Real-Time Testing With TTCN-3

Diana Alina Serbanescu[1],Victoria Molovata[2], George Din[3],
Ina Schieferdecker[3,4], Ilja Radusch[1]

[1]DCAITI, Berlin, Germany
[2]Politechnica, Bucharest, Romania
[3]Fraunhofer FOKUS, Berlin, Germany
[4]TU Berlin, Berlin, Germany

Abstract. Reactive real-time software is used when safety is the issue and the margin for errors is narrow. Such software is used in automotive, avionics, air traffic control, space shuttle control, medical equipment, nuclear power stations, etc. As the timeliness of those systems is critical, it needs to be assured and tested. However, real-time properties require automated tests as manual tests are untimely and imprecise. This paper reviews Testing and Test Control Notation Version 3 (TTCN-3) as a means for real-time testing and proposes extensions to enable real-time test systems in TTCN-3. Small examples demonstrate the usage of the new constructs. Real-time operating systems are analyzed and reviewed, to enable the realization of real-time test systems based on TTCN-3.

1 Introduction

As the software industry undergoes a high and rapid development process, the software products become increasingly diverse and complex. Therefore, the demands regarding performance and reliability enlarges. An important part in the industry is the field of embedded real-time systems, which operate in environments with strict timing constraints. Embedded real-time systems find their applicability in a variety of domains where consideration of these timing requirements is critical important or even crucial (for example in automotive, avionics and robotic controllers). Dealing with an increased level of complexity of the software products, the possibility of errors occurring during the development process is an unavoidable fact. Software testing can be costly, but avoiding the testing may generate more expensive risks, especially in places where human lives are at stake. Together with the emergence of real-time embedded technologies, the demand for developing suitable means for testing those systems has increased, so as to systematically achieve the desired level of reliability. Compared to classical systems, embedded real-time applications require even more powerful testing techniques because they not only have to provide a certain functionality, but they have to provide that functionality in a predefined, well-determined amount of time, sometimes this amount of time being very short. Therefore, the test system (TS) should be well instrumented for measuring the timing attributes of the system under test (SUT) and for providing test stimuli in appropriate time, as required by the test procedures.

This paper discusses, in Section 2, the properties a test system should have in order to test real-time embedded applications and provides further, a solution for designing such test systems. This solution is based on the TTCN-3. This language for designing and specifying tests is an internationally standardized testing language, constantly developed and maintained by the European Telecommunications Standards Institute (ETSI). It gained popularity during the last years for its successful applications, especially in the telecommunication domain. Although TTCN-3 is an expressive and flexible language for writing tests, it does not fully provide notions for describing real-time aspects. Therefore, we took the liberty to enhance the language with new concepts necessary for testing real-time features. Such extensions to the language and their usage are presented in Section 3 of this study.

Our target is a *real-time test system* that tests real-time applications not only remotely, by way of some form of dedicated communication links, but also within the proximity of the tested system itself, as it is embedded on a board physically attached to the system. This allows to deploy test systems even where interconnectivity is hard to achieve, as for example, on online tests of electronic control units (ECUs) in automotive and avionics. For achieving that, the test system should be able to define abstract real-time test constraints for the tested system and for itself. After its abstract definition, the test system should be low level implemented on a specific embedded platform, which consists of both the real-time operating system and real-time hardware. As embedded systems dispose of minimal hardware resources, the selection of the right real-time operating system (RTOS), constitutes an important step. The criteria used for selecting an appropriate real-time operating system is presented in Section 4.

For the execution of the tests, one has to fill in the gap between the high level test specifications and the real-time executable code. In Section 5, a practical example is presented, consisting of a simple real-time application for automotive and a demonstrative test designed for the example. The test is implemented both using the enhanced TTCN-3 notation and the application programmer interface (API) provided by the operating system. It is interesting to present the two perspectives, one abstract and the other very specific, in order to emphasize the correlations between them. While the TTCN-3 test specification is more concise and simple to use, the system implementation is rather complex. Hence, the aim is to have this code generated, so that a future goal is to develop a compiler for automated transformation of tests into executable code, customized for the specific platform, as discussed in Section 6.

2 Real-time Testing

Software testing is the process of executing a program or system with the intent of assuring its quality [1] and of finding errors [2]. Testing involves any activity aimed at evaluating attributes or capabilities of programs or systems, and finally, determining if they meet all of the requirements.

Both white-box testing and black-box testing approaches are used. White-box testing denotes testing with the knowledge of the internal structure of the SUT. In contrast, black-box testing, which is also called functional testing, is purely based on the requirements of the SUT. The internal structure of the SUT is considered to be unknown. Black-box testing is mainly used in the higher levels of system design, that is, for integration, system or acceptance testing. Different to white-box testing, test cases for black-box testing are often written by using special test specification and test implementation languages. TTCN-3 is such a test language and its primary domain is black-box testing. There exist several types of black-box testing which focus on different kinds of requirements or on different test goals, such as conformance testing, interoperability testing, performance testing, system testing, acceptance testing and so forth. Conformance testing is functional black-box testing where the functionality of the SUT is tested. This paper considers mainly conformance testing of real-time embedded systems.

A simple conformance test system is that where the SUT is seen as a black-box, where only the inputs and outputs are visible. The outputs of the SUT are reactions of certain stimuli induced to it by the TS. These outputs are captured by the TS and matched against some predefined templates and if they coincide, the requirements are considered to be satisfactory, and the test is passed. Otherwise, the test fails.

