Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

SEN

Software Engineering

Software ENgineering

Simulated time for testing railway interlockings with TTCN-3

S.C.C. Blom, N. Ioustinova, J.C. van de Pol,
N. Sidorova

REPORT SEN-E0503 FEBRUARY 2005

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Simulated time for testing railway interlockings with TTCN-3

ABSTRACT
In this report, we first give an overview of software systems based on Vital Processor Interlocking (VPI). Interlockings guarantee safety of railway control systems, so testing these software systems is a key issue. We show why testing such systems with real time and scaled time is inefficient. We also provide a time semantics for simulated time that is more suitable for testing VPI's software. We provide a solution that allows simulated time for TTCN-3 test systems. TTCN-3 is a standard language for specifying and executing test suites. In the context of the TT-MEDAL project, TTCN-3 is applied to various domains, in particular to testing railway and automotive systems. TTCN-3 supports real-time and scaled-time testing but not simulated-time testing. The solution is based on a distributed termination detection algorithm that we extend to provide the main ingredients of simulated time: idleness detection and correct time progress. We implemented our solution as a TTCN-3 module and several Java classes that can be reused for testing other systems that have characteristics similar to those of VPIs.

2000 Mathematics Subject Classification: 68M15
1998 ACM Computing Classification System: I.6.5
Keywords and Phrases: testing, real time, scaled time, simulated time, interlockings, discrete time, TTCN-3
Note: This work is done within the project "Test and testing Methodologies for Advanced Languages (TT-Medal)"

Simulated Time for Testing Railway Interlockings with TTCN-3

Stefan Blom Natalia Ioustinova Jaco van de Pol
Department of Software Engineering, CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
E-mail: Stefan.Blom@cwi.nl, Natalia.Ioustinova@cwi.nl

Natalia Sidorova
Department of Mathematics and Computer Science
Eindhoven University of Technology Den Dolech 2, P.O. Box 513, 5612 MB Eindhoven, The Netherlands
E-mail: n.sidorova@tue.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

In this report, we fist give an overview of software systems based on Vital Processor Interlocking (VPI). Interlockings guarantee safety of railway control systems, so testing these software systems is a key issue. We show why testing such systems with real time and scaled time is inefficient. We also provide a time semantics for *simulated time* that is more suitable for testing VPI's software.

We provide a solution that allows simulated time for TTCN-3 test systems. TTCN-3 is a standard language for specifying and executing test suites. In the context of the TTMedal project [10], TTCN-3 is applied to various domains, in particular to testing railway and automotive systems. TTCN-3 supports real-time and scaled-time testing but not simulated-time testing.

The solution is based on a distributed termination detection algorithm that we extend to provide the main ingredients of simulated time: idleness detection and correct time progress. We implemented our solution as a TTCN-3 module and several Java classes that can be reused for testing other systems that have characteristics similar to those of VPIs.

Keywords: testing, real time, scaled time, simulated time, interlockings, discrete time, TTCN-3.

1. Introduction

Railway control systems are safety-critical and therefore we have to ensure that they are designed and implemented correctly. Interlocking is a layer of railway control systems that guarantees safety. It allow to execute commands given by a user only if they are safe; unsafe commands are rejected. Interlockings also react in dangerous situations that can lead to trains derailments and collisions. In this paper we propose a testing method for interlockings and indicate the characteristics of systems for which this method will be suitable as well.

The software part of the interlocking is a program that consists of a large number of guarded assignments. The program defines a control cycle of the system that starts with reading inputs, then proceeds by evaluating the guards and finally issues outputs. After being idle for some time, the interlocking repeats these activities. The total time of the active and the idle phases of the control cycle is a fixed time period. Although the environment of the system changes continuously, the system sees only snapshots of the environment made at the beginning of each control cycle. So the environment is discrete from the system's point of view. The system is *timed*, delays are used to guarantee safety. To keep the logic of the system simple and safe, the delays are chosen based on the worst case assumptions about the environment behavior. We try and choose a time semantics that is the most suitable and efficient to test this kind of systems.

Real time is usually considered to be the most adequate choice when testing timed systems. In real time, the system clock is driven by a physical clock. For safety reasons, the length of the active phase is much smaller then the length of the control cycle. Therefore, the total time spent by the system on being idle is much larger than the time spent on real computations. Hence, testing interlockings in real time, we would waste a large amount of time on idle phases.

Testing interlockings, we actually test a software system, so we have the control over timing of test executions. The most quick, naive solution would be to test the system using scaled time. In scaled time, the moment when a time factor is set is remembered as initial moment. Scaled time is calculated as initial time plus the product of a time factor and a difference between the physical current time and the initial moment, e.g. each physical second, the system clock progresses by 5 seconds.

