Linköping Studies in Science and Technology

Dissertation No. 1241

# Testability of Dynamic Real-Time Systems

by

**Birgitta Lindström**

Department of Computer and Information Science
Linköping universitet
SE-581 83 Linköping, Sweden

Linköping 2009

ISBN: 978-91-7393-695-8
ISSN: 0345-7524

Printed by LiU-tryck, Linköping 2009

*In loving memory of my daughter Sofia who supported and encouraged my studies but never got the chance to see the work finished*

# Abstract

This dissertation concerns testability of event-triggered real-time systems. Real-time systems are known to be hard to test because they are required to function correct both with respect to what the system does and when it does it. An event-triggered real-time system is directly controlled by the events that occur in the environment, as opposed to a time-triggered system, which behavior with respect to when the system does something is constrained, and therefore more predictable. The focus in this dissertation is the behavior in the time domain and it is shown how testability is affected by some factors when the system is tested for timeliness.

This dissertation presents a survey of research that focuses on software testability and testability of real-time systems. The survey motivates both the view of testability taken in this dissertation and the metric that is chosen to measure testability in an experiment. We define a method to generate sets of traces from a model by using a meta algorithm on top of a model checker. Defining such a method is a necessary step to perform the experiment. However, the trace sets generated by this method can also be used by test strategies that are based on orderings, for example execution orders.

An experimental study is presented in detail. The experiment investigates how testability of an event-triggered real-time system is affected by some constraining properties of the execution environment. The experiment investigates the effect on testability from three different constraints regarding preemptions, observations and process instances. All of these constraints were claimed in previous work to be significant factors for the level of testability. Our results support the claim for the first two of the constraints while the third constraint shows no impact on the level of testability.

Finally, this dissertation discusses the effect on the event-triggered semantics when the constraints are applied on the execution environment. The result from this discussion is that the first two constraints do not change the semantics while the third one does. This result indicates that a constraint on the number of process instances might be less useful for some event-triggered real-time systems.

**Keywords:** Testability, Software testing, Real-time systems, Timeliness, Model-based testing.

# Sammanfattning

Denna avhandling handlar om testbarhet hos händelsestyrda realtidssystem. Realtidssystem är erkänt svåra att testa eftersom dessa system har krav på sig att fungera korrekt både med avseende på vad systemet gör och när det gör det. Händelsestyrda realtidssystem styrs direkt av de händelser som inträffar i omgivningen till skillnad från tidsstyrda system vars beteende med avseende på när systemet gör något är hårt kontrollerat och därmed mer förutsägbart. I den här avhandlingen så är det just tidsaspekten som står i fokus och vi visar hur testbarheten påverkas av några olika faktorer då systemet ska testas för punktlighet.

Avhandlingen innehåller en översikt över den forskning som fokuserar på testbarhet hos programvara och realtidssystem. Översikten ligger till grund för hur avhandlingen valt att mäta testbarhet i ett experiment. Avhandlingen presenterar även en metod för att generera mängder med spår i en modell med hjälp av en metaalgoritm som arbetar mot en model checker. Metoden är nödvändig för att genomföra experimentet men de spår som genereras kan även användas för testmetoder där man fokuserar på ordningar, exempelvis exekveringsordningar.

Avhandlingen redovisar ett experiment av hur testbarheten hos händelse-styrda realtidssystem påverkas av att man inför vissa egenskaper hos exekveringsmiljön. Begränsande egenskaper som påminner om dem som finns hos tidsstyrda realtidssystem men som inte anses förändra den händelsestyrda semantiken. Experimentet omfattar tre sådana egenskaper som rör exekveringsavbrott, observationer och antal instanser av samma processtyp. Dessa egenskaper har tidigare pekats ut som avgörande faktorer för testbarheten. Resultaten visar ett stöd för detta vad gäller de första två egenskaperna medan den tredje egenskapen inte alls tycks påverka testbarheten.

Slutligen redovisar avhandlingen hur den händelsestyrda semantiken påverkas då man inför de föreslagna egenskaperna hos exekveringsmiljön. Slutsatsen från denna diskussion är att två av egenskaperna inte förändrar semantiken medan en av dem har en påverkan på semantiken som innebär att den egenskapen kan vara mindre lämplig att införa hos vissa händelsestyrda realtidssystem.

# Acknowledgements

A project of this size is not possible to pursue without support and encouragement from other people. There is a large number of people that I owe my gratitude. Without them this thesis would never have been written.

- First and foremost I want to thank my family for their loving support and patience; my husband Börje and my daughters Linnéa and Maria. I want to thank you for all your understanding when the thesis seemed to occupy all of my time.

