Simulated Time for Testing Railway Interlockings with TTCN-3[*]

Stefan Blom[1], Natalia Ioustinova[1], Jaco van de Pol[1,3], Axel Rennoch[2], and Natalia Sidorova[3]

[1] Centrum voor Wiskunde en Informatica, SEN2,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
`Stefan.Blom@cwi.nl, Natalia.Ioustinova@cwi.nl, Jaco.van.de.Pol@cwi.nl`
[2] Fraunhofer FOKUS, Kaiserin-Augusta-Alee 31, D-10589, Berlin, Germany
`axel.rennoch@fokus.fhg.de`
[3] Eindhoven University of Technology, Dept. of Math. and Computer Science,
P.O. Box 513, 5612 MB Eindhoven, The Netherlands
`n.sidorova@tue.nl`

Abstract. Railway control systems are timed and safety-critical. Testing these systems is a key issue. Prior to system testing, the software of a railway control system is tested separately from the hardware. Here we show that real time and scaled time semantics are inefficient for testing this software. We provide a time semantics with *simulated time* and show that this semantics is more suitable for testing of software of railway control systems.
TTCN-3 is a standardized language for specifying and executing test suites. It supports real time and scaled time but not simulated time. We provide a solution that allows simulated time testing with TTCN-3. Our solution is based on Dijkstra's distributed termination detection algorithm. The solution is implemented and can be reused for simulated time testing of other systems with similar characteristics.
Keywords: testing, real time, discrete time, scaled time, simulated time, interlockings, TTCN-3.

1 Introduction

Railway control systems are safety-critical and therefore we have to ensure that they are designed and implemented correctly. The interlocking is a layer of railway control systems that guarantees safety. It allows 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 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* that is

[*] This work is done within the project "TTMedal. Test and Testing Methodologies for Advanced Languages (TT-Medal)" [15]

repeated by the system. The control cycle consists of two phases: an active phase and an idle phase. The *active phase* starts with reading the inputs, then proceeds by evaluating the guards and by computing new output values, and finally issues outputs. After the active phase, the system becomes idle for the rest of the control cycle. The point of the control cycle where the idle phase starts is further referred to as an *idleness point.* The total time of the active and the idle phases of the control cycle is fixed. Although the environment of the system changes continuously, the system sees only snapshots of the environment made at the beginning of each control cycle. Thus 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. In this paper, 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. In the interlocking, the length of the active phase of the control cycle is much smaller than 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, with testing interlockings in real time, we waste a large amount of time on idle phases.

When testing interlockings, we actually test a software system, so we have the control over the timing of test executions. The most simple, naïve solution is to test the system using *scaled time.* Scaled time is calculated as initial time plus the product of a time factor and a difference between the current physical time and the initial moment. The larger the factor is the faster we can execute tests. Choosing the time factor is however not as simple as it seems. The time factor must be small enough to make the longest active phase fit into the scaled control 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 the 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 on being idle is still 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 discrete 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 progresses only when the system is idle.

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 beginning of each control cycle. This provides natural discretization of the system behavior. Interlockings are designed in such a way that the duration of the control cycle is much 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 cannot be tested in field.

We have chosen TTCN-3 to implement our solution for testing interlockings. TTCN-3 is a language with the syntax and the operational semantics standardized by ETSI [9, 3, 4]. TTCN-3 was originally developed for real-time testing of telecommunication systems. A TTCN-3 test executable has predefined standard interfaces [5, 6] that allows to offer TTCN-3 solutions that do not depend on the implementation details of a system under test (SUT). Therefore, applying TTCN-3 to domains other than telecommunication systems is potentially beneficial. Implementing simulated time for existing TTCN-3 interfaces is however not straightforward.

In simulated time, a test system and an SUT should agree on simulated time. To guarantee this, we provide a mechanism that detects an idleness point of an SUT together with a test system for each control cycle and then synchronizes them on time progression. A TTCN-3 test system and an SUT usually consist of several concurrent components, so we extend a distributed termination detection algorithm [2] to decide on idleness of all components and to synchronize them on 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 adapters. The solution is general and can be used to test systems other than interlockings.