The larger the factor is the faster we can execute tests. However, choosing the time factor is not as simple as it seems. The time factor must be small enough so that the longest control computation fits into the scaled cycle. Hence, we have to determine the largest possible time factor that still satisfies this condition. Determining the largest time factor is difficult, time-consuming and potentially error-prone. Any simple change in the system or in a test suite implies that the factor has to be determined again.

Even if we have found the time factor, it still would not be optimal for testing. The time spent on computations differs from cycle to cycle. If computations in one control cycle take ten times as much time as computations in the other ten cycles, the total time spent by the system idle will be much larger than the total computation time. Hence, testing with scaled time is not the best choice for this kind of systems.

In this paper, we propose a solution based on simulated time where the system clock is a logical clock. Simulated time is based on the assumption that the time spent by the system on computations is negligible compared to the duration of the external events. Therefore, the computations are considered to be instantaneous and time may progress only when the system is idle.

Simulated time is often considered to be unsuitable for testing timed systems. However, it is still as good as real time for testing interlockings. The reasons why simulated time is adequate for testing this kind of systems are the following: The length of the control cycle is fixed by the design of the system. The environmental changes are seen by the system as snapshots made at the begin of each control cycle. This provides natural discretization of the system behavior. For safety reasons, the maximal time spent within a control cycle on computations must be smaller than the minimal time within which the system must react on the changes in the environment. Therefore, we may safely use simulated time for testing this kind of systems. 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 case of varying computation times and allows adequate simulation of the environment in case the system can not be tested in field.

We have chosen TTCN-3 to implement our solution for testing interlockings. It is a language with syntax and operational semantics standardized by ETSI [6, 2, 3]. TTCN-3 has predefined standard interfaces [4, 5]. They allow to offer TTCN-3 solutions that do not depend on the technical details of a system under test. Therefore, applying TTCN-3 to domains other than telecommunication systems is potentially beneficial [1].

TTCN-3 was originally developed for real-time testing telecommunication systems so implementing simulated time for existing TTCN-3 interfaces is not straightforward. Here we provide a mechanism that first detects whether a test system and a system under test are idle and then synchronizes them on the time progression. A TTCN-3 test system and an SUT usually consists of several concurrent components, so we modify an distributed termination detection [1] to decide on idleness of all the components and to provide correct time progression. Our implementation consists of a TTCN-3 module and Java classes for simulated time. The TTCN-3 module supports simulated time within the TTCN-3 executable entity. The Java classes provide implementation of simulated time for platform and system

[1] This work is done within the project "Test and testing Methodologies for Advanced Languages (TT-Medal)" [10]

adapters. The solution is general and can be used for testing systems other than interlockings.

In Section 2 we give a survey on railway control systems and describe a control cycle typical for interlockings. In Section3 we define a time semantics for testing interlockings. Section 4 provides a survey on a general structure of a TTCN-3 test system [4]. We present our approach to implementing simulated time for TTCN-3 test systems in Section 5. The details of the implementation are given in Section 6. We conclude with Section 7 where we discuss outline the future work, discuss the restrictions under which our approach is working and propose possible ways to remove them.

2. Testing Railway Interlockings

Railway control systems consist of three layers: infrastructure, interlocking, logistic. 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. Figure 1 provides a schematic view of the road map of the railway yard at Hoorn–Kersenboogerd in the Netherlands. In this figure, the objects 52D, 62A-C, 66A-C, 69A-B, 70A-C, 73A-B, 74A-B denote tracks, the objects 60, 62, 64, 66, 68, 70, 72, 74 denote signals, the objects 69, 73 denote points, and object 35.0 denotes a level crossing.

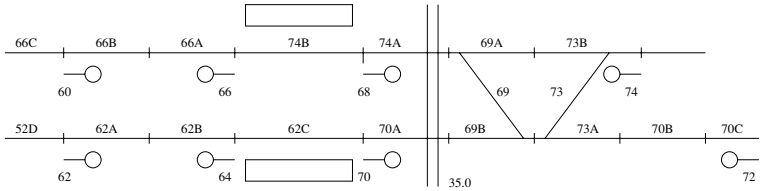

Figure 1: Railway yard

A railway track can be either occupied or unoccupied. Points are either in a normal position or in a reverse position. Signals can show red, yellow, flashing yellow or green color which can be labelled by a number imposing a speed limit. A level crossing can be open or closed. Infrastructure is responsible for executing commands like moving points into position, setting up signals, detecting presence of trains, detecting if all the lamps of the signals are functioning etc. The state of the infrastructure changes continuously.

