to assure the correctness of distributed real-time systems with respect to functional and real-time behaviour.

The procedures for testing functional behaviour are defined in the international ISO/IEC standard 9646 *Conformance Testing Methodology and Framework* (CTMF) [12]. Even though CTMF focuses on conformance testing of OSI protocol entities, CTMF has been applied successfully to other types of functional testing. Part 3 of CTMF defines the test specification language *Tree*

and Tabular Combined Notation (TTCN). The latest version of TTCN has been published in 1999 as European technical report by the European Telecommunications Standards Institute (ETSI) [3]. With *PerfTTCN* (*Performance TTCN*) [15] and *RT-TTCN* (*Real-Time TTCN*) [16] two approaches exist to extend TTCN for real-time and performance testing.

PerfTTCN extends TTCN with concepts for performance testing. These concepts are: (1) *performance test scenarios* for the description of test configurations, (2) *traffic models* for the description of discrete and continuous streams of data, (3) *measurement points* as special observation points, (4) *measurement declarations* for the definition of metrics to be observed at measurement points, (5) *performance constraints* to describe the performance conditions that shall be met, and (6) *performance verdicts* for the judgement of test results. The PerfTTCN concepts are introduced mainly on a syntactical level by means of new TTCN tables. Their semantics are described in an informal manner only.

RT-TTCN is an extension of TTCN in order to test *hard* real-time requirements. On the syntactical level, RT-TTCN supports the annotation of TTCN statements with two timestamps for earliest and latest execution times. On the semantical level, the TTCN snapshot semantics has been refined and, in addition, RT-TTCN has been mapped onto timed transition systems [8].

The *Testing and Test Control Notation* (TTCN-3) [4, 7] has been developed from 1998 to 2001 by an ETSI experts team as successor language for TTCN. TTCN-3 is based on a textual core notation that has the look and feel of a common programming language. In comparison to TTCN, TTCN-3 includes no OSI and conformance testing specific constructs, but provides several additional concepts like, e.g., *dynamic test configurations*, *procedure-based communication* and *module control part*. The development of TTCN-3 concentrated on features for functional testing. Thus, some major concepts needed for real-time and performance testing are missing.

This paper tries to close this gap by proposing *TIMED*TTCN-3 as a real-time extension for TTCN-3. *TIMED*TTCN-3 introduces (1) a *new test verdict* to judge real-time behaviour, (2) supports *absolute time* as a means to measure time and to calculate durations, (3) allows to *delay* the execution of statements for defining time dependent test behaviour, (4) supports the specification of *synchronization conditions* for test components and (5) provides a means for the *online and offline evaluation* of real-time properties. Our first experiments give evidence that *TIMED*TTCN-3 covers most PerfTTCN and RT-TTCN features while being more intuitive in usage. Moreover, the *TIMED*TTCN-3 extensions are more unified than the other extensions by making full use of the expressiveness of TTCN-3 . Thus only a few changes to the language are needed.
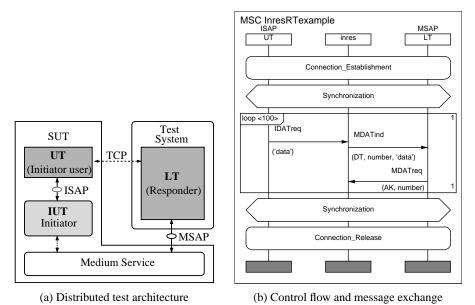
(a) Distributed test architecture   (b) Control flow and message exchange

Figure 1.   Test architecture and control flow for the Inres test case example

2. An Inres-based Example

The concepts of *T*IMEDTTCN-3 will be explained by a test case for the *Inres* protocol [9]. As shown in Figure 1a, the test case is written for the distributed test method of CTMF [12]. The *Implementation Under Test* (IUT) is an *Initiator* implementation. The *Upper Tester* (UT) function plays the role of an Initiator user and the *Lower Tester* (LT) function plays the role of a Responder entity. The UT has a direct connection with the IUT whereas the LT only has indirect access to the lower interface of the IUT via a *Medium Service*. UT and LT coordinate themselves by *Test Coordination Procedures* (TCP).

The example is designed to test the real-time properties *latency* and *mean arrival time* for the exchange of 100 data packets.[1] Its principle control flow and message exchange is presented by the Message Sequence Chart in Figure 1b. The test case starts with a preamble that establishes a connection between UT and LT. Afterwards, UT and LT synchronize in order to ensure that both tester functions are in a correct state to execute the test body. The test body includes the sending of 100 data packets from UT to LT. The LT must acknowledge the correct reception of each data packet. Otherwise, the SUT will retransmit the data packet or, after three retransmissions, release the connection.[2] At the

---

[1]Please note that in order to perform meaningful non-functional testing as described in this paper, it is necessary that an IUT has passed the related functional test cases.