- My advisors; Sten, Jeff, Paul, at an early stage, Jonas and, at the end, Zebo. Thank you for all the feedback, your patience during long discussions and for believing in me when the thesis work seemed to go on forever.

- All of you that have read the thesis and provided me with useful feedback; Ammi, Bengt, Lionel, and committee members. Without you, my work would have been a lot harder.

- My co-authors on some of the papers included; Mats and Robert. Thank you for interesting discussions and for making my work a lot more fun.

- My current and past colleagues in the research group; Sanny, Gunnar, Marcus, Alexander, Ronnie, Jörgen and Joakim. Thank you all.

- A special thanks to Betty and Björn who supported me and my family when we needed it most.

- Finally, I want to thank all of you wonderful people that have shown me your friendly support during these years; my mother Gudrun and Alfred, my brother Bosse and Meri, my sister Margareta and Paul, Anders and Helena, Håkan, Marianne, friends and colleagues. Life would be a lot more empty without you.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Introduction

Software systems today tend to be more sophisticated and complex. A consequence of this is higher demands on the software industry to cope with the increased complexity on all levels in the development process. This is especially true for verification activities. As system complexity increases, the act of verification becomes more difficult. It is therefore necessary to identify ways to understand and control this complexity to build safe and reliable systems. This is a problem in many dimensions. There is a need for better test techniques that help testers identify the most efficient and effective test suites, i.e., test suites that reveal faults at a cost the developers can afford. Software testing has matured during the last decade but there are areas where there is a lack of good techniques. One such area is concurrent systems and especially real-time systems. There is a need for better tools that can control the test execution so that exactly the test cases are executed that are intended to be executed. There is a need for tools that allow us to observe the test execution and thereby recognize erroneous behavior. Finally, system testability should be considered already during the design phase. To do that more information is needed together with a better knowledge about system testability and the effect that different design choices may have on it.

This work considers relationships between system design and system testability. If it is possible to design systems with higher testability, the effort to test the system can decrease and testers can perform better testing in a structured way. In turn, better testing can help developers build higher quality systems. There are two reasons for this. The first reason is that with better testability testers and developers can find and remove more faults. This will of course have an effect on quality. The second reason is that structured testing, where tests are controlled and observed, can help developers gain a better understanding of the system. The better the understanding of the system is the better is the chance to improve its quality.

The focus in this dissertation lies on dynamic, event-triggered real-time systems, which are known to be inherently harder to test than corresponding time-triggered real-time systems (Schütz 1993). The time-triggered design gives good support for testability and is therefore often preferred. There are, however, situations when an event-triggered design is preferred or even necessary. In these situations, the designers have little information about how testability can be supported by the system design. Still, such systems have to be tested and testability will have a significant impact on both the cost of testing and the resulting level of quality.

This work investigates a set of design choices to determine what impact they have on the level of testability in event-triggered systems. The specific design choices are in the form of constraints on the execution environment.

The relation between the system testability and the execution environment is considered to be mutual because constraints on the execution environment may lead to improved testability and requirements on high testability may lead to constraints on the execution environment.

This dissertation discusses testability in the context of testing for timeliness on a system level. Constraints on the execution environment are selected and their impact on testability is investigated. The choice of constraints is based on previous work by Mellin (1998) and Birgisson, Mellin & Andler (1999) that define an upper bound on test effort for event-triggered real-time systems. The goal with this dissertation is to determine whether applying the proposed set of constraints on the execution environment increases testability in event-triggered real-time systems, as proposed in their work. This dissertation includes an empirical study (see Chapter 6) where the experimental goal is to see whether the results from this study support previous work on the relation between the selected execution environment constraints and system testability. Our results indicate that some of the constraints affect testability while others have no effect at all. A method for trace-set generation is also defined in this dissertation (see Chapter 5). The method is a necessary part of the study but can also be used for model-based testing and opens up for new testing criteria suitable for concurrent systems.

## 1.1.1 List of Publications Included in Thesis

This dissertation is based on the papers below. With each paper is an explanation of which parts of the paper are included in what chapters of this dissertation. Unless specified, co-authors have had the role of advisors.

1. B.Lindström, A. J. Offutt and S. F. Andler. Testability of dynamic real-time systems: An Empirical Study of Execution environment implications, In *Proceedings of The 1st IEEE International Conference on Software Testing, Verification and Validation (ICST), pages 112-120, Lillehammer, Norway, April 2008.*
   This paper describes an empirical study that explores the effect on testability when varying some parameter settings in the execution environment. The major parts in this paper, including experimental set-up, implementation, execution and analysis, are included in Chapter 6. Conclusions and related work from the paper are part of Chapter 8 in this dissertation.