Logistic is responsible for the user interface, by which human experts can give control instructions for the railway yard to guard trains. An interlocking guarantees that execution of these instructions do not cause train collisions or derailments. Thus, it is responsible for the safety of a railway control system. If an interlocking considers a command unsafe, the execution of the command is postponed until the command can be safely executed or discarded.

Here we consider interlocking systems based on Vital Processor Interlocking (VPI) that is used nowadays in USA, Netherlands, Spain, Italy and Australia. A VPI is usually implemented as a machine which executes hardware checks and a program for moving trains through a railway yard expressed as a large number of guarded assignments. They represent dependencies between various objects of a railway yard like points, signals, level crossings and delays on electrical devises. Assignments, guards and their combinations depend on a configuration of railway tracks, particular for each station. An example of a VPI specification can be found in [11].

A VPI is a safety critical system, so testing of VPIs is a key issue. Certainly, searching for flaws in VPIs should be automatized. In the TTMedal project [10], we define an approach to testing VPI's software with TTCN-3. This work is done in cooperation with engineers of ProRail that take care of capacity, reliability and safety on Dutch railways. They have formulated general safety requirements for VPI's. We are going to use these requirements to develop a TTCN-3 test system for VPI's.

A program implementing VPI's has several read-only input variables, auxiliary variables used for computations and several writable variables that correspond to 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 fases an active phase and an idle phase. The active phase starts with inputting new values for input variables. The values of input variables are determined by the infrastructure and the logistic. After the values are latched by the the program, it uses them to evaluate the guards. Depending on the values of the guards, the program computes new values for internal variables and finally decides on new outputs. The values of the output variables are transmitted to the infrastructure where they are used to manage signals, points, level crossings and trains. The rest of the control cycle the system stays idle. The duration of the control cycle is fixed. Moreover, delays are used to ensure the safety of the system. For example, when freed a train track must remain unoccupied for a certain time. Hence VPIs are *timed systems*. Further, we are going to propose a time semantics suitable for testing VPI's software.

3. Time Semantics for Testing Interlockings

Originally TTCN-3 was developed for real time testing telecommunication systems. We are going to use it for testing VPI's. Testing VPI's software in real time, we would spend time waiting for idle phase of each control cycle. Imagine about 1000 tests each of 5 minutes. Executing all of them would require about 80 hours. The control cycle of VPI's is repeated each second. If the active phase takes in average 0.5 seconds, we would basically lose 40 hours on idle phases.

We are testing VPI's software separately from hardware. That gives us the control over timing of test execution, so we could try to solve the problem by using scaled time. For testing with scaled time, we have to determine a time factor. Scaled time is calculated as initial time plus physical time, which passed since the initial moment, multiplied by the time factor. Testing VPI's software, we can scaled only idle phase of the control cycle. Time spent on active phases will still be determined by a hardware running the VPI's program, so active phases can not be scaled. Therefore, we have to chose a safe time factor so that the longest active phase still fits into the scaled control cycle.

Determining a safe time factor, we have to take into attention not only the longest active phase of the VPI's program but also the longest active phase of the test system. Hence, determining a time factor could be difficult, time consuming and potentially error-prone. Minor changes made in the program or in the test suite could lead to the change of duration of active phases. Even if we have determined a time factor, it is not optimal. The active phase of VPI's program together with the test system can take different time from one control cycle to another. So scaled time is hardly optimal for testing interlockings.

In this section we try to determine which time semantics is the most suitable for testing VPI's software.

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 [7]. Since the duration of a control cycle in VPIs is fixed and the program observes the changes in the environments only once per a control cycle, the choice for discrete time is obvious.

Often, it has been argued that models where any action takes some non-zero time allow more faithful descriptions. However, the VPI's are designed in such a way that an active phase is always fits into the control cycle. For safety reasons, the duration of a control cycle should be smaller than the time period within which the system must react on the environmental changes. Inputs are checked only once per a control cycle. In sequel, the actual duration of operations in an active phase is *negligible* compared to the duration of the control cycle and to the reaction time of the system. Therefore, we can treat an active phase as instantaneous. This assumption does not prevent from modelling actions that take some time. Whenever it is necessary, we can put an explicit time delay before an action or split it into start- and finish-events with time delay between them.