The rest of the paper is organized as follows: Section 2 provides a brief survey on a general structure of a TTCN-3 test system [5]. In Section 3 we describe particularities of railway control systems and interlockings. In Section4 we define a time semantics for testing interlockings. In Section 5, we present the implementation of simulated time for TTCN-3 test systems. We conclude with Section 6 where we discuss the limitations of our solution, propose possible ways to resolve them and outline future work.

2 TTCN-3 test systems

TTCN-3 is intended for specification of (abstract) test suites [9]. The specifications can be generated automatically or developed manually. A specification of a test suite is a TTCN-3 *module* which possibly imports some other modules. Modules are the TTCN-3 building blocks which can be parsed and compiled autonomously. A module consists of two parts: a definition part and a control part. The first one specifies test cases. The second one defines the order in which these test cases should be executed.

A test suite is executed by a TTCN-3 test system whose general structure is defined in [5]. Fig. 1 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 the user interface 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 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. A configuration can be changed 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 the ports of 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 will completely create its required configuration of components and connections when its execution starts.

To specify time delays, TTCN-3 supports a timer mechanism. Timers are local, namely each timer belongs to a certain test component. For each test component, there exists a timeout list. A test component can start a timer for a certain duration by operation START, stop a timer by operation STOP, check

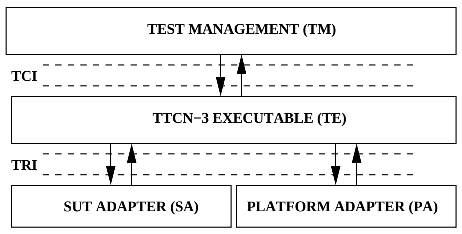

Fig. 1. General structure of a TTCN-3 test system

whether a timer is running by operation RUNNING, read the elapsed time of a running timer by operation READ and consume timeouts from the timeout list by operation TIMEOUT. A timer can be *active* or *inactive*. An active timer runs from 0 up to the specified duration. When the specified duration is reached, a timer expires, i.e. it adds a timeout to the timeout list of the test component and becomes inactive. Operation TIMEOUT allows a test component to consume a timeout message from its list.

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. Timers are distinguished by unique timer identifiers (TID). The runtime interface [5] (TRI) allows the TE entity to invoke external functions and the operations on timers implemented by the PA entity. For invocations of some TTCN-3 operations, there exists a direct correlation to invocations of TRI operations. TTCN-3 timer operations START, STOP, READ, RUNNING are realized by TRI operations TRISTARTTIMER, TRISTOPTIMER, TRIREADTIMER, TRITIMERRUNNING respectively. These operations are invoked by the TE entity and performed by the PA entity.

If the TE invokes TRISTARTTIMER, the PA starts the indicated timer with the specified duration. If the TE invokes TRISTOPTIMER, the PA stops the timer. If the TE calls TRIREADTIMER, the PA returns the time elapsed from the moment of starting the timer. In case the timer has not been started or already expired, the PA returns zero. If the TE calls TRITIMERRUNNING, the PA replies whether the timer is active or not.

The PA is responsible for expiring the timers. If an active timer reaches its specified duration, the PA deactivates the timer and notifies the TE about the expiration by calling TRI operation TRITIMEOUT. On the invocation of this operation, the TE entity adds the timeout to the timeout list of the corresponding test component. When starting, stopping or expiring a timer whose timeout is still in the timeout list, the TE removes the timeout message from the timeout list.

In the next section, we give a short overview of railway control systems and describe the control cycle typical for railway interlockings.

3 Testing 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 devises and ensure the safety of the railway system. An example of a VPI specification can be found in [1]. 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 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, e.g. "when freed, a train track must remain unoccupied for 120 seconds". VPIs control infrastructure objects. 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*. Further we are going to propose a time semantics suitable for testing VPI software.

4 Time Semantics for Testing Interlockings

Originally TTCN-3 was developed for the real time testing of telecommunication systems. We use it here for testing VPIs. When testing VPI software in real time, we waste time on idle phases of each control cycle. Imagine that we have to execute 1000 tests of 6 minutes each. Executing all of them will require thus 100 hours. Suppose that the control cycle of VPI is repeated each second and that the active phase takes in average 0.2 seconds. Then we will lose 80 hours on idle phases.

