# Challenges in Testing Real-Time Systems

Birgitta Lindström
University of Skövde
28 PUBLICATIONS 160 CITATIONS

# Challenges in Testing Real-Time Systems

Mats Grindal[i]
magr@enea.se

Enea Realtime AB
Box 232
SE-183 23 Täby
Sweden

Birgitta Lindström
birgitta.lindstrom@ida.his.se

Distributed Real-Time Systems Research Group
Department of Computer Science
University of Skövde
Sweden

## *Abstract*

Testing real-time systems presents more challenges than testing non-real-time systems; since, in addition to the value domain, the temporal domain also has to be considered. A number of design issues affect the testing strategies and the testability of the system. This paper gives a brief introduction to some of these design issues and explains how testing is affected by the different possible choices. The main conclusion is that testers need to partake in the design phase to safeguard the testability of the system, an issue that is otherwise easily overlooked.

Testing the temporal domain of a real-time system also affects potential tools for automatic test execution. The last part of this paper is devoted to explain the major requirements on testing tools stemming from the need to test in the temporal domain. The conclusion is that automating the test execution is a necessity for a successful test.

## 1 Introduction

More than 99% of the processors produced today are used in embedded systems [Tur99]. Many of these embedded systems have real-time requirements (e.g., cellular phones, fuel injection systems, modems, and video recorders). Still, to the best of our knowledge, little attention has been devoted to develop the theory and the best practice of testing real-time systems.

Although testing principles developed for non-real-time systems are applicable for real-time systems, the fact that time is a parameter in the testing complicates many issues. This paper is intended to give an overview of some of the more important issues affected by the need for considering time and explain the impact on the testing by these issues. In some cases, general advice about how to handle certain problems from a tester's point of view is given.

As is shown, many of these issues are related to hardware and software design decisions. This paper, therefore, is hoped to stimulate the discussion between testers and designers, resulting in better informed design decisions. It is argued that testers should participate as early as possible in the development process, to help increase the quality of the specifications. One example of this is the Rational Unified Process [Kru00]. This paper goes one step beyond this and concludes that testers should, not only review specifications but also, participate in the design process since design decisions have profound effect on the testing. It is also worth noting that our conclusions about early tester involvement substantiate the claims of Koomen and Pol in their Test Process Improvement (TPI) method [KP99]. One of the twenty key areas in TPI is "moment of involvement". The highest maturity level of this key area is to involve testers at the initiation of the project to add focus on testability.

Before going into details of the different issues, the terminology and basic concepts used in this paper is established in section 2. Each design issue affecting testing is described in a separate subsection in section 3. The dependencies between these subsections are kept to a minimum. In section 4 the ideas presented in sections 2 and 3 are put in the context of tools for testing and transformed into requirements on a real-time test case execution tool. Section 5 concludes this paper by summarizing the most important ideas and information.

## 2 Preliminaries

The following sections provide the necessary background on real-time systems and testing.

### 2.1 Real-time Systems

Many computer systems, including most real-time systems can be viewed as in Figure 1. Input, in this paper called an *event*, initiates a computation, in this paper called a *task*. Upon termination, the task produces a *result*. Loosely, a task can be understood to be an arbitrary computation. A *real-time task* is a task that must complete at an intended point in time [KV93]. In practice, it is usually enough if the real-time task completes before the intended point in time, that is, the *deadline.*

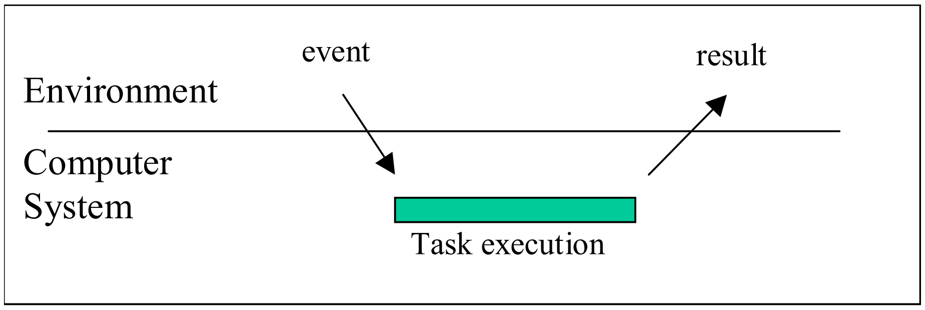

**Figure 1 Simple model of a computer system.**

This definition of a real-time task can be used to define a *real-time system* as a system that contains at least one real-time task [Sch93]. This is only one of many possible definitions of a real-time system with the common denominator that in real-time systems, both the value and the time domains are important. The reason for choosing this and the previous definitions in this section is to include as many real-time systems as possible.

