Simulated Time for Host-Based Testing with TTCN-3 ⋆

Stefan Blom [a], Thomas Deiß [b], Natalia Ioustinova [c], Ari Kontio [d], Jaco van de Pol [c,e], Axel Rennoch [f], Natalia Sidorova [e]

[a] *Institute of Computer Science, University of Innsbruck, 6020 Innsbruck, Austria*

[b] *Nokia Research Center, Meesmannstrasse 103, D-44807 Bochum, Germany*

[c] *CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands*

[d] *Nokia Research Center, Itämerenkatu 11-13, 00180 Helsinki, Finland*

[e] *Department of Mathematics and Computer Science, Eindhoven University of Technology, Den Dolech 2, 5612 AZ Eindhoven, The Netherlands*

[f] *Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, D-10589, Berlin, Germany*

Abstract

Prior to testing embedded software in a target environment, it is usually tested in a host environment used for developing the software. When a system is tested in a host environment, its real-time behavior is affected by the use of simulators, emulation and monitoring. In this paper, the authors provide a semantics for *host-based testing* with *simulated time* and propose a simulated-time solution for *distributed* testing with TTCN-3, which is a standardized language for specifying and executing test suites. The paper also presents the application of testing with simulated time to two real-life systems.

Corresponding author: Natalia Ioustinova, tel: +31-20-5924247.

Key words: testing, distributed systems, real time, discrete time, simulated time, telecom, railway control systems, TTCN-3.

⋆ This work has been done within the project "TT-medal. Test and Testing Methodologies for Advanced Languages (TT-Medal) sponsored by Information Technology for European Advancement program (ITEA)" [1]

Email addresses: Stefan.Blom@uibk.ac.at (Stefan Blom), thomas.deiss@nokia.com (Thomas Deiß), ustin@cwi.nl (Natalia Ioustinova), ari.kontio@nokia.com (Ari Kontio), Jaco.van.de.Pol@cwi.nl (Jaco van de Pol), axel.rennoch@fokus.fhg.de (Axel Rennoch), n.sidorova@tue.nl (Natalia Sidorova).

3 April 2007

1 Introduction

Software testing is of key importance for the quality of embedded systems. Although no testing regime can find all errors, testing helps to detect bugs and increases confidence in the trustworthiness of a system. Prior to testing in a *target* environment embedded software is usually tested in a *host* environment used for its development. When being tested, embedded software is executed to check whether it meets certain quality requirements and to detect failures. The requirements are expressed in the form of *test cases* that are executed by a *test system*. The test system sends stimuli to the system under test (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. Wrong timing of stimuli from the test system to the SUT may lead to failures even in a correct system. Therefore, a test system should be an adequate representation of a real environment of the SUT with respect to time behavior.

Testing is an incremental process where components and (sub-)systems are integrated and tested. Many software and hardware components and services are not available for testing until the end of the development process. When testing a component or a sub-system, a host environment makes use of *simulation* and *emulation* techniques to execute it. *Monitoring* and *instrumentation* are used to observe the external behavior of embedded software. Ideally, the effect of simulation, emulation and monitoring on the real-time behavior should be negligible, not influencing test results. However, solutions satisfying this requirement are often expensive and product-specific. In case the effects of simulation, emulation and monitoring significantly change the real-timed behavior, real-time testing might miss to find errors or might report non-existing errors.

A simple naive solution would be to test with *scaled time*. Scaled time is calculated as initial time plus the product of a time factor and the difference between the current physical time and the initial moment. The larger the factor the faster tests can be executed. Choosing the optimal time factor is, however, not trivial. The effects of a host environment are difficult to calculate particularly for distributed systems. Moreover they may vary through the execution. Thus using scaled time does not always solve the problem.

In this paper the authors propose a time semantics for testing with *simulated time*. In simulated time, the system clock is modeled by a discrete logical clock and time progression is modeled by a `tick` action. 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. For testing distributed systems, simulated time facilitates debugging by synchronizing time progression at all nodes and providing the ability to stop or suspend time progression. Simulated time improves thus controllability of testing and debugging. In general, simulated time can be seen as scaled time with a dynamic time factor that is determined automatically. Since the factor is dynamic, the approach is efficient in the case of varying computation times.

The authors have chosen the Testing and Test Control Notation version 3 (TTCN-3) to implement the proposed solution for testing with simulated time. TTCN-3 is a language for specifying test suites. It has a syntax and operational semantics standardized by the European Telecommunications Standards Institute (ETSI) [2–4]. TTCN-3 was originally developed for real-time testing of telecommunication systems. A TTCN-3 test executable has predefined standard interfaces [5,6,4]. Standardized interfaces of TTCN-3 allow the definition of test suites on a level independent of a particular implementation or platform, which significantly increases the reuse of TTCN-3 test suites. TTCN-3 interfaces provide support for distributed testing, which makes TTCN-3 particularly beneficial for testing embedded systems. TTCN-3 has already been successfully applied to the testing of embedded systems not only in telecommunication but also in automotive and railway domains [7,8]. The time semantics of TTCN-3 has been intentionally left open to enable the use of TTCN-3 with different time semantics [3]. Implementing simulated time using existing TTCN-3 interfaces is, however, not straightforward.

In this paper the authors provide a framework for host-based simulated-time testing with TTCN-3. They define the semantics of simulated time for host-based testing with TTCN-3 and discuss which time constraints can be *adequately* tested in a host environment with simulated time. Furthermore, the authors provide a solution for implementing simulated time for a *distributed* TTCN-3 test system and argue its correctness. The solution allows the execution of TTCN-3 test suites in real and in simulated time without having to change them.

The rest of the paper is organized as follow. In Section 2, some aspects of host-based testing of timed systems are discussed. In Section 3, the authors define a semantics for host-based testing with simulated time and discuss the adequacy of test results for host-based testing with real and with simulated time. In Section 4, two applications of host-based testing with simulated time are illustrated: one from the railway domain and another one from the telecom domain. Section 5 provides a brief survey on the general structure of a distributed TTCN-3 test system and formalizes requirements to a distributed TTCN-3 test system with simulated time. Section 6 contains requirements to the entities of a TTCN-3 test system which enable implementing simulated

time. Section 7 presents a solution for simulated time host-based testing with TTCN-3. Section 8 concludes the paper and provides a discussion of related work.

2 Host-based testing of timed systems

Testing software in a *target* environment can be expensive and even impossible due to the absence of the target environment. Software failures that occur when testing *safety-critical* embedded systems (e.g. railway control systems) in a target environment can be not only dangerous but also disastrous. The time period when the target environment is available for testing software can be very short, for example updating banking software often happens at night. Therefore software is first tested in a host environment to find and fix as many software errors as possible prior to running any test case in the target environment. Increasing the degree of *host-based testing* before testing in a target environment is a widely accepted approach to ensure the quality of developed software.

Embedded systems are typically *timed*, i.e. a system has to interact with its environment under certain timing constraints. Timing constraints are imposed both on the system and on its target environment. The environment is responsible for the timing of stimuli to the system. In this section the authors discuss how a host environment affects the timed behavior of embedded software.

