TTCN-3 for Distributed Testing Embedded Software *

Stefan Blom[1], Thomas Deiß[2], Natalia Ioustinova[4], Ari Kontio[3], Jaco van de Pol[4,6], Axel Rennoch[5], and Natalia Sidorova[6]

[1] Institute of Computer Science, University of Innsbruck, 6020 Innsbruck, Austria
[2] Nokia Research Center, Meesmannstrasse 103, D-4480 Bochum, Germany
[3] Nokia Research Center, Itämerenkatu 11-13, 00180 Helsinki, Finland
[4] Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
[5] Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, D-10589, Berlin, Germany
[6] Dept. of Math. and Computer Science, Eindhoven University of Technology, Den Dolech 2, 5612 AZ Eindhoven, The Netherlands

Stefan.Blom@uibk.ac.at, thomas.deiss@nokia.com, ari.kontio@nokia.com, ustin@cwi.nl, Jaco.van.de.Pol@cwi.nl, axel.rennoch@fokus.fhg.de, n.sidorova@tue.nl

January 16, 2006

Abstract. TTCN-3 is a standardized language for specifying and executing test suites that is particularly popular for testing embedded systems. Prior to testing embedded software in a target environment, the software is usually tested in the host environment. Executing in the host environment often affects the real-time behavior of the software and, consequently, the results of real-time testing.
Here we provide a semantics for *host-based testing* with *simulated time* and show which kind of timing constraints can be *adequately* tested with simulated time. We provide a simulated-time solution for *distributed* testing with TTCN-3. We also report on two case studies where we used simulated time for testing with TTCN-3.
Keywords: (distributed) testing, discrete time, simulated time, TTCN-3.

1 Introduction

The Testing and Test Control Notation Version 3 (TTCN-3) is a language for specifying test suites and test control [10]. Its syntax and operational semantics are standardized by ETSI [3, 4]. The previous generations of the language were mostly used for testing systems from the telecommunication domain. TTCN-3 is however a universal testing language applicable to a broad range of systems. Standardized interfaces of TTCN-3 allow to define test suites and test control on the level independent from a particular implementation or a platform [6, 5]. Independency of an implementation significantly increases the reuse TTCN-3 test suites e.g. for different system versions or platforms. TTCN-3 also provides a great flexibility when defining test suites for distributed systems. That also

* This work is done within the project "TTMedal. Test and Testing Methodologies for Advanced Languages (TT-Medal) sponsored by Information Technology for European Advancement Programm (ITEA)" [9]

makes TTCN-3 particularly beneficial for testing modern embedded systems. TTCN-3 has already been successfully applied to test embedded systems not only in telecommunication but also in automotive and railway domains [8, 1].

When being tested, a system under test (SUT) is executed to check whether it meets certain quality requirements and to detect failures. The quality requirements are expressed in the form of *test cases* that are executed by a *test system*. The test system sends stimuli to the SUT and checks whether the response of the SUT matches one of the responses expected by the test case. Note that both the SUT and the test system can be *distributed*. If a failure is detected, the cause of the failure should be found by *debugging* and corrected.

Modern embedded systems consist of many timed components working in parallel, which makes testing and debugging even more difficult. Potential software errors can be too expensive and dangerous to test on a *target environment* where the system is supposed to work. In practice, embedded software is tested in the *host environment* used for developing the system. That allows to fix huge amount of errors prior to testing in the target environment.

The (*host*) environment differs from the (*target*) environment. When being developed, the actual system does not exist until late stages of development. *Environment simulations* are used to represent target environments. If the target operating system is not available, *emulating the target OS* is used to provide message communication, time, scheduling, synchronization and other services necessary to execute embedded software. *Monitoring* and *instrumentation* are used to observe the order and the external events of an SUT.