Real-time systems are often classified according to the cost of missing a deadline. Locke [Loc86] describes four different classes (soft, firm, hard essential, and hard critical) of real-time systems based on the cost of missing a deadline. In a *soft* real-time system, completing a task after its deadline will still be beneficial even if the gain is not as big as if the task had been completed within the deadline and it may be acceptable to occasionally miss a deadline. In a *firm* real-time system, completing a task after its deadline will neither give any benefits nor incur any costs. In a *hard essential* real-time system, a bounded cost will be the result of missing a deadline. An example may be lost revenues in a billing system. Finally, missing a deadline of a *hard critical* system will have disastrous results, for instance loss of human lives.

Different events give rise to event types. Two events are of the same *event type* if they have entered the system through the same channel and they only differ in the time of entrance and possibly in the specific value. Event types may be very different from each other. Examples of event types are temperature readings, keyboard commands and pushing a call button of an elevator. When designing real-time systems the types and frequencies of events are important to consider. An event type may be periodic, sporadic or aperiodic. An event type is *periodic* if an event of that type occurs with a regular (and known) period. An event type is sporadic if events of that type may occur any time but there is a known minimum inter-arrival time between two consecutive events of that type. An event type is *aperiodic* if nothing is known about how often events may occur or when it is known that events may occur anytime. The peak load that is assumed to be generated by the environment is called the *load hypothesis* [KV93] and it is often formulated in terms of types and frequencies of the events from the environment.

## 2.2 Testing

*Testing*, that is, dynamic execution of test cases [BS98], has two main goals. These goals are assessing and increasing reliability [FHL+98]. Testing as a means of *assessing the reliability*, relies on choosing and executing test cases based on some operational distribution and monitoring the number of encountered failures. A *failure* is defined as a deviation of the software from its expected delivery or service [BS98]. Testing as means of increasing the reliability, builds on selecting test cases that are assumed to be especially likely to cause failures. The observed failure is analyzed to find the cause of the failure, which is the *fault* [BS98]. The fault is removed and the reliability is assumed to increase. Many different test methods exist (e.g. equivalence partitioning, boundary value analysis, state-based testing, syntax testing, [Bei90]) that are all assumed to generate test suites containing test cases especially prone to revealing failures.

The strategies and methods used for testing the value domain in non-real-time systems can to a large extent be used without alteration in real-time systems. A good example of this is the DO178b standard for testing avionics systems [DO92]. This standard requires the testing of the most safety critical parts of the avionics system to reach 100% Modified Condition Decision Coverage (MCDC). MCDC is a code coverage criterion totally independent of the temporal properties of the application.

The main challenge in testing real-time systems is that they need to be tested in the temporal domain as well as the value domain [Sch93]. Testing in the temporal domain has several implications. The input to the test object may need to be issued at a precise moment. The temporal state of the test object at the start of the test execution may need to be controlled. The timing of the result may need to be observed. There is a potential for non-determinism etc.

Two central concepts are observability and controllability. *Observability*, that is, the functionality facilitated by the system to observe or monitor what the system does, how it does it, and when it does it [Sch93], and *controllability*, that is, the functionality available to the user to control the (re-) execution of a test case [Sch93]. The *testability* of a system is defined as the attributes of software that bear on the effort needed for validating the [modified] software [ISO91]. The testability of a system depends to a large extent on the observability and controllability of the system [Sch93].

## 3 Design Trade-offs for Testability in RT Systems

The following sections describe how testability is affected by different design decisions.

### 3.1 Overview

When designing a real-time system there are many decisions to make. The outcomes of most of these decisions affect the testing in one way or another. In the following subsections we will describe and discuss issues relating to some of these decisions. We have selected issues that are likely to be present in many real-time systems and that have a major impact on the testing. Scheduling and the choice of design paradigm are the first two issues covered. These two issues are closely related and have a major impact on the testing strategy that can be used [LMA02]. The next two issues: tracing and support for state manipulation give concrete examples of how to achieve observability and controllability in a real-time system. The final issue, caching, is motivated by the fact that most commercial processors today contain one or more levels of caches. Introduction of caching in a system may increase the performance of the system but there are severe drawbacks from a testability perspective.

### 3.2 Scheduling

In practice, most real-time system contains more than one task. Sometimes two or more tasks will be possible to execute at the same time. In these cases there must be rules for in which order the tasks should be executed since only one task at a time can use a processor[ii] or other resources. Determining the order of the task execution based on the supplied rules is called *scheduling*. Scheduling can be either static or dynamic.