In host-based testing, a target environment is replaced by an environment simulation. Ideally, the timing of stimuli in the *simulated environment* should not differ significantly from the timing of stimuli in the target environment. Developing simulators that adequately mimic the target environment is, however, often unfeasible due to high costs and time limitations imposed on the whole testing process.

When executed, embedded software interacts with an operating system (OS) that provides communication, time, scheduling and synchronization services. In host-based testing, the services of the target operating system are often emulated. To obtain adequate test results, the emulation should be accurate with respect to the time required by the target operating system to provide the above mentioned services. High timing accuracy is difficult to achieve when *emulating the target OS*.

To assess the correctness of embedded software it is necessary to observe the timing and the order of external events in the SUT. This is usually done by *monitoring* or by *instrumentation*. To obtain adequate test results, changes introduced by monitoring or instrumentation into the real time behavior of

the SUT should be *negligible*, i.e. they should not lead to changes in the timed behavior of the SUT.

Monitoring introduces a software tool or hardware equipment that runs concurrently with an SUT and logs observable events. Non-intrusive monitoring, i.e. monitoring not affecting real time behavior of an SUT is expensive and hard to achieve because it requires a product-specific hardware-based implementation.

Instrumentation enables observability by inserting additional code into the code of an SUT. This extra code collects information about the SUT's behavior during test execution. To overcome the probe effects induced by instrumentation, instrumentation should be kept active in the real product so that the test version and the real version behave similarly. This solution would, however, increase the size of the code and sometimes significantly decrease the system's performance.

Simulations, emulations of the target operating system, monitoring and instrumentation affect the real-time behavior of an SUT. This restricts the class of timing constraints that can be validated with host-based testing. Timing constraints are often divided into two categories: performance constraints and behavioral constraints [9]. *Performance constraints* are concerned with setting limits on the latency and throughput of a system. *Behavioral constraints* specify logical correctness of a system. A behavioral requirement can, for example, state that a system or a component produces an output if a certain stimulus does not arrive within certain time limits. This paper focuses on testing logical correctness rather than performance.

3 Simulated time for host-based testing

In this section, the time semantics for host-based testing with simulated time is defined and guidelines on using testing with simulated time are formulated. The adequacy of results obtained by host-based testing in real and in simulated time is discussed.

Normally, it is assumed that real-time systems operate in "real" continuous time. Although an environment of an embedded system changes continuously, the system observes only snapshots of the environment. That provides a natural discretization of the environment's behavior. Moreover, a less expensive, discrete time solution is, for many systems, as good as dense time in the modeling sense, and better than the dense one when testing and verification are concerned [10,11]. Therefore, the authors choose to work with discrete time.

An embedded system should react to all important changes that happen in its environment. That means that the system should take snapshots of the environment often enough to catch the important changes. Moreover, the system's computations and communication should be fast enough for the system to respond to the environment's changes on time.

The authors consider the class of systems where (i) the snapshots are taken often enough to allow to the system to see the important changes in the environment and (ii) external delays are significantly larger compared with the duration of normal computations and communication within the system. If the system satisfies these requirements, the duration of computations within the system is *negligible* compared to the external delays and can be safely treated as *instantaneous* or zero time.

Host-based testing makes use of environment simulations, OS emulations, monitoring and instrumentation which significantly affect timed behavior of the system. To obtain test results independent of the changes caused by OS emulations, the authors assume that the OS services are instantaneous (i.e. provided in zero time). To get rid of probe effects induced by monitoring or instrumentation, they are treated as being instantaneous as well.

The assumption that communication and computation is instantaneous implies that time progress can never take place if some action is still enabled, or in other words, time progress has the least priority in the system and may take place only when the system is *idle*. This property is known as *minimal delay* or *maximal progress* [12]. The assumption that actions are instantaneous does not prevent one from modeling actions that take some time. Whenever necessary, an explicit time delay can be put before an action or an action can be split into start- and finish-events. Time progress is referred to as `tick` and the period of time between two `tick`s is referred to as a time slice.

The concept of timers is usually used to express time-dependent behavior. A timer can be either active or deactivated. An active timer keeps the information about the time left until its expiration. When the delay becomes zero, the timer expires and becomes deactivated. An expiration of a timer produces a timeout. If the system is idle, the *time progresses* by action `tick` by the minimal timer value of the currently active timers. If the delay left until timer expiration reaches zero, the timer expires within the current time slice. Timers ready to expire within the same time slice expire in an arbitrary order.

For testing purposes, the focus is on *closed* systems (a test system together with an SUT) consisting of multiple components communicating with each other. A *component* is *idle* if and only if it cannot proceed by performing computations, receiving messages or consuming timeouts. The idleness of a single component is called *local idleness*. A *system* is *idle* if and only if all

components of the system are idle and there are no messages or timeouts that can still be received during the current time slice. Such messages and timeouts are referred to as *pending*. The idleness of the whole system is called *global idleness*. Subsequently, this time semantics is called *simulated time*.

Definition 1 (Global Idleness) *A closed system is* globally idle *if and only if all components are locally idle and there are no messages and no timeouts pending.*

To assess the adequacy of testing with simulated time, consider first the nature of inadequate test results. A *false positive* refers to situations where an execution of a system passes a test case but there exists another execution of the system in the target environment on which the test case fails. A *false negative* refers to situations where a test case fails for some execution but for all executions of the system in the target environment the test case passes. Note that false positives are possible both in the host and in the target environment whereas false negatives are only possible in the host one.

False negatives can be caused by an inadequate host environment. Building an adequate host environment is a challenge both in the real and in the simulated time frameworks. However, the discretization in the environment's behavior assumed for simulated time may alleviate this challenge to some extent. Even if a test case passes in the target environment, failing the test case in the host environment shows that there can be an environment where this test case fails. This knowledge can be useful when migrating the system to another platform or modifying the target environment.

Simulated time is convenient for debugging because it allows suspending time progression in the SUT and the test system, inspecting the current situation with a debugger without stopping test execution completely and later on resuming the test execution from the suspension point. Although helpful in many ways, host-based testing with simulated time has the same limitations as host-based testing in real time and testing in general as far as false positives and false negatives are concerned.

4 Case Studies

In this section, two case studies illustrate the applicability of host-based testing with simulated time. The first one is from the railway domain and the second is from the telecommunication domain. [1]

[1] Non-disclosure agreements with the companies involved prevent release of any details about the test systems, the SUTs and statistics on test execution.

Railway Interlockings Railway control systems consist of three layers: infrastructure, logistic, and interlocking. The infrastructure represents a railway yard; the logistic layer is responsible for the interface with human experts, who give control instructions for the railway yard to guide trains. The interlocking guarantees that the execution of these instructions does not cause train collisions or derailments. If the interlocking considers a command as unsafe, the execution of the command is postponed until the command can be safely executed or discarded.

Testing the interlocking in a target environment is safety-critical and expensive. Therefore, it may take place only when a high level of confidence in the logical correctness of the system has been achieved by host-based testing. The authors tested the interlocking software in a host environment where the interlocking software and the target environment have been simulated.