A lot of safety requirements to VPIs are timed. They describe dependencies between infrastructure

objects in a period of time, e.g. "if a level crossing has been open for ten seconds, then...". Delays are used to ensure safety in VPIs. Often delays are employed to wait for a certain situation in the environment to stabilize. In VPIs, delays take multiple control cycles. Together with negligible duration of an active phase, that leads us to conclusion that we may safely use logical clock instead of physical one, namely, we may use *simulated time*. The time-progress has the least priority in the system, and time may progress only if both VPI's software and a TTCN-3 test system are *idle*. This property is already known as *minimal delay* or *maximal progress* [8]. Further, we will refer to the time progress action as `tick` and to the period of time between two `tick`'s as a time slice.

In TTCN-3, timers are used to express time delays. An expiration of a timer is a natural way to model an interrupt from hardware or a trigger for a software event. Both interrupt and software event must be handled, and they must be handled exactly once, i.e. when taking an event guarded by a timer condition, the timer which triggered this event became deactivated (otherwise, the system could handle one event several times). Time progresses by decreasing the values of all active timers by one time unit. Next we give a short overview of TTCN-3 and consider how simulated time can be realized in the TTCN-3 framework.

4. TTCN-3 test systems

TTCN-3 is intended for specification of test suites [6]. The specifications can be generated automatically or developed manually. A specification of a test suite is given by a TTCN-3 *module* that can import 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 an order in which these test cases should be executed.

A test suite is executed by a TTCN-3 test system which general structure is defined in [4]. Fig. 2 illustrates this structure. The Test Management (TM) entity controls the order of execution of test cases and logs test events. Typically, this entity also implements interface with a user of the test system. The TTCN-3 executable (TE) entity actually executes or interprets a test suit. 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 test system. The SA entity also propagates messages and calls from the TE entity to the SUT and notifies the TE about messages and calls from the SUT. The platform adapter (PA) realizes platform dependant issues like external functions and time.

The TE entity executes TTCN-3 modules. A call of a test case can be seen as an invocation of an independent program. Starting a test case leads to creating a *configuration*. A configuration consists of several test components running in parallel and communicating with each other and with an SUT by message passing or by procedure calls. The first test component created at the starting point of test case execution is a main test component (MTC).

For communication purposes, a test component owns a set of ports. Each port has **in** and **out** direction. **in** directions are represented by infinite FIFO queues, **out** directions are directly linked to the communication partner, i.e. outgoing information is not buffered. TTCN-3 distinguishes between ports that are used for communication within the test system and the ports that serve for communication with the SUT.

A configuration is created dynamically by performing configuration operations CREATE, CONNECT, MAP, START and STOP that allow to create a test component, to map and connect its ports to ports of the other components, to start the component with a certain behavior and finally to stop it. The behavior of a test component is defined by a function given as a reference to the START operation. All components and ports are implicitly destroyed at the termination of each test case, so each test case shall completely create its required configuration of components and connections when its execution starts.

To specify time delays, TTCN-3 supports timer mechanism. Timers are local, namely each timer belongs to a certain test component. A test component can start the timer, stop the timer, verify whether the timer is still running. A test component can check the time elapsed since the timer was

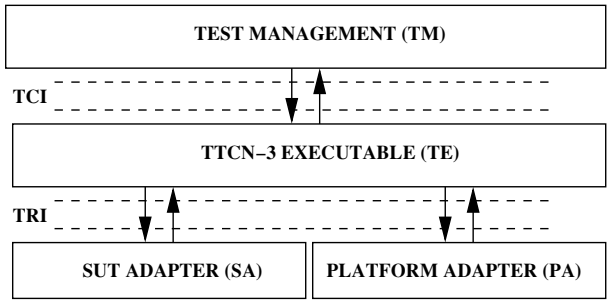

Figure 2: General structure of a TTCN-3 test system

started. Timers can be started with a given or default duration or delay. A test component can also to check expiration of the timer.

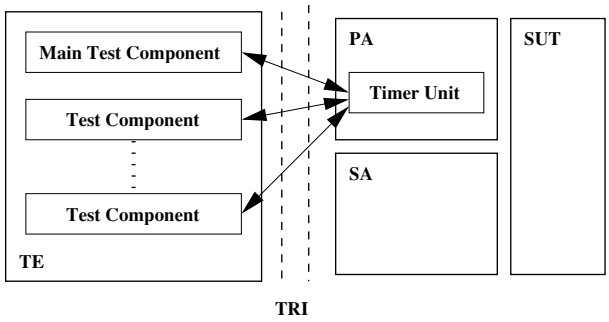

Figure 3: Real Time in a TTCN-3 Test System

An implementation of timers is platform dependent so the timer instances created in the TE and operations on them are implemented by the PA entity. The timers are distinguished by unique timer identifiers (TID). The runtime interface [4] (TRI) allows the TE entity to invoke external functions and the operations on timers implemented by the PA entity. A test component in the TE can start a timer for a certain period of time using triStartTimer interface. It can stop a timer using triStopTimer. The component can obtain the time elapsed since the start of the timer by the invocation of triReadTimer operation. It can also check a current status of the timer by invoking triTimerRunning.