[2]The retransmission of data packets and the exceptional connection release are not shown in Figure 1b.

```
(1)     testcase InresRTexample(integer sequenceStartNum)
(2)     runs on InitiatorUserType system InresSystemType {

(3)        var ResponderType responder := null;
(4)        var float sendTime := 0.0;
(5)        var default myDefault := null;

(6)        myDefault := activate(InitiatorDefault);      // Default activation
(7)        map(self:ISAP, system:ISAP);

           // Creating/mapping/connecting/starting the Responder PTC
(8)        responder := ResponderType.create(self.timezone)
(9)        map(responder:MSAP, system:MSAP);
(10)       connect(self:CP, responder:CP);
(11)       responder.start(ResponderBehaviour(SequenceStartNum));

(12)       InitiatorPreamble(); // Preamble for connection establishment

(13)       CP.send(boolean:true); // Initial synchronization
(14)       CP.receive(boolean:true); // Default handles alternatives

(15)       sendTime := self.now + 5.0; // Send the first time in 5.0s

           // Sending of 100 data packets in a loop
(16)       for (var integer i := 1; i <= 100; i := i + 1) {
(17)          resume(sendTime); // Wait until 'sendTime'
(18)          log (TimestampType:{self.now, self.timezone, IDATreq});
(19)          ISAP.send(IDATreqType:{self.now});

(20)          sendTime := sendTime + 0.01; // Send periodically every 10ms
(21)       }

(22)       CP.send(boolean:true); // Final synchronization
(23)       CP.receive(boolean:true); // Default handles alternatives

(24)       verdict.set(pass); // Everything is OK
(25)       InitiatorPostamble(); // Postamble for connection release
(26)    }
```

Figure 2.   TIMEDTTCN-3 test case description

end of the test body, UT and LT synchronize again and perform a postamble to
release the connection.

The *TIMEDTTCN-3* code for the behaviour of the *main test component* (**mtc**)
is shown in Figure 2. The **mtc** is the UT and plays the role of an Initiator user.
Lines 1 and 2 provide the test case interface, i.e., name, formal parameters,
component types for **mtc** (**runs on** clause) and *abstract test system interface*
(**system** clause). Lines 3-7 describe variable declarations, a default activation
and the mapping of **mtc** ports onto ports of the abstract test system interface.

The creation of the LT component, the mapping of LT ports onto ports of the
abstract test system interface, the connection of LT and **mtc** ports, and the start of
the LT component are specified in lines 8-11. The preamble *InitiatorPreamble*
is called in line 12 and the initial synchronization by means of an UT-initiated
handshake with boolean synchronization messages is shown in lines 13 and 14.
The time for sending the first data packet is determined in line 15. The body
of the test case consists of the **for** loop specified in lines 16-21. The loop body

```
(1)     function ResponderBehaviour(integer expectedNum) runs on ResponderType {
(2)        var float receiveTime := 0.0, sendTime := 0.0;
(3)        var MDATindType receivedMessage;
(4)        myDefault := activate(ResponderDefault); // Default activation
(5)        ResponderPreamble(); // Preamble for connection establishment
(6)        CP.receive(boolean:true); // Initial synchronization
(7)        CP.send(boolean:true);
           // Receiving 100 data packets in a loop
(8)        for (var integer i := 1; i <= 100; i := i + 1) {
(9)          alt {
(10)           [ ] MSAP.receive(MDATindType:{DT, expectedNum, ?}))
                                                       -> value receivedMessage {
(11)                receiveTime := self.now;
(12)                log(TimestampType:{receiveTime, self.timezone, MDATind});
(13)                sendTime := receivedMessage.data; // Extract the send time
(14)                // Latency online evaluation
(15)                if (evalLatency(sendTime, receiveTime, 0.001, 0.005) == conf) {
(16)                    verdict.set(conf);
(17)                }
(18)                MSAP.send(MDATreqType:{AK, expectedNum, 0.0});
(19)                expectedNum := toggle(expectedNum);
(20)             }
(21)           [ ] MSAP.receive(MDATindType:{DT, toggle(expectedNum), ?}) {
(22)                MSAP.send(MDATreqType:{AK, toggle(expectedNum), 0.0});
(23)                repeat;
(24)             }
(25)          }
(26)        }
(27)        CP.receive(boolean:true); // Final synchronization
(28)        CP.send(boolean:true);
(29)        verdict.set(pass); // Everything is OK
(30)        ResponderPostamble(); // Postamble for connection release
(31)     }
```

Figure 3.   *T*IMEDTTCN-3 behaviour of the Responder test component

is repeated 100 times and specifies that a data packet is sent every 10ms (lines 19 and 20). The test case terminates with the final synchronization (lines 22 and 23), the setting of a **pass** verdict (line 24) and the call of the postamble *InitiatorPostamble* (line 25).

The behaviour of the LT is specified by the *T*IMEDTTCN-3 function shown in Figure 3. The function can be structured into three parts. The first part consist of declarations (lines 2 and 3), a default activation (line 4), the call of the preamble *ResponderPreamble* (line 5) and the initial synchronization (lines 6 and 7).