We are testing VPI software separately from hardware. That gives us the control over the 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 that has passed from the initial moment multiplied by the time factor. When testing VPI software, we can scale *only the idle phase* of the control cycle. Time spent on active phases will still be determined by a hardware running the VPI program, so active phases cannot be scaled. Therefore, we have to choose 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 account not only the longest active phase of the VPI program but also the longest active phase of the test system. Therefore, determining a time factor is difficult, time consuming and potentially error-prone. Minor changes made in the program or in the test suite can lead to a change of the duration of active phases. Even if we determined a time factor, this time factor is still not optimal. The duration of the active phase of the VPI program together with a test system can differ from one control cycle to another. That means that we still lose time on idle phases of control cycles with a short active phase. Scaled time is not optimal for testing interlockings.

In this section we try and determine which time semantics is the most suitable for testing VPI 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 testing and verification are concerned [11]. The duration of the control cycle of VPIs is fixed. The program sees only snapshots of the environment at the beginning of each control cycle, meaning the program observes the environment as a *discrete* system. Therefore, 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, VPI software is designed in such a way that an active phase always fits into the control cycle. The duration of a control cycle is smaller than the time period within which the system must react on the environmental changes. In sequel, the actual duration of the active phase is *negligible* compared to the duration of the control cycle and to the reaction time of the system. Therefore, we can treat the active phase as instantaneous.

Time constraints in a VPI program are expressed by time delays that are much longer than the duration of the control cycle. Together with the negligible duration of an active phase, that leads us to the conclusion that we may safely use a logical clock instead of a physical one, namely, we may use *simulated time.* In simulated time, the time progress has the least priority in a system, and time may progress only if the system is *idle.* This property is known as *minimal delay* or *maximal progress* [13]. We refer to the time progress action as `tick` and to the period of time between two `tick`s as a time slice. Further we define the notion of idleness more formally.

Here we consider closed systems consisting of multiple components. Timers are used to express time constraints of the system. We say that a *component* is *idle* iff it cannot proceed by performing computations or by receiving messages. As a consequence, all **in** ports of an idle component are empty and the timeout list of the component is empty as well. Otherwise, the component could still proceed by receiving a message or by consuming a timeout. Further we refer to the idleness of a single component as *local idleness*. We say that a *system* is *idle* iff all components of the system are idle and none of the active timers can expire in the current time slice. We refer to the idleness of the whole system as *global* idleness. If the system is globally idle, the time progresses by action `tick` that increases the elapsed time of active timers by one. If a timer has reached its specified duration, it expires within the current time slice. Timers ready to expire within the same time slice expire in an arbitrary order.

In the next section, we provide a TTCN-3 solution for testing with simulated time.

5 Simulated time in TTCN-3

TTCN-3 is developed for real time testing, simulated time is not included as an option of a TTCN-3 test system. Our goal is to implement simulated time within the existing structure of a TTCN-3 test system using only standard TRI interface and without introducing any changes into the syntax and the semantics of the TTCN-3 language. Here, we consider a closed system formed by an SUT and a TTCN-3 test system. In simulated time, we have to keep time of all system components synchronized. Therefore, we should provide a mechanism that allows to detect the idleness point of the system in each control cycle and to implement `tick`-steps.

An SUT is idle if it cannot progress further by performing internal computations or by receiving input messages from the test system, i.e. all its **in** ports have to be empty. When doing black-box testing, we do not have control over the computations of an SUT and we cannot observe its internal FIFO queues. Therefore, we make two assumption about an SUT: an SUT supports an interface notifying us of its status (active or idle); an SUT supports an interface for time progression. These are reasonable assumptions when interlockings are concerned. Interlockings have the control cycle with an explicit input/output structure, thus extending VPI software with such interfaces is straightforward.

A TTCN-3 test system is idle if all its entities are idle. The TE entity is idle if all the test components are idle, i.e. they cannot 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 it is not performing any external function and there are no timers that have reached their specified duration but not expired yet. The SA cares for the communication with the SUT, so we use it to decide on the idleness of the SUT.

5.1 Distributed termination detection algorithm of Dijkstra

To decide on the global idleness of the system, we employ the well-known distributed termination detection algorithm of Dijkstra [2]. The algorithm allows to decide on the 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 differentiates 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 communication is reliable, meaning, no message is lost.

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 basic message. An active component can always become idle. The system is terminated only if all components have the idle status and all channels are empty. The Dijkstra's algorithm allows one of the components, for example the 0-component, to detect whether termination has been reached.