After a timer, which has previously been started, expires, the platform adapter notifies the TE via triTimeout interface and the timeout is added into the timeout list of the corresponding test component. Stopping, restarting or expiring a timer that has already expired leads to removing its timeout message es from the timeouts list. Figure 3 illustrates time-related communication in a TTCN-3 test system. We refer to the part of the PA implementing timers as a Timer Unit. Further we consider, how simulated time can be realized in a TTCN-3 test system using existing interfaces.

5. Simulated time in TTCN-3

TTCN-3 is developed for real time testing, simulated time is not an option of a TTCN-3 test system. Our goal is to implement simulated time within existing structure of a TTCN-3 test system using only standard TRI interface and without introducing any changes into syntax and semantics of the TTCN-3 language. In principle, there are two problems, we have to solve to make simulated time possible in TTCN-3: (1) *idleness detection*, namely, checking whether all entities of a TTCN-3 test

system and an SUT are idle and (2) *correct* time progression. Correct time progression means that both the test system and the SUT have the same notion of time and receiving messages from future is not possible.

Receiving message from future is possible if any two entity running in parallel disagree on current simulated time. For example, one of the test components implemented by the TE entity sets a timer with a delay of 1 second. Then it is doing some work that will take about 0.1 second. The component is going to cancel the timer after the work is done. The PA entity dictating time receives the set request. Further, it has nothing to do, so it considers itself idle, progresses time and immediately issues the timeout. As result, the test component gets the timeout from the future. The test component is still in the current time slice while the platform adapter already jumped to the next one.

In a TTCN-3 test system, there is no shared memory, information is transmitted between the entities via ports. In case, message-based communication is used, it is difficult to predict when sent message is received by the recipient. There is no simple way for the entities of a TTCN-3 test system and an SUT to agree on simulated time. We have extended the distributed termination detection algorithm [1] to detect idleness of a TTCN-3 test system and to provide time progression.

A TTCN-3 test system is idle if the TE entity, the PA entity and the SUT are idle. The TE entity is idle if all the test components are idle, i.e. they can not progress further by receiving new messages or by performing computations, meaning, the timeout-lists are empty and the channels are empty as well. The PA is idle if there are no timers that have reached their expiration time but not expired yet. Otherwise, the PA can activate one of the test components by notifying the TE entity about a new timeout. The SA cares for the communication with the SUT, so we can use it to detect the idleness of the SUT.

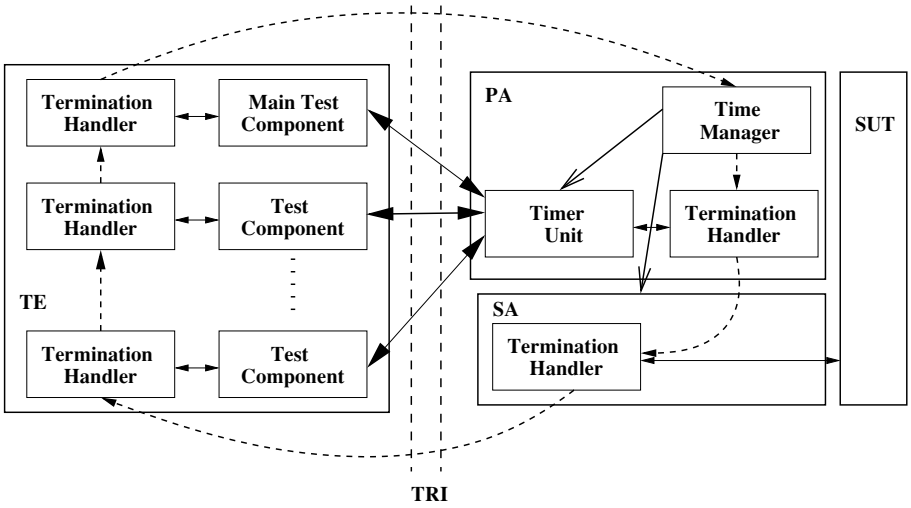

Figure 4: Simulated Time in a TTCN-3 Test System

Building the idleness detection into functions that define behaviors of the test components is error-prone and time-consuming. We provide simulated time as a stand alone solution that can be reused for any TTCN-3 test system if testing with simulated time is required. To implement simulated time, we refined the general structure of a TTCN-3 test system. The refinement illustrated in Fig. 4 concerns the TE entity, the PA and SA entities only. Since we do not want to build the solution into the code specifying test components, we introduce a termination handler for each entity that have to be checked for idleness. We also introduce one time manager that manages all termination handlers. The termination handlers and the time manager are connected into a ring as it is showed in Fig. 4. The ring is used for idleness detection only.