The second part (test body) consists of a **for** loop (lines 8-26). The loop body is repeated 100 times and includes an **alt** statement with two alternatives. The first alternative (lines 10-20) describes the expected message exchange: A correct data packet is received (line 10), the actual time is retrieved and

recorded (lines 11 and 12), the send time is extracted from the received message (line 13), the latency is checked (line 15), if the latency requirement is violated, the new test verdict **conf** (Section 3) is set (line 16), finally, the data packet is acknowledged (line 18) and the sequence number of the next correct data packet is computed (line 19). The second alternative describes the case when the previous acknowledgement got lost and, therefore, the previous data packet is re-transmitted by the IUT. The reception of the re-transmitted data packet is described in line 21 and its re-acknowledgement is specified in line 22.

The third part describes the final synchronization (lines 27 and 28), the setting of the **pass** verdict (line 29) and the call of *ResponderPostamble* (line 30).

The test case specifies the expected message exchange only. Erroneous and unexpected responses received from the SUT are considered to be handled by defaults (line 6 in Figure 2 and line 4 in Figure 3).

The *TIMED*TTCN-3 code in Figure 2 and Figure 3 includes the real-time extensions **self.now**, **resume**, **self.timezone**, the new verdict **conf**, a modified syntax for the **log** statement and a new parameter for the **create** operation. These extensions will be explained in the following sections.

3. Non-Functional Verdicts

The TTCN-3 verdicts indicate whether a test case was successful (**pass**), inconclusive (**inconc**) or faulty (**fail**) with respect to functional requirements. By introducing the possibility to test non-functional requirements, additional information concerning the test outcome is needed: A test case may *pass* with respect to both functional and non-functional behaviour or it may *pass* only with respect to the functional behaviour while the non-functional requirements are violated.[3]

Since non-functional behaviour can be observed only in combination with functional behaviour on which the non-functional requirements are imposed, it is not meaningful to make any statements on non-functional test results if the functional behaviour is not conforming to the functional requirements. In the special case of a tester malfunction which may lead to a wrong real-time measurement, the **error** verdict will be set by the tester.

Even in case of a *functional inconclusive*, no statement can be made on non-functional test results, since such an inconclusive case may have other non-functional requirements than the pass case which is subject of testing. Hence, distinctive verdicts are just needed in case of a *functional pass*.

Besides the existing **pass** verdict which is used to indicate a *functional pass* with an associated *non-functional pass*, *TIMED*TTCN-3 introduces the new ver-

---

[3]In the following, the terms *functional pass*, *non-functional pass*, etc. are used to describe the test outcome with respect to functional and non-functional behaviour.

Current value	New verdict assignment value				
of verdict	pass	conf	inconc	fail	none
pass	pass	conf	inconc	fail	pass
conf	conf	conf	inconc	fail	conf
inconc	inconc	inconc	inconc	fail	inconc
fail	fail	fail	fail	fail	fail
none	pass	conf	inconc	fail	none

Figure 4. *T*ᴵᴹᴱᴰTTCN-3 overwriting rules for the test verdicts

dict **conf** (as abbreviation for *conforming*) to indicate a *functional pass* with an associated *non-functional fail*. Due to the introduction of the new verdict, the overwriting rules for verdicts are refined. They are presented in Figure 4. The new verdict **conf** is inserted between the verdicts **pass** and **inconc**.

An example for the usage of the new **conf** verdict can be found in Figure 3. If the latency requirement is violated (checked in the **if** statement in line 15), **conf** is assigned to the local verdict of the Responder test component (line 16). Due to the overwriting rules of *T*ᴵᴹᴱᴰTTCN-3, a **conf** verdict will not be overwritten by the **verdict.set**(**pass**) statement in line 29 (see also row 2, column 1 of Figure 4). The existing verdict **fail** is still available to express a *functional fail* (see Figure 11, lines 13 and 24).

4. Time Extension for TTCN-3

For the handling of time, TTCN-3 provides a *timer* mechanism. This timer mechanism is designed for supervising the functional behaviour of an IUT, e.g., to prevent the blocking of a test case or to provoke exceptional behaviour. But, it is too slow and too clumsy for the test and measurement of real-time properties, because the measurement of durations is influenced by the TTCN-3 snapshot semantics and by the order in which the port queues and the timeout list are examined. TTCN-3 makes no assumptions about the duration for taking and evaluating a snapshot. Thus, exact times can not be measured.

Furthermore, TTCN-3 has no concept of *absolute time*, i.e., a test component cannot read and use its local system time. In real-time testing, the absolute time is necessary to check relationships between observed test events and to coordinate test activities. In case of synchronized clocks in a distributed test environment, the system time may be exchanged among test components to check real-time requirements that cannot be measured locally or for the timely coordination of test activities.

As a consequence of these considerations, *T*ᴵᴹᴱᴰTTCN-3 has the concept of *absolute time* in order to support real-time testing. In case of a distributed test environment, the test cases may define the requirements for the synchronization of clocks of different test components.

4.1 Absolute Time

Absolute time is related to *clocks* that provide the actual value of time. We assume that each test component has access to such a clock, but make no assumptions about the number and the synchronization of these clocks.[4] Furthermore, we assume that the resolution of such clocks and the speed of the tester are adequate for the real-time requirements which are subject of testing.

