

# Testability of Dynamic Real-Time Systems: An Empirical Study of Constrained Execution Environment Implications

Birgitta Lindström

University of Skövde, P.O. Box 408  
SE-541 28 Skövde, Sweden  
birgitta.lindstrom@his.se

Jeff Offutt

George Mason University  
Fairfax, VA 22030, USA  
offutt@gmu.edu

Sten F. Andler

University of Skövde, P.O. Box 408  
SE-541 28 Skövde, Sweden  
sten.f.andler@his.se

## Abstract

*Real-time systems must respond to events in a timely fashion; in hard real-time systems the penalty for a missed deadline is high. It is therefore necessary to design hard real-time systems so that the timing behavior of the tasks can be predicted. Static real-time systems have prior knowledge of the worst-case arrival patterns and resource usage. Therefore, a schedule can be calculated off-line and tasks can be guaranteed to have sufficient resources to complete (resource adequacy). Dynamic real-time systems, on the other hand, do not have such prior knowledge, and therefore must react to events when they occur. They also must adapt to changes in the urgencies of various tasks, and fairly allocate resources among the tasks. A disadvantage of static real-time systems is that a requirement on resource adequacy makes them expensive and often impractical. Dynamic real-time systems, on the other hand, have the disadvantage of being less predictable and therefore difficult to test. Hence, in dynamic systems, timeliness is hard to guarantee and reliability is often low. Using a constrained execution environment, we attempt to increase the testability of such systems. An initial step is to identify factors that affect testability. We present empirical results on how various factors in the execution environment impacts testability of real-time systems. The results show that some of the factors, previously identified as possibly impacting testability, do not have an impact, while others do.*

## 1 Introduction

A *real-time system* is required to have *timeliness* (i.e., a timely behavior). The penalty of a missed deadline can be particularly high for hard real-time systems. Hard real-time systems are therefore usually designed so that their timing behavior is highly predictable. Common approaches for this includes scheduling tasks statically and assuming there will always be enough resources (*resource adequacy*). As a consequence, such systems have a predictable behavior that is easy to test and also enhances the ability to verify timeliness with formal methods. We refer to such real-time systems as *static RTS*.

Building static RTSs requires knowledge about the worst case execution time and maximum need for resources such as shared data and bandwidth. Designing for resource adequacy is often expensive and infeasible when the environment is unpredictable. The alternative is to use an approach that includes dynamic, online scheduling. Such systems react to events by deciding how to react on a case-by-case basis. A new task may be triggered in response to the event. The new task is assigned a priority based on urgency and criticality and the queue is rescheduled. If the new task has a higher priority than the task that is currently executing, the executing task is preempted in favor of the new task. We call such real-time system *dynamic RTS*. The drawback of dynamic RTSs is that they are inherently harder to test than static RTSs [13]. The major reason is that execution of a dynamic RTS is decided by the often unpredictable environment while execution of a static RTS is decided by time. Moreover, a static RTS has harder constraints on its execution environment than dynamic RTSs and

these constraints result in predictable behavior.

The problem that a real-time systems designer sometimes faces is that building a static RTS is too expensive due to an unpredictable environment and building a dynamic RTS is too risky due to low testability. We do not yet know how to build dynamic RTSs with enough support to test timeliness properties. This is the problem we are trying to address in this research, specifically in terms of assessing and increasing testability in dynamic RTSs.

According to the IEEE Standard Glossary of Software Engineering Terminology, *testability* is the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met [8]. The real challenge is to increase testability in dynamic systems while maintaining the dynamic semantics. An important step to building dynamic RTSs that are easier to test is to understand how testability is affected by the execution environment.

