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-44807 Bochum, Germany
[3] Nokia Research Center, Itämerenkatu 11-13, 00180 Helsinki, Finland
[4] CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
[5] Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, D-10589, Berlin, Germany
[6] Eindhoven Univ. of Techn., 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

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 a a simulated-time solution for *distributed* testing with TTCN-3.

Keywords: TTCN-3, distributed testing, simulated time.

1 Introduction

The Testing and Test Control Notation Version 3 (TTCN-3) is a language for specifying test suites and test control [17]. Its syntax and operational semantics are standardized by ETSI [4,5]. 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 a level independent of a particular implementation or a platform [6,7], 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 test embedded systems not only in telecommunication but also in automotive and railway domains [2,9].

* 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)" [15].

I. Virbitskaite and A. Voronkov (Eds.): PSI 2006, LNCS 4378, pp. 98–111, 2007.

Modern embedded systems consist of many timed components working in parallel, which complicates testing and debugging. Potential software errors can be too expensive 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 most 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, target operating system emulations, monitoring or instrumentation should not affect 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 without 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 as 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 implies 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 [2] we proposed host-based testing with *simulated time* for non-distributed applications. There we implemented simulated time on the level of TTCN-3 specifications. Here we provide a framework for host-based testing of *distributed* embedded systems with TTCN-3, where simulated time is implemented at the level of test adapters. 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.

The rest of the paper is organized as follows. Section 2 provides a brief survey on the general structure of a distributed TTCN-3 test system. In Section 3 we provide the time semantics for host-based testing with simulated time. In Section 5, we give an overview of two case studies where simulated time has been used two test two systems: one from telecommunication and one from

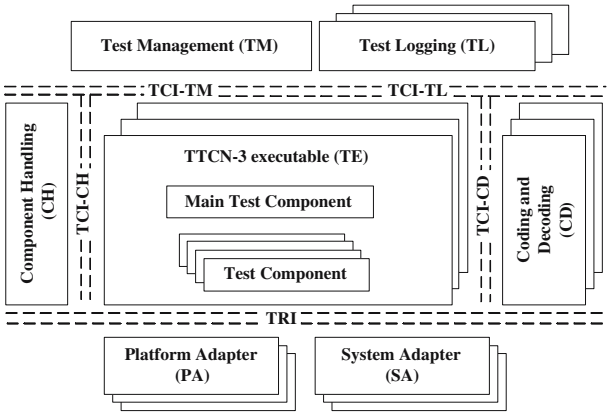

Fig. 1. General structure of a distributed TTCN-3 test system

transportation domain. In Sections 4, we present our testing framework. We conclude in Section 6 with discussing the obtained results.

2 TTCN-3 Test Systems

TTCN-3 is a language for the specification of test suites [8]. 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 [6] and illustrated in Fig. 1. The TTCN-3 executable (TE) entity actually executes or interprets a test suite. 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 zero, the timer expires and becomes deactivated. The expiration of a timer results in producing 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 SUT 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 run-time 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 [14]. Each of the TS_i has an instance of the TE entity TE_i equipped with an SA_i, a test logging (TL) entity TL_i, a 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) [7].

3 Simulated Time in TTCN-3

Here we first define the time semantics for testing with simulated time and then proceed with concretizing it for TTCN-3 test systems.

The first choice to be made is between dense and discrete time. It is normally assumed that real-time systems operate in "real", continuous time (though some physicists contest against the statement that the changes of a system state may occur at any real-numbered time point). However, a less expensive, discrete time solution is for many systems as good as dense time in the modelling sense, and better than the dense one when verification is concerned. Therefore we chose to work with discreet time.

We consider a class of systems where (i) the snapshots are taken with a speed that allows the system to see the important changes in the environment and (ii) external delays are significantly larger compared to the duration of normal computations 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.

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 transition has the least priority in the system and may take place only when the system is *idle*: there is no transition enabled except for time progress and communication with the environment. It means that some actions are urgent, as a process may block the progress of time and

enforce the execution of actions before some delay. This property is usually called *minimal delay* or *maximal progress* [13].