In *static* scheduling, the execution order of the tasks is determined in advance. In a simplified description, as soon as the execution of one task is finished the processor finds the next task to execute by looking in a table containing the pre-calculated order. Often these pre-calculated orders are cyclic.

In *dynamic* scheduling there is no pre-calculated execution order. Instead there is a set of rules for how to resolve a conflict when two or more tasks want to execute at the same time. One common approach is to assign a priority to each task and force the tasks to execute in priority order. It is not uncommon to let higher priority tasks interrupt lower priority tasks. This is called *preemption*.

### 3.3 Design paradigms

There are two different design paradigms for real-time systems: time-triggered and event-triggered [Sch93]. The main difference between the two design paradigms is when communication between the real-time system and the environment is performed. A *time-triggered system* only communicates with its environment at predefined points in time. Events that has occurred in the environment since the latest communication point will not be detected and reacted upon by the real-time system until the next communication point. Similarly a result computed by the time-triggered real-time system will not be passed back to the environment until the next communication point. Figure 2 illustrates this concept. The consequence of communicating with the environment only at specific points in time is that the system works in cycles (… read events, execute tasks, write results…). This in turn means that the cycle time, that is, the time between two communication points need to be long enough for the worst-case execution time for any anticipated combination of tasks. Overload situations cannot be handled and should not be possible since the system is designed for an assumed worst case. The normal way of implementing a time-triggered system is by polling and it is common to use static scheduling in time-triggered systems.

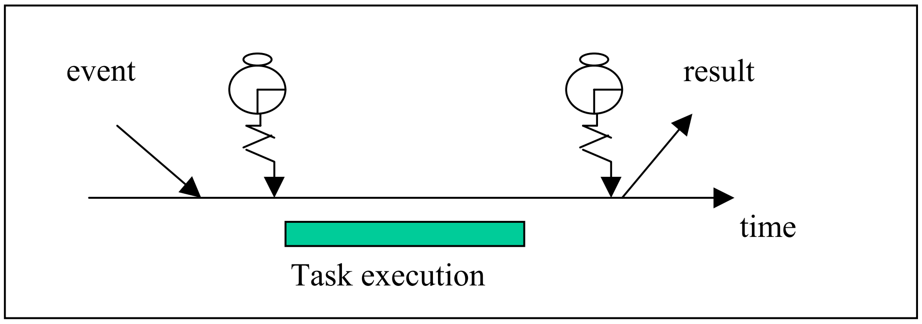

**Figure 2 Observation and reaction to an event in a time-triggered system.**

In an *event-triggered system* there are no special points in time when communication has to occur between the real-time system and its environment. Instead events are observed and reacted upon as they happen. Produced results are similarly communicated to the environment as soon as they are ready. This can be seen in Figure 3. Event-triggered systems must be scheduled dynamically and a normal way of achieving this is by using interrupts.

In contrast to time-triggered systems, an event-triggered system may face overload situations. The effect is that the event-triggered system must be designed to handle such situations dynamically, in a best-effort manner. This means that deadlines can occasionally be missed and it is the responsibility of the designer to minimize the damage in such situations. In many event-triggered systems it is crucial to guarantee a minimum level of service. Taking a car as an example, the braking service is critical to uphold at all times, whereas the climate control service can be allowed to fail during an overload situation. Therefore, it is essential to make a correct load hypothesis and design the system with enough resources to maintain this minimum level of service under the load hypothesis.

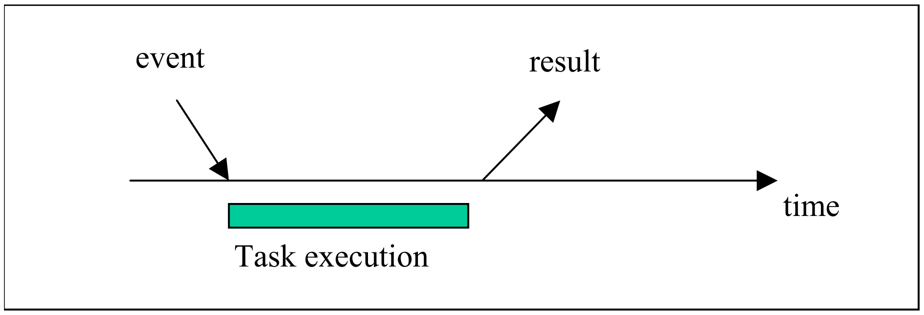

**Figure 3 Observation and reaction to event in an event-triggered system.**

The choice of design paradigm has consequences for a number of properties of the real-time system. Unfortunately the desired values of the different properties are in conflict with each other both for time-triggered and event-triggered systems. Figure 4 gives a simplified overview of how some of the different properties are in conflict with each other for the two design paradigms. This will be further explained in the proceeding sections.