Ideally, using environment simulations, emulating target operating system, monitoring or instrumentation should not affects the real-time behavior of an SUT. In practice, developing simulators and emulators with high timing accuracy is often unfeasible due to high costs and time limitations imposed on the whole testing process. Monitoring not affecting real time behavior of an SUT is expensive and often requires a product-specific hardware-based implementation. In host-based testing, using simulators, emulating target OS, monitoring or instrumentations usually *affects* the real-time behavior of the SUT. If the effects significantly change timed behavior, real-time testing is not optimal and leads to inadequate test results.

Here we propose host-based testing with *simulated time* where the system clock is modelled there a logical clock and time progression is modelled by a tick-action. The calculations and actions within the system are considered to be *instantaneous*. The assumption about instantaneity of actions leads us to the conclusion that time progress can never take place if there is still an untimed action enabled, or in other words, the time progress has the least priority in the system and may take place only when the system is *idle*. We refer to the time progress action as `tick` and to the period of time between two `tick`s as a time slice. We assume that the concept of timers is used to express time-dependent behavior. Further, we refer to this time semantics as *simulated time*.

In [1] we proposed host-based testing with *simulated time* for non-distributed applications. Implementing simulated time on the level of TTCN-3 specifications, this solution was not scalable for distributed testing and led to developing test suites only suitable for testing with "simulated time". Here we provide a framework for host-based testing of *distributed* embedded systems with TTCN-3. The framework allows to use the same test suites for host-based testing with simulated time and for testing with real time in the target environment. Further we report on two case studies, one from the telecommunication domain and another one from the transportation domain where we have applied simulated time for testing with TTCN-3.

The rest of the paper is organized as follows: Section 2 provides a brief survey on a general structure of a distributed TTCN-3 test system. In Section3 we provide a time semantics for host-based testing with simulated time. In Sections 4, we present a solution for simulated time host-based testing with TTCN-3. We conclude with Section 5 where we also provide related works.

2 TTCN-3 test system

TTCN-3 is intended for specification of (abstract) test suites [7]. The specifications can be generated automatically or developed manually. A specification of a test suite is a TTCN-3 *module* which possibly imports some other modules. Modules are the TTCN-3 building blocks which can be parsed and compiled autonomously. A module consists of two parts: a definition part and a control part. The first one specifies test cases. The second one defines the order in which these test cases should be executed.

A test suite is executed by a TTCN-3 test system whose general structure is defined in [5] and illustrated in Fig. 1. The TTCN-3 executable (TE) entity actually executes or interprets a test suite, i.e. the TE entity executes TTCN-3 modules. A call of a test case can be seen as an invocation of an independent program. Starting a test case leads to creating a *configuration*. A configuration consists of several test components running in parallel and communicating with each other and with an SUT by *message passing* or by *procedure calls*. The first test component created at the starting point of a test case execution is the main test component (MTC). For communication purposes, a test component owns a set of ports. Each port has **in** and **out** directions: infinite FIFO queues are used to represent **in** directions; **out** directions are linked directly to the communication partners.

The concept of timers is used in TTCN-3 to express time-dependent behavior. A timer can be either active or deactivated. An active timer keeps an information about the time left until its expiration. When the time left until the expiration becomes null, the timer expires and becomes deactivated. An expiration of a timer results in producing a timeout that can be further consumed by one of the components. Timers are local, namely each timer belongs to a certain test component. An expiration of a timer leads to enqueueing the timeout at the timeout-list of the component.

The Platform Adapter (PA) implements timers and operations on them. The System Adapter (SA) implements communication between a TTCN-3 test

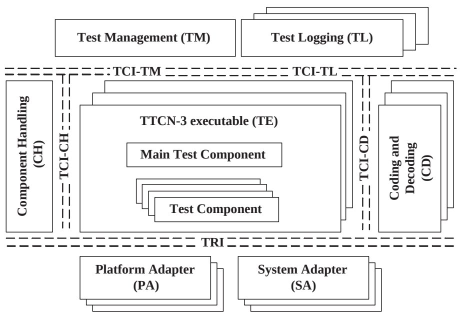

Fig. 1. General structure of a distributed TTCN-3 test system

system and an SUT. It adapts message- and procedure-based communication of the TTCN-3 test system to the particular execution platform of the SUT. The runtime interface (TRI) allows the TE entity to invoke operations implemented by the PA and the SA.