The tested interlocking system is based on Vital Processor Interlocking (VPI) that is used nowadays in Australia, some Asian countries, Italy, the Netherlands, Spain and the USA. A VPI is implemented as a machine which executes hardware checks and a program consisting of a large number of guarded assignments. The assignments reflect dependencies between various objects of a specific railway yard like points, signals, level crossings, and delays on electrical devices and ensure the safety of the railway system.

The VPI program has several read-only input variables, auxiliary variables used for computations and several writable variables that correspond to the outputs of the program. The program specifies a control cycle that is repeated with a fixed period by the hardware. The control cycle consists of two phases: an active phase and an idle phase. The active phase starts with reading new values for input variables. The infrastructure and the logistic layer determine the values of the input variables. After the values are latched by the program, it uses them to compute new values for internal variables and finally decides on new outputs. The values of the output variables are transmitted to the infrastructure and to the logistic, where they are used to manage signals, points, level crossings and trains. Here we assume that the infrastructure always follows the commands of the interlocking. The rest of the control cycle the system stays idle.

The length of the control cycle is fixed by the design of the system. Delays are used to ensure the safety of the system. A lot of safety requirements to VPIs are timed. They describe dependencies between infrastructure objects over a period of time. The objects of the infrastructure are represented in the VPI program by input and output variables. Thus the requirements defined in terms of infrastructure objects can easily be reformulated in terms of input and output variables of the VPI program. Hence VPIs are time-critical systems.

For safety reasons, time spent by the system on communication and computation must be much smaller than the minimal time within which the system must react to the changes in the environment. Thus the system satisfies the requirements formulated for application of simulated time and simulated time may be safely used for testing the system in the host environment.

The authors performed host-based testing of the interlocking with simulated time. The standard requirements were formulated for the interlocking with a general configuration. All requirements were of the form: initial situation, action, expected results. To develop test cases, we had to (1) map the general configuration to a particular configuration of the station; (2) map the initial situation to the stimuli for the SUT; (3) map the final situation to the output values expected from the SUT; (4) define default values for objects of the station that are not involved in the tested situation but still can influence it; (5) formulate time requirements for tested actions. We specified the test cases in TTCN-3.

When we implemented a simulator for VPI, two things quickly became clear: First, the simulator is a lot faster than the real hardware. Second, the running time of the simulator is far less predictable. The reason for this unpredictability is that the simulator spends less time on a slice without events than on a slice in which an event happens. In practice the difference is often an order of magnitude. Typically, time slices without events outnumber those with events by an order of magnitude, so the gain of using simulated time is considerable.

The experiments showed that our approach to host-based testing with simulated time allows to detect violations of safety requirements in interlocking software. The conclusion drawn from the experiments is that testing with simulated time is an adequate host-based testing method for this type of systems. Moreover, it is a low-cost method compared to its alternatives.

GSM/WCDMA Mobile Phone Application The system from the telecommunication domain is embedded software for a dual-mode mobile terminal that supports both WCDMA (Wideband Code Division Multiple Access) [13] and GSM (Global System for Mobile Communication) [14]. Besides other control functionality, the software considered implements the handover control between WCDMA and GSM: When a phone user first establishes a voice call using WCDMA and then moves outside of WCDMA coverage, the phone is able to continue the voice call service over GSM without noticeable disturbance. The SUT is a timed system. For example handover from WCDMA to GSM should be accomplished within certain time bounds. Otherwise handover would become visible to an end-user.

The authors tested the software in a host environment. The services of the target operating system were emulated on a workstation and executed together

with the software. In the test system, representing the network side, the peer entity to the tested software was implemented in TTCN-3. The lower protocol layers were implemented in a C-based library for finite state machines and the air interface was replaced by an Ethernet connection.

Typically, a message exchanged between the SUT and the test system causes several messages to occur in the lower protocol layers. One could view the occurrence of messages within the complete system as bursty. Processing these message bursts was fast compared to the duration of timers in the tested software. Therefore the requirements for applying simulated time are satisfied in this case. We used host based testing with simulated time to check behavioral time-dependent features of the SUT.

To implement simulated time, idleness had to be detected in the tested software, the protocol implementations, and in the TTCN-3 part of the system. At the time of implementing this system the TTCN-3 control interface had not been defined and the authors used a proprietary API. This API was similar to the corresponding TCI operations and the idleness detection could be implemented.

Testing the SUT with the developed test system allowed to debug the test system. Throughout the test execution the SUT could be suspended and inspected with a debugger. These inspection time intervals could be arbitrary long. As the SUT could not become idle while being suspended, no timer expired in such an interval and test case execution could continue after such a long interval. The test system was implemented and shown to be working.

The solution for testing with simulated time in TTCN-3 is also applicable to other systems with similar characteristics.

5 TTCN-3 test systems

In this section, an overview of a distributed TTCN-3 test system is given and requirements for implementing a distributed TTCN-3 test system are formulated.

TTCN-3 is a language for the specification of test suites [15]. The specifications can be generated automatically or developed manually. A specification of a test suite is a TTCN-3 *module* which might import other modules. Modules are the TTCN-3 building blocks that 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.

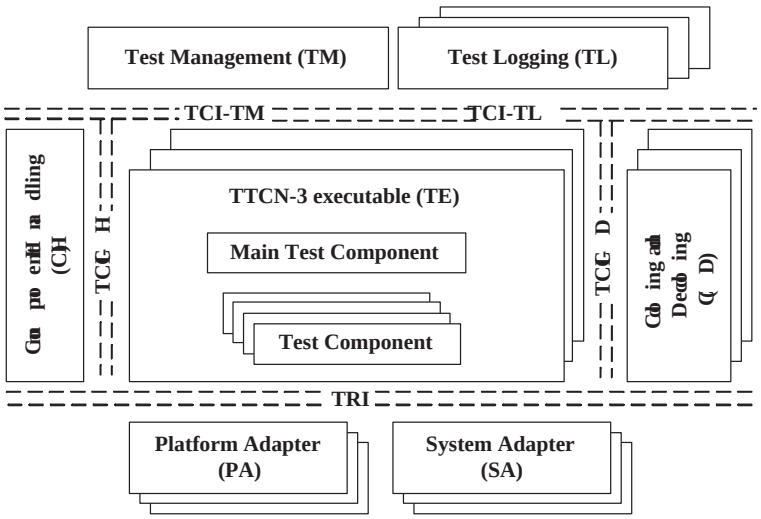

Fig. 1. General structure of a distributed TTCN-3 test system (TS)

A test suite is executed by a TTCN-3 test system whose general structure is defined by the TRI (TTCN-3 Runtime Interface) standard [5] and illustrated in Fig. 1. The TTCN-3 executable (TE) entity actually executes or interprets a test suite. A call to a test case can be seen as an invocation of an independent program. Starting a test case creates a *configuration*. A configuration consists of several test components running in parallel. The first test component created at the starting point of a test case execution is the main test component (MTC). The test components communicate with each other and with an SUT by *message passing* or by *procedure calls*. For communication purpose, a test component owns a set of ports. Each port has **in** and **out** directions: infinite FIFO queues are used to represent the **in** directions; the **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 the information about the time left until its expiration. When this time becomes zero, the timer expires and becomes deactivated. The expiration of a timer produces a timeout. The timeout is enqueued at the component to which the timer belongs.