For the handling of time values either a new type is needed, or the time values have to be mapped onto an existing basic type. Due to numerous possible time representations, e.g., the UNIX approach to count the seconds since 1.1.1970 [10] or a structured type with fields for year, month, day, hour etc., a common new type for time values is not easy to define.

For simplicity, *Timed*TTCN-3 uses the existing **float** type and follows the UNIX approach, i.e., time is counted in seconds and the absolute time is represented by the number of seconds since a fixed point in time. In contrast to the UNIX scheme, *Timed*TTCN-3 does not define a fixed starting point for the time measurement. For that, *Timed*TTCN-3 supports the usage of absolute time by the operations **now** and **resume**:

- The **now** operation is used for the retrieval of the current *local time*. The local character of the **now** operation is reflected by its application to the **self** handle, i.e., **self.now** is the expected call statement for the **now** operation. Operation **now** returns a float value that equals the current absolute time when the operation is called. The mapping of the float value onto a concrete daytime, e.g., year, month, day, hour, etc., is considered to be outside the scope of *Timed*TTCN-3 and has to be provided by the test equipment, e.g., in form of additional conversion functions.

- The **resume** operation provides the ability to delay the execution of a test component. The argument of the **resume** operation is considered to be an absolute time value, i.e., the point in time when the test component shall resume its execution. If required, a relative time can easily be specified by using the current time as reference time, e.g., waiting for 3 seconds can be described by **resume**(**self.now** + 3.0).

An example for the usage of the absolute time extension is shown in Figure 2. The current time is retrieved in line 15. It is used to calculate the sending time of the first data packet. The sending time is used by the **resume** operation in line 17. The test component will resume when the specified time is reached.

---

[4]From a conceptional point of view, synchronized test components share the same clock, even though in a real implementation, the clocks are synchronized by using a synchronization protocol [13, 14].

4.2 Synchronization of Clocks

Time values are observed and used locally by the test components. Time values that are observed in different test components may be exchanged and used for further computations. But this only makes sense if the clocks of the involved test components are synchronized. The synchronization itself is outside the scope of *T*IMEDTTCN-3 and should be guaranteed by the test equipment, but requirements for clock synchronization may very well be expressed in *T*IMEDTTCN-3. These requirements may be used by a *T*IMEDTTCN-3 compiler to distribute test components in an adequate manner or by a *T*IMEDTTCN-3 runtime environment to execute synchronization procedures for the test devices.

Timezones. Most specification and implementation languages either support *local time* or *global time*. Local time means that each behavioral entity, e.g., an SDL process or a TTCN-3 test component, has its own local time. Global time means that all behavioral entities share the same *global* time. Global time is perfect for the purpose of real-time testing, because all test components have by definition the same global time and are synchronized.

However, neither local nor global time are realistic assumptions for real-time testing situations. A real-time test environment typically consists of several devices. If synchronization among two or more test components is required to reach the goal of a test case, the components have to be executed either on the same device or on synchronized devices.

The developer of real-time test cases should not care about synchronization procedures and the distribution of test components himself, but he can support their implementation by identifying test components which have to be synchronized. For this purpose, *T*IMEDTTCN-3 supports the *timezones* concept.

A timezone is an (optional) attribute that can be assigned to a test component when the component is created. Test components with the same attribute are considered to be synchronized, i.e., they have the same absolute time. A test component can only have one timezone attribute. Components without timezone attributes are considered to be not synchronized with other components.

Implementation of the Timezones Concept. *T*IMEDTTCN-3 implements the timezones concept by using an enumeration type with the reserved name **timezones**. The user has to specify the timezone attribute values by defining the **timezones** type in the module definitions part of a *T*IMEDTTCN-3 module. This type has an implicit member of name **none** which indicates no clock synchronization. The usage of an enumeration type only makes sense if the number of timezones is finite and known.

In *T*IMEDTTCN-3, a timezone attribute is associated with a test component when the component is created, i.e., the timezone attribute is an optional pa-

rameter of the **execute** and the **create** operations. The timezone attribute of an **mtc** is assigned by using an **execute** operation. Attributes of all other test components are assigned by means of the **create** operations.

The flexibility of the timezones concept can be improved by making the timezones visible to the test components. This is implemented in *TIMED*TTCN-3 by means of a special **timezone** function which returns the timezone of the component that called the function. In the case of a non-synchronized test component, the value **none** is returned. Like the **now** operation, the **timezone** operation is always applied to the **self** handle of a test component, i.e., **self.timezone** is the expected call statement for the **timezone** operation. The timezone information may be exchanged among test components to check if synchronization conditions are satisfied, or used to create several synchronized components.
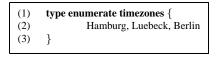
```
(1)    type enumerate timezones {
(2)            Hamburg, Luebeck, Berlin
(3)    }
```

Figure 5.   Definition of timezones

The usage of the timezone concept is shown in Figures 2, 5 and 10. Figure 5 presents the definition of timezones *Berlin*, *Hamburg* and *Luebeck*. In our test case example, the **mtc** is created by the **execute** statement in line 5 of Figure 10 and receives the timezone attribute *Luebeck*. The behaviour of the **mtc** is shown in Figure 2. The **mtc** creates the test component *Responder* (line 8) and assigns its own timezone to the new component, i.e., **mtc** and *Responder* are considered to be synchronized. A *TIMED*TTCN-3 run-time environment may use this information to ensure this synchronization condition.