A test system (TS) can be distributed over several test system instances TS_1, ..., TS_n each of which runs on a separate test device. Each instance of the test system TS_i has an instance of the TE entity TE_i equipped with with a system adaptor SA_i, a test logging TL entity TL_i, a platform adapter PA_i and a coder/decoder CD_i running on the node. One of TE's instances is identified to be the main one. It starts executing a TTCN-3 module and calculates final testing results.

The Test Management (TM) entity controls the order of the invocation of modules. Test Logging (TL) logs test events and presents them to the test system user. The Coding and Decoding (CD) entity is responsible for the encoding and decoding of TTCN-3 values into bitstrings suitable to be sent to the SUT. The Component Handling (CH) is responsible for implementing distribution of components, remote communication between them and synchronizing components running on different instances of the test system. Instances of the TE entity interact with the TM, the TLs, the CDs and the CH via the TTCN-3 Test Control Interface (TCI) [6].

3 Simulated time in TTCN-3

For testing purposes, we focus on *closed* systems (a test system together with an SUT) consisting of multiple components communicating with each other. We say that a *component* is *idle* iff it cannot proceed by performing computations, receiving messages, consuming timeouts. Further we refer to the idleness of a single component as *local idleness.* We say that a *system* is *idle* iff all components of the system are idle and there are no messages or timeouts that still can be received during the current time slice. Further we call such messages and timeouts *pending.* We refer to the idleness of the whole system as *global idleness.*

$$[\forall i = 1..n\colon (SA_i = idle) \wedge (PA_i = idle) \wedge (SA_i = idle)] \wedge SUT = idle \quad (1)$$
$$\sum\nolimits_{i=1..n} SA_i Sent2SUT = EnqdBySUT \quad (2)$$
$$SentBySUT = \sum\nolimits_{i=1..n} EnqdBySA_i \quad (3)$$
$$\sum\nolimits_{i=1..n} TCISentByTE_i = \sum\nolimits_{i=1..n} TCIEnqd2TE_i \quad (4)$$
$$\forall i = 1..n : TRISentByTE_i = TRIEnqdBySA_i PA_i \quad (5)$$
$$\forall i = 1..n : TRISentBySA_i PA_i = TRIEnqdByTE_i \quad (6)$$

Fig. 2. Necessary conditions for global idleness

Definition 1 (Global Idleness). *We say that a closed system is* globally idle *iff all components are locally idle and there are no messages, no timeouts pending.*

If the system is globally idle, the *time progresses* by action `tick` that decreases time left until expiration of active timers by one. If a delay left until expiration of a timer reaches zero, the timer expires within the current time slice. Timers ready to expire within the same time slice expire in an arbitrary order. Note that in a distributed system, we have to ensure that all components progress time before starting in the next time slice.

The time semantics of TTCN-3 has been intentionally left open to enable the use of TTCN-3 with different time semantics [4]. Nevertheless, the focus has been on using TTCN-3 for real-time testing so not much attention has been paid to implementing other time semantics for TTCN-3 [10]. Existing standard interfaces TCI and TRI provide excellent support for real-time testing but lack operations necessary for implementing simulated time [6, 5].

Our goal is to provide a solution for implementing simulated time for a distributed TTCN-3 test system. Developing a test suite for host-based testing costs time and efforts. Therefore, we want the test suites developed for host-based testing with simulated time to be reusable for real-time testing in the target environment. That means we focus on providing a solution that can be implemented on the level of adapters, not on the level of TTCN-3 code. In this way, the same TTCN-3 test suites can be used both for host-based testing with simulated time and for real-time testing in the target environment. Also providing such a solution inevitably means extending existing TRI and TCI interfaces, we try to keep these extensions minimal.

According to the definition of global idleness, we need to detect the situation when all components of the system are locally idle and there are no messages or timeouts pending. We reformulate this definition in terms of necessary and sufficient conditions for detecting global idleness of the closed system. Further we take into account only messages-based communication. Extending the conditions and the solution to procedure-based communication is straightforward.