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* if and only if it cannot proceed by performing computations, receiving messages or consuming timeouts. We refer to the idleness of a single component as *local idleness*. We say that a *system* is *idle* if and only if all components of the system are idle and there are no messages or timeouts that still can be received during the current time slice. We call such messages and timeouts *pending*. We refer to the idleness of the whole system as *global idleness*.

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

If the system is globally idle, the *time progresses* by the action `tick` that decreases time left until expiration of active timers by one. If the delay left until the 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. Further, we refer to this time semantics as *simulated time*.

The time semantics of TTCN-3 has been intentionally left open to enable the use of TTCN-3 with different time semantics [5]. 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 [17]. Existing standard interfaces TCI and TRI provide excellent support for real-time testing but lack operations necessary for implementing simulated time [6,7].

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. Therefore we provide 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. Although providing such a solution inevitably means extending the TRI and TCI interfaces, we try to keep these extensions minimal.

According to the definition of global idleness, we need to detect situations when all components of the system are locally idle and there are no messages and no timeouts pending. We reformulate this definition in terms of necessary and sufficient conditions for detecting global idleness of the closed system. For the sake of simplicity, 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 an 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. Global idleness requires all the entities to be

$$[\forall i = 1..n\colon (TE_i = idle) \wedge (PA_i = idle) \wedge (SA_i = idle)] \wedge SUT = idle \quad (1)$$
$$\sum_{i=1..n} SA_i SentSUT = EnqdSUT \quad (2)$$
$$SentSUT = \sum_{i=1..n} EnqdSA_i \quad (3)$$
$$\sum_{i=1..n} TCISentTE_i = \sum_{i=1..n} TCIEnqdTE_i \quad (4)$$
$$\forall i = 1..n : TRISentTE_i = TRIEnqdSA_iPA_i \quad (5)$$
$$\forall i = 1..n : TRISentSA_iPA_i = TRIEnqdTE_i \quad (6)$$

Fig. 2. Global Idleness Condition

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 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 (*SASentSUT*) are enqueued by the SUT (*EnqdSUT*) and that 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 is the same as the number of messages enqueued by the corresponding TE_i (see conditions (5-6) in Fig. 2).

It is straightforward to show that the system is still active if one of the conditions in Fig. 2 is not satisfied. If all conditions in Fig. 2 are satisfied then all entities of the test system and the SUT are idle and there are no timeouts/messages that still can be delivered and activate them, thus the closed system is globally idle.

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

Thus to implement the simulated time for TTCN-3, we need to detect situations where conditions (1-6) in Fig. 2 are satisfied and enforce time progression.

4 Distributed Idleness Detection and Time Progression in TTCN-3

Detecting global idleness of a distributed system is similar to detecting its termination. Our algorithm for simulated time is an extension of the well-known distributed termination detection algorithm of Dijkstra-Safra [3].

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 or a timeout. 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. We require synchronous communication between a component and its idleness handler to guarantee the correctness of the extension of the algorithm.