In addition, real-time systems have to respect some special requirements, for which the matching of outputs is insufficient and the timings at which those outputs were received, are also relevant. This means that functionality must be accomplished within a certain time interval, its starting or ending should be marked by precisely defined points timely, but allowing some tolerance. An example of a test logic with real-time requirements for the TS and SUT is given in Figure 1. Regarding time, there are two critical sections in this test example: first, there is t_max_1, a time constraint for the SUT that indicates the interval in which the reaction to the first stimuli should be received by the TS; the second is t_max_2, which indicates the time that should elapse at the TS side, between the receiving of the first reactions from the SUT and the sending of the second stimuli to the SUT.

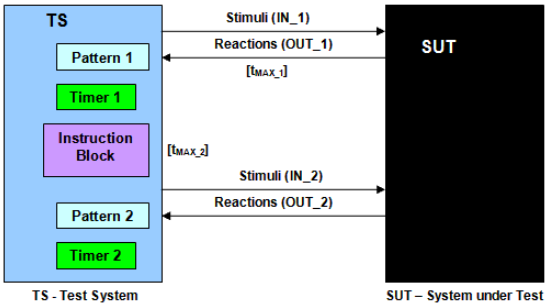

Fig. 1. Reactive real-time TS

3 Real-time Language Extensions for TTCN-3

Following the presentation in the previous section of the requirements that a real-time test system should follow, the most important demands for a real-time language are now introduced. After that, we try to fit the TTCN-3 language in this perspective, analyzing it as a real-time language for writing real-time test systems. Also, a discussion follows, concerning insufficiencies of TTCN-3 in defining real-time specifications and possible solutions for some of them are provided.

3.1 Requirements of a Real-time Language

It is well known that a real-time software must be guaranteed to meet its timing constraints [3]. Therefore, one of the most important requirement of a real-time language is that it must be designed so that its *programs can be guaranteed to meet their deadlines at compile-time* (that means they should be calculability analyzable). Real-time applications must also be very reliable and have a long life expectancy. Therefore, the language should provide *strong typing*, *structured constructs* and *be modular*. To reduce the costs allocated for maintenance, real-time programs should be easy to understand, be readable and simple to modify. This general maintainability requirement is greatly aided if the language is *readable*, *well-structured* and *not very complex*. Moreover, a real-time language should provide error handling mechanisms. Because many real-time programs involve multiprogramming, a real-time language should include *process definitions* and *process synchronization mechanisms*.

TTCN-3 is an expressive testing language providing all the necessary features for writing reliable and maintainable test systems for a wide range of application domains. It is a modular and well-structured language for testing not only conformance, but also other qualitative and quantitative features of the targeted systems. TTCN-3 allows multiprogramming and distribution due to the concept of test component which can be associated with a task.

Despite its advantages, TTCN-3 is not expressive enough for designing real-time tests. There are several problems when an atempt is made to design such a test. The first problem is *the precision* of time when it is recorded or checked by the TS or when it is associated with certain events. There is the semantic of timers that was not intended for suiting real-time properties, but conceived only for catching (typically longterm) timeouts. When using timers, the measurement of durations is influenced by the TTCN-3 snapshot semantics and by the order in which receive and timeout operations are ordered in the alt statement. TTCN-3 makes no assumptions about the duration for taking and evaluating a snapshot. Thus, exact times cannot be measured using ordinary timers. As they are now, timers shall be used for detecting or provoking the absence of signals and to take care that a test case eventually terminates if it is blocked for some reason, but not for specifying real-time requirements. Furthermore, the process of matching the received feedback from the SUT against the expected templates may take arbitrary long, though finite in time, as it depends on the structure

and size of the templates [4]. Having no restriction on the number of snapshots, on the structure of test data and on templates, introduces time *nondeterminism* and thus, the matching time cannot be properly estimated. Therefore, it would not be real-time. Nondeterministic *delays* are also introduced at the implementation level of different TTCN-3 primitives, for example, at the adaptor layer. For certain critical operations it is very important to have the possibility to impose *limits for the execution time*, upper and lower limits, and TTCN-3 language lacks in providing instruments for achieving measurements for these actual times of execution, or for imposing those time limits. Another problem is that timers are always local to a test component. They cannot be made global variables, and thus it is impossible to test real-time properties which are imposed on events that occur at different test components. This is also a matter of time *synchronization* and time *consistency* for different components, possibly running on different machines. When dealing with distributed systems it is important to have mechanisms for time synchronization.

Before tackling some of those problems of TTCN-3, by introducing new concepts to overcome the inconveniences, an overview of the existing work in the area is performed.

3.2 Related Work

There exist already several approaches that have been proposed in order to extend TTCN (Tree and Tabular Combined Notation, an earlier version of TTCN-3) and TTCN-3 for real-time and performance testing.