The closed system consists of a TTCN-3 test system and the SUT. A *distributed* TTCN-3 test system (TS) consists of n test system instances running on different test devices. Further we refer to the test instances i as TS$_i$. Each of the TS$_i$ consists of a TE$_i$, SA$_i$ and PA$_i$. The global idleness requires all the entities to be in the idle state (see condition 1 in Fig. 2). Condition 1 is necessary but not sufficient to decide on the global idleness of the closed system. There still can be some message or timeout pending which can activate one of the idle entities.

"No messages or timeouts pending" means that all sent messages and timeouts are already enqueued at the input ports of the receiving components. When testing with TTCN-3, we should ensure that

- There are no messages pending between the SUT and the TS, i.e. all messages sent by the SA ($SASent2SUT$) are enqueued by the SUT ($EnqdBySUT$) and that all messages sent by the SUT ($SentBySUT$) are enqueued by the SA ($EnqdBySA$) (see conditions 2-3 in Fig. 2).
- There are no remote messages pending at the TCI interface, i.e. the all messages sent by all instances the TE entity via the TCI interface ($TCISEntByTE$) are enqueued at the instances of the TE entity ($TCIEnqd2TE$) (see condition 4 in Fig. 2).
- There no messages pending at the TRI interface, i.e. the number of messages sent by the TE$_i$ via the TRI ($TRISentByTE$) should be equal to the number ($TRIEnqdBySAPA$) of messages enqueued by the SA$_i$ and the PA$_i$, and the number of messages sent by the SA$_i$ and the PA$_i$ is the same as the the number of messages enqueued by the TE$_i$ (see conditions 5-6 in Fig. 2).

Lemma 2. *The closed system is globally idle iff the conditions 1-6 in Fig. 2 are satisfied.*

Further we propose a solution for detecting whether the conditions 1-6 in Fig. 2 are satisfied and for progressing time in the closed system.

4 Distributed idleness detection and time progression in TTCN-3

Detecting global idleness of a distributed system is similar to detecting its termination. We extend the well-known distributed termination detection algorithm of Dijkstra-Safra [2] to decide on the global idleness and to progress time.

In the closed system, each component has a status that is either active or idle. Active components can send messages, idle components are waiting. An idle component can become active only if it gets a message. An active component can always become idle.

In the Dijkstra-Safra's algorithm, termination detection was built into the functionality of components. We separate global idleness detection from normal functionality of a component by introducing an idleness handler for each component of the closed system. Since TTCN-3 is mainly used in the context of black-box testing where we can only observe external actions, we consider the SUT as a single component implementing certain interfaces in order to be tested with simulated time. In a distributed TTCN-3 test system, we consider instances of the TS as single components. To guarantee the correctness of the extension of the algorithm, we assume synchronous communication between a component and its idleness handler.

To decide on the global idleness we introduce a *time manager*. The time manager can be provided as a part of SUT or as the part of the test system. The time manager and the idleness handlers are connected into a unidirectional *ring*.

Time Manager The time manager initializes the global idleness detection, decides on the global idleness and progresses time by sending an idleness token along the ring. The token consists of a global flag and a global message counter. The flag can be "IDLE" meaning that there are not active components in the closed system, "ACTIVE" meaning that maybe one of the components is still active, "TICK" meaning time progression and "RESTART" meaning reactivating the system in the new time slice. The counter keeps track of messages exchanged between the components.

The time manager initiates idleness detection by sending an idleness token with the counter equal to 0 and the flag equal to "IDLE" to the next idleness handler along the ring. The time manager detects global idleness if it receives back the idleness token with the counter equal to zero, meaning there are no messages etc. pending between instances of the TS and the SUT, and the flag equal to "IDLE" meaning that all instances of the TSs and the SUT are idle. Otherwise it repeats idleness detection in the same time slice.