To decide on global idleness we introduce a *time manager*. The time manager can be provided as a part of SUT or as a 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 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 no 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 next 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 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. That synchronizes all the TS's instances and the SUT on time progression. 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 the 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 class implements interface `Runnable` [1]. The idleness handler communicates with the other handlers via operation `IdlenessTokenSend()` that allows to receive an idleness token from

```
public synchronized void PAIdle(int TRISentPA, int TRIEnqdPA)
{TRISentSAPA+=TRISentPA; TRIEnqdSAPA+=TRIEnqdPA; idlePA=true; notify();}

public synchronized void SAIdle(int TRISentSA, int TRIEnqdSA,
int SASentSUT, int SUTEnqdSA)
{TRISentSAPA+=TRISentSA; TRIEnqdSAPA+=TRIEnqdSA;
SASUTcount+=SASentSUT-SUTEnqdSA; flagSA=true; idleSA=true; notify();}

public synchronized void TEIdle(int TCISentTE, int TCIEnqdTE,
int TRISentTE, int TRIEnqdTE)
{this.TRISentTE+=TRISentTE; this.TRIEnqdTE+=TRIEnqdTE;
 TCITEcount+=TCISentTE-TCIEnqdTE; 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 &(TRISentTE==TRIEnqdSAPA)&
         (TRIEnqdTE==TRISentSAPA)&(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)
         {TRISentTE=0; TRIEnqdTE=0; TRISentSAPA=0; TRIEnqdSAPA=0;
          SASUTcount=0; idlePA=false; flagSA=true; flagTE=true;pa.Tick();}
         if (msg.tag==IdlenessToken.RESTART){pa.Restart();}
         NextHandler.IdlenessTokenSend(msg);
         }
..... wait();}
}
```

Fig. 3. Idleness Handler for TS_i

one neighbor and propagate it further to the next one. For this purpose the idleness handler keeps the reference `NextHandler` to the next handler along the ring. 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 TS_i is locally idle iff the TE_i, the SA_i and the PA_i are idle and there are no messages/timeouts 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 several local counters. `TRISentTE` and `TRIEnqdTE` keep the number of messages sent and enqueued by the TE_i via the TRI interface. `TRISentSAPA` and `TRIEnqdSAPA` provide analogous information for the SA_i and the PA_i. These four counters are necessary to detect 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.

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 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 possibly 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 use TCI-operation `TEIdle( int TCISentTE, int TCIEnqdTE, int TRISentTe, int TRIEnqdTE)` called by a TE_i at the idleness handler when the 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 changing the value of `idleTE` to *true*, updating the local counters `TRISentTE`, `TRIEnqdTE` and `TCITEcount` and setting `flagTE` to *true*.

To detect local idleness of the PA_i, we use operation `PAIdle(int TRISentPA, int TRIEnqdPA)` 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 local counters `TRISentSAPA` and `TRIEnqdSAPA`.

To detect local idleness of an SA_i we use operation `SAIdle(int TRISentSA, int TRIEnqdSA, int SASentSUT, int SUTEnqdSA)` called by SA at the idleness handler when an active SA_i becomes idle. `TRISentSA` and `TRIEnqdSA` denote the numbers of internal messages sent and enqueued by the SA_i. Parameters `SASentSUT` and `SUTEnqdSA` keep 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 use operations `TEActivate()` called by the CH at the idleness handler when a remote message is being enqueued at the idle TE_i, `SAActicvate()` called by the SA_i at the idleness handler when an idle SA_i gets a message or a timeout, and `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 local idleness of the TS_i is implemented by the method `run()`. Local idleness of the TS_i is detected iff status variables `idleSA`, `idlePA` and `idleTE` are *true* and all internal messages sent via the TRI interface have been enqueued. This is expressed by local idleness condition at the first if-statement of 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 up-to-date information about the external messages exchanged

between instances of the TS and the external messages exchanged between the TS and the SUT by updating the idleness token and sending it further along the ring to the time manager.

If `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 `flagSA` is *true*, the number of messages exchanged with the SUT has change. 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 one of TS instances or the SUT may still be 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.

The solution proposed in this section strongly resembles the termination detection algorithm of Dijkstra-Safra when detection of messages pending on the level of TCI and communication with an SUT is concerned. The condition detected by an idleness handler in order to decide on local idleness of an instance of the TS, guarantees that all entities of the TS_i are idle and no messages/timeouts are pending on the level of TRI.

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 Case Studies

In this section we consider two case studies: one from the telecommunication domain and another one from the railway domain.