We cannot decide on termination only by looking at the status of the components. The idle status of the components is necessary but not sufficient in this case. The status of a component changes from idle to active only by receiving a basic message, so we have to keep the track of all the messages in the network. 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 that is referred to as a *token*. The termination token consists of a global message counter and a global flag. The 0-component initiates a termination detection 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 none of the components has the active status, which is to be checked by passing the token along the ring.

If the next component has the active status, it keeps the token until the status of the component becomes idle. If the component has the 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 propagates the flag by changing the global flag to *true*, meaning, that maybe one of the system components is still active. 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 the 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*.

5.2 An extension of the distributed detection algorithm

We extend the Dijkstra's distributed termination detection algorithm to decide on global idleness of the system and to provide time progression. Trying to build an ad-hoc idleness detection into the functions that define the behavior of the test components is error-prone and time-consuming. Therefore, we provide simulated time as a stand alone solution that can be reused for any TTCN-3 test system with simulated time. To check local idleness of the system components, we introduce an *idleness handler* for each system component, i.e. for each test component, for the PA entity and for the SA entity. To decide on global idleness and to progress time, we introduce a time manager. The time manager and the idleness handlers are connected into a ring illustrated in Fig.2. (There the dashed lines represent the border of the original system and the channels within the system.) Although this solution brings a certain overhead, it is generic and independent of the details of a test suit.

The implementation of simulated time consists of a TTCN-3 module and several Java classes. The TTCN-3 module defines the idleness handlers and time progression for the test components. The module can be imported by a specification of a test suite. The Java classes implement a time manager, a timer unit, idleness handlers for the timer unit and for the SUT. The classes are part of the platform and system adapters respectively. When implementing our approach, we have used a series of tools for TTCN-3-based testing provided by the TestingTech company [14].

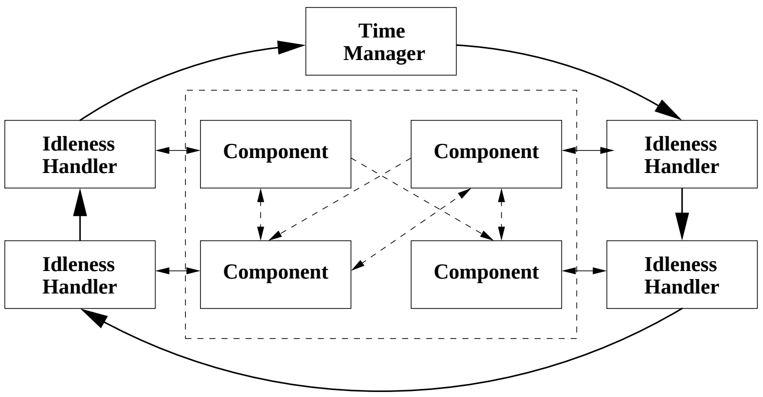

Fig. 2. A Closed System with Simulated Time

To implement simulated time, we have to detect global idleness, not termination. After idleness is detected time progresses and detecting global idleness starts in the next time slice again. So we have to ensure time progression and restarting the idleness detection in each time slice. For the sake of simplicity, we consider here only communication based on message passing. The same approach can be used in the case of communication based on procedure calls.

The original algorithm works with two kinds of messages: basic messages and token. In a TTCN-3 test system, we also have to deal with time progression and timeouts. Timeouts are a special kind of messages that are not sent via usual ports but placed into timeout lists. Timeouts can disappear from lists as a result of stopping or resetting timers. The original algorithm works only in case all sent messages are received. Not all timeouts are received by the components. Some of timeouts are lost by stopping and resetting timers, 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 in the system.

Time Manager A time manager initializes idleness detection, decides on global idleness and realizes time progression. The time manager initiates idleness detection by sending an idleness token. As in the original algorithm, the idleness token has a global message counter and a global flag. In order to support time progression, we extend the set of values of the global flag carried by the idleness token. In the original algorithm, the global flag was *true* or *false*. In the extended version, the global flag can be "IDLE_TAG" meaning that there are not active components in the system, "ACTIVE_TAG" meaning that maybe one of the system components is still active, and "TICK_TAG" meaning that time progresses by one time slice. The time manager initiates idleness detection by sending an idleness token with the counter equal to 0 and the flag equal to "IDLE_TAG" to the next idleness handler along the ring.