This paper describes an empirical study of the impact of the execution environment constraints on testability of dynamic RTSs. Testability is discussed in the context of testing for timeliness on a system level. Constraints on the execution environment are varied and their impact on testability estimated. The choice of constraints is based on previous work by Birgisson, Mellin and Andler that defines an upper bound on test effort for dynamic RTSs [2]. The experimental goal is to see whether our results support their claims of how the constraints affect testability. Our hypothesis is that each constraint in their formulae is significant for testability. We also evaluate whether the formulae can be used as an approximation of testability, that is, if the upper bound is sufficiently tight.

## 2 Background

The effort to test a system, *test effort*, is affected by a number of different aspects such as test process, degree of automation, and the skills of the test team. An important aspect is the testability of the system. Testability is a concept that has proven to be hard to define. Several inconsistent definitions exist [8, 14, 17]. The reason is that testability is an emergent property of the system itself but the support for testing that is given by the system depends on what we are testing for. Hence, a system may have high testability with respect to some testing activities (for example, logical correctness in a software unit) and low with respect to other (for example, performance in a target system). It is, therefore, our opinion that testability is a system property that can best be defined and estimated with respect to the purpose of

a testing activity. This is also reflected in the definition of system testability used in this paper: *System testability is the degree to which a system has a design or implementation that supports selection, execution, observation, and analysis of tests targeting verification of required system properties.* This paper focuses on testing of timeliness and the required property here is timeliness. We refer to this as *timeliness testability*.

Concurrency, resource allocation policy, and online scheduling are major factors that affect testability in dynamic RTSs. Execution of concurrent processes is interleaved in some order decided by a dynamic scheduler that bases each decision on current state. Race conditions and small variations in timing may result in different execution orders (that is, interleavings in a task set). Introducing accelerating hardware such as caches and pipes means that variations in timing are increased (that is, the difference between best case and worst case with respect to elapsed time is increased) and thereby harder to predict. Hence, the same input may lead to different behavior with respect to when things happen. The consequence is not only that it is hard to repeat tests, it also makes the results less trustworthy since meeting a deadline on one test execution does not guarantee it will be met the next time that test is run. A final observation is that the problem with several potential execution orders tends to get worse when the system is stressed by a high task load and event bursts. These are precisely the situations that a tester would use to provoke the system to miss a deadline. When the load is normal, race conditions and preemptions are more rare and the resulting behavior with respect to execution order much more predictable.

A common approach to handling the problem of having several potential execution orders for the same test case is to execute the same test suite several times to get statistical confidence for the result. However, this approach works better when testing for efficiency (average response time) than when testing for timeliness (worst response time). When running a test for timeliness, we want to increase confidence that a deadline will be met under **all** circumstances. The average response time is of no concern for timeliness [15]. From a tester's point of view, it is the worst cases we want to execute, by for example, executing with overload or reduced capacity. Focusing on the worst cases and adverse circumstances distinguishes timeliness testing from functional testing, which usually focuses on test cases representative for an operational profile.

As the number of potential execution orders increases for test cases included in the test suite, it becomes harder to gain sufficient confidence for timeliness with the statistical approach. Therefore, we consider the number of

potential execution orders to be a reasonable metric for estimation of timeliness testability. This metric of timeliness testability is used in this study. This metric goes in line with previous work [13].

## 2.1 Previous work

Kopetz [9] points out that testability of a real-time system depends on the architecture and must be considered during the design phase [9]. The most complete work concerning testability of distributed real-time systems was by Schütz [13]. Schütz presents a formula that expresses an upper bound on the number of control paths for a time-triggered system. Furthermore, Schütz shows how the formula applies to dynamic event-triggered systems.