The implementation of simulated time consists of a TTCN-3 module and several Java classes. The TTCN-3 module defines the termination handlers and time progression for the test components. The TTCN-3 module is imported by a specification of a test suite in case the use of simulated time is more efficient than real or scaled time. The Java classes implement a time manager, a timer unit, a termination handler of the timer unit, and the termination handler of the SUT. The classes are the part of platform and system adapters respectively. When implementing our approach, we have used a series of tools for TTCN-3-based testing provided by the TestingTechn company [9].

6. Implementation of Simulated Time

The original termination detection algorithm considers a network of N nodes. Each node can be running or idle. Running nodes can send messages, idle nodes are waiting. Each message sent is received after some period of time. An idle node can become running only if it gets a message. The change from running to idle can happen spontaneously. The system is terminated in the state where all nodes are idle and channels are empty. The algorithm allows to one of the nodes, for example node 0, to detect whether this state has been reached.

An idle node can become running only by receiving a message, so we have to keep the track of all the messages. Each node has a counter that is increased when the node receives a message and decreases when the node sends a message. The channels of the system are empty when the sum of all the counters is zero. Besides a counter, every node also has a color (true or false). The nodes are initially false. A node becomes true if it sends a message.

The 0-node initiates a termination detection by sending a false token with the zero counter to the N-1 node. If the next node is running, it keeps the token, otherwise it adds its counter to the counter of the token and forwards the token to the next node. If the node is true it colors the propagated token into true, otherwise the color of the token stays unchanged. After forwarding the token, each node changes its color to false. The termination is detected if the 0-node gets back a white token such that the sum of the token counter and the counter of the 0-node is zero.

To apply the original algorithm for our case, we need to extend it. We are detecting idleness, not termination. After idleness is detected time may progress and the idleness detection should start in the new time slice. So we have to ensure time progression and restarting the detection in each time slice. Here we consider an implementation for the case with communication based on message passing. The same approach can be simplified to be used for the case with communication based on procedure calls.

The original algorithm works with two kinds of messages: basic messages exchanged by the nodes and termination messages. In a TTCN-3 test system, we deal with three kinds of messages: basic messages, idleness/time progression messages and timeouts. Timeouts are special kind of messages, that are not sent via usual ports but placed into a special timeout list. Timeouts can disappear from the list in result of stopping and resetting timers. The original algorithm works only in case all sent messages are received. We can not guarantee that for timeout messages in a TTCN-3 test system, so we handle timeouts separately from basic messages. For basic messages, we assume that the channels of TTCN-3 test system are reliable and that no dynamic reconfiguration takes place.

Termination handler. Here we define the general behavior of a *termination handler*. Fig. 5 illustrates a TTCN-3 function defining the behavior of a termination handlers. Note that the termination handler is defined for the case with asynchronous communication, so all the messages received from the test components are acknowledged. A termination handler communicates with its termination client and with its immediate neighbors along the ring. Every time a termination client receives message, sends a message or consumes a timeout, it informs the termination handler about it.

A termination handler has an idle status, a color and a counter. Initially, the idle-status of a termination handler is false. A termination client sending messages is obviously active. After sending a message, a termination client informs its termination handler by the “SEND”-message. By receiving “SEND”, the termination handler increases the counter, changes the color to true and sends an acknowledgment to the client.

An idle termination client becomes non-idle if it receives a message. The client notifies its termination handler by the "RECV"-message. By receiving "RECV", the termination handler decreases the counter, changes the color to true, changes the idle-status to false and sends an acknowledgment to the client.

An idle termination client becomes non-idle also if it consumes a timeout. When consuming a timeout, the termination client notifies its termination handler by the "ACTIVATE"-message. The "ACTIVATE" message triggers the termination handler to change the handler's color to true, the idle-status to false and to send an acknowledgment to the client.

A non-idle termination client can become idle spontaneously. In this case, it informs the termination handler by the "IDLE" message. If a termination handler receives "IDLE", it changes the idle-status to true and sends an acknowledgment to the client.