The Platform Adapter (PA) implements timers and operations on them. The System Adapter (SA) implements communication between a TTCN-3 test 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 TTCN-3 runtime interface (TRI) allows the TE entity to invoke operations implemented by the PA and the SA.

The Test Management (TM) entity controls the order of the invocation of modules. The 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 a format suitable to be exchanged 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 Control Interface (TCI) [6].

A test system (TS) can be distributed over several test system instances TS_1, ..., TS_n each running on a separate test device [16]. Each TS_i has an instance TE_i of the TE entity equipped with the corresponding SA_i, the test logging (TL) entity TL_i, a PA_i and a coder/decoder CD_i running on the node. One of the TE's instances is identified to be the main one. It starts executing a TTCN-3 module and computes final testing results.

5.1 Simulated time requirements for a TTCN-3 test system

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

The goal of our research is to provide a solution for implementing simulated time for distributed TTCN-3 test systems. Developing a test suite for host-based testing costs time and effort. Therefore, the test suites developed for host-based testing with simulated time should be reusable for real-time testing in the target environment. Here, the authors provide a solution that can be implemented on the level of adapters instead 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. Although providing such a solution inevitably means extending the TRI and TCI interfaces, the authors try to keep these extensions minimal.

According to the definition of global idleness (Def. 1), it is important to detect situations when all components of the system are locally idle and there are no messages and no timeouts pending. Here this definition is reformulated in terms of a necessary and sufficient condition to detect global idleness of the closed system.

Procedure-based communication in TTCN-3 treats procedure calls, replies, and exceptions as special kinds of messages. Analogous to the `send` and `receive` operations for messages, the operations `call` and `getcall` are responsible for sending and receiving procedure calls. The operations `reply` and

$$[\forall i = 1..n\colon idle(\mathrm{TE}_i) \wedge idle(\mathrm{PA}_i) \wedge idle(\mathrm{SA}_i)] \wedge idle(\mathrm{SUT}) \quad (1)$$

$$\textstyle\sum_{i=1..n} \mathrm{SASentSUT}_i = \mathrm{EnqdSUT} \quad (2)$$

$$\mathrm{SentSUT} = \textstyle\sum_{i=1..n} \mathrm{EnqdSA}_i \quad (3)$$

$$\textstyle\sum_{i=1..n} \mathrm{TCISentTE}_i = \sum_{i=1..n} \mathrm{TCIEnqdTE}_i \quad (4)$$

$$\forall i = 1..n : \mathrm{TRISentTE}_i = \mathrm{TRIEnqdSAPA}_i \quad (5)$$

$$\forall i = 1..n : \mathrm{TRISentSAPA}_i = \mathrm{TRIEnqdTE}_i \quad (6)$$

Fig. 2. Global Idleness Conditions

`getreply` are used to send and to receive replies of procedures. The operations `raise` and `catch` do the same for exceptions [4]. Therefore, the treatment of procedure calls, replies and exceptions in idleness detection would not differ much from treating messages when detecting idleness. Handling calls, replies, and exceptions on the TRI and the TCI levels is also similar to handling messages [5,6]. For the sake of simplicity, in the rest of the paper the authors restrict communication to message-based communication.

The closed system consists of a TTCN-3 test system and an SUT. A *distributed* TTCN-3 test system (TS) consists of n test system instances running on different test devices. In the following the test instance i is denoted as TS_i. Each TS_i consists of a TE_i, SA_i and PA_i. 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 global idleness of the closed system. There still can be messages or timeouts 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, the following conditions have to be ensured:

- There are no messages pending between the SUT and the TS, i.e. all messages sent by the SA (SASentSUT) are enqueued by the SUT (EnqdSUT) and all messages sent by the SUT (SentSUT) are enqueued by the SA (EnqdSA) (see conditions (2-3) in Fig. 2).
- There are no remote messages pending at the TCI interface, i.e. all messages sent by all instances of the TE entity via the TCI interface (TCISentTE) are enqueued at the instances of the TE entity (TCIEnqdTE) (see condition (4) in Fig. 2).
- There are no messages pending at the TRI interface, i.e. the number of messages sent by every TE_i via the TRI (TRISentTE) should be equal to the number (TRIEnqdSAPA) of messages enqueued by the corresponding SA_i and PA_i, and the number of messages sent by every SA_i and PA_i TRISentSAPA is the same as the number of messages enqueued by the corresponding TE_i, TRIEnqdTE (see conditions (5-6) in Fig. 2).

Proposition 2 *A closed system is globally idle if and only if conditions (1-6) in Fig. 2 are satisfied.*

Thus to implement simulated time for TTCN-3, it is important to detect situations in which conditions (1-6) in Fig. 2 are satisfied and enforce time progression in the form of `tick` actions.

6 Local idleness of PA, SA and TE

The entities of the test system have to provide information on their status to detect global idleness. This section defines the requirements on the behavior of PA, SA and TE that have to be satisfied to enable implementing a TTCN-3 test system with simulated time. For the sake of readability, the requirements are formulated in terms of messages that should be issued and received by PA, SA and TE entities to report their status and to progress time. (Note that these requirements can be easily reformulated in terms of interfaces provided and required by PA, SA, TE and CH entities.)

A PA_i is responsible for implementing timers. It is assumed that the PA_i maintains the list of deactivated timers, the list of timers ready to expire, and the list of active timers that are not ready to expire in the current time slice. A PA_i is *idle* iff it does not perform any computations and none of the timers is ready to expire. Otherwise the PA_i may proceed with expiring the timer and sending a timeout to the TE_i. Local idleness of the PA_i is easy to detect by checking the list of ready timers. Whenever the list of ready timers becomes empty, PA_i sends a message IDLE parametrized by the number TRISENT of messages sent and the number TRIENQD of messages received and enqueued by the PA_i via the TRI respectively.

The list of ready timers has to be updated at each time-progression step. To enforce time progression, an idle PA_i has to be able to receive message TICK. On receiving this message, the PA_i updates the list of ready timers. To trigger the expiration of timers in the next time slice, the PA_i has also to be able to receive message RESTART. Receiving this message triggers expiring all ready timers by the PA_i.

Another possibility to activate the PA_i is to start a timer with zero-delay. In this case the timer is ready to expire in the current time slice. Every time the TE_i calls the TRISTARTTIMER operation at an idle PA_i, the PA_i has to send message ACTIVE to indicate its activation within the same time slice.

An SA_i is *idle* iff it does not perform any computations and there are no messages the SA_i still can deliver to the TE_i or to the SUT within the same

time slice. An idle SA_i becomes active iff it receives a message from the TE_i or from the SUT. In case the SA_i makes use of timers (for example to mimic channels with delays), it can also be activated by timeouts from the PA_i.