5.   Evaluation of Real-Time Properties

While functional behaviour is basically tested by using sequences of **send** and **receive** operations, real-time requirements can be tested by relating particular points in time to each other [1, 2, 11]. The essence of the various real-time requirements can be broken down to the relationship of points in time. Mathematical formulae can be used to evaluate whether the points in time of interesting events fulfill a certain real-time requirement or not.

To obtain those points in time, functional TTCN-3 test cases are instrumented by statements which generate timestamps. *TIMED*TTCN-3 implements this approach by making use of the possibility to read absolute time values which serve as timestamps. The mathematical formulae which are applied on the timestamps can be coded as TTCN-3 functions. Those *evaluation functions* return a judgement which indicates whether a requirement is fulfilled or not. An *online* or an *offline evaluation* of timestamps is possible:

- *Online evaluation* is needed if it is not possible to separate functional and non-functional requirements, i.e., a non-functional property directly influences the functional behaviour of a testcase. In such a case, evaluation of non-functional observations must be performed during the testrun in order to react on the result of the evaluation. Online evaluation has the drawback of cluttering the testcase and slowing down the performance of the testcase which may be undesirable for time-critical testcases.

- *Offline evaluation* may be used if the non-functional requirements which are subject of testing have no influence on the functional reaction of a testcase. In this case, the code just needs to be instrumented by statements that log the relevant timestamps. Based on the timestamps in the logfile, the non-functional property can be evaluated when the testrun has finished. Offline evaluation has the advantage of having a low impact on the performance of a testcase, since timestamps have to be logged only during the testrun.

5.1 Online Evaluation

For performing online evaluation, timestamps have to be evaluated during the testrun, e.g., by calling a special evaluation function with timestamps as actual parameters. In a distributed test architecture, non-functional requirements may involve timestamps which have been collected by different test components. In this case, the evaluating component needs to obtain timestamps from other components. To achieve this, timestamps can either be piggybacked in the payload of some IUT signals or be communicated directly among test components by using coordination messages. For implementing online evaluation, the new concepts of *T*ɪᴍᴇᴅTTCN-3 which have been introduced so far, are sufficient.

In our example (Section 2), online evaluation is used to check the fulfillment of a non-functional *latency* requirement. In case of a violation, the local test verdict of the Responder test component is set to **conf** (line 16 in Figure 3).

The online evaluation of the latency requirement covers timestamps of several test components. Hence, the remote timestamps have to be transferred to the evaluating component. The evaluation function is called in the Responder test component (line 15). The **receive** operation for the *MDATind* signal (line 10) is local to the Responder component. The related timestamp is obtained by calling **now** and stored in variable *receiveTime* (line 11). In contrast, the corresponding **send** operation is performed by the **mtc** and the associated timestamp is piggybacked to the payload of the *IDATind* signal (line 19 in Figure 2).[5]

---

[5]In our Inres example, the payload of the *IDATind* signal is considered to be of type **float**. In the more general case, the **float** value has to be encoded into the particular payload type.

```
(1)     function evalLatency(float timeA, float timeB,
                         float lowerbound, float upperbound) return verdicttype {
(2)     var float latency:=timeB-timeA;
(3)     if ((latency<upperbound) and (latency>lowerbound)) {
(4)       return pass;      // non-functional pass
(5)     }
(6)     else {
(7)       return conf;      // non-functional fail
(8)     }
(9)   }
```

Figure 6.   *TIMED*TTCN-3 online evaluation function

The Responder component extracts the piggybacked timestamp from the received signal and assigns it to variable *sendTime* (line 13 in Figure 3). Afterwards, *evalLatency* (line 15) is called. The actual parameters of this online evaluation function call are send and receive time as well as the boundaries $1ms$ and $5ms$ which describe the incarnation of the latency real-time requirement.

The evaluation function *evalLatency* (Figure 6) checks the condition related to latency ($lowerbound < t_{receive} - t_{send} < upperbound$) of related timestamps (line 3). Depending on the result, it returns either a **pass** or a **conf** verdict (lines 4 and 5) which may be used by the calling entity for further decisions.

5.2   Offline Evaluation

When using offline evaluation, the evaluation function is merely called after the test execution. *TIMED*TTCN-3 offers a means to record timestamps in a logfile during a testrun in order to evaluate them afterwards. The final test verdict is a composition of the functional test verdict and result of the subsequent offline evaluation. To enable offline evaluation of real-time requirements, *TIMED*TTCN-3 refines the existing logfile concept of TTCN-3.

TTCN-3 assumes that one global or several local logfiles exist and provides a **log** statement to enable logging of comments. The number of logfiles is not specified, the logging mechanism is not described and neither module control nor test components can access the global or local logfiles. For an efficient offline evaluation, module control and test components need access to the logfiles and the contents of the logfiles have to be specified more formally.