- *PerfTTCN* (*Performance TTCN*) [5] extends TTCN with concepts for performance testing, such as: *performance test scenarios* for the description of test configurations, *traffic models* for the description of discrete and continuous streams of data, *measurement points* as special observation points, *measurement declarations* for the definition of metrics to be observed at measurement points, *performance constraints* to describe the performance conditions that should be met, *performance verdicts* for the judgement of test results. The PerfTTCN concepts are introduced mainly on a syntactical level by means of new TTCN tables. Their semantics is described in an informal manner and realized by a prototype.
- *RT-TTCN* (*Real-Time TTCN*) [6] is an extension of TTCN in order to test *hard* real-time requirements. On the syntactical level, RT-TTCN supports the annotation of TTCN statements with two timestamps for earliest and latest execution times. On the semantical level, the TTCN snapshot semantics has been refined and, in addition, RT-TTCN has been mapped onto timed transition systems.
- *TimedTTCN-3* [7] is a real-time extension for TTCN-3, which covers most and *RT-TTCN* features while being more intuitive in usage. Moreover, the TimedTTCN-3 extensions are more unified than the other extensions by making full use of the expressiveness of TTCN-3. Therefore only a few changes to the language are needed. It introduces the following features: *A new test verdict* to judge real-time behavior; *Absolute time* as a means to

measure time and to calculate durations and this is the reason for using the operation **now** at the current local time retrieval; *Delays* to postpone the execution of statements as the new statement **resume** provides the ability to delay the execution of a test component; *Timed synchronization* for test components; TimedTTCN-3 supports the *timezones* concept, by which those test components can be identified and must be synchronized in time; *Online and offline evaluation* of real-time properties.

These new concepts are all useful and are utilized additionally for real-time test specifications. However, they are means of verifying the real-time properties of the SUT in particular and do not guarantee that the TS in itself is real-time, or that the TS is able to stimulate and respond timely to the SUT queries. In order to impose a real-time execution to the TS, a further introduction of control mechanisms at semantical and syntactical level of the language is described.

– *ContinuousTTCN-3* [8] introduces basic concepts and means for handling continuous real world data in digital environments. TTCN-3 is enhanced with concepts of stream-based ports, sampling, equation systems, and additional control flow structures to be able to express continuous behavior. In ContinuousTTCN-3 time is also very important, and the problem of imprecise timers is mentioned. The concept of global time is overtaken from TimedTTCN-3 and it is enhanced with the notion of sampling and sampling time.

In paper [4] some of the presented limitations of the existing treatment of time within TTCN-3 are illustrated as follows: the problems with snapshot semantics, with assignment of test verdict and with synchronization of distributed test configurations. The proposed approach is that of giving general guidelines for a more accurate measurements of the real-time using the actual capabilities of the language within its boundaries.

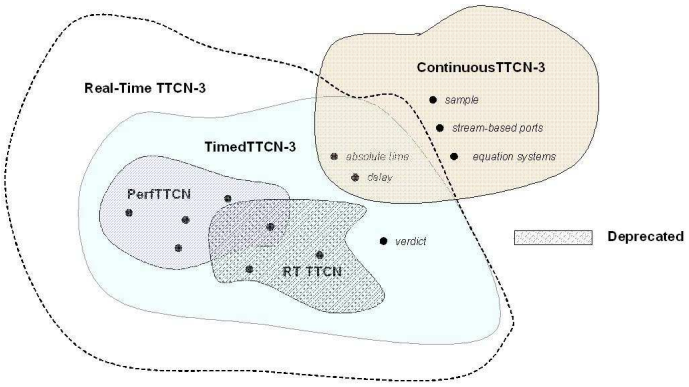

Fig. 2. TTCN-3 overview of proposed extensions

3.3 New Concepts for a Real-time TTCN-3

In the following, we discuss and introduce new instruments for dealing with real-time requirements in order to solve the problems presented before. Carefully selected new additions to the language were developed in a minimal set of concepts build upon the predefined ones. The set is not fully developed yet and it only covers some aspects. The concepts are syntactically and semantically presented and integrated into the language.

Clocks And Timers Precision timing and synchronization are key factors for real-time applications. Period clocks and timers are the basic instruments for achieving them. Therefore, we start the conceptual part with these elements, although they do not represent an original contribution of this paper. An equivalent definition for clocks is overtaken from TimedTTCN-3 [7] and timers are presented as they are already in the standard, relegating for the implementation the cumbersome work of making them real-time efficient.

A real-time clock is an incremental counter with fixed intervals, called the clock resolution. Generally, clocks are used for extracting absolute and local time values and for introducing intended delays into executions of test components. Absolute time is the time counted from a fixed point in time for the global system and the local time is the time counted on a local component. TTCN-3 does not provide the *clock*-concept, but thanks to the enhancements introduced by the TimedTTCN-3 [7], the functionality of a clock can be replaced. For extracting absolute and local time as a float value, representing the number of seconds from the established fixed point in time, primitive **now** is used. For local time retrieval the **now** operation should be applied to the **self** handle, for example, **self.now**. The primitive defined for introducing delays is **resume** which takes as argument an absolute time value, designating the moment for resuming its execution.

A timer is a complement of a clock. While a clock increments time, a timer decrements its value and generates a signal when that value reaches zero. Timers can be also used for counting relative time in a system where relative time is the time counted from a point in time, relative to the execution flow. In a general way, timers are used for controlling the sequence of event execution. There are situations when timers are used as standalone instructions, for example, when they are intended to delay processes or to indicate passage of time, and situations where they are used in dependence upon other statements, for example, associated to a receive event. In that case, the timer has the function to impose a time constraint on that event. Nevertheless, as discussed, it is not a reliable means for measuring the precise time. In order to keep consistence the old notion of timer is maintained as it is, but the implementation should overcome the limitations of snapshot semantics. We propose to implement each timer as a task that is programmed to trigger a signal after its expiration period of time.