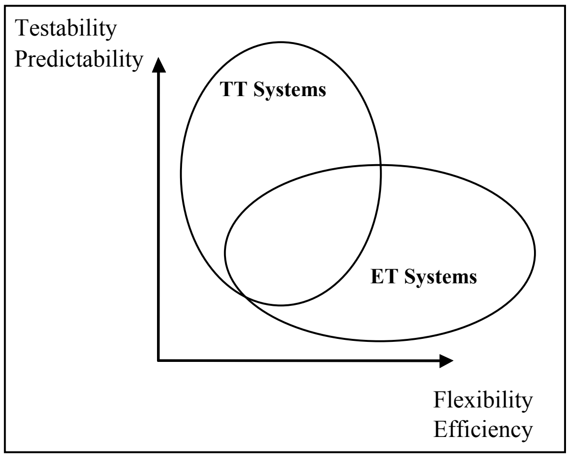

**Figure 4 Trade-off between testability/predictability and flexibility/efficiency for the two design paradigms**

- Testability
  - Time-triggered design. The order of events occurring within the same observation interval is insignificant. All of the events are communicated at the same time, that is, at the next communication point. Due to static schedule, the tasks corresponding to the events are executed in the same order regardless of the actual order of the events. This means that there is a finite, albeit large, number of possible behaviors for a time-triggered system, which is why time-triggered systems can be tested with systematic coverage exploration. Moreover, the controllability is increased since the tester only need to bother about in which observation interval an event is input to the system.
  - Event-triggered design. Different order of events may lead to different behavior since the dynamic scheduling algorithm continuously changes the order of task execution based on the current situation, that is, the state of the system and the incoming event. Further, for two event sequences with the same order, the exact time of occurrence of an individual event may affect the result, again due to the dynamic behavior of the scheduling. This makes testing more difficult. In addition, many of the dynamic scheduling algorithms are heuristic, which means that the result of executing the same input with the same timing starting from the same state might yield different results. This leads to the observation that testing with systematic coverage exploration is not feasible in event-triggered systems. Instead we must use statistical testing with tailored loads.

- Predictability
  A system is predictable if the effect of a task can be unambiguously derived from knowledge of that task and its execution environment. Randomness, heuristics, and race-conditions all have a negative impact on predictability. Usually predictability focuses on the observable end result. However, in this article we are not only concerned with the observable end result (e.g., that a deadline was met) but also to some extent how the result was obtained (e.g., which interleaving of tasks that was actually executed). Thus, we will use the term predictability in this wider meaning.
  - Time-triggered design. The static scheduling and the insignificance of the order of events in the same observation period of these systems increase the predictability of the system since the interleaving of tasks is completely determined in advance. A high predictability makes it possible to use systematic coverage criteria in the testing. Predictability is also of extreme importance in hard critical real-time systems since such systems require high confidence that the system will work in all situations.
  - Event-triggered design. The dynamic scheduling and in particular the heuristics involved in the scheduling decision decreases the predictability of the behavior for a given input sequence. This will of course decrease the confidence in the test results and also make regression testing harder. This is one of the main reasons why statistical test methods should be used for event-triggered systems. In statistical methods the same input sequence may be executed many times to increase

the confidence that the system works correctly under all circumstances for that input sequence. Mimicking the operational conditions of the system is also more important for this reason for event-triggered systems. It is important to note that for most event-triggered systems testing alone will not permit that 100% confidence is gained of the reliability of the system. This is one of the reasons why event-triggered systems are seldom in practice used for hard real-time systems.

- Flexibility
  - Time-triggered design. A time-triggered system is inflexible when it comes to changing the system or altering the load or fault hypotheses. Alterations exceeding possible spare capacity for future extensions in the system or change of the prerequisites require at least that the static schedule is recomputed and the system reintegrated and re-tested. Sometimes the system even needs to be redesigned, in particular if the change increases the resource demand.
  - Event-triggered design. An event-triggered system is flexible by its nature. The dynamic scheduling and its ability to handle overloads make the event-triggered systems suitable in less predictable environments. However, it is important to note that a change to the system, even strict removals of parts of the system, effectively invalidates all previous test results.