An upper bound on test effort for dynamic RTSs was defined by Mellin [12]. Mellin presents a formula that expresses the upper bound for test effort in a dynamic RTS. The formula is further refined to allow for more than one shared resource [2]. The current paper suggests that some of the properties from time-triggered static systems, such as sparse time base, in combination with designated preemption points, and a maximum number of concurrently executing tasks, should be adopted in the dynamic design. The result is still a dynamic, event-triggered system but with a level of testability that approaches time-triggered systems. The formula expresses the upper bound on the test effort as a function of a set of execution environment constraints (see Equations 1 through 4).

$$FSTAT = ESTAT * BSTAT * PSTAT \quad (1)$$

$$\begin{aligned} ESTAT(s, n) &= \sum_{k=0}^n \binom{n}{k} s^k \\ &= \sum_{k=0}^n \binom{n}{k} s^k 1^{n-k} \\ &= (s + 1)^n \end{aligned} \quad (2)$$

$$\begin{aligned} PSTAT(p, q, t) &= \left( \sum_{k=0}^q (p + 1)^k \right)^t \\ &= \left( \frac{(p + 1)^{q+1} - 1}{p} \right)^t \\ &\text{where } p > 0 \end{aligned} \quad (3)$$

$$BSTAT(q, t, C) = \sum_{c \in C} \frac{\prod_{j=1}^r (q * t - \sum_{k=1}^{j-1} c_k)}{\prod_{n \in elem(c)} card(n, c)!} \quad (4)$$

1. *FSTAT*: The upper bound for the number of combinations of event sequences and states with respect to experienced preemptions and current blockings for each task.
2. *ESTAT*: The number of distinct event sequences for an interval with  $s$  observation points during the interval and  $n$  distinct events that may occur during the same interval.
3. *PSTAT*: The number of preemption states where  $q$  is the upper bound for concurrently executing tasks of the same type,  $t$  is the number of task types, and  $p$  is the upper bound on the number of preemptions a task can experience during execution.
4. *BSTAT*: The number of distinct blocking states. As in *PSTAT*,  $q$  is the upper bound for concurrently executing tasks and  $p$  is the upper bound on the number of preemptions.  $C$  is a bag of all possible blocking scenarios. A blocking scenario  $c_i$  is defined by the number of resources that any task is blocked on and the number of tasks that are blocked on the resource. An example of a blocking scenario is [5, 3, 3], where tasks are blocked at three different resources and the numbers represent the number of blocked tasks for each resource.

The next section describes an experiment to evaluate these formulae.

## 3 Experimental Setup

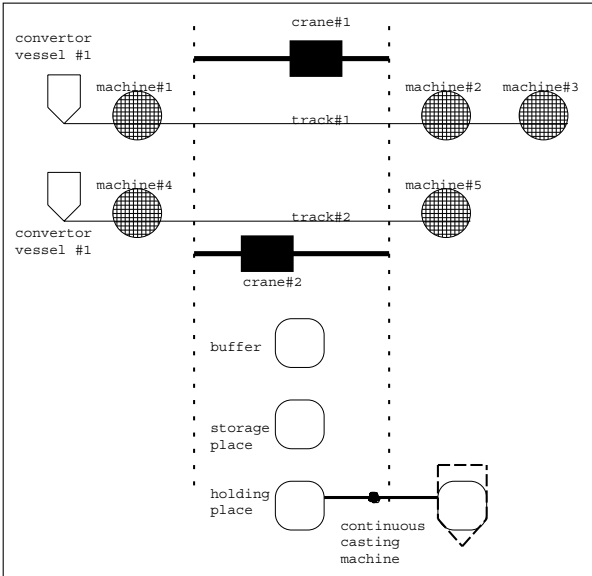
The basic idea for this investigation is to model a dynamic RTS with its execution environment, vary the investigated constraints, and study the effect on the number of potential execution orders. In this section, we describe how this experiment was conducted.

### 3.1 Purpose of experiment

The purpose of our work is to evaluate the previous theoretical work. The equations provide appealing models, but they must be verified empirically to be useful. Specifically, the experiment tries to answer three questions: (i) do the proposed constraints (the values of  $s$ ,  $q$  and  $p$ ) affect testability in the same way as suggested by the formulae, (ii) does the formula give a true upper bound for testability, and (iii) is the formula an appropriate approximation of testability as we measure it (is the bound sufficiently tight).