A termination token circulating in the ring consists of a counter and a tag. A tag can have one of the following values: IDLE, RUNNING, and TERM. The meaning of IDLE is that every node in the ring reported IDLE so far. RUNNING means that there may be someone in the ring who is RUNNING. TERM means that the idleness is detected in the TTCN-3 test system. Initially, the counter is 0 and the tag is IDLE.

If an idle termination handler with the "false" color gets a token, it adds its counter to the counter of the token and forwards the token to the neighbor on the ring without changing the tag. In case an active termination handler gets a token, it keeps the token until it becomes idle. If the color of the handler is still true, the handler changes the tag of the token to RUNNING, changes its color to false, updates the counter of the token and forwards the token.

In case, the termination handler gets the token with the TERM-tag, it reinitializes its counter, sets its idleness status and color to true. Now the handler is ready for the next time slice.

Transformation of the TTCN-3 code. The idleness detection works correctly only if the TTCN-3 code for test components follows certain specification pattern and the whole system is configured correctly. By correct configuration we mean that each test component should include ports for communication with its termination handler. Each test component should have an inout port for communication with its termination handler. Moreover, each test component (including main test component) should have a unique termination handler created and connected with it by the MTC. The termination handlers and a time manager should be connected into a ring. The SimulatedTime module implementing simulated time should be imported. No dynamic reconfiguration should be possible.

By the specification pattern, we mean that the code specifying behavior of test components should satisfy the following conditions.

- Every TTCN-3 blocking operation (`receive`, `timeout`, `alt`) must be preceded by sending "IDLE" to the port of a termination handler and by receiving an acknowledgment.
- All `receive` statements should be followed by sending "RECV" to the port of a termination handler and by receiving an acknowledgment.
- A timeout statement should be followed by sending "ACTIVATE" to the port of a termination handler, by receiving an acknowledgment and stopping the timer.
- Every `send` statement should be followed by sending "SEND" to the port of a termination handler and by receiving an acknowledgment.

The specification pattern can be implemented as an automatic transformation of TTCN-3 specifications. Further we consider a time manager and a timer unit components that are part of the platform adapter.

Time Manager. A time manager realizes idleness detection and time progression. It initiates a round of idleness detection by sending on the ring a termination token with the IDLE tag and the

zero counter. If the time manager receives the token back with the zero counter and the IDLE tag, it detects idleness. After detecting idleness, the time manager sends the token with the TERM tag to reinitialise the handlers for idleness detection in the next time slice. After the time manager gets back the token with the TERM tag, it can safely trigger time progression and start new round of the idleness detection. In case, the time manager receives the token back with the non-zero counter or a tag different from IDLE, it repeats the idleness detection in the current time slice.

Timer Unit. A timer unit implements the TRI operations on timers. A timer can be active or deactivated. The timeout messages are treated differently from usual messages in the system. The expiration of a timer causes issuing a timeout message at some point of the current time slice. An expired timer, which timeout- message is already issued but not consumed yet, keeps a TTCN-3 test system active. An expired timer, which timeout- message is not issued yet, keeps a TTCN-3 test system active as well. The timer unit is idle if all active timers are not going to expire in the current time slice and all timeout-messages for the expired timers are issued and consumed.

We use a termination handler to check idleness of the timer unit. Our solution for timer unit keeps active timers in three tables: an "active" table for active timers that are not going to expire in the current time slice, an "waiting" table for expired timer, which timeout-message is not issued yet, and a "timeouted" table for expired timers, which timeout-message is already issued but not consumed yet.

The timers kept in the "active" table are referred further as blocked timers, because they can not do anything until the next time slice or a restart. The members of the second table are called "waiting", because they are expired but their timeouts still have to be issued in this time slice. The timers in the third table are referred as "timeouted".

Starting a timer to value lager than zero leads to deleting the timer from all the tree tables and to adding the timer into the first table. In case there are no other "waiting" or "timeouted" timers, the timer unit reports to the termination handler "IDLE".

Staring a timer with the zero value leads to deleting the timer from all the tree tables and adding the timer into the "waiting" table. This timer will cause a timeout during the current time slice. Therefore, the timer unit sends "ACTIVATE" to its termination handler.

Stopping a timer leads to removing the timer from all the tree tables. In case there are no other "waiting" or "timeouted" timers, the timer unit reports to the termination handler "IDLE" as well.

Issuing a timeout moves an expired timer from the "waiting" table to "timeouted" table. If no timer expires in the current time slice, the timer unit reports itself "IDLE" to the termination handler.

Time progression issued by the time manager causes moving timers that are going to expire during the new time slice into the "waiting" table and notifying the termination handler by the "ACTIVATE" message.