The Logfile Concept.   A *TIMED*TTCN-3 logfile is basically a list of values of arbitrary TTCN-3 types. A logfile is of type **logfile** and it is possible to handle logfiles by means of variables or to pass them into functions.

Each *TIMED*TTCN-3 test component has its own local logfile. A local logfile is initialized when the owning component is created. When test execution finishes, i.e., the main test component terminates, the local logfiles are merged

Operation name	Return type	Function
first(*sortkey, value*)	**boolean**	Select and sort logfile by *sortkey* and move to first matching entry in the logfile
next(*value*)	**boolean**	Move to the next matching entry
previous(*value*)	**boolean**	Move to the previous matching entry
retrieve	type of *sortkey* used as parameter of **first**	Retrieve entry from current logfile position

Figure 7.   Overview of *T*IMEDTTCN-3 *logfile* operations

into a global one. *T*IMEDTTCN-3 does not specify the internal mechanisms that are needed for storing and maintaining logfiles[6], but defines four functions for accessing the entries of a logfile (Figure 7).

Logging of Events.   *T*IMEDTTCN-3 refines the TTCN-3 **log** statement in order to write information into logfiles. But in TTCN-3, the argument of the **log** statement is an arbitrary string. In *T*IMEDTTCN-3, the argument can be the value of any arbitrary valid type. For offline evaluation, a special structured type with a timestamp field may be specified. A corresponding offline evaluation function may only consider logfile entries of the special type in order to judge the fulfillment of the real-time requirement.

Logfile Operations.   For retrieving entries of a logfile, *T*IMEDTTCN-3 offers means for sorting a logfile by a certain field of the logfile's entries. Since a logfile may contain values of arbitrary types, sorting and retrieving is only possible for a certain type which has to be specified. According to the order which is imposed by sorting, the first, the next or the previous logfile entry may be retrieved. For this purpose, *T*IMEDTTCN-3 uses an internal cursor which points to an entry in the logfile. This cursor can be moved and the value at the current cursor position may be retrieved.

The operation **first** serves two purposes: It selects the entries of the logfile by their types and sorts them. In addition, it moves the cursor to the first matching entry in the logfile. The first parameter of **first** specifies the field which is used as a sorting key. This is done using the TTCN-3 template notation: A "**?**" indicates the field which is used as sorting key, all other fields must be set to "**-**". The type of the template is used to select the type of entries which are regarded by the logfile operations presented in Figure 7. The second parameter can be used to further restrict the value of the entry, i.e., the internal cursor is

---

[6]The mechanisms for storing and maintaining logfiles are considered to be implementation specific and therefore outside the scope of *TIMED*TTCN-3.

Operation name	Return type	Description
getlog	**logfile**	Get logfile
getverdict	**verdicttype**	Get global verdict
setverdict(. . .)	–	Set global verdict

Figure 8.   Overview of *TIMED*TTCN-3 operations for testrun handles

```
(1)    type record TimestampType {
(2)        float logtime;
(3)        timezones componentzone;
(4)        Messages messagename;
(5)    }

(6)    type enumerate Messages {
(7)        IDATreq, MDATind;
(8)    }
```

Figure 9.   Data types used for offline evaluation in the Inres example

moved to the first entry that matches the second parameter. The same matching mechanisms which are available for TTCN-3 **receive** statements apply.

The operations **next** and **previous** place the internal cursor to the next matching entry before or after the current cursor position. The order to which **next** and **previous** refer to is imposed by the sorting resulting from operation **first**. The parameter of **next** and **previous** is used in the same way as the second parameter of **first**. More complex search operations may be build from these basic search operations. The three operations **first**, **next** and **previous** return **true** when the matching entry is found in the logfile, otherwise **false**. The value of the last matched entry, i.e., the value at the current cursor position, can be retrieved by the **retrieve** operation.

The Testrun Handle.   For the handling of global logfiles, *TIMED*TTCN-3 introduces the concept of *testrun handles* and changes the semantics of the **execute** statement. A testrun handle is basically a pointer of type **testrun** which is returned by the **execute** statement and which gives access to the results of a testrun, i.e., the test verdict and global testlog. The operations which can be applied on a testrun handle are shown in Figure 8. The **getlog** operation is used to retrieve the logfile of a testrun. The operations **getverdict** and **setverdict** are used to retrieve and set the global verdict after a testrun. The change of the final testrun verdict might be necessary, if an offline evaluation shows that a non-functional property is not fulfilled. For the **setverdict** operation the same overwriting rules as defined in Section 3 apply.

Local Handling of Logfiles.   Global logfiles can be retrieved by applying the **getlog** function to testrun handles (Figure 8). *TIMED*TTCN-3 also allows to

apply the **getlog** function to **self** handles, i.e., a test component may access its own logfile in order to perform a local offline evaluation after the collection of timestamps.