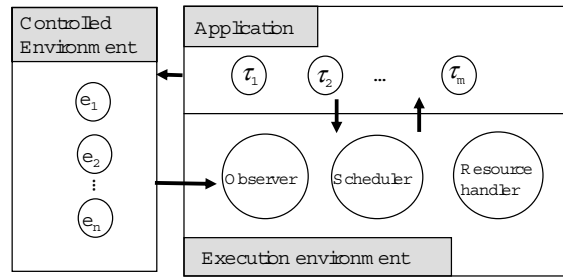
### 3.2 A real-time system model

The subject of our study is a control application for a steel plant SIDMAR. The steel plant has been previously described in detail and used in several studies [1, 4, 5, 6]. The plant consists of two cranes, two converter vessels, five machines, two normal tracks, a buffer, a storage place, a holding place, and a continuous casting machine. Figure 1 shows how these components interact.



**Figure 1. Layout of the steel production plant. The figure is taken from Fehnker [6].**

- **Converter vessels:** This is where the pig iron (raw cast iron) enters the system. It is poured portion-wise into steel ladles.
- **Tracks:** The ladles can move autonomously along two normal tracks. Moving from one track to another requires a crane.
- **Cranes:** Two cranes are available to move the ladles between tracks, the buffer, the storage place and the holding place. The crane is also needed for moving empty ladles from the casting machine to the storage place.
- **Machines:** SIDMAR has five machines of three different types. Machines #1 and #4 are identical as are machines #2 and #5. The quality of the steel depends on the order with which the machines treat the iron.



**Figure 2. The modeled system.**

- **Continuous casting machine:** This is where the steel leaves the system. The casting machine consists of two parts, a holding place and the casting machine itself. The casting machine works as a merry go round. An empty ladle can only leave the casting machine when the holding place has a full ladle.
- **Buffer:** The buffer can hold at most five ladles and can be used to pass ladles between the cranes.
- **Storage place:** Empty ladles are placed on the storage place. An empty ladle can only be transported to the storage place if the holding place contains a full ladle. Moving an empty ladle from the casting machine requires a crane.

We model the control application for the steel plant as a dynamic RTS. This means that we identify a set of triggering events and task types that respond to such events. Each task is assumed to be executed by a process. A subset of these tasks, with their natural dependencies with respect to timing and shared resources, is used in the study.

We have developed a formal model of the system in timed automata (TA). The model, summarized in figure 2, consists of three parts: (1) the application, (2) a controlled environment, and (3) an execution environment.

1. **Execution environment:** The execution environment contains three TA processes that model a scheduler, a resource handler, and an observer.
  - **Scheduler:** Scheduling is dynamic and the schedule is recalculated whenever a task is triggered, blocked, finished or releases a resource. The schedule uses the earliest deadline first policy, EDF.
  - **Resource handler:** Resource handling is dynamic and based on the first in first out (FIFO)

policy. Hence, this resource handler implements the same behavior as a set of FIFO semaphores, one for each shared resource.

- Observer: The role of the observer is to observe events in the controlled environment and communicate observations to the scheduler. The observer ensures that event occurrences are observed by the system at the next observation point.
2. Controlled environment: The controlled environment contains a set of TA processes,  $e_1 \dots e_n$ . Each TA process simulates a triggering event from a sensor signal.
  3. Application: The application contains a set of TA processes,  $\tau_1 \dots \tau_m$  where each TA process simulates execution of a task. On a signal from the observer, the TA process moves from an idle state to a ready state where it stays until it is first in the ready queue.

The system model is varied with respect to investigated constraints. Testability is estimated by counting the potential execution orders in each variant with the model checker UPPAAL [10].