In TTCN-3, events can be considered to be *dependent* or *independent* from each other. In Listing 1.1, *P.receive* and $T1.timeout$ are dependent events because the execution of one influences the other, timer $T1$ imposes a time constraint on the *P.receive* operation. In order to control this relation, tasks asso-

Listing 1.1. Dependent and independent receive-timeout events

```
...                                                                    1
port P;
timer T1, T2;                                                          3
alt {
    [] P.receive(response){ ... }                                      5
    [] T1.timeout{ ... }
}                                                                      7
...
T2.start(100);                                                         9
T2.timeout;
P.send(message);                                                       11
...
```

ciated to these events, in fact to the statements, synchronize with each other. The way in which the alt instruction is exited depends upon which of the two operations is executed first.

In the case of independent events, presented in Listing 1.1, timer $T2$ functions as a delay for the entire process and is not a time constraint to determine the arrival of another event in a timely manner, for example, the sending of a message to happen in a given time. In Listing 1.1, lines 14 and 15 can be replaced by the **resume(self.now + 100)** statement.

The Try-Catch Statement Time constraints are considered to be central elements to describe the real-time properties of a system. They are needed to control both the timing of the TS and the SUT, depending on the instructions to which the time constraints apply. When applied to the communication statements for receiving events, they indicate the responsiveness of the SUT. When applied to other simple or compound statements grouped together in blocks or standalone, those constraints indicate whether the TS itself respects certain time requirements.

To assure that a sequence of statements, or instructions, executes in a specified period of time, the instruction block can be protected by a **try-catch** construct. This wraps an instruction block by a time constraint together with an exception handling if the time constraint is not met. As an argument to the **try-catch** we have a float value which is preceded by the ***absolute*** or ***relative*** keyword. The float value represents a time value which can be absolute or relative. If dealing with an absolute value of the time it means that when reaching that point in time the execution of the enclosed instructions should have been terminated. Otherwise a real-time exception is generated. Whereas dealing with a relative time, the float value indicates the time units in which the block of statements should execute. If the time overpasses, the real-time exception is raised.

If the float value is negative or zero, a real-time exception is generated immediately. The exception can be handled immediately and the handling behavior is described in its following brackets. The instructions contained inside a **try-catch** statement can be simple or compound. They can be *ConfigurationStatements*, *TimerStatements*, *AltConstruct*, *RepeatStatements* or *CommunicationStatements* [9].

Listing 1.2. Generic try-catch statement bounding time execution for a set of statements

```
try (absolute|relative FloatValue) {
   Statement_1;
   Statement_2;
   ...
   Statement_N;
} catch (rtexception) {
   // exception handler
}
```

Real-time Exceptions Time guards for instruction blocks that do not contain communication with the SUT designate the time that should elapse on the TS side. If the time constraint is not respected, the TS is responsible for a test failure and an exception handling mechanism for protecting against TS errors should be activated. A new type of exception shall be introduced, **rtexception**, which will occur only when a deadline is missed by the TS. Also, special exceptions should be introduced for the delays of the SUT. The exception handling mechanism can define alternative actions depending on the test requirements for hard, firm, soft, or real real-time systems:

- the test will be stopped and a verdict will be set to a value corresponding to the error;
- the test can continue if the missed deadlines were not critical for the system;
- the test can continue by repeating the statements for a specified number of times.

Special Instructions The statements that require special attention are sending/receiving to/from a communication port. They are special because they enable the communication and interaction with the SUT. In many cases, it is necessary that a **send** operation must be executed in a timely manner. Another case is that sending to a port should be done every x time units where a time unit can vary from microseconds to hours, days or even more, depending on the test requirements. The construction in Listing 1.3 is very similar to the previous diagrams. However, a dedicated time constraint for a **send** statement is given. When encountering this construction, a separate thread is created for sending the message to the SUT. The first parameter represents the maximum duration of the **send** operation. The second parameter represents the interval between two consecutive send operations. The third parameter of the **try-catch** construction should be a integer value, indicating the number of times the message should be send to the SUT.

If the first parameter is negative or zero, an exception will be generated. If the second parameter is negative or zero or if the third parameter is zero, the sending of the package will not be repeated.

In this example, real-time exceptions could be obtained in three situations: (1) if the send instruction takes longer to execute than the time indicated by the first parameter (2) or if the send operation cannot be scheduled in time due to the overloading of the system (3) or an inappropriate scheduling politic. If

Listing 1.3. Time constraints imposed on a **send** operation

```
try (relative 0.1, 0.2, 3) {
   P.send(message);
} catch (rtexception rte) {
   // exception handling behavior
}
```

Listing 1.4. Time constraints imposed on a **receive** event

```
alt {
   try(relative 0.001) [] P.receive(message1) {
       Statement_1;
       Statement_2;
       ...
       Statement_N;
   } catch (rtexception rte){
       // exception handling behavior
   }
   try(relative 0.01)  [] P.receive(message2) {
       Statement_1;
       Statement_2;
       ...
       Statement_N;
   } catch (rtexception rte){
       // exception handling behavior
   }
}
```

the first parameter is preceded by absolute keyword, indicating absolute time, it will be automatically increased with the interval value at every cycle. The **try-catch** statement can also be used in the previous indicated manner with just one parameter, and then, the instruction will be executed only once.

In case of a **receive** statement, the constraint is put as part of the alternative statement and refers to the timed response of the SUT plus the time for matching. When a new message arrives, it will be matched conforming to the classical matching mechanism, and if it enters a branch, it verifies also if the timing requirement was respected or not. Depending on this it will execute further the block of statements contained inside the branch or the exception handling behavior.