If the time manager receives the idleness token back with the zero message counter and the global flag with value "IDLE_TAG", it detects global idleness. Otherwise, it repeats idleness detection in the same time slice. If global idleness is detected, the time manager changes the global flag of the token to "TICK_TAG" and sends it along the ring to reinitialise the handlers for idleness detection in the next time slice. After the time manager gets back the token with the "TICK_TAG" global flag, it safely triggers time progression and then starts the idleness detection in the next time slice. Since all time issues are realized by the PA entity, we implement the timer manager as a part of the PA entity.

Idleness handler Here we define the general behavior of an *idleness handler*. A TTCN-3 function in Fig. 3 specifies the behavior of idleness handlers. In the Dijkstra's algorithm, termination detection was built into the functionality of components. We separate idleness detection from normal functionality of components by introducing idleness handlers. To guarantee the correctness of this extension of the algorithm, the communication between a component and its idleness handler is synchronized. An idleness handler acknowledges each message received from its component. An idleness handler and its component

```
function IdlenessHandler() runs on IdlenessComponent {
  var Token token; var boolean TokenPresent := false;
  var boolean flag := true; var boolean idle := false;
  var integer count := 0;
  while(true){
    alt {
      [] Comp.receive(SEND)
         { count := count+1; Comp.send(ACK);}
      [] Comp.receive(RECV)
         {count := count-1; flag := true; idle := false; Comp.send(ACK); }
      [] Comp.receive(ACTIVATE)
         {flag := true; idle := false; Comp.send(ACK); }
      [] Comp.receive(IDLE)
         {idle := true; Comp.send(ACK); }
      [] RingIn.receive(Token) -> value token
         {TokenPresent := true;}
       }
    if (idle and TokenPresent)
       {if (token.flag==IDLE_TAG or token.flag==ACTIVE_TAG)
         {if (flag){token.flag:=ACTIVE_TAG; flag:=false;}
          token.count:=token.count+count;}
        if (token.tag==TICK_TAG)
          {log("time progression");
           count:=0; flag:=true; idle:=true;}
       RingOut.send(token);
       TokenPresent := false;
       }}}
```

Fig. 3. A TTCN-3 idleness handler

communicate via port Comp. Ports RingIn and RingOut are used by a handler to receive a token from a previous handler and resp. to send a token to the next handler along the ring.

The local message counter, the status represented by the variable *idle* and the local flag of the component are now kept by its idleness handler. Initially, the idle status is *false*, meaning the component is potentially active, the local flag is *true*, meaning the token does not have up-to-date information about the component yet, and the local message counter is zero. The idleness handler keeps track of all the messages sent and received by the component, the component informs the idleness handler about receiving a basic message, sending a basic message or becoming idle by sending "RECV", "SEND", and "IDLE" messages respectively. In case of sending, receiving a message or becoming idle, the idleness handler follows the original distributed termination detection algorithm.

In a TTCN-3 test system, a component can become active also if it consumes a timeout. Therefore, the component client notifies its idleness handler about consuming a timeout by the "ACTIVATE" message. This message triggers the idleness handler to change the the local flag to *true* and the idle status to *false*.

Forwarding the idleness token with an "IDLE_TAG" global flag or "ACTIVE_TAG" global flag happens on the same conditions as forwarding the termination token with the *false* and *true* global flag respectively. In case the idleness handler gets the token with the "TICK_TAG"-flag, it reinitializes the local message counter counter, sets the local idleness status and the local flag 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 of test components follows certain specification pattern and the whole system is configured correctly. By correct configuration we mean that each test component has an port for communication with a unique idleness handler. Moreover the handlers together with the time manager are connected into a ring. The SimulatedTime module implementing simulated time is imported. No dynamic reconfiguration is 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`, `done`, etc.) is preceded by sending "IDLE" to its idleness handler;
– every `receive` statement is followed by sending "RECV" to the idleness handler;
– every `send` statement is followed by sending "SEND" to the idleness handler;
– a timeout statement should be followed by sending "ACTIVATE" to the idleness handler
– sending "IDLE", "RECV", "SEND" and "ACTIVATE" are followed by receiving an acknowledgment from the idleness handler;
– an acknowledgment for "ACTIVATE" is followed by stopping the timer to inform the PA that the timeout is consumed.