### 3.3 Independent variables

The variables that this study varies are three parameters in the formulae for upper bound on test effort, Equations 1 through 4. Each parameter is a property of the constrained execution environment. The parameters are:

1. Number of observations during the observed interval, called *observePoints* in the text,  $s$  in the formulae.
  - The length of the observed interval,  $g_a$ , is based on the maximum response time. An event that occurs at time  $t$  should have no impact on the behavior at time  $t + g_a$  since any task it might have triggered no longer remains in the system.
  - The number of observations, *observePoints* ( $s$ ), during the observed interval is  $\frac{g_a}{g_o}$  where  $g_o$  is the time granularity of the observations.
  - *observePoints* is varied by varying the granularity of the sparse time base used by the observer in the TA system.
2. Maximum number of concurrently executing tasks of the same type, called *maxTasks* in the text,  $q$  in the formulae.

- Tasks are triggered as a response to events in the controlled environment.
  - Triggering of the task is delayed until the next observation point by the observer.
  - Triggering of the task may be further delayed until less than maximum number of tasks are executing.
  - *maxTasks* is controlled by the number of TA processes specified in the TA system
3. Maximum number of preemptions a task may encounter, called *maxPreempts* in the text,  $p$  in the formulae.
    - Tasks are executing in non-preempted intervals between designated preemption points.
    - A task that has encountered the maximum number of preemptions completes execution in non-preemptive mode.
    - *maxPreempts* is implemented as a constant in the TA system.

A *test scenario* in this context is a series of steel ladles entering the system on the conveyor belt at machine #1 or #4 (see Figure 1). A test scenario specifies the number of arriving steel ladles and a small span of time for each arrival. The reason to have a span of time rather than an exact point in time is that we want to mimic a test situation where the exact point in time is hard to control.

As a steel ladle is being processed and moves between different machines, it generates a series of sensor signals. Timers keep track of delays between these signals. For example, it takes an amount of time to move from one sensor to another. Again, we model this as spans of time rather than exact points in time. For example, it does not take exactly  $x$  time units to move from position A to B, it takes between  $x-1$  and  $x+1$  time units depending on, for example, the speed of a belt or crane.

The choice of test scenarios in our study is a trade-off between our intention to stress the system with a burst of events and the limitation imposed from model checking. We therefore, selected test scenarios that were as stressing as could be handled by the model checker. The underlying assumption is that if a significant impact is shown for these scenarios, then this impact will be at least as significant for a test scenario where the event burst is worse.

Four scenarios were specified and for each such test scenario, the values of the variables *observePoints*, *maxTasks*, and *maxPreempts* are varied. Each variation results in a TA model, referred to as *model variants*.

### 3.4 Execution

Each scenario was included in a model and the set of model variants were specified for all scenarios. The models in a set are identical except for the values of the independent variables *observePoints*, *maxTasks*, and *maxPreempts*. Hence, differences in the number of potential execution orders depend on the different values of these controlled parameters. Each model variant is fed to a model checker used to determine potential execution orders.

Finding the complete set of execution orders in a large model is difficult for two reasons. A model checker typically determines a single trace and the state-space exploration algorithm can use significant memory. The *state-space explosion problem* refers to the exponential size of state space with respect to the size of the input model [7]. We therefore developed a tool to derive all execution orders that a model may generate.

The tool repeatedly invokes the model checker until all execution orders in the currently explored model are found. The tool also mitigates the problem with memory consumption by using a guide automaton. This technique reduces the memory needed for each invocation of the model checker. This method is further described in our previous paper [3].

## 4 Number of Observation Points

This section presents the results concerning the effect of the number of observation points (*observePoints*, or parameter *s* in Formulae 2) on testability. Four different scenarios were used and the value of *observePoints* is varied from 500 to 2500. Figure 3 shows the result from three scenarios. The fourth scenario is excluded from the figure for presentation reasons<sup>1</sup>. The results show that the number of execution orders grows with a constant rate for all scenarios. The fourth scenario gives a higher number of execution orders, which is explained by the fact that this scenario contains one more input event compared to the other scenarios.