The proposed **try-catch** statement is intended to be a powerful construct and therefore, the relation between this statement and other statements of TTCN-3 should be carefully analyzed. Other important example will be the combination with a **default** behavior. The **defaults** will be treated as a normal branch for an alternative. It is supposed that the **try-catch** statement guards the behavior defined for the associated **default**. All the situations will be approached when the operational semantic for all the introduced new concepts will be defined.

Code Spanning Limitations We have introduced the concepts *time guards* and *real-time exceptions* in order to enable TTCN-3 to stipulate real-time test specifications. These test specifications can be used to check real-time properties of the SUT and to make the test system itself have a time-deterministic behavior. Nevertheless, there are elements in TTCN-3 which increase the code complexity and affect time determinism. Those elements are, for example, the **alt** statement which can have a large number of alternative branches, or a large number of nest-

ing with other **alt** statements inside, **altstep** invocations, and similar. The type structure and nesting of templates are an additional load factor for the test system and in particular critical during the matching operation. There are other statements such as **for** loops or function invocations which enable an infinite number of executions. Therefore, in addition to the new concepts, we have to introduce a means to limit those aspects of a test in order to increase the chance of the TS to conform with its real-time requirements. For example, the test designer will be able to attach to an **alt** statement an upper limit for the number of branches and for the number of snapshots being taken into consideration (see Listing 1.5). Those limits are represented as integer numbers attached to the statements. Therefore, at compile time, a static prediction for the timings of the system can be obtained. At runtime, those timings will be also influenced by the load and other factors of the current state of the test system. This mechanism is extremely useful for the specialist to rapidly regulate the timing behavior of the test system without making great modifications into the code. Those limitations could be regarded as annotations introduced to bring in additional information into the system. This information could be used by the compiler for optimization purposes. Code spanning limitations could be gracefully manipulated by the test designer in order to deliberately separate important aspects of the functional behavior from the ones of real-time behavior. An alternative solution, using the existing resources of the language, would be to define a global boolean variable as for example **with_rt**. The variable can be used in an **if** statement with a **true** value to guard the behavior relevant for real-time evaluation and with a **false** value to guard the behavior important for functional evaluation, but which can be dropped when real-time tests are performed. Nevertheless, the solution proposed with the new constructs is more elegant and increases the flexibility and configurability of the tests. Based on the indication incrusted into the code, a great challenge would be the compiler itself, which should be designed in such a manner that it would be able to generate a fully optimized code, suitable for embedded systems.

In Listing 1.5 the first parameter of the **alt** statement indicates the number of the branches that should be taken into consideration and the second parameter indicates the number of snapshots that should be taken into account. The branches are considered in top down oder. The parameters are natural numbers.

4 Real-time Platforms

The market of operating systems (OS) is continuously developing due to multiples and more sophisticated requirements. One of the key necessities is to support embedded real-time applications in which the OS must guarantee the timeliness and the correctness of the code processing. Many OS claim to be real-time operating systems (RTOS), but often only by reviewing the OS specifications, or arriving detailed information, can one truly identify those operating systems that enable real-time applications.

Listing 1.5. Limitations

```
// only the first 10 branches and
// only the first 2 snapshots
// are taken into consideration

alt (10, 2){
            [] Branch1{
                  for(var integer i:=1; i<3; i++){
                      ...
                  }
            }
            [] Branch2{...}
             ...
            [] Branch10{...}
            [] Branch11{...}
            [] Branch12{...}
}
```

The process of selecting the right RTOS is important and, at the same time, critical. It involves knowing all the specifications of different real-time operating systems, in an abundant market of available real-time operating systems, from micro kernels to commercial ones. The design space available to an RTOS is very broad. Selecting the RTOS based on specific features is a multidimensional search problem where each dimension corresponds to a RTOS characteristic. This requires an exhaustive research quest, tremendous computing resources and valuable time.

A wide variety of real-time operating systems are available to suit most projects and pocketbooks [10], [11]. Our search revealed sixteen real-time operating systems that deserve worth further investigation. These were merely the ones that could be applied to construct a real-time test system. We were specially interested in currently maintained open source projects. This consideration left four RTOS candidates ([12], [13], [14], [15]) to be evaluated in detail and ranked conforming to our specific requirements.

Based on specific requirements from the automotive domain and on obvious need for performance, reliability and cost-effectiveness common to every real-time project, we have divided the selection criteria in two parts. First, one envelopes general points of view such as supported languages, portability, latest update, commercial status, available API and information about development and support (Table 1). Secondly, it includes more specific features of real-time operating systems such as scheduling algorithms, type of RT (soft of hard), priority levels, kernel ROM size, kernel RAM size, multi-process support, interrupt latency, task switching time, type of interprocess communication (IPC) mechanism, memory management, task management and so forth.