Agreements on an SUT. A TTCN-3 test system is idle if the SUT is idle as well. We check the idleness of an SUT by using a termination handler in the system adapter. To make idleness detection possible, an SUT should provide interfaces notifying system adapter whether it is idle or running. The system adapter informs an SUT about time progression issued by a time manager, so an SUT should provide an interface time progression as well.

7. Conclusion and Future Works

In this paper, we provided a time semantics that is the most efficient for testing VPI software. Testing with simulated time, we do not need to spend time waiting for idle phases as in real time testing. Simulated time can be considered as scaled time with dynamic time factor that is defined automatically. Hence simulated time provides the most fair and effective scaling. We also defined the criteria that need to be satisfied to ensure safe application of simulated time.

We provided a "simulated time" solution for TTCN-3 test systems. The solution is based on an extension of distributed termination' detection algorithm [1]. We implemented our approach as a stand alone solution that can be used for any TTCN-3 test system if testing with simulated time is necessary.

The original algorithm that we use as a basis for idleness detection works correctly only if the channels in s system are reliable, i.e. no basic message gets lost. The TTCN-3 language provides operations that allow dynamic reconfiguration and clearing the contents of channels. Our current solution has two limitations: no distributed testing, no dynamic reconfiguration.

Dynamic reconfiguration means dynamically adding, removing test components and, consequently, mapping and unmapping ports. Dynamic reconfiguration is potentially dangerous because a reconfiguration can cause loosing messages. An easy solution removing this limitation would be allowing reconfiguration between idleness detection and time progression when the channels are empty anyway. In principle, clearing and resetting the channels may safely take place only if the system is idle. If this solution is not flexible enough, one could extend our approach to simulated time with a safe reconfiguration protocol. The protocol should guarantee that channels are empty when reconfiguration takes place.

Distributed testing, where a test system consists of multiple copies of a TTCN-3 test system considered here, is hardly possible with the current solution for simulated time because such a system would have multiple copies of a time manager instead of a mandatory single copy. This can be solved by disabling time managers in slave copies and by extending termination ring across all the copies. However it is not clear yet for which class of systems distributed testing with simulated time is reasonable.

As future work, we are going to use our solution for developing a test suite for VPI's. The current solution is safe but not optimal with respect to the number of acknowledgment messages sent by termination handlers. We plan to optimisme the current solution in order to minimize their number.

```
function TermHandler() runs on TermComponent {
   var TermMsg msgrecvd;
   var boolean TermMessagePresent := false;
   var boolean dirty := true;
   var boolean idle := false;
   var integer count := 0;
   while(true){
       alt {
         [] TermServer.receive(SEND) ->{
           count := count+1; dirty := true;
           TermServer.send(ACK);}
          [] TermServer.receive(RECV) ->{
           count := count-1; dirty := true;
                       idle := false;
           TermServer.send(ACK); }
          [] TermServer.receive(ACTIVATE) ->{
           dirty := true; idle := false;
           TermServer.send(ACK); }
          [] TermServer.receive(IDLE) ->{
           idle := true;
           TermServer.send(ACK); }
           [] TermIn.receive(termmsg) ->
                     value msgrecvd{
                     TermMessagePresent := true;}
            }
       if (idle and MessagePresent)
                       {if (msgrecvd.tag==IDLE_TAG
                             or msgrecvd.tag==RUNNING_TAG)
                                {if (dirty){
                     msgrecvd.tag:=RUNNING_TAG;
                     dirty:=false;
                           }
                 msgrecvd.count:=msgrecvd.count+count;
                     }
           if (msgrecvd.tag==TERM_TAG){
               log("time progression");
               count:=0; dirty:=true;
               idle:=true;}
           TermOut.send(msgrecvd);
           MessagePresent := false;
               }
        }
}
```

Figure 5: A TTCN-3 termination handler

References

1. 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.
2. ETSI ES 201 873-1 V2.2.1 (2003-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. ETSI Standard.
3. ETSI ES 201 873-4 V2.2.1 (2003-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics. ETSI Standard.
4. ETSI ES 201 873-5 V1.1.1 (2003-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI). ETSI Standard.
5. ETSI ES 201 873-6 V1.1.1 (2003-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI). ETSI Standard.
6. Jens Grabowski, Dieter Hogrefe, György Rthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An introduction into the testing and test control notation (TTCN-3). *Computer Networks, Volume 42, Issue 3*, pages 375–403, June 2003.
7. T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In Werner Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer, 1992.
8. 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.
9. <testing_tech> Testing Technologies. http://www.testingtech.de.
10. TTMedal. Testing and Testing Methodologies for Advanced Languages. http://www.tt-medal.org.
11. 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, 1989.