The results suggest that the effect parameter *observePoints* has on testability is linear rather than exponential, which is suggested by previous work [2, 13]. Formula 2 gives the upper bound on the number of potential event sequences. This means that the input domain grows exponentially with *observePoints*. However, unless we intend to perform exhaustive testing, the number of test cases that we need to execute does not necessarily grow as rapidly as the size of the input domain. Moreover, a

<sup>1</sup>The result from the fourth scenario shows a linear growth when *observePoints* is increased but the number of orders is approximately one magnitude higher for the fourth scenario.

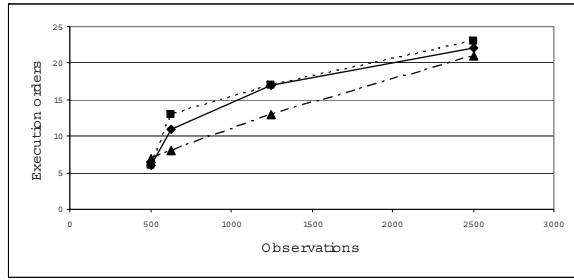


Figure 3. Effects of the number of observation points.

large input domain does not necessarily imply low testability since testability is more a question of how difficult it is to define, execute and analyze the tests. Our results show that the larger *observePoints* is, the more the predictability of the dynamic system is affected. Hence, it gets harder to control the execution of a selected test case and the confidence gained from test execution decreases because the same test may have different behaviors in different executions.

## 5 Concurrency

This section presents the results concerning the effect of the number of concurrently executing tasks of the same type (*maxTasks*, or parameter *q* in Formulae 3 and 4) on testability. The same four scenarios used to investigate the impact from observation granularity were now checked to investigate the impact from concurrency. Parameter *maxTasks* was varied between 1 and 3. The experiments found no impact from this parameter. Several explanations for this are possible:

- The selected scenarios might not be appropriate to provoke multiple execution orders.
- The selected policies for scheduling and resource handling might not be appropriate for investigation of this parameter.
- The parameters have little or no impact on testability.

It is hard to argue that the parameter has no effect on testability just because we could not find a scenario where the impact was shown. However, there are intuitive reasons to believe that this parameter is less interesting from a testability perspective:

- Our TA model uses an EDF scheduling policy. A real system might use another policy but a dynamic

RTS bases its scheduling decisions on the tightness of the deadlines and/or the importance (criticality) of the tasks.

- Importance of a task is generally decided by the task type and its associated value function (i.e., the penalty of a missed deadline).
- Deadline of a task is decided by the type of triggering event and when the event was observed by the system.
- Two events of the same type cannot be observed at the same point in time according to the underlying assumptions for Formula 2 [2].

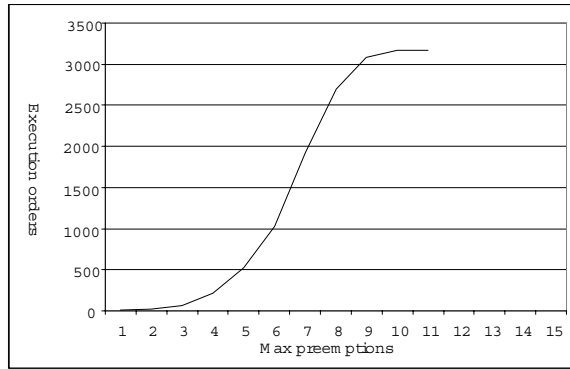
Hence, if several concurrently executing tasks of the same type are present, they have the same criticality but different deadlines and therefore a predictable priority order. This leads to a predictable execution order since the task with the higher priority will execute first. It might be possible that a locking protocol together with specific blocking scenarios can provoke an execution where a task preempts a previously triggered task of the same type. However, locking protocols, such as the Stack resource protocol [16], are predictable and are therefore likely to have little impact on the number of execution orders. Investigation of the impact on testability from such protocols is therefore left for future work.