An active SA_i issues message IDLE when the SA_i has no messages to deliver in the current time slice. This message is parametrized by the number TRISent of messages sent by the SA_i, the number TRIEnqd of messages enqueued at the SA_i, and by SUTSent and SUTEnqd, which provide the analogous information about messages exchanged between the SA_i and the SUT.

In case an idle SA_i receives a message from the SUT or from TE_i, or a timeout from the PA_i, it reports this activation by sending message ACTIVE. Note that queues of the SA_i do not have to be empty when reporting local idleness. Delivering messages can be delayed for one or more time slices.

There can be several test components running in a single TE_i. Each test components can be either active or idle. Whenever a message or a timeout is enqueued at a port of a component, this test component becomes *active*. Whenever a test component is waiting for a message or a timeout to receive but there are no messages to process, the *test component is idle*. A TE_i is *idle* iff all test components running on it are idle and there are no messages and timeouts pending in the TE_i.

The goal of this work is to obtain a TTCN-3 test suite that is executable both with simulated and with real time, without having to introduce changes in order to switch from one mode to another. Therefore, detecting the idleness of a TE_i has to be performed by the TE_i itself. For test execution, the TE_i keeps track of components running on it and messages exchanged via the TCI and TRI interfaces. Moreover, the TE_i also has access to the content of the internal queues. Thus the functionality of a runtime system can be easily adapted to detect the situations when all test components running on a test device are idle.

A TE_i should send message IDLE when the active TE_i becomes idle. The message is parametrized by TCISent and TCIEnqd, which keep track of messages exchanged via the TCI, and by TRISent and TRIEnqd that capture the same information for messages exchanged via the TRI interface. To detect activating an idle TE_i by a message via the TCI, an idle TE_i activated by a remote message should issue message ACTIVE.[2] Note that the parameters provide the numbers of messages exchanged since the last time an entity has reported idleness or since the initialization (for the first time slice).

[2] Nokia has submitted a change request to ETSI to introduce an operation tciAllAWait(TRIcounter1, TRIcounter2, TCIcounter1, TCIcounter2) with the same functionality to the TCI interface. The change request is currently pending.

7 Global idleness detection

The conditions for detecting global idleness are similar to the conditions that have to be detected to decide on a termination of a distributed system consisting of N components communicating with each other [17]. In this section, the authors extend the well-known distributed termination detection algorithm of Dijkstra-Safra [18] to detect global idleness and to implement time progression.

Distributed termination detection algorithm of Dijkstra-Safra The distributed termination detection algorithm of Dijkstra-Safra [18] detects termination of a system of N components. Each component has a unique identity that is a natural number from 0 to $N-1$. The algorithm distinguishes two kinds of messages: (i) *basic* messages exchanged by the components; (ii) termination detection messages. The main assumption important for the correctness of the algorithm is that the communication is reliable, meaning that no message is lost. (That is a reasonable assumption for embedded systems.)

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 receives a basic message. An active component can become idle without receiving any stimuli from outside. The system terminates only if all components have idle status and all channels are empty. The Dijkstra-Safra algorithm allows one of the components, for example the 0-component, to detect whether termination has been reached.

One cannot decide on termination by only looking at the status of the components. The idle status of the components is necessary but not sufficient. The status of a component changes from idle to active only by receiving a basic message, so one has to keep track of all the messages in the network. To do this, each component has a local message counter. A component decreases its counter when it receives a basic message. When a component sends a basic message, it increases its message counter. Moreover, each component has a local flag. The flag is initially *false*, and it turns *true* only when the component receives a basic message.

The components are connected into a ring that is used to transmit the termination message referred to as a *token*. The termination token consists of a global message counter and a global flag. The 0-component initiates the termination detection algorithm by sending a termination token with the counter equal to 0 and the flag equal to *false* to the next component in the ring. The 0-component expects that no messages are pending in the network and no component has active status, which is to be checked by passing the token along the ring.

If the component with the token has active status, it keeps the token until its status becomes idle. If the component has idle status, it modifies the token by adding its local message counter to the global message counter. If the value of the local flag of the component is *true*, the component sets the global flag to *true*, meaning that maybe one of the system components is still active. Otherwise the global flag remains unchanged. Then the component forwards the token to the next component along the ring. After forwarding the token, the component changes its local flag to *false*, meaning that the token already got the up-to-date information about this component. The termination is detected by the 0-component only if the component gets back the token with the global flag equal to *false* and the sum of the global message counter with the local message counter of the 0-component is zero. In this case the 0-component can be sure that all other components have idle status and there are no messages pending in the FIFO queues representing the channels. Otherwise, the 0-component starts a new round of termination detection by sending a termination token with the counter equal to 0 and the flag equal to *false*.

7.1 An extension of the distributed detection algorithm

The authors extend the Dijkstra-Safra distributed termination detection algorithm to decide on the global idleness of a closed system and to trigger time progression. In the Dijkstra-Safra algorithm, termination detection is built into the functionality of a component. Here the global idleness detection is separated from the 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 one can only observe external actions, the SUT is considered as a single component which has to implement certain interfaces in order to be tested with simulated time. Instances of the TS are considered as single components in a distributed TTCN-3 test system. The authors require synchronous communication between a component and its idleness handler to guarantee the correctness of the extension of the algorithm.

To decide on the global idleness the authors introduce a *time manager* which corresponds to the 0-component in the Dijkstra-Safra algorithm. The time manager can be provided as a part of the SUT or as a part of the test system. The time manager and the idleness handlers are connected into a ring.

Time Manager The time manager initializes the global idleness detection, decides on the global idleness and enforces time progression. In a distributed system, one needs to ensure that all components take the time progression step prior to starting the new time slice.

To ensure correct time progression, the progress of time is divided into two phases: (i) time progress; (ii) reactivation of components in the new time slice. Time progression causes reinitializing of the idleness handlers in the new time slice and propagating time progress to platform adapters. When all instances of the TS and the SUT are informed about time progress, they may issue timeouts in the new time slice. If an instance of TS or the SUT was allowed to issue timeouts immediately after it receives information about time progression, it could lead to receiving messages "from the future" by an instance of the TS or by the SUT where time has not progressed yet. After reactivating all the instances of TS and the SUT, the time manager proceeds with detecting idleness in the new time slice.

Detecting global idleness, time progression, and reactivating the instances of the TS and the SUT in the new time slice are done by sending a token along the ring. As in the original algorithm, the idleness token has a counter, which keeps track of external messages exchanged by instances of the TS and the SUT, and a global flag.

To support time progression and reactivation in the next time slice, the set of values of the global flag carried by the token is extended. In the original algorithm, the global flag was either *true* or *false*. In the extended version, the flag can be "IDLE", meaning that there are no active instances in the TS, "ACTIVE", meaning that maybe one of the TS instances or the SUT is still active, "TICK", meaning time progression, and "RESTART", meaning reactivating the system in the new time slice.