Based on our specific requirements, we selected FreeRTOS as the most suitable RTOS. FreeRTOS is a very small, simple and concise operating system, making it suitable for small applications on small platforms. Since the majority of code is written in C language, it is highly portable and has been ported to many platforms. A strong advantage of FreeRTOS is that the code includes a demo project for each supported platform, demonstrating how to use the code on that specific platform. Unfortunately, this feature was not found at the other systems, where installation, configuration and development required more ef-

Table 1. A General RTOS Selection

	eCos	FreeRTOS	RTAI	RTEMS
Languages Support	Assembly, C, C++, Ada95	C, Assembly	C	C, C++, Ada95
Target CPUs Support	x86, PowerPC, ARM, MIPS, Altera NIOS II, Calmrisc16/32, Freescale 68k ColdFire, Fujitsu FR-V, Hitachi H8, Hitachi SuperH, Matsushita AM3x, NEC V850, SPARC	ARM architecture (ARM7, Cortex-M3) , AVR, AVR32, HCS12, MicroBlaze, MSP430, PIC microcontroller (PIC18, PIC24, dsPIC), Renesas H8/S, x86, 8052	X86 (with and without FPU and TSC), PowerPC, ARM (StrongARM; ARM7: clps711x-family, Cirrus Logic EP7xxx, CS89712, PXA25x), MIPS	ARM, Blackfin, ColdFire, Texas Instruments C3x/C4x DSPs, H8/300, x86, 68K, MIPS, Nios II, PowerPC, SuperH, SPARC
Development Status	eCos 2.0, May 2003	FreeRTOS 4.4.0, July 2007	RTAI 3.5, February 2007	RTEMS 4.6.6, April 2006
Source Model/ License	Open source/ eCos License (GPL with exceptions)	Open source/ Modified GPL	Open source	Open source/ Modified GPL
API	POSIX (1003.1b), ITRON, "classic/native" API in C and Ada	Well written custom API, based on "classic" API in C	Custom API derived from RTLinux V1 API	uITRON 3.0 API, POSIX 1003.1b, BSD standards
Development Information/- Support	Books, papers/ Mailing list	Web tutorials/ Forum	Incomplete documentation/ Web support, mailing list	Wiki/ Contractual support

fort. Its strength is in its small size, making it possible to run where most other operating systems would not fit.

5 A Practical Example

The actual mapping of the basic and new introduced concepts of TTCN-3 language to a RTOS platform is explained by a simple test case for an example taken from the automotive domain. The chosen example consist of an embedded system on a MC9S12NE64 demo board [16] attached to a car door. The system is controlling various basic units of the door, for example, the window lifter, flashing light indicator, electrical central locking system. The system changes its internal states when receiving signals via RS232 serial interface. Those signals are in fact, some control strings with the role of triggering a special basic functionality of the door such as driving up the window, turning on the flash indicator and so forth. When receiving such a string, the system enters a different state, executes the function associated with that state, and sends back a

response string to the serial interface. The basic functions of the door could be combined so as to form safety applications such as when an accident occurs, the window should be driven down, the flash indicator should blink and the door should be automatically unlocked. Therefore, it is important to assure that the basic functionality is happening real-time. One can put several real-time constraints on such as: "the window should be driven down within ten milliseconds; the flash signal should be turned on within one millisecond; it should remain on for ten milliseconds then turn off."

In Figure 3 the test settlement is presented. On the left side we have the embedded system connected to the door, and on the right there is the TS consisting of a PC upon which is installed the FreeRTOS. The PC is an Intel Pentium 4 CPU with 3.20GHz and 1.00 GB of RAM. The operating system installed on the PC is Windows XP Professional version 2002. FreeRTOS has been optimized for an embedded system environment, having lack of resources. Although our aim is to develop a TS based on an embedded platform, in the development phase we prefer to use a PC with a port for FreeRTOS that runs on an integrated environment from the WATCOM open source project [17] (the distribution is for Windows). The PC is connected to the board through a serial cable. The real-time requirement that one wishes to test can be formulated in this manner: "The flash signal should be turned on within 30 milliseconds." For testing this requirement only, the signal is sent several times instructing for a repeated flashing. After sending one signal, and the flash is on, it is assumed that it automatically turns off after a fixed period of time. After expiring this period of time one can send another signal for turning it on again. This example epitomizes the new introduced concepts.

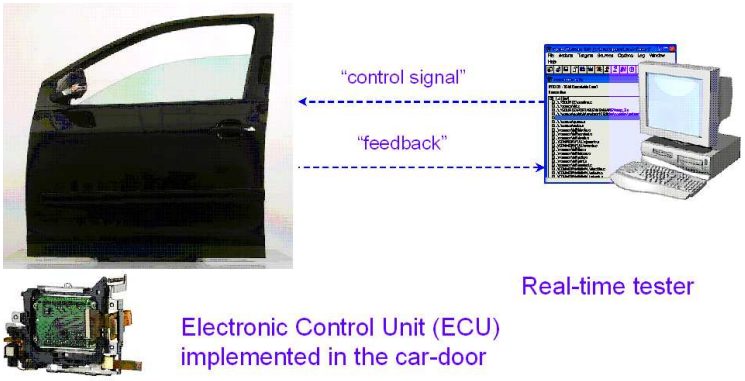

Fig. 3. Test Setup

5.1 TTCN-3 Test Specification

The TTCN-3 partial code of the behavior of the real-time test component is shown in Listing 1.6. The mapping between the port of the test component and