2. B. Lindström, R. Nilsson, M. Grindal, A. Ericsson, S. F. Andler, B. Eftring, and A. J. Offutt. Six Issues in Testing Event-Triggered Systems, Technical report HS-IKI-TR-07-005, University of Skövde, 2007.
   This is a joint paper for the TETReS research group and it contains a list of issues that are recognized to be harder when testing dynamic systems. Part of the background and discussed issues are included in Chapter 2. Part of the described approach (Improving Testability) is included in Chapter 3. Birgitta wrote about real-time systems in the background section and several parts in the result section (including Issues in real-time systems, Monitor execution and control and Design trade-offs). These are the parts that are included in the thesis.

3. B. Lindström, P. Pettersson and J. Offutt, Generating Trace-Sets for Model-based Testing. In *Proceedings of The 18th IEEE International Symposium on Software Reliability Engineering (ISSRE'07)*, pages 171-180, Trollhättan, Sweden, November

2007.
This paper presents a method for generating sets of traces when model-checking rather than single traces. The major parts in this article, from the background to the results, are included in Chapter 5 in this dissertation, while the discussion and related work are included in Chapter 8. Finally, part of the problem description and motivation are included in Chapter 4. Paul contributed the description of timed automata and substantial feedback on the rest of the article.

4. B. Lindström and P. Pettersson, Model-Checking with Insufficient Memory Resources, Technical report HS-IKI-TR-06-005, University of Skövde, 2006.
   This paper presents a method that dynamically divides a state-space into partitions during model-checking. As dynamic real-time systems are prone to the state space explosion problem, this method divides a problem into sub-problems that can be solved independently, thereby mitigating the state space explosion. The major parts from this paper, all sections from the introduction to the results, are discussed in Chapter 5 while the included case study is discussed in Chapter 6. Related work is discussed in Chapter 8. Paul contributed the description of timed automata and substantial feedback on the rest of the article.

5. B. Lindström, and J. Mellin, Work in Progress: Testability Experiments, In *Proceedings of Real Time in Sweden 2005 (RTiS 2005)*, Special Session on Testing of Event-Triggered Real-Time Systems, pp. 101-106, 2005.
   This paper presents the method and some preliminary results of the work described in this dissertation. Background and previous work are discussed in Chapter 3 in this dissertation while the approach and results are discussed in Chapter 6.

6. B. Lindström. System testability and the execution environment. Thesis proposal, University of Skövde, Sweden, 2003.
   This work was presented to the Department of Computer Science at University of Skövde. The proposal describes and motivates the research problem. The background is included in Chapter 2. Parts of the previous work and the problem description in the paper are described in Chapter 3.

7. B. Lindström, J. Mellin, and S. F. Andler. Testability of Dynamic Real-Time Systems. In *Proceedings of Eight International Conference on Real-Time Computing Systems and Applications (RTCSA2002)*, pages 93-97, Tokyo, Japan, March 2002.
   This paper focuses on the motivation and underlying theory for this dissertation. The introduction and discussion about system testability are discussed in Chapter 2 in this dissertation while the theory and the discussion about the implications from the execution environment on testability is discussed in Chapters 3 and 6. Finally, the sections concerning conclusions, related work, contribution and future work are discussed in Chapter 8.

### 1.1.2 Thesis Overview

The dissertation is arranged as follows.

The rest of this chapter gives the necessary background for this thesis. Section 2.1 gives an overview of software testing. Section 2.2 describes two different types of design for real-time systems, the (dynamic) event-triggered and the (static) time-triggered design. Section 2.3 discusses efficient and effective testing.

Chapter 3 discusses previous work, which forms a basis for this thesis and motivates the research problem. The problem statement is also given here. Previous work is described in Section 3.1 and the problem definition is given in Section 3.2.

Chapter 4 contains a survey of testability research and an elaborated discussion about the author's view on testability in real-time systems and how it can be estimated. Chapter 5 gives a method that uses model-checking to generate trace-sets instead of single traces while, at the same time, mitigating the state space explosion problem. Chapter 6 describes the impact on testability from the execution environment. Chapter 7 contains a discussion of the effect on the dynamic, event-triggered semantics the constraints may give.

Finally, Chapter 8 contains discussion, related work and conclusions.

# Chapter 2

# Background

This chapter introduces the concepts that are used throughout this thesis and presents a background for the thesis work. This Chapter is based on material presented in the background sections in Papers 2, 7 and 6. Section 2.1 discusses software testing. Section 2.2 presents two different design paradigms for real-time systems and the trade-off decisions that has to be made by the designer. Finally, Section 2.3 discusses efficient and effective testing.

## 2.1 