5.1 2G/3G Mobile Phone Application

Here we consider embedded software for a 2G/3G dual-mode mobile terminal that supports both WCDMA and GSM. GSM (Global System for Mobile Communication) [11] is a mobile network technology that has a global footprint in providing the second generation (2G) mobile services like voice, circuit-switched and packet-switched data and short message service (SMS). WCDMA (Wideband Code Division Multiple Access) [10] is one of the 3G mobile network technologies that meets the performance demands of mobile services like the Mobile Internet, including Web access, audio and video streaming, video and IP calls. WCDMA provides a cost efficient wireless technology for high data throughput.

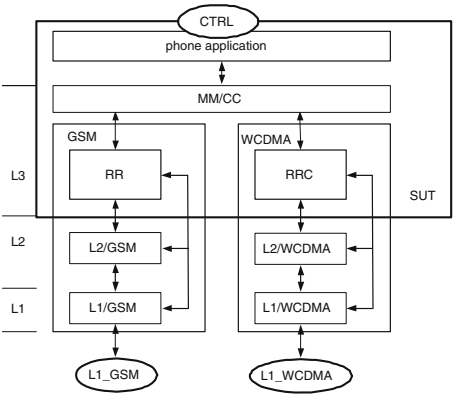

Fig. 4. Structure of embedded software for a 2G/3G dual-mode mobile terminal

Equipping the third generation mobile phones with both WCDMA and GSM technologies enables seamless, practically worldwide mobile service for their end-users [10,11].

The software for a dual-mode WCDMA/GSM phone implements an interworking mechanism for both technologies (see Fig. 4). In case a phone user first establishes a voice call using WCDMA technology and then moves outside of WCDMA coverage, the software is able to provide voice call service over GSM. Handovers from WCDMA to GSM and vice versa are handled in such a way that no noticeable disturbance happens.

In this case study, an implementation of the third layer have been tested. It combines the functionality of the third layers of WCDMA and GSM respectively, solves WCDMA to GSM (and vice versa) handover issues and a mobile terminal application (see Fig.4). In order to access and to control the implementation of the third layer, the actual system under test (SUT) also includes layers 1 and 2 and a mobile terminal application. 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.

We have tested the SUT on a workstation, so the air interface connecting the mobile terminal to the network is simulated by an Ethernet connection. The network is mimicked by a test system that interacts with the SUT through the simulated interfaces. There are three points available to control and observe the SUT: CTRL can be used to control the phone driver on the top of the SUT, L1_GSM and L1_WCDMA are used to exchange messages between the SUT and the test system.

To test the implementation of the third layer of a 2G/3G mobile terminal, we mimic the GSM/WCDMA air interfaces by Ethernet connections, emulate services of the target operating system and simulate the mobile terminal hardware. The test system simulates behavior of layers 1-3 of the mobile network. The OS services have been emulated. We used host based testing with simulated time to check behavioral time-dependant features of the SUT.

At the time of developing the test system for this case study the TCI had not been defined yet, hence proprietary APIs of the TTCN-3 tool had to be used to implement it. The operations at this API are however comparable to the operations in the TCI relevant for message exchange and indicating idleness. Despite these technical differences to the approach in this paper, it is possible to achieve that the implementation of simulated time is not visible on the level of TTCN-3 code.

Testing the SUT with the developed test system sufficiently increased the possibilities for debugging the SUT. Throughout the test execution the test system the SUT could be suspended and inspected with a debugger. These time intervals could be arbitrarily long, but due to the usage of simulated time no timer expired in such an interval and testing could be continued after such a long interval.

5.2 Railway Interlockings

Railway control systems consist of three layers: infrastructure, logistic, and interlocking. The infrastructure represents a railway yard that basically consists of a collection of linked railway tracks supplied with such features as signals, points, and level crossings. 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. Thus it is responsible for the safety of the railway system. If the interlocking considers a command as unsafe, the execution of the command is postponed until the command can be safely executed or discarded. Since the interlocking is the most safety-critical layer of the railway control system, we further concentrate on this layer.