Listing 1.6. TTCN-3 Test Specification Sample Code

```
testcase rtTest() runs on RTComponent system System {
    ...                                                                    2
    map(self.P, system.P);
    // sending the string for bringing the system into initial state       4
    P.send(SYS_INIT_CODE);
    try(relative 0.03){                                                    6
        P.receive(feedbackInit);
    }catch(var rtexception rte){                                           8
        log(rte);
        setverdict(fail);                                                  10
    }
    // periodically sending the receive statement                          12
    try(relative sendDelay, sendPeriod, nrOfTimes){
        P.send(BLINK_ON_CODE);                                             14
    }catch(var rtexception rte) {
        log(rte);                                                          16
        setverdict(fail);
    }                                                                      18
    for(var int i:= 0; i<=nrOfTimes; i:=i+1){
        try(relative 0.03){                                                20
            P.receive(feedbackBlinkOn){
                finished := finished + 1;                                  22
                if(finished == nrOfDatasets) {
                    setverdict(pass);                                      24
                    stop;
                }                                                          26
            }
        }catch(var rtexception rte){                                       28
            log(rte);
            setverdict(fail);                                              30
        }
    } // end for                                                           32
} // end testcase
```

the port of the abstract SUT is performed (Line 3). This mapping should be implemented for the serial port. Then, the initialization string is sent to the SUT without any special time requirement (line 6). We wait for the SUT to move into its initial state and to send back a confirmation string which we capture (Lines 8-13). We introduce here the new **try-catch** statement for imposing time requirements to the receive operation. We should expect the feedback from the SUT no more than 30 milliseconds. This should be a real-time restriction for the SUT and therefore, the implementation of receive operation should introduce no delay. If there is no message received from the SUT within the indicated time, then a real-time exception will be captured and logged, and the verdict of the test is **fail**. After the initialization phase we send the command string for turning on the flash signaller for a number of times (Lines 15-20). We can observe here the new construct introduced for cyclic real-time restricted behavior. The first **send** should be triggered after *sendDelay* relative delay, and the next send operation should occur after every *sendPeriod* interval. The *sendPeriod* represents also the time required for the flash signaler to automatically turn off. We are expecting the feedback from the SUT for every send message in a **for** loop (Lines 23-37). For the **receive** operation inside the loop (Lines 24-36), we have the time constraint of 30 milliseconds expressed in the similar manner as previously described. This means that in 30 milliseconds from the **send** operation, we should receive confirmation from the SUT, which means that the function was executed and the flash signaler was turned on.

Listing 1.7. TTCN-3 FreeRTOS tSend Task Code

```
void v_tSend(void *pvParameters) {                                                    1
    ...
        for( ;; ) {                                                                   3
          if(i==0) vTaskDelay(tSendDelay); /* Suspend the current task for a given time */
      else vTaskDelay(tSendPeriod);                                                   5
           ...
      timel[i] = xTaskGetTickCount();  /* Send the string to the serial port */       7
          vSerialPutString( xPort, codes[codeId], strlen(codes[codeId]));
          /* Create the tExp task in order to wait for the response for this data set */ 9
          xTaskCreate( v_tExp, tExpName, STACK_SIZE, &i, mainMTC_TASK_PRIORITY, &tExpHandle[i] );
          i++;  //increment the counter i                                            11
    }
}                                                                                     13
```

Listing 1.8. TTCN-3 FreeRTOS tExp Task Code

```
void v_tExp(void *pvParameters) {                                                     1
   ...
   for(;;) {                                                                          3
       vTaskDelay(tExpConst);
       ...                                                                            5
       vTaskEndScheduler();
   }                                                                                  7
}
```

5.2 Test System Implementation at the FreeRTOS Level

The implementation using the FreeRTOS API is complex and not very easy to read. Therefore, we intent to illustrate only a few samples representing the basic implementation of $tExp$ timers, the implementation of the thread associated with the *send* instruction, as well as, the implementation of the thread associated to the *receive* instructions. These aspects are illustrated in Listings 1.7, 1.8 and 1.9 respectively. The synchronization between the processes is made using primitives $vTaskDelay$, for delaying a task for a defined period and $vTaskEndScheduler$, for interrupting the execution of other threads, which were provided by FreeRTOS.

It is important to note that these tasks are dependent on each other and therefore they are synchronized accordingly. Each time the $tSend$ task sends a message to the serial port (Line 10, Listing 1.7), it creates also a $tExp$ task (Line 15 and 15, Listing 1.7) corresponding to a timer. It can be observed that the first *send* operation is delayed with *tSendDelay* and the following are sent after each *tSendPeriod* expiration (Lines 5 and 6, Listing 1.7).

Each timer task contains a $vTaskDelay$ instruction (Line 4, Listing 1.8) which delays the running of the task for a period $tExpConst$ that represents the timer's expiration time. If the given time elapses, the timer task becomes active. By using of the primitive function $vTaskEndScheduler()$ all the other processes are killed (Line 6,Listing 1.8) and the test execution is finished. However, this is not done before assigning the verdict $fail$ to the current test (Lines 11 and 17, Listing 1.8). From the behavior of the receive thread we can observe that if a message was received (Line 5, Listing 1.9) and the message corresponds to the expected one (line 8, Listing 1.9), the timer thread associated with that message is killed (Line 12, Listing 1.9). The time difference between the sending

Table 2. The Results For The Presented Example

Sent_at (ticks)	Received_at (ticks)	Interval (ticks)	Constraint (ticks)	Verdict
2000	2002	2	3000	**pass**
3000	3001	1	3000	**pass**
4000	4001	1	3000	**pass**
5000	5118	118	3000	**pass**
6000	6002	2	3000	**pass**
7000	7001	1	3000	**pass**
8000	8031	31	3000	**pass**
9000	9001	1	3000	**pass**
10000	10001	1	3000	**pass**
11000	11541	541	3000	**pass**

of a message and the response of the message is calculated (Line10, Listing 1.9). Therefore that timer is deactivated.