- Efficiency
  - Time-triggered design. In a time-triggered system, the schedule usually is static. Sporadic tasks are scheduled as if they where periodic. The tasks execute with their worst-case execution time. When the worst case in terms of execution time or arrival rate for sporadic tasks differ much from the average case, then there is a waste of resources. The reason is that whenever a task completes in less time than its worst case execution time, the unused time cannot be used for anything else.
  - Event-triggered design. This paradigm demands dynamic scheduling. Sporadic tasks are scheduled on arrival. A more critical task than the currently executed task leads to a preemption of the current task whereas a less critical task than the current task has to wait. The execution time varies, which means that if there is a difference between average-case execution time and worst-case, then the efficiency is much higher in an event-triggered system than it would be in a time-triggered. The reason is that any unused time may be used for less critical work by dynamically scheduling an appropriate task.

Pure time-triggered or event-triggered systems are rare. Many systems have characteristics from both paradigms. A natural trade-off for systems with mixed criticality is to design the critical part of a system according to the time-triggered paradigm for predictability reasons and design the rest of the system according to the event-triggered paradigm for efficiency reasons. This can be done if it is possible to separate the critical parts from the non-critical parts. During the trade-off discussions, it is important to consider the testing issues as well as all other properties.

## 3.4 Traces

A common approach to facilitate observability during testing is to use traces. The test object is instrumented by extra code, usually write statements that prints values of interesting variables. During test execution these write statements produce a log, which is analyzed after the test execution to determine the test results or identifying the cause of a failure. The instrumentation of the code introduces a *probe effect* [Gai86]. The probe effect occurs because the behavior of the instrumented system under test is different from the final version of the system.

The probe effect is more severe in real-time systems than in non-real-time systems. There are two reasons for this. First, the traces need to contain more information to include the temporal information, which leads to more instrumentation code being inserted. Second and more importantly, each code instruction that we add to the code will necessarily affect the temporal behavior since each instruction will add execution time. This means that the test results obtained from an instrumented version of the test object might not be valid for a non-instrumented version of the same test object. It is important to note that the changed behavior due to the probe effect often increases but may sometimes even decrease the response time for a specific task due to the changed interleaving of the task execution. If it were always the case that response times were increased an instrumented test object meeting its deadlines would imply that the corresponding uninstrumented object would also meet the deadlines. Since the timing is affected, we might get different interleavings among the tasks due to the extra instructions. If there is a race condition, a deadline might be met that would not have been met if it were not for the extra instructions. The conclusion is that a changed timing behavior in a real-time system may change the results themselves, not only the timing of the results.

A common approach to deal with the probe effect is to leave the instrumentation in the final product. This requires, however, that there is a mechanism for hiding unwanted information for the end user and that using this mechanism in itself does not alter the timing of the application.

## 3.5 Support for system state manipulation

Often, a test case requires the system under test to have a certain internal state as a starting point of the test case. A big challenge for the tester is to achieve the required state prior to test case execution. The state of the system shall, in this scope, be interpreted as any requirement imposed by the test case on the system under test. Obviously, the nature of these requirements depend on the test case but common examples of such requirements include certain variables having specific values, task queues containing specific tasks, and certain amount of dynamic memory already allocated. For real-time systems these prerequisites may also include timing requirements, for instance that a certain task has already executed during exactly 10 ms.

Achieving the right system state prior to test case execution is strongly related to controllability since controllability includes all means of preparing and controlling the test case execution. However, for the tester the process of achieving the right system state may also include a check that the right state really has been achieved. In these cases, some aspects of observability are also included.

Dick and Faivre [DF93] describe two methods of achieving a required internal state prior to test case execution. One method is to demand from the system under test to provide special test-bed functions, which can place the system under test directly in any desired state. The other method is to start in an idle state and then execute other test cases and set-up scripts that will result in the system having the desired state. Although from a testing point of view having specific test-bed functions clearly is preferable, there are several drawbacks with this approach. Sometimes it will not be possible to include specific test functionality, for instance if software components are imported from somewhere else. Even if possible, it might not be practical due to the cost and complexity introduced. Finally, if dedicated test-bed functions are built-in to the system, care must be taken so that these functions will not become a security hazard. Thus, using test cases and set-up scripts will in many cases be the only option, even if this method restricts the controllability.

Although this discussion has not specifically mentioned real-time systems, the same reasoning apply with the addition that controlling a system in the temporal domain is even harder than controlling a system in the value domain. In practice, testing real-time systems when there are requirements on the time domain prior to test execution will require a statistical approach or a trial-and-observe approach. The statistical approach builds on the assumption that if we repeat the test cases enough times, we will have achieved the desired conditions at least one time, but we do not explicitly check that this is the case. The trial-and-observe approach require that there are means for observing the actual state before test case execution was performed, thus making it possible to determine if the desired conditions were met.