## 6 Preemptions

This section describes the results concerning the effect of the number of allowed preemptions (*maxPreempts*, or parameter  $p$  in Formulae 3) on testability. We investigated the scenarios used in previous experiments and found little impact. The reason was that in these scenarios, with their small number of events, there were very few preemptions even when we allowed an unlimited number of them. In order to study the effect of the constraint we needed a scenario with a potentially high number of preemptions, e.g., an emergency situation with alarm signals. The scenarios used previously only included the normal arrival of a few steel ladles and this could not stress the system enough to study this parameter.

Three different scenarios were selected. The scenarios included bursts of high-priority interrupts leading to preemptions and possible race conditions on both CPU and other shared resources and can therefore, be considered as worst cases. It is, however, worth noticing that the limitation of model checking never let us try any scenario that was near a real worst case situation.

All that we can do is try to provoke as bad a case as possible. Again, the assumption is that the impact shown on the number of execution orders will not decrease if we add more events to the event burst. These scenarios are particularly hard to predict with respect to the dynamic schedule. This makes such scenarios interesting test cases from a timeliness perspective. The results of varying the maximum number of preemptions in such scenarios is shown in Figure 4. Only one scenario is shown for presentation reasons<sup>2</sup>.



**Figure 4. Result of varying the maximum number of preemptions.**

The maximum number of preemptions is varied from 1 to 11 and as shown in Figure 4, the number of execution orders increases exponentially. There is a point of saturation for each scenario. Beyond this point, an increase of *maxPreempts* has no effect on the execution orders for the scenario. The saturation occurs when *maxPreempts* approaches a limit on the number of preemptions that a task in the particular scenario can experience if *maxPreempts* is unbounded.

The results show that the maximum number of preemptions has a significant impact on the number of execution orders. The exponential growth of the number of execution orders support the theory from previous work [2, 11] and suggests that an upper bound on *maxPreempts* is a good candidate for increasing testability.

## 7 Results

Section 3.1 introduced three questions to be addressed by the experiment. The first question is whether *observePoints*, *maxTasks* and *maxPreempts* affect testability in the same way as the formulae in section 2 sug-

<sup>2</sup>The other two scenarios also showed an exponential growth followed by a saturation. However, the magnitude differ and the saturation occurred at different points.

gest. The experimental results indicate that *observePoints* and *maxPreempts* do affect testability while *maxTasks* has no effect at all, given the policies selected for scheduling and resource handling in the experimental setting. The effect on testability from *maxPreempts* is exponential, which agrees with the formulae. The effect on testability from *observePoints* is linear in the experimental data rather than exponential. The exponential impact from *observePoints* suggested in previous work, [2, 13], refers to the size of the input domain. This experiment ignores the size of the input domain and focuses on the effect on the execution orders. The results show that there is an impact on testability from *observePoints* other than the sole size of the input domain.

The second question is whether the formulae give a true upper bound on testability. The data in this experiment agree with this statement.

The third question is whether the formulae give an appropriate approximation of testability. The data indicate a negative answer to this question. Given our view of testability, the effect from *observePoints* is much lower than suggested by the formulae. Moreover, our results indicate that there is no effect on testability from *maxTasks*.

## 8 Conclusions

In this work, we have assumed that exhaustive testing is not an option and that the size of the input domain has less impact on testing than other factors such as controllability and observability. A dynamic RTS reacts to events in the environment by online decisions about execution and schedule. Due to elements that are not controlled in such systems, the behavior with respect to timing and execution order is less predictable in a dynamic RTS than in a corresponding static RTS. We have therefore chosen the number of execution orders as a reasonable approximation of testability when timeliness testing is our goal. This approximation assigns the highest level of testability to the time-triggered design of static RTS, where there is only one potential execution order for each input sequence.