Listing 1.9. TTCN-3 FreeRTOS tReceive Task Code

```
void v_pReceive(void *pvParameters){
   ...
       for( ;; ) {
              ...
              xGotChar = xSerialGetChar( xPort, &cRxedChar, xBlockTime );
              ...
              if(match(responseStr,responses[resId])) { // validate the response
                 time2[i] = xTaskGetTickCount();
                 timedif[i] = time2[i] - time1[i];
                 ...
                 vTaskDelete(tExpHandle[i]);
              }
   }
}
```

We observe that the function for retrieving the time is $xTaskGetTickCount$ (Line 9, Listing 1.7 and Line 9, Listing 1.9). For a tick rate of 10^5 Hz, we obtained the numbers listed in Table 2. All the tests were passed. The table contains the relative times, measured in numbers of clock ticks for sending and for receiving the message. The real-time constraint was 3000 ticks which means 30 milliseconds.

6 Conclusions and Future Work

This paper demonstrates a possibility of realizing real-time test systems by using the TTCN-3 testing language. The extended version of TTCN-3 presented in this paper has features for describing, analyzing and testing real-time properties of systems. The basic language concepts together with new concepts for defining time constraints, handling time-critical behavior and so forth are employed.

Although the project is in an early stage that covers only parts of an extended TTCN-3 mapping to a RTOS platform, the first objectives were already achieved:

– description and design of simple real-time tests using the existing and newly introduced TTCN-3 features;
– experiments with a selected RTOS platform;
– experiments on different ways of realizing TTCN-3 real-time concepts using the primitives provided by the real-time operating system platform;
– realization of a simple real-time test which demonstrates the power of the new concepts.

This paper provides only a validation of real-time tests through empirical results. A formal validation was not targeted in this paper, but it may represent the subject of a further research. Also, further work is needed for analyzing the full potential of enriching TTCN-3 with dedicated real-time features. The fundament of the work will be a specific real-time semantic for TTCN-3, which limits current unlimited elements of TTCN-3 executions, for example, by putting an upper number for the number of snapshots per alternative statement and so forth. Only this will allow to give execution time limits for TTCN-3 statements, making altogether TTCN-3 a real-time test language. Furthermore, our implementation of the new features and semantics on a real-time operating system will be continued. For that, this analysis and experimentation with the API's of other real-time operating systems such as RTAI or RTEMS will be continued. As long as the goal is to run TTCN3 inside an embedded operation system, a very advanced TTCN3 compiler should be developed, which is the greatest challenge of all. Although the derivation of code for embedded systems from abstract test specification may seem at first sight to be a cumbersome task, this goal can be achieved through efficient and well optimized mapping patterns.

References

1. L. Osterweil, "Strategic directions in software quality," *ACM Comput. Surv.*, vol. 28, no. 4, pp. 738–750, 1996.
2. G. J. Myers, *The Art of Testing*, 1st ed. John Wiley & Sons, 1979.
3. W. A. Halang and A. D. Stoyenko, *Constructing Predictable Real Time Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 1991.
4. R. Sinnott, "Towards more accurate real-time testing," in *The 12th International Conference on Information Systems Analysis and Synthesis (ISAS 2006)*, Orlando, Florida, 2006.
5. I. Schieferdecker, B. Stepien, and A. Rennoch, "Perfttcn, a ttcn language extension for performance testing." [Online]. Available: citeseer.ist.psu.edu/243959.html
6. T. Walter and J. Grabowski, "Real-time ttcn for testing real-time and multimedia systems," 1997. [Online]. Available: citeseer.ist.psu.edu/walter97realtime.html
7. Z. Dai, J. Grabowski, and H. Neukirchen, "Timed ttcn-3 - a real-time extension for ttcn," 2002. [Online]. Available: citeseer.ist.psu.edu/article/dai02timedttcn.html
8. I. Schieferdecker and J. Gromann, "Testing embedded control systems with ttcn-3," in *Testing Embedded Control Systems with TTCN-3*. Springer Berlin / Heidelberg: Kluwer, B.V., 2007, pp. 125–136.
9. E. E. . .-. V3.2.1, "Methods for testing and specification (mts); the testing and test control notation version 3; part 1: Ttcn-3 core language," 2007-02.

10. “Dedicated systems encyclopedia rtos list.” [Online]. Available: www.dedicated-systems.com/encyc/buyersguide/products/Dir1048.html
11. “Frequently asked questions of the comp.realtime newsgroup.” [Online]. Available: http://www.omimo.be/encyc/publications/faq/rtfaq.htm
12. “Freertos — open source, mini real time kernel.” [Online]. Available: www.freertos.org
13. “Rtai — the realtime application interface for linux.” [Online]. Available: www.rtai.org
14. “ecos — open source real-time operating system intended for embedded applications.” [Online]. Available: www.ecos.sourceware.org
15. “Rtems — real-time operating system for multiprocessor systems.” [Online]. Available: www.rtems.com
16. “Mc9s12ne64 demonstration kit.” [Online]. Available: http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=DEMO9S12NE64
17. “Open source watcom c, c++, and fortran cross compilers and tools.” [Online]. Available: www.openwatcom.org
18. H. W. Neukirchen, “Languages, tools and patterns for the specification of distributed real-time tests,” Ph.D. dissertation, Mathematisch-Naturwissenschaftlichen Fakultten der Georg-August-Universitt zu Gttingen, 2004.
19. V. Molovata, “Realizing real-time test systems using ttcn-3, diploma thesis,” 2007.
20. J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock, “An introduction to the testing and test control notation (ttcn-3),” *Comput. Networks*, vol. 42, no. 3, pp. 375–403, 2003.