Here we consider interlocking systems based on Vital Processor Interlocking (VPI) that is used nowadays in Australia, some Asian countries, Italy, the Netherlands, Spain and the USA [12]. 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. An example of a VPI specification can be found in [16]. In the TTMedal project [15], we develop an approach to testing VPI software with TTCN-3. This work is done in cooperation with engineers of ProRail who take care of capacity, reliability and safety on Dutch railways. They have formulated general safety requirements for VPIs. We use these requirements to develop a TTCN-3 test system for VPIs.

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 duration of the control cycle is fixed. 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 in 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 be easily reformulated in terms of input and output variables of the VPI program. Hence VPIs are *time-critical systems*.

We have tested VPI software without access to the target VPI hardware. To execute VPI program, we simulated the VPI hardware/software interfaces and the VPI program itself. The duration of the control cycle of VPI program is fixed. The VPI program sees only snapshots of the environment at the beginning of each control cycle, meaning the program observes the environment as a *discrete* system. Timing constraints in a VPI program are expressed by time delays that are much longer than the duration of the control cycle. That leads us to the conclusion that we may safely use simulated time to test VPI software.

Based on the concept of the simulated time we have developed a test system for executing the test cases. The experiments showed that our approach to host-based testing with simulated time allows to detect violations of safety requirements in interlocking software.

6 Conclusion

In this paper we proposed a simulated-time framework for host-based testing of distributed systems with TTCN-3. Simulated time has been successfully applied to testing and verification of systems where delays are significantly larger than the duration of normal events in the system (see e.g. [2]). Our framework contributes to the repeatability of test results when testing embedded software and also solves some time-related debugging problems typical for distributed embedded systems. It allows to use the same test suites for simulated time and for real time testing. We also provide two case studies where host-based testing with simulated time has been applied to two systems: one from telecommunication domain and one from transportation domain.

References

1. K. Arnold, J. Gosling, and D. Holmes. *Java(TM) Programming Language*. Java Series. Addison Wesley, 2005.
2. 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*, LNCS to appear, pages 10–25. Springer, 2005.

3. E. W. Dijkstra. Shmuel Safra's version of termination detection. EWD998-0, Univ. Texas, Austin, 1987.
4. 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.
5. ETSI ES 201 873-4 V3.1.1 (2005-06). MTS; TTCN-3; Part 4: TTCN-3 Operational Semantics.
6. ETSI ES 201 873-5 V1.1.1 (2005-06). MTS; TTCN-3; Part 5: TTCN-3 Runtime Interface (TRI).
7. ETSI ES 201 873-6 V1.1.1 (2005-06). MTS; TTCN-3; Part 6: TTCN-3 Control Interface (TCI).
8. 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.
9. S. Hendrata. Standardisiertes Testen mit TTCN-3: Erhöhung der Zuverlässigkeit von Software-Systemen im Fahrzeug. *Hanser Automotive: Electronics+Systems*, (9-10):64–65, 2004.
10. H. Holma and A. Toskala. *WCDMA for UMTS- Radio Access for Third Generation Mobile Communications.* John Wiley and Sons, 2004.
11. H. Kaaranen, A. Ahtiainen, L. Laitinen, S. Naghian, and V. Niemi. *UMTS Networks: Architecture, Mobility and Services.* John Wiley and Sons, 2005.
12. U. Marscheck. Elektronische Stellwerke-internationale *Ü*berblick. *SIGNAL+DRAHT*, 89, 1997.
13. 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.
14. 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.
15. TTMedal. Testing and Testing Methodologies for Advanced Languages. http://www.tt-medal.org.
16. F. J. van Dijk, W. J. Fokkink, G. P. Kolk, P. H. J. van de Ven, and S. F. M. van Vlijmen. Euris, a specification method for distributed interlockings. In W. Ehrenberger, editor, *Proc. 17th Conference on Computer Safety, Reliability and Security - SAFECOMP'98, Heidelberg*, volume 1516 of *Lecture Notes in Computer Science*, pages 296–305. Springer, 1998.
17. C. Willcock, T. Deiß, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3.* Wiley, 2005.