Example. The smooth interworking of all *T*ᴵᴹᴱᴰTTCN-3 concepts for offline evaluation will be explained by means of our test case example (Section 2). For logging test events, the data types shown in Figure 9 have been defined. Values of type *TimestampType* will be logged. Its field values describe the logtime (line 2), the timezone of the logging component (line 3) and the type of the message which causes the log event (line 4). The message type is described by an enumeration type (lines 6-8).

Local logfile entries are written by the **mtc** before sending an *IDATreq* message (line 18 in Figure 2) and by the Responder test component after the reception of a correct *MDATind* message (line 12 in Figure 3).

Figure 10 shows the *T*ᴵᴹᴱᴰTTCN-3 module control part for the Inres example. The control part starts with variable declarations for the handling of a testrun, a logfile and a verdict value (line 2-4). The testcase *InresRTexample* is executed with *Luebeck* as timezone attribute for the **mtc** (line 5) and the **execute** statement returns a testrun handle which is assigned to variable *myTestrun*. The verdict is retrieved from the testrun and stored in variable *myVerdict* (line 6).

If *myVerdict* is **pass** (checked in line 7), the logfile is retrieved (line 8) and the offline evaluation function *evalMeanArrivalTime* is called (line 9). The actual parameters are the message identifier *MDATind* for which the mean arrival time should be checked, the timezone value *Luebeck* for the identification of relevant logfile entries, the time bounds of $10ms$ and $15ms$ that define the requirement to be checked, the integer value 100 that defines the number of relevant timestamps and the reference to the logfile to be evaluated. Finally, the result of the offline evaluation is assigned to the final verdict of the testrun (line 10).

```
(1)     control {
(2)        var testrun myTestrun; // Variable for testrun handling
(3)        var logfile myLog; // Variable for testlog handling
(4)        var verdicttype myVerdict;

(5)        myTestrun := execute(InresRTexample(0), Luebeck);

(6)        myVerdict := myTestrun.getverdict; // Retrieval of verdict
(7)        if (myVerdict == pass) {
(8)           myLog := myTestrun.getlog; // Retrieval of testlog
(9)           myVerdict := evalMeanArrivalTime(MDATind, Luebeck,
                            0.01, 0.015, 100, myLog); // Offline evaluation
(10)          myTestrun.setverdict(myVerdict); // Change of testrun verdict
(11)       }
(12)    }
```

Figure 10. *T*ᴵᴹᴱᴰTTCN-3 control part for the offline evaluation

```
(1)    function evalMeanArrivalTime(Messages messageId, timezones zone, float lowerbound,
            float upperbound, integer count, logfile timelog)    return verdicttype {
(2)      var float timeSum:=0, averageArrivalTime;
(3)      var TimestampType stampA, stampB;
(4)      if (timelog.first(TimestampType:{?,-,-},
                 TimestampType:{?, zone, messageId}) == true) { // search
(5)        stampA := timelog.retrieve; // Get current timestamp entry
(6)        for (var integer i := 2; i <= count; i := i + 1) {
(7)          if (timelog.next(TimestampType:{?, zone, messageId}) == true) { // search
(8)            stampB := timelog.retrieve; // Get current timestamp entry
(9)            timeSum:= (stampB.logtime - stampA.logtime) + timeSum;
(10)           stampA := stampB;
(11)         }
(12)         else {
(13)           return fail; // Wrong number of messages indicates functional problem
(14)         }
(15)       }
(16)       averageArrivalTime := timeSum / (count-1);
(17)       if ((averageArrivalTime < upperbound) and (averageArrivalTime > lowerbound)) {
(18)         return pass; // return non-functional pass
(19)       }
(20)       else {
(21)         return conf; // return non-functional fail
(22)       }
(23)     }
(24)     return fail; // Wrong number of messages indicates functional problem
(25)   }
```

Figure 11.   *TIMED*TTCN-3 offline evaluation function

The offline evaluation function *evalMeanArrivalTime* is shown in Figure 11. It implements the mathematical formula for mean arrival time based on times-tamps, namely $\left[\sum_{i:=2}^{n}(t_i - t_{i-1})\right]/(n-1)$ and subsequently verifies that the mean arrival time falls within the range (*lowerbound*, *upperbound*).

In order to iterate through the timestamps of *MDATInd* messages, **first** (line 4 in Figure 11) and **next** (line 7) are used. Since the **first** operation in line 4 sorts the logfile by the *logtime* field, the timestamp entries are matched in ascending order. If **first** or **next** fails, the logfile contains less matching timestamps than expected. This is an indication for a non-conforming behaviour of the IUT. Hence, evaluation is aborted with a **fail** verdict (lines 13 and 24).

The **retrieve** operation (lines 5 and 8) yields the value of the last successfully matched entry, which is used to calculate the mean arrival time. Based on the final value of the calculation, either **pass** or **conf** is returned (lines 18 and 21).