The specification pattern can be implemented as an automatic transformation of TTCN-3 specifications. Further we consider implementation of the timer unit in the PA.

Timer Unit. A timer unit implements the TRI operations on timers. Our solution for timer unit keeps active timers in three tables: a "blocked" table for active timers that are not going to expire in the current time slice, a "ready" table for timers ready to expire, and an "expired" table for expired timers, whose timeout message is not consumed yet.

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

Starting a timer with a value greater than zero leads to deleting the timer from all three tables and to adding the timer into the "blocked" table. Stopping a timer leads to removing the timer from all the tree tables. Issuing a timeout moves an expired timer from the "ready" table to the "expired" table. In case there are no other "ready" or "expired" timers, the timer unit reports to its idleness handler "IDLE".

On time progression issued by the time manager, the timer unit increases the elapsed time of all active timers by one, moves the timers that expire in the next time slice into the "ready" table and notifies its idleness handler by the "ACTIVATE" message.

6 Conclusion and Future Work

Using formal methods for the verification of railway control systems is an active area of research. Model checking [8] and theorem proving [7] have been successfully applied to untimed verification of interlockings. Several domain-specific languages [1, 10] have been developed to support automatic verification, validation and system testing.

In this paper, we provided a time semantics that is the most efficient for testing VPI software. When testing with simulated time, we do not waste time on idle phases as in real time testing. Simulated time can be considered as scaled time with a dynamic time factor that is defined automatically. Hence simulated time provides a fair and effective scaling.

We provided a "simulated time" solution for TTCN-3 test systems. The solution is based on an extension of the well-known distributed termination detection algorithm [2]. We implemented our approach as a stand alone solution that can be used for any TTCN-3 test system when testing with simulated time is necessary. This work together with other case studies within the TT-Medal project showed the necessity of simulated time for testing. Formulating proposals for changing TRI so that it allows a straightforward implementation of simulated time in a TTCN-3 test system is the subject of future work.

The Dijkstra's algorithm that we use as a basis for idleness detection works correctly only if the channels of the 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 implementation has two limitations: no distributed testing, no dynamic reconfiguration. Dynamic reconfiguration means dynamically adding and removing test components and, consequently, mapping and unmapping ports. Dynamic reconfiguration is potentially dangerous because a reconfiguration can lead to loosing messages. An easy solution is to forbid dynamic reconfiguration of non-idle components.

Distributed testing, where a test system consists of multiple instances of a TTCN-3 test system, is not possible with the current implementation of our solution 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. The current implementation of the solution is available on `www.cwi.nl/∼ustin/stime.html`.

Acknowledgments We 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 VPIs. We also appreciate discussions with Antti Huima (Conformiq Company, Finland) that helped us to improve our solution.

References

1. 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.
2. E. W. Dijkstra, W. H. J. Feijen, and A. J. M. v. Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, June 1983.
3. ETSI ES 201 873-1 V2.2.1 (2003-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. ETSI Standard.
4. ETSI ES 201 873-4 V2.2.1 (2003-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics. ETSI Standard.
5. ETSI ES 201 873-5 V1.1.1 (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.
6. 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.
7. W. J. Fokkink. Safety criteria for the vital processor interlocking at hoornkersenboogerd. In J. Allan, C. A. Brebbia, R. J. Hill, G. Sciutto, and S. Sone, editors, *Proc. 5th Conference on Computers in Railways - COMPRAIL'96, Volume I: Railway Systems and Management, Berlin*, pages 101–110. Computational Mechanics, 1996.
8. S. Gnesi, D. Latella, G. Lenzini, C. Abbaneo, A. M. Amendola, and P. Marmo. An automatic spin validation of a safety critical railway control system. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 119–124. IEEE Computer Society, 2000.
9. J. Grabowski, D. Hogrefe, G. Rthy, 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.
10. A. E. Haxthausen and J. Peleska. Automatic verification, validation and test for railway control systems based on domain-specific descriptions.
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. 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. <testing_tech> Testing Technologies. http://www.testingtech.de.
15. TTMedal. Testing and Testing Methodologies for Advanced Languages. http://www.tt-medal.org.