If the time manager detects global idleness, it progresses time by sending the token with flag "TICK" along the ring. After all instances of the TS and the SUT are informed about time progress, the manager reactivates the components of the system by sending the token with flag "RESTART" along the ring. After the reactivation, the time manager restarts idleness detection in the new time slice.

Idleness handler for TS$_i$ We first consider the idleness handlers for TS$_i$. An idleness handler for the SUT is a simplified version of a TS$_i$ idleness handler. A fragment of the Java class `IdlenessHandlerTS` in Fig. 3 illustrates the behavior of an idleness handler for an instance of the TS$_i$. The idleness handler communicates with the other handlers via operation `IdlenessTokenSend()` that allows to receive an idleness token from one neighbor and propagate it further to the next one. For this purpose the idleness handler keeps the reference to the next handler along the ring `NextHandler`. The idleness handler decides of the local idleness of the TS$_i$, propagates the idleness token along the ring and triggers time progression at the PA$_i$. The TS$_i$ is locally idle iff the TE$_i$, the SA$_i$ and the PA$_i$ are idle and there are not messages/timeout pending between the TE$_i$, the SA$_i$ and the PA$_i$.

Messages exchanged by the TE$_i$, the SA$_i$ and the PA$_i$ via the TRI interface are internal wrt. the TS$_i$. Messages exchanged by the TE$_i$ via the TCI interface and the messages exchanged by the SA$_i$ with the SUT are external wrt. the TS$_i$. To keep information about external and internal messages, idleness handler maintains the several local counters. `TRISentByTE` and `TRIEnqd2TE` keep the number of messages sent and enqueued by the TE$_i$ via the TRI interface. `TRISentBySAPA` and `TRIEnqd2SAPA` providing analogous information for the SA$_i$ and the PA$_i$. These four conters are necessary to detect the local idleness of

```
public synchronized void PAIdle(int TRISentByPA,
       int TRIEnqd2PA){
       TRISentBySAPA += TRISentByPA;
       TRIEnqd2SAPA += TRIEnqd2PA;
       idlePA=true; notify(); }

public synchronized void SAIdle(int TRISentBySA, int TRIEnqd2SA,
                                int SASent2SUT, int SUTEnqd2SA){
        TRISentBySAPA += TRISentBySA;
        TRIEnqd2SAPA  += TRIEnqd2SA;
        SASUTcount    += (SASent2SUT-SUTEnqd2SA);
        flagSA=true; idleSA=true; notify();}

public synchronized void TEIdle(int TCISentByTE, int TCIEnqd2TE,
                                int TRISentByTE, int TRIEnqd2TE){
     this.TRISentByTE += TRISentByTE;
     this.TRIEnqd2TE  += TRIEnqd2TE;
     TCITEcount       += (TCISentByTE-TCIEnqd2TE);
     flagTE=true; idleTE=true; notify(); }

public synchronized void PAActivate(){idlePA=false; }
public synchronized void SAActivate(){idleSA=false; }
public synchronized void TEActivate(){idleTE=false; }

 public synchronized void run(){
      IdlenessToken msg=null;
      for(;;) {
         if (idlePA & idleSA & idleTE &
             (TRISentByTE==TRIEnqd2SAPA) &
             (TRIEnqd2TE==TRISentBySAPA) &
             (buffer!=null)) {
                 msg=buffer; buffer=null;
                    if (msg.tag==IdlenessToken.IDLE |
                        msg.tag==IdlenessToken.ACTIVE){
                         if (flagTE | flagSA)
                             {msg.tag=IdlenessToken.ACTIVE;}
                         if (flagTE) {msg.count+=TCITEcount;
                                      TCITEcount=0; flagTE=false;}
                         if (flagSA) {msg.count+=SASUTcount;
                                      SASUTcount=0; flagSA=false;}
                        }
                    if (msg.tag==IdlenessToken.TICK) {
                         TRISentByTE=0; TRIEnqd2TE=0;
                         TRISentBySAPA=0; TRIEnqd2SAPA=0;
                         SASUTcount=0; idlePA=false;
                         flagSA=true; flagTE=true;
                         pa.Tick();
                         }
                   if (msg.tag==IdlenessToken.RESTART)
                      {pa.Restart();}
                   NextHandler.IdlenessTokenSend(msg);
                   }
           ..... }
  }
```

Fig. 3. Idleness Handler for TS_i

the TS_i. **`TCITEcount`** and **`SASUTcount`** keep the number of external messages exchanged by the TS_i via the TCI interface and with the SUT.