6.   Summary and Outlook

We presented *TIMED*TTCN-3, a real-time extension for TTCN-3, and demon-strated its usage by applying it to a testcase for the Inres protocol. By introducing absolute time for test components, it is possible to wait until an absolute point

in time and to collect timestamps. Timestamps may be evaluated online during the testrun or offline afterwards. For offline evaluation, *T*IMED TTCN-3 offers a flexible log mechanism with local and global logfiles. The log mechanism also enables an evaluation of non-functional properties which are not real-time related. For example, failure rates for a fixed amount of data packets can be checked offline by logging correct as well as erroneous message receptions without any time information. *T*IMED TTCN-3 can also be used for distributed test architectures, since it supports the specification of synchronization conditions for clusters of clock-synchronized test components. This allows to compare timestamps captured at different, but synchronized test components.

A module of pre-defined timestamp type definitions and evaluation functions can be provided in order to facilitate the usage of *T*IMED TTCN-3. In this way, the real-time testcase developer simply needs to select the appropriate evaluation function from the pre-defined library and instrument the testcase accordingly.

In this paper we did not address the formal semantics of the new *T*IMED TTCN-3 constructs. Most *T*IMED TTCN-3 extensions can be explained by an extension of the existing TTCN-3 semantics. Only the concept of absolute time in combination with the notion of synchronized components and the **resume** operation requires new real-time semantics. These features allow the description of time dependencies among test components, i.e., absolute time values influence the behaviour in different test components.

We also did not cover load generation. For most real-time tests, it is necessary to establish some background load to obtain a realistic environment. This may easily be achieved by using either an external load generator or by explicitly implementing a load generator using *T*IMED TTCN-3 statements. As described in [6], an external load generator may be controlled from TTCN-3 by abstract service primitives which are passed to the load generator by an adaptor port.

Furthermore, we did not address the issue of online evaluation functions with memory, i.e., variables of evaluation functions keep their values between subsequent calls. Such "static" variables are valuable for the online evaluation of real-time requirements like floating average. A simple workaround for "static" variables is possible by declaring such variables as component variables.

A further open issue is the graphical presentation of *T*IMED TTCN-3 test cases. In the same manner as a graphical format (GFT) is currently developed for TTCN-3 [5], extensions of GFT are needed for a graphical presentation of *T*IMED TTCN-3.

Our current work concentrates on applying the proposed extensions in a larger case study in order to assess the capabilities of *T*IMED TTCN-3. This includes also the implementation of the language extensions in our TTCN-3 toolset. Our future work will focus on the mentioned open issues, especially on the real-time semantics and the graphical representation of *T*IMED TTCN-3 test cases.

18

References

[1] ATM Forum Performance Testing Specification (AF-TEST-TM-0131.000). The ATM Forum Technical Committee, 1999.

[2] Traffic Management Specification Version 4.1 (AF-TM-0121.000). The ATM Forum Technical Committee, 1999.

[3] ETSI Technical Report (TR) 101 666 (1999-05): Information technology - Open Systems Interconnection Conformance testing methodology and framework; The Tree and Tabular Combined Notation (TTCN) (Ed. 2++). ETSI, 1999.

[4] ETSI European Norm (EN) 101 873-1 (2001-06): The Tree and Tabular Combined Notation version 3; Part 3: TTCN-3 Core Language. ETSI, 2001.

[5] ETSI Technical Report (TR) 101 873-3 (2001-06): The Tree and Tabular Combined Notation version 3; Part 3: TTCN-3 Graphical Presentation Format (GFT). ETSI, 2001.

[6] Roland Gecse, Péter Krémer, and János Szabó. HTTP Performance Evaluation with TTCN. In H. Ural, R.L. Probert, and G. von Bochmann, editors, *Testing of Communicating Systems*, volume 13. Kluwer Academic Publishers, 2000.

[7] J. Grabowski, A. Wiles, C. Willcock, and D. Hogrefe. On the Design of the New Testing Language TTCN-3. In H. Ural, R.L. Probert, and G. von Bochmann, editors, *Testing of Communicating Systems*, volume 13. Kluwer Academic Publishers, 2000.

[8] T. Henzinger, Z. Manna, and A. Pnueli. Timed Transition Systems. In *Real-Time: Theorie and Practice*, volume 600 of *Lecture Notes in Computer Science*, 1991.

[9] D. Hogrefe. Report on the Validation of the Inres System. Technical Report IAM-95-007, Universität Bern, November 1995.

[10] IEEE Standard 1003.1: Information technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language]. IEEE, 1996.

[11] Request for Comments 1193: Client requirements for real-time communication services. Internet Engineering Task Force (IETF), 1990.

[12] Information technology – Open Systems Interconnection – Conformance testing methodology and framework. ISO/IEC, 1994-1997. International ISO/IEC multipart standard No. 9646.

[13] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21, 1978.

[14] P. Ramanathan, K.G. Shin, and R.W. Butler. Fault-Tolerant Clock Synchronization in Distributed Systems. *IEEE Computer*, 23, 1990.

[15] I. Schieferdecker, B. Stepien, and A. Rennoch. PerfTTCN, a TTCN Language Extension for Performace Testing. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, volume 10. Chapman & Hall, 1997.

[16] T. Walter and J. Grabowski. A Framework for the Specification of Test Cases for Real-Time Distributed Systems. *Information and Software Technology* (41), 1999.