## 3.6 Caching

A cache is a small memory located on the processor chip together with the processor. Since the cache memory is located close to the processor, an access to the cache memory is several times faster than an access to the ordinary memory, which is implemented on a separate memory chip. A program with code and data stored in the cache memory will thus execute faster than if the program is stored in ordinary memory. However, the area on the processor chip allocated to the cache memory is usually rather small, since most of the area is used for the processor itself, so a complete program (e.g., code and data) will normally not fit in the cache memory.

An efficient use of the cache would be to fetch data and instructions that will soon be needed by the program and store it in the cache, and remove all data from the cache that is not needed anymore. This is also done in practice by a cache replacement algorithm, but unfortunately it is not always possible to make perfect guesses which data are needed in the near future and when data is used for the last time and thus can be replaced. The rules governing these guesses is called a *cache replacement policy*. The actual execution time will vary whether or not the needed code and data is in the cache when needed. This in turn depends on the replacement policy of the cache, the size of the cache, and the initial contents of the cache. This means that the introduction of a cache into a computer system will introduce some unpredictability of the execution time, unless the contents of the cache can be completely controlled and this is usually not the case in commercial processors.

For testing of non-real-time systems the effect of caches in the system is normally not an issue, since variations in the execution time of a particular task generally will not affect the actual result of that task. However, for testing of real-time systems, caching will have a profound effect. Given a task that completes within its deadline on a system with caches, what can be concluded? In one extreme situation all memory accesses will have been made to the cache, resulting in a minimal execution time for the task. In the other extreme situation all memory accesses will have been made to the ordinary memory, resulting in a maximal execution time. The problem for the tester is that it is usually not known what the caching situation was.

The standard way of handling caching during test of real-time systems is to disable the caches completely during testing under the assumption that this represents a worst-case, and if tasks complete on time without caching, they will complete on time also with caches enabled. While this is true for an unloaded system it is not necessarily true in an event-triggered system where other tasks are executing at the same time. The reason is that when caches are enabled, the interleaving of the tasks will probably be different due to changed execution times of the tasks. This change of interleaving might in turn cause a higher priority task to block the processor for the task under test, making it miss its deadline.

Thus, the ultimate solution to the cache problem from a testing perspective in event-triggered real-time systems is to remove the caches from the system altogether. If caching is used, the only practical testing strategy for event-triggered real-time systems is statistical testing.

For time-triggered systems, the pre-calculated schedules should be based on the worst case execution time [PK89], which in a cached system occurs when the data needed happens to be non-cached. Thus, it is hard to for time-triggered systems to benefit from caches.

## 4 Testing tool Issues

The area of testing tools for non-real-time systems has already received considerable attention elsewhere [FG99], [Hay95] so this section will only focus on features needed in tools for testing real-time systems. For most types of tools the real-time aspect of the developed system does not affect the applicability of the tool. This is especially true for administrative tools, such as tools for test management, traceability, and error reporting. However, as soon as [parts of] the test case execution shall be automated in a real-time system the timing aspects need to be considered.

It is quite difficult to find commercial testing tools for automatic test execution of real-time systems. There are several reasons for this. One reason is that the support for timeliness is difficult to implement, especially if the system under test is regarded as a black box, which is a necessity for a commercial program. For instance, the time granularity of the tool needs to be as small or smaller than the time granularity of the system under test. Another reason for the difficulty of finding commercial tools for automatic test execution of real-time systems is that many real-time systems are embedded and lacking standardized interfaces. Also many real-time systems have specialized application domains with very specific demands. Imagine for instance the different demands of mobile phones, brake-by-wire systems, and pacemakers. Still another reason is that some systems are built using state-of-the-art technology, which means that there hardly exist tools that support the new technology. The overall implication of this reasoning is that most tools for automatic test execution of real-time systems are built in-house.

Since building tools in-house is often the only option, based on the contents of this paper, we will give a small overview of features that might be handy in a tool for automatic test execution of real-time systems. Even if a commercially available tool is considered, this overview may serve as a checklist for evaluation of the tool. The order of the items in the list is not significant.

- Probe-effect
  In many cases the tool needs to intervene with the system under test. For embedded systems it is quite common that a part of the test tool is located on the target system to facilitate detailed observation despite limited means of communication. In other cases the tool might rely on specific timing information that is obtained by instrumenting the code with special test instructions. In both cases the test tool will inflict a probe-effect on the system under test. In such cases it is beneficial to have the ability to determine the amount of probe-effect caused by the tool. To some extent this can be achieved by having the tool measuring itself. A complementing approach is to make theoretical calculations of the probe-effect based on known benchmark figures and the actual results of the test execution.