The time manager initiates idleness detection by sending the 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 the idleness token with the counter equal to zero, meaning there are no messages 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 global idleness is detected, the time manager changes the flag of the token to "TICK" and sends the token along the ring. After the time manager has received again the token with the flag "TICK", it triggers the start of a new time slice by sending the token with flag "RESTART". Upon receiving the token with flag "RESTART", the time manager restarts idleness detection in the new time slice.

Idleness handler for $\mathbf{TS}_i$ The idleness handler decides on local idleness of the TS_i, propagates the idleness token along the ring and triggers time progression at the PA_i. The idleness handler for the SUT is a simplified version of the TS_i idleness handler and is explained at the end of this section. The

behavior of an idleness handler for TS_i is specified as a Promela process [19].

The syntax of Promela is similar to that of C, but the **if** and **do**-loop use Dijkstra's guarded command syntax [20]. That is,

do
:: GUARD1 ⟶ RESPONSE1
:: GUARD2 ⟶ RESPONSE2
od

means: if GUARD1 is true, execute RESPONSE1; if GUARD2 is true, execute RESPONSE2; if both are true choose non-deterministically and repeat until a **break** statement is executed. Moreover, the syntax for sending (receiving) a message MSG with parameters P, Q on channel CNAME is CNAME!MSG,P,Q (CNAME?MSG,P,Q). Receiving is often used in guards, where the message received can both be stored in a variable and compared against a given value. In channel declarations, the size of the buffer is specified. A buffer size of 0 specifies that communication is synchronous rather than asynchronous.

Fig. 3 contains the declarations of channels and variables plus the main event loop. The body of the event loop is split over Fig. 4 and 5, which contain the handling of local and global messages, respectively.

The idleness handler receives an idleness token via channel RINGIN and sends it further via channel RINGOUT (declared in Fig. 3, l. 2,3). Channels TE, PA, SA (l. 4-6) serve for communication with the PA_i, the SA_i and the TE_i respectively.

Messages exchanged by the TE_i, the SA_i, and the PA_i via the TRI interface are referred to as *internal* wrt. the TS_i. Messages exchanged by the TE_i via the TCI interface and messages exchanged by the SA_i with the SUT are referred to as *external* wrt. the TS_i. The idleness handler receives messages IDLE and ACTIVE from the TE_i, the SA_i and the PA_i and sends messages TICK and RESTART to the PA_i (declared in l. 1). IDLE messages are parametrized with the number of messages exchanged via TRI, TCI, and with the SUT as defined in Section 6.

The TS_i is locally idle if and only if the TE_i, the SA_i, and the PA_i are idle and there are no messages or timeouts pending between the TE_i, the SA_i, and the PA_i. The idleness handler maintains several local counters to collect information on the number of internal and external messages exchanged (l. 11-13). TRISENTTE and TRIENQDTE keep the number of messages sent and enqueued by the TE_i via the TRI interface. TRISENTSAPA and TRIENQD-SAPA provide analogous information for the SA_i and the PA_i. These four counters are necessary to detect the local idleness of 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. The last two counters contain

```
1  mtype={IDLE, ACTIVE, TICK, RESTART};
2  chan RingIn = [1] of {mtype, int, bool};
3  chan RingOut = [1] of {mtype, int, bool};
4  chan TE=[0] of {mtype, int, int, int, int};
5  chan PA=[0] of {mtype, int, int};
6  chan SA=[0] of {mtype, int, int, int, int};
7  active proctype TSIdlenessHandler(){
8      bool flagSA = true, flagTE = true;                          /* local flags */
9      bool idleSA = false, idlePA = false, idleTE = false;        /* idleness status */
10     bool buffer = false;                                        /* presence of idleness token */
11     int TRISentTE = 0, TRIEnqdTE = 0, TCITEcount =0; /* local counters of TE */
12     int TRISentSAPA = 0, TRIEnqdSAPA = 0;        /* local counters of SA and PA */
13     int SASUTcount = 0;                                         /* local counter of SA/SUT */
14     mtype tokenFlag;                                            /* global flag */
15     int tokenCount;                                             /* global counter */
16     int TRISent, TRIEnqd, TCISent, TCIEnqd, SUTSent, SUTEnqd;
17     do {
18         /* TE, PA and SA report idle or active, see Fig. 4 */...
19         /* detection local idleness of TS and propagation of idleness token, see Fig. 5 */...
20         }
21     od
22     }
```

Fig. 3. Idleness Handler for TS_i

information necessary to decide on the global idleness.

The information about external messages exchanged by the TS_i is propagated to the time manager by updating the token. To ensure that the same information is used *at most* once for updating, the idleness handler keeps two flags: one for TE_i and one for SA_i (flagTE, flagSA, l. 8). The flags indicate whether TCITEcount and SASUTcount contain information that has not been propagated yet. The PA_i is not involved in communication with the SUT and the rest of the TS, information on messages exchanged by the PA_i is only important to detect local idleness of the TS_i, thus there is no need for a local flag for the PA_i. The idleness handler keeps information on the status of the TE_i, SA_i and PA_i in variables idleTE, idleSA and idlePA respectively (l. 9).

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

In Fig. 4, if the idleness handler receives an IDLE message from the TE_i (l. 1), it sets the flag flagTE to *true*, changes idleTE to *true*, and modifies the local counters TRISentTE, TRIEnqdTE and TCITEcount (l. 2-5). Now the flag flagTE indicates that the information about the TS_i carried by the token is not up-to-date anymore. Similarly to the original Dijkstra-Safra algorithm, the number of messages sent by the TE_i via the TCI interface to the rest of the test system is added to the TCITEcount-counter and the number of messages received via the TCI interface and enqueued by the TE_i is subtracted from the counter.

```
1  :: TE? IDLE, TCISent, TCIEnqd, TRISent, TRIEnqd ⟶{      /* TE reports idle */
2       flagTE = true; idleTE = true;
3       TRISentTE  = TRISentTE + TRISent;
4       TRIEnqdTE  = TRIEnqdTE + TRIEnqd;
5       TCITEcount = TCITEcount + (TCISent−TCIEnqd);
6       }
7  :: PA?IDLE, TRISent, TRIEnqd ⟶{                        /* PA reports idle */
8       idlePA = true;
9       TRISentSAPA = TRISentSAPA + TRISent;
10      TRIEnqdSAPA =  TRIEnqdSAPA + TRIEnqd;
11      }
12 :: SA?IDLE, TRISent, TRIEnqd, SUTSent, SUTEnqd ⟶ {    /* SA reports idle */
13      flagSA = true; idleSA = true;
14      TRISentSAPA = TRISentSAPA + TRISent;
15      TRIEnqdSAPA = TRIEnqdSAPA + TRIEnqd;
16      SASUTcount  = SASUTcount + (SUTSent−SUTEnqd);
17      }
18 :: TE?ACTIVE,_,_,_,_ ⟶ idleTE = false;                 /* TE reports active */
19 :: PA?ACTIVE,_,_    ⟶ idlePA = false;                 /* PA reports active */
20 :: SA?ACTIVE,_,_,_,_ ⟶ idleSA = false;                /* SA reports active */
```

Fig. 4. PA_i, SA_i and TE_i report IDLE or ACTIVE

Receiving an IDLE message from the PA_i (l. 7) results in changing the status-variable idlePA to *true* and updating the local counters TRISentSAPA and TRIEnqdSAPA by the number of messages sent TRISent and the number of message enqueued TRIEnqd by the PA_i, resp. (l. 8-10)

Receiving an IDLE message from the SA_i (l. 12) results in changing the flag of SA_i to *true*, setting the status of SA_i to *true*, and updating the local counters TRISentSAPA, TRIEnqdSAPA, and SUTSAcount (l. 13-16). The number TRISent of messages sent by the SA_i is added to TRISentSAPA; the number TRIEnqd of messages enqueued by the SA_i is added to TRIEnqdSAPA. Similarly to the original algorithm, the number of messages exchanged with the SUT is increased by the number SUTSent of messages sent by the SA_i to the SUT, and decreased by the number SUTEnqd of messages from the SUT enqueued by the SA_i.