The load a tester would use to stress the system under test is higher than could possibly be used in a study like this. This is, of course, due to the state-space explosion problem. We used long running tasks and short interrupts but the load was never higher than twelve concurrently executing tasks. It is always possible to question whether this load is representative or not. However, the load was high enough to allow us to observe the effect on testability. This effect is likely to be even higher when the system under test is undergoing tests for timeliness.

## 8.1 Contribution

We have investigated the impact on testability from a set of execution environment constraints. The constraints are proposed to increase testability in dynamic RTSs. We have shown that two of the proposed constraints, designated preemption points and observation points, are good candidates for increasing testability in dynamic RTSs. The third constraint, concurrently executing tasks of the same task type, showed no impact on testability. The results show that the formulae for an upper bound on test effort of dynamic RTSs (Equations 1 through 4) are not adequate to be used as an approximation for testability.

## 8.2 Future work

One thing that needs to be investigated is whether these results remain when we change the policies for scheduling and/or resource handling. This study handled resources by emulating simple FIFO semaphores. It would help to repeat the experiments using a protocol that consider the task priorities when allocating shared resources.

The results show that the formulae for an upper bound on test effort is not sufficiently tight to be used as an approximation of testability. It is therefore necessary to find a closer bound. It is possible that an approximation is more useful to the designer than a true upper bound.

## References

- [1] G. Behrmann, T. Hune, and F. Vaandrager. Distributed timed model checking - How the search order matters. In *Proc. of 12th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Chicago, Juli 2000. Springer-Verlag.
- [2] R. Birgisson, J. Mellin, and S. Andler. Bounds on Test Effort for Event-Triggered Real-Time Systems. In *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, December 1999.
- [3] B.Lindström, P.Pettersson, and J.Offutt. Generating Trace-Sets for Model-based Testing. In *Proceedings of the 18th International Symposium on Software Reliability Engineering, ISSRE'07*, pages 171–180, Trollhättan, November 2007.
- [4] R. Boel. Automatic synthesis of schedules in a timed discrete event plant. In *Proceedings of ADPM2000*, 2000.
- [5] R. Boel and F. J. Montoya. Modular Synthesis of Efficient Schedules in a Timed Discrete Event Plant. In *Proceedings of the 39th IEEE Conference on Decision and Control*, volume 1, pages 16–21, December 2000.
- [6] A. Fehnker. Scheduling a steel plant with timed automata. In *Real-Time Computing Systems and Applications (RTCSA)*, pages 280–286, 1999.

- [7] G.J.Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
- [8] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*, 1990.
- [9] H. Kopetz. The Design of Real-Time Systems. *Software Engineering Journal*, 6(3):72–82, 1991.
- [10] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.
- [11] B. Lindström, J. Mellin, and S. Andler. Testability of Dynamic Real-Time Systems. In *Proceedings of Eighth International Conference on Real-Time Computing Systems and Applications (RTCSA2002)*, pages 93–97, Tokyo, Japan, March 2002.
- [12] J. Mellin. Supporting system-level testing of applications by active real-time database systems. In *Proceedings of the 2nd International Workshop on Active, Real-Time, and Temporal Database Systems*, volume 1553 of *Lecture Notes in Computer Science*, pages 194–211. Springer, 1998.
- [13] W. Schütz. *The Testability of Distributed Real-Time Systems*. Kluwer Academic Publishers, 1993.
- [14] I. Standard ISO/IEC 9126. Information technology – Software product evaluation – Quality characteristics and guidelines for their use. International Organization for Standardization, International Electrotechnical Commission, Geneva, 1991.
- [15] J. A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10):10–19, Oct. 1988.
- [16] T.P.Baker. Stack-Based Scheduling of Realtime Processes. *The Journal of Real-Time Systems*, 3:67–99, 1991.
- [17] J. M. Voas and K. W. Miller. Software Testability: The New Verification. *IEEE Software*, 12(3):17–28, May 1995.