- Event injection
  Event injection or stimuli of the system under test is an important aspect of a testing tool for automatic test execution. For a real-time system timing of the input is important. One useful feature of a tool is to be able to release stimuli to the system under test at a specific [predefined] time. Another, related and equally useful, feature is to be able to release stimuli to the system under test with specified delay relative to an event, which is either another release by the tool or an event occurring in the system under test that is perceived by the tool.

  Ordinary load generation tools intended for non-real-time systems usually support some form of customization of the distribution of the load. Such and related features are of course useful in the real-time case as well.

- Observation of timing
  Observation of events generated by the system under test is also an important area when automating the test execution. Part of deciding if a timeliness test case passes of fails is to determine if the deadline was met. There are basically two methods to implement such a check. Either the tool supports timers or the events perceived by the tool are time-stamped.

  A tool supporting timers is usually limited to measure elapsed time of stimuli-response pairs, where the stimuli originates from the tool, since the timer has to be triggered somehow. An advantage is that there are neither probe-effect nor overhead introduced by the tool since all of the intelligence is outside of the system under test. Another advantage with timers is that the pass/fail decision can be made in real-time if the timers are implemented as time-outs. A drawback is that signal transfer and processing times outside the system under test is included in the round-trip delay. A challenge in the timer approach is when there is not a one-to-one correspondence between stimuli and response or when stimuli-response pairs are interleaved.

  The other solution having events time stamped requires the tool to perform post-analysis to calculate the actual round-trip delays this prohibits making the pass/fail decision in real-time, but gives the advantage of handling events not only originating from the test tool itself. If the time stamping is made by the system under test this will either result in a probe-effect or extra overhead (if the time stamping is kept in the final system). In addition, the communication between the tool and the system under test is increased since the time stamps are added to the events. If the tool, on the other hand, time stamps the events the probe effect and/or overhead are removed and the communication to the tool is reduced, but again we face the problem with inclusion of transfer and handling times outside the system under test.

- Synchronization
  As soon as there are more than one locus of control in a computer system there might be a clock synchronization problem. In the case of a tool for automatic test execution, if the clocks of both the

tool and the system under test are used then a synchronization problem arises. Obviously the same applies in a distributed real-time system with multiple clocks. Clock synchronization can be achieved in two ways. Either a global clock is used as a master clock frequency, or the local clocks are used together with a clock synchronization protocol. In either case a real-time network is required.

Another, more coarse-grained, synchronization problem arises whenever the test tool supports distribution. Many tools for load generation allow multiple clients to generate the load, in order to generate larger loads than a single machine can manage. In such cases the different clients need synchronization with respect to at least start and stop of load generation. It is quite an implementation challenge if high precision is needed in such synchronization, and a real-time network is soon needed, increasing both the complexity and cost of the tool.

- Resynchronization in case of failure
  More advanced tools for automatic execution of test cases, real-time or not, supports resynchronization of the test execution after a failure has been detected. The idea is that after a failure, the test tool can take actions to restore the system under test into a known state, and resume test case execution an a point in the test suite after the test case, which failed.

  Due to the complexity in the actions needed to diagnose and reset the system under test, even for non-real-time systems resynchronization after a failure is a big challenge. It is an even bigger challenge, when real-time aspects need to be taken into account. Recent research on the topic reports promising but not yet widely available results. Form instance, Iorgulescu and Seviora [IS97] report results on real-time supervision and diagnose of a telephony system. Their work is based on a specification written in Specification and Description Language (SDL). The specification is used to generate possible but erroneous states, which are compared with the actual erroneous state. When the problem has been diagnosed, suitable actions corrective actions can be taken to resume the test case execution. All this is done in real-time, which is a prerequisite in timeliness testing.

## *5 Conclusions*

Real-time systems are more challenging to test than non-real-time systems. The main reason is that in real-time systems tests have to be performed both of the temporal and the value domain. This affects both observability and controllability and makes them both harder to achieve, making real-time systems less testable than non-real-time systems.

A number of design decisions will affect the testability of the real-time system. Often performance and flexibility of the real-time system is in direct conflict with the testability of the system. Thus, it is of great importance that the tester participates in these design decisions, so that the issue of testability is not lost. In addition, the tester may safeguard the implementation of explicit test support functionality, e.g., support for tracing and possibilities to turn caches off and on. Such test support functionality may alleviate the testing to a large extent.

The two main design paradigms, event-triggered and time-triggered require very different testing approaches. In event-triggered real-time system the only feasible testing strategies are based on statistical methods. Time-triggered real-time systems may under certain conditions be tested using systematic coverage exploration.

The challenges in testing real-time systems are visible also in the tools for automatic test case execution. For testing problems with fine time granularity the use of tools for automatic test case execution is necessary. However, due to the diversity of real-time systems it is hard to build general-purpose tools. Custom-designed tools are often expensive to build but often the only option. In particular if the real-time system is proprietary or based on edge technology.