Even if the TE_i, the SA_i, and the PA_i are reported to be idle, they will not necessarily remain locally idle until the next time slice. An idle TE_i remains idle until it receives a message or a timeout. An idle PA_i can be reactivated by setting a timer to the current time. A message from the SUT or a message from the TE_i activates an idle SA_i. If activated, the entity sends message ACTIVE to the idleness handler. The idleness handler changes the corresponding status variable to *false* (see Fig. 4, l. 18-20).

In Figure 5, the first two lines allow TS_i to receive the token and store it.

Local idleness of the TS_i may be concluded (l. 4,5) if the status variables idleSA, idlePA and idleTE are *true* and there are no messages between TE_i, PA_i and SA_i. The latter holds if all messages sent by TE_i via TRI are already enqueued by SA_i and PA_i (i.e. TRISentTE is equal to TRIEnqdSAPA), and

```
1  /* receiving idleness token */
2      :: RingIn?tokenFlag,tokenCount ⟶ buffer = true;
3  /* detection local idleness TS, propagation of token */
4      ::  ( idlePA ∧ idleSA ∧ idleTE ∧ (TRIsentTE == TRIenqdSAPA) ∧
5            (TRIenqdTE == TRIsentSAPA)                    /* TS is locally idle */
6          ∧ buffer )⟶                                     /* and TS holds the token */
7          {if
8           :: (tokenFlag==IDLE ∨ tokenFlag==ACTIVE) ⟶
9              { if
10                :: (flagTE)  ⟶ {tokenFlag = ACTIVE;
11                tokenCount = tokenCount + TCITEcount;
12                TCITEcount = 0; flagTE = false; }
13             fi;
14             if
15                :: (flagSA) ⟶ {tokenFlag=ACTIVE;
16                tokenCount = tokenCount + SASUTcount;
17                SASUTcount = 0; flagSA = false; }
18             fi
19            }
20          :: (tokenFlag==TICK) ⟶ {                       /* time progression */
21                TRIsentTE = 0; TRIenqdTE = 0;
22                TRIsentSAPA = 0; TRIenqdSAPA = 0;
23                SASUTcount = 0; idlePA = false;
24                flagSA = true; flagTE = true;
25                PA!TICK,0,0 ;}
26          :: (tokenFlag== RESTART) ⟶ PA!RESTART,0,0;  /* start new time slice */
27          fi;
28          buffer = false;
29          RingOut!tokenFlag,tokenCount;                 /* propagate token */
30         }
```

Fig. 5. Detection local idleness, propagation and time progression

the messages sent by the SA_i and the PA_i via TRI are already enqueued by the TE_i (i.e. TRIenqdTE is equal to TRIsentSAPA).

Since the TE_i, the SA_i, and the PA_i report the number of the enqueued messages only when they become idle (see Section 6), conditions (1-3) in Fig. 6 imply local idleness of the TS_i.

If the local idleness conditions are satisfied and the idleness handler possesses the idleness token with flag "IDLE" or "ACTIVATE" (l. 8), 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 (l. 9-19) and sending it further along the ring (l. 28,29). If the TE has reported being idle at least once since the last visit of the token, i.e. the flagTE is *true* (l. 10), then the number of messages exchanged via TCI has possibly changed. Thus the handler updates the counter of the idleness token. The value kept by TCIcount is added to the counter of the idleness token (l. 11-12). Analogously, if the SA has reported being idle at least once since the last visit of the token, i.e. flagSA is *true* (l. 15), the number of messages exchanged with the SUT has possibly changed. Thus the idleness handler updates the token counter by adding the value kept in SASUTcount (l. 16,17). If at least one of the flags is *true*, the flag of the token changes to "ACTIVATE" (l. 10,15) meaning that one of TS instances or the SUT is still possibly active.

$$idle(\mathrm{SA}_i) \wedge idle(\mathrm{PA}_i) \wedge idle(\mathrm{TE}_i) \quad (1)$$

$$\mathrm{TRI}\textsc{SentTE}_i = \mathrm{TRI}\textsc{EnqdSAPA}_i \quad (2)$$

$$\mathrm{TI}\textsc{SentSAPA}_i = \mathrm{TRI}\textsc{EnqdTE}_i \quad (3)$$

Fig. 6. Local idleness of TS_i

If the idleness handler holds an idleness token with flag "TICK" (l. 20), it prepares to detect idleness in the next time slice by setting all the flags to *true*, setting IDLEPA to *false*, sending a TICK message to the PA_i (l. 21-25), and sending the token to the next handler along the ring (l. 28,29). The PA_i is activated by time progression, so it should report idleness at least once per time slice. Both TE_i and SA_i may, however, remain idle during a time slice, i.e. they do not necessarily report idleness in every time slice. Therefore, the status of TE_i and of SA_i remains idle until explicit activation.

If the idleness handler gets an idleness token with flag "RESTART" (l. 26), it sends message RESTART to the PA_i and propagates the token to the next idleness handler (l. 29).

Proposition 3 *An idleness handler for a TS_i detects local idleness of the TS_i iff conditions (1-3) in Fig. 6 are satisfied.*

Idleness Handler for SUT It does not matter whether the time manager is implemented by the test system or by the SUT as long as there is precisely one time manager per closed system. Moreover, the idleness handler of the SUT should support sending and receiving of an idleness token.

An SUT is idle iff it cannot progress further by performing computations or by exchanging messages with the test system. When doing black-box testing, one does not have control over the computations of an SUT and one cannot observe its internal FIFO queues. Therefore the following assumption is made on an SUT: The SUT implements an idleness handler, idleness detection, and time progression analogous to the one described for the TS. The idleness handler of the SUT propagates the idleness token iff the SUT is idle. When the SUT's handler propagates the token, it increments the token's counter by the number of messages sent by the SUT to the test system and decrements it by the number of messages 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.

If the SUT's handler gets the token with flag "TICK", the SUT forces time progression in the system by marking the timers as ready to expire in the next time slice. If the SUT's handler gets the token with flag "RESTART", the idleness handler triggers such ready timers. In both cases the token is

propagated to the next handler in the ring.