Two flags (for TE_i and SA_i) kept by the idleness handler show whether **`TCITEcount`** or/and **`SASUTcount`** respectively contain the up-to-date information that is not known to the idleness token. Since the PA_i communicates neither with the SUT nor with the other instances of the TS, information on messages etc. exchanged by the PA_i is only important to detect local idleness of the TS_i. Therefore, there is no need for a flag for the PA_i. The idleness handler keeps information on the status of the TE_i, SA_i and PA_i in the variables **`idleTE`**, **`idleSA`** and **`idlePA`** respectively.

Initially, the statuses are *false* meaning TS_i is potentially active. The flags are initiated to *true*, meaning the idleness token does not have the up-to-date information about messages exchanged by the TS_i via TCI and messages exchanged by the TS_i and the SUT. The counters are initially zero.

To detect global idleness, the TE_i, the SA_i and the PA_i should support a number of interfaces. To detect idleness of a TE_i, we assume TCI-operation `TEIdle(int TCISentByTE, int TCIEnqd2TE, int TRISentByTe, int TRIEnqd2TE)` called by a TE_i at the idleness handler, when an active TE_i becomes idle. The first two parameters keep track of external messages exchanged via the TCI and the last two parameters capture the same information for internal messages. Calling this operation leads to change of TE_is status $idleTE$ to $false$, updating the local counters `TRISentByTE`, `TRIEnqd2TE` and `TCITEcount` and setting flag $flagTE$ to $true$.

To detect idleness of the PA_i, we assume operation `PAIdle(int TRISentByPA, int TRIEnqd2PA)` called by PA at the idleness handler when an active PA_i becomes idle. Two parameters correspond to the number of messages sent and the number of messages received by the PA_i via the TRI respectively. Calling `PAIdle` at the idleness handler leads to changing variable `idlePA` to *true* and updating the local counters `TRISentbySAPA` and `TRIEnqd2SAPA`.

To detect local of an SA_i we assume operation `SAIdle(int TRISentBySA, int TRIEnqd2SA, int SASent2SUT, int SUTEnqd2SA)` called by SA at the idleness handler when an active SA_i becomes idle. `TRISentBySA` and `TRIEnqd2SA` denote the numbers of internal messages sent and enqueued by the SA_i. Parameters `SASent2SUT` and `SUTEnqd2SA` keep the analogous information about external messages exchanged between the SA_i and the SUT. Calling `SAIdle()` leads to changing the status of SA_i to *true*, updating the local counters and changing the flag of SA_i to *true*.

The TS_i can be activated by receiving an external message. To detect an activation, we assume operation `TEActivate()` called by the CH at the idleness handler when a remote message is being enqueued at the idle TE_i, operation `SAActicvate()` called by the SA_i at the idleness handler when an idle SA_i gets a message or a timeout and operation `PAActivate()` called by the PA_i at the idleness handler when an idle PA_i is activated. Calling these operation leads to updating the idleness status of the corresponding entity to *false*.

Checking the local idleness of the TS_i is implemented by the method `run()`. The local idleness of the TS_i is detected iff status variables `idleSA`, `idlePA` and `idleTE` are *true* and all messages internal messages sent via the TRI interface have been reported enqueued. This is expressed by the local idleness condition at the first if-statement of the method `run()`.

If the local idleness conditions are satisfied and the idleness handler is in the possession of the idleness token with flag "IDLE" or "ACTIVATE" , the handler propagates the up-to-date information about external messages exchanged by the TS_i to the time manager by updating the idleness token and sending it further along the ring.

If the `flagTE` is *true* then the number of external messages exchanged by the TS_i via TCI has changed since the last detection round, thus the idleness handler adds `TCIcount` to the counter of the idleness token. If the number `flagSA` is *true*, the number of messages exchanged with the SUT has changed thus the idleness handler updates the token's counter by adding the number of messages sent by the SA$_i$ to the SUT and subtracting the number of messages from the SUT enqueued by the SA$_i$. If at least one of the local flags is *true* the flag of the token changes to “ACTIVATE” meaning maybe one of TS instances or the SUT is still active.

If the idleness handler gets an idleness token with flag ”TICK”, it prepares for detecting idleness in the next time slice by setting all the flags to *true*, setting `idlePA` to *false*, calling operation `Tick()` at the PA$_i$, and sending the token to the next handler along the ring. Upon `Tick()`, the PA$_i$ look-ups the timers ready to expire in the new time slice. If the idleness handler gets an idleness token with flag ”RESTART”, it calls operation `Restart()` at the PA$_i$ and propagates the token to the next idleness handler. Upon `Restart()`, the PA$_i$ expires the ready timers. The status of TE$_i$ and of SA$_i$ remains idle until explicit activation because both TE$_i$ and of SA$_i$ may remain idle during a time slice.

Idleness Handler for SUT We assume that the SUT implements an idleness handler for itself. The idleness handler of the SUT propagates the idleness token iff the SUT is idle. When the SUT’s handler propagates the token, it adds to the token’s counter the number of messages etc. the SUT has sent to the test system, subtracts the number of messages etc. from the TS enqueued by the SUT since the SUT has been idle last time. The handler also has to change the token’s flag to ”ACTIVATE” if the SUT has been activated at least once since the last visit of the idleness token.

Corollary 3. *The solution for simulated time proposed in Section 4 detects global idleness iff the conditions 1-6 in Fig. 2 are satisfied.*

5 Conclusion

In this extended abstract we proposed a framework for host-based simulated-time testing with TTCN-3. Simulated time is suitable for testing and verifying a class of systems where delays are significantly larger than the duration of normal events in the system. When used for testing, simulated time ensures that an SUT and a test system agree on time and advance time together. This contributes to the repeatability of test results when testing embedded software and also solves some time-related debugging problems typical for distributed embedded systems.

In [1] we proposed host-based testing with *simulated time* for non-distributed applications. The solution provided in this paper is suitable for distributed testing. It allows to use the same test suites for simulated time and for real time testing. The solution can be implemented only on the level of test adapters. The full paper will provide proofs and also discuss which kind of time constraints can adequately tested with simulated time.

References

1. S. Blom, N. Ioustinova, J. van de Pol, A. Rennoch, and N. Sidorova. Simulated time for testing railway interlockings with TTCN-3. In C. Weise, editor, *FATES’05*,

volume to appear of *Lecture Notes in Computer Science*, pages 10–25. Springer, 2005.

2. E. W. Dijkstra. Shmuel Safra's version of termination detection. note EWD998-0, Univ. Texas, Austin, 1987.
3. ETSI ES 201 873-1 V2.2.1 (2003-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. ETSI Standard.
4. ETSI ES 201 873-4 V2.2.1 (2003-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics. ETSI Standard.
5. ETSI ES 201 873-5 V1.1.1 (2005-06). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI). ETSI Standard.
6. ETSI ES 201 873-6 V1.1.1 (2005-06). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI). ETSI Standard.
7. J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An Introduction into the Testing and Test Control Notation (TTCN-3). *Computer Networks, Volume 42, Issue 3*, pages 375–403, June 2003.
8. S. Hendrata. Standardisiertes testen mit ttcn-3: Erhhung der zuverlssigkeit von software-systemen im fahrzeug. *Hanser Automotive: Electronics+Systems*, (9-10):64–65, 2004.
9. TTMedal. Testing and Testing Methodologies for Advanced Languages. http://www.tt-medal.org.
10. C. Willcock, T. Deiß, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3.* Wiley, 2005.