## 6 Acknowledgements

The authors are greatly indebted to Jonas Mellin and Robert Nilsson for the large amount of help in writing and reviewing this article. Further, the authors would like to thank the members of the Distributed Real-Time Systems Research Group in the Computer Science Department of the University of Skövde for the constructive advice on how to present this material.

Much of the inspiration to write this paper comes from discussions with various persons at the Testing Department of Enea Realtime AB and at the Distributed Real-Time Systems Research Group in the Computer Science Department of the University of Skövde. The authors express their gratitude to all those persons who unknowingly have contributed to this work.

## 7 References

[Bei90] B. Beizer, *Software Testing Techniques*, second edition, Van Nostrand Reinhold, ISBN 0-442-20672-0, 1990

[BS98] BS 7925-1 *Software Testing Vocabulary*, British Standardisation Institute, 1998

[DF93] J. Dick and A. Faivre. *Automating the Generation and Sequencing of Test Cases from Model-based Specifications*, Proceedings of FME'93: Industrial-Strength Formal Methods, Springer-Verlag Lecture Notes in Computer Science Volume 670, 1993, Odense Denmark, pp. 268-284

[DO92] *DO-178B Software Considerations in Airborne Systems and Equipment Certification*, RTCA Inc, 1828 L Street NW, Suite 805, Washington, DC 20036, 1992, URL: http://www.rtca.org

[FG99] M. Fewster and D. Graham, *Software Test Automation: Effective Use of Test Execution Tools*, 1st edition, Addison-Wesley Pub Co; ISBN: 0-201-33140-3; 1999

[FHL+98] P.G. Frankl, R. G. Hamlet, B. Littlewood, and L. Stringini, *Evaluating Testing Methods by Delivered Reliability*, IEEE Transactions on Software Engineering, Vol. 24, No. 8, Aug., 1998

[Gai86] J. Gait. *A Probe Effect in Concurrent Programs*. Software – Practice and experience, 16(3):225-233, Mar. 1986

[Hay95] L. Hayes, *Automated Testing Handbook*, Software Testing Institute; ISBN: 0-970-74650-4, 1995

[IS97] R. Iorgulescu and R.E. Seviora. *A Method for Continuos Real-time Supervision*. Software Testing, Verification and Reliability, Vol. 7, pp 69-98, 1997

[ISO91] International Standard ISO/IEC 9126. *Information technology – Software product evaluation – Quality characteristics and guidelines for their use*, International Organization for Standardization, International Electrotechnical Commission, Geneva

[KP99] T. Koomen and M. Pol, *Test Process Improvement A practical step-by-step guide to structured testing*, Addison-Wesley and ACM Press, ISBN 0-201-59624-5, 1999

[Kru00] P. Kruchten, *The Rational Unified Process An Introduction Section Edition*, Addison-Wesley, ISBN 0-201-70710-1, 2000

[KV93] H. Kopetz and P. Veríssimo, *Real Time and Dependability Concepts*, Chapter 16 in Distributed Systems, second edition, Addison-Wesley, edited by S. Mullender, ISBN 0-201-62427-3, 1993

[LMA02] B. Lindström, J. Mellin, and S.F. Andler, *Testability of Dynamic Real-Time Systems*, Proceedings of Eighth International Conference on Real-Time Computing Systems and Applications (RTCSA2002), Tokyo Japan, 18-20 March 2002, pp. 93-97

[Loc86] C.D. Locke, *Best-Effort Decision Making for Real-Time Scheduling*, Technical Report CMU-CS-86-134, Department of Computer Science, Carnegie-Mellon University, USA, 1986

[PK89] P. Pushner and Christian Koza, *Calculating the Maximum Execution Time of Real-Time Programs*, Journal of Real-Time Systems, Vol. 1, No 2, Sept. 1989, pages 159-176.

[Tur99] Jim Turley, *Embedded Processors by the Numbers,* Embedded Systems Programming Vol. 12, No 5, May 1999, URL: http://www.embedded.com

[Sch93] W. Schütz, *The Testability of Distributed Systems*, Kluwer Academic Publishers, ISBN 0-7923-9386-4, 1993

---

[i] M. Grindal also holds a  part time position at the Distributed Real-Time Systems Research Group at the Department of Computer Science, University of Skövde

[ii] We have chosen to omit multiprocessor systems from this discussion since we want to focus on the general issues of testing real-time systems. Synchronization is one aspect that will add further testing difficulty in a multiprocessor system. Synchronization is briefly covered in section 4.