According to the Dijkstra-Safra termination detection algorithm [18], the solution proposed in this section detects global idleness if and only if all test system instances are idle and there are no messages pending between instances of the test system and the SUT. That means that the global idleness is detected iff conditions (2-4) in Fig. 2 are satisfied. According to Prop. 3, local idleness of a TS_i is detected if and only if conditions (1-3) in Fig. 6 are satisfied. This, together with local idleness of the SUT satisfies of conditions (1) and (5-6) in Fig. 2.

Proposition 4 *The solution for simulated time proposed in Section 7 detects global idleness iff conditions (1-6) in Fig. 2 are satisfied.*

8 Conclusion and Related Work

Blom et al. [8] proposed host-based testing with *simulated time* for non-distributed applications. There, simulated time is implemented at the level of TTCN-3 specifications. This paper provides a framework for host-based testing of *distributed* embedded systems with TTCN-3, where simulated time is implemented at the level of test adapters. Moreover, this framework allows the use of the same test suites for host-based testing with simulated time and for testing with real-time in the target environment. Simulated time also improves the controllability of testing and debugging.

The solution provided allows the implementation of simulated time for distributed testing with TTCN-3. Time simulation is implemented at the level of test system adapters and does not affect a TTCN-3 test suite. That means that the same TTCN-3 test suite can be executed both in real and in simulated time. The solution for testing with simulated time is based on an extension of a well-known distributed termination detection algorithm [18]. The usefulness of host-based testing with simulated time has been confirmed by two case studies: one from the railway and another one from the telecommunication domain.

Related Work Using simulation techniques is common for software testing and verification. Blom et al. [10] proposed simulated time for verifying reactive systems. Using simulated time for host-based testing is motivated by the case studies performed during the TT-medal project [1] and the paper of Latvakoski et al. [21] discussing time simulation methods that can be used to reduce external non-determinism when testing embedded software for communicating systems. These papers, however, provide no simulated-time solution for distributed testing.

A paper close to ours in spirit and technique is that of Alvarez and Christian [22] where the CESIUM testing environment is proposed for simulation-based testing of communication protocols for dependable embedded systems. Alvarez and Christian aim at *distributed* testing whereas CESIUM is a *centralized* simulation engine that executes on a distributed set of tasks in a *single* address space. Moreover, this paper focuses on *host-based testing* of logical correctness, while the approach by Alvarez and Christian [22] tries to compute some *performance predictions* on the behavior of embedded software. Dai et al. [23] introduce Timed TTCN-3, a TTCN-3 extension for performance testing. Timed TTCN-3 assumes that the test components have access to synchronized clocks but the synchronization itself is not addressed.

Distributed and parallel simulation techniques also make use of distributed termination detection algorithms. Although testing in a host environment often relies on simulation techniques, the goal of host-based testing is different from the goals of distributed and parallel simulation formulated by Fujimoto [24]. In simulation, a simulation model is executed to predict the system behavior and performance in real world. In host-based testing, a host environment makes use of simulations to execute embedded software and to *validate* its behavioral (time-dependent) features. For example, Mattern extends Dijkstra's distributed termination detect algorithm [25] to approximate global virtual time (GVT) [26]. This paper employs a time semantics different from one presented in this paper. There, each of the components has a local clock and may increase it at any time. Although this paper and the one of Mattern [26] are both devoted to detecting a certain global state of a distributed system, the purpose of the extensions and the conditions being detected by the extensions are different.

Acknowledgments The authors would like to thank Daan van der Meij (ProRail, The Netherlands) and Wan Fokkink (Free University of Amsterdam) who provided us with detailed information on Virtual Processor Interlockings. The authors also appreciate discussions with Antti Huima (Conformiq Company, Finland) who helped us to improve our solution. Finally, the authors are grateful to the reviewers of this paper for their detailed and useful comments and suggestions.

References

[1] TTMedal. Testing and Testing Methodologies for Advanced Languages. http://www.tt-medal.org.

[2] ETSI ES 201 873-1 V3.1.1 (2005-06). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language.

[3] ETSI ES 201 873-4 V3.1.1 (2005-06). MTS; TTCN-3; Part 4: TTCN-3 Operational Semantics.

[4] C. Willcock, T. Deiß, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3*. Wiley, 2005.

[5] ETSI ES 201 873-5 V1.1.1 (2005-06). MTS; TTCN-3; Part 5: TTCN-3 Runtime Interface (TRI).

[6] ETSI ES 201 873-6 V1.1.1 (2005-06). MTS; TTCN-3; Part 6: TTCN-3 Control Interface (TCI).

[7] S. Burton, A. Baresel, and I. Schieferdecker. Automated testing of automotive telematics systems using TTCN-3. In *Proceedings of 3rd Workshop on SYSTEM TESTING AND VALIDATION, Paris 2004*. Fraunhofer, 2004.

[8] 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 3997 of *LNCS*, pages 1–15. Springer, 2005.

[9] B. Dsarathy. Timing constraints of real-time systems: constructs for expressing them, methods for validating them. In *Proc. real-time systems symposium: Los Angeles, California*, pages 197–204. IEEE Computer Society, 1982.

[10] S. Blom, N. Ioustinova, and N. Sidorova. Timed verification with mCRL. In M. Broy and A. V. Zamulin, editors, *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2003.

[11] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In W. Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer, 1992.

[12] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proc. of the Real-Time: Theory in Practice, REX Workshop*, pages 526–548. Springer-Verlag, 1992.

[13] H. Holma and A. Toskala. *WCDMA for UMTS- Radio Access for Third Generation Mobile Communications*. John Wiley and Sons, 2004.

[14] H. Kaaranen, A. Ahtiainen, L. Laitinen, S. Naghian, and V. Niemi. *UMTS Networks: Architecture, Mobility and Services*. John Wiley and Sons, 2005.

[15] 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.

[16] I. Schieferdecker and T. Vassiliou-Gioles. Realizing Distributed TTCN-3 Test Systems with TCI. In D. Hogrefe and A. Wiles, editors, *TestCom*, volume 2644 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2003.

[17] J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *J. Syst. Softw.*, 43(3):207–221, 1998.

[18] E. W. Dijkstra. Shmuel Safra's version of termination detection. EWD998-0, Univ. Texas, Austin, 1987.

[19] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.

[20] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[21] J. Latvakoski and H. Honka. Time simulation methods for testing protocol software embedded in communicating systems. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *IWTCS*, volume 147 of *IFIP Conference Proceedings*, pages 379–394. Kluwer, 1999.

[22] G. A. Alvarez and F. Cristian. Simulation-based testing of communication protocols for dependable embedded systems. *J. Supercomput.*, 16(1-2):93–116, 2000.

[23] Z. R. Dai, J. Grabowski, and H. Neukirchen. Timed TTCN-3 - A Real-time Extension for TTCN-3. In I. Schieferdecker, H. König, and A. Wolisz, editors, *TestCom*, volume 210 of *IFIP Conference Proceedings*, pages 407–424. Kluwer, 2002.

[24] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley Series on Parallel and Distributed Computing. Wiley, 2000.

[25] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, June 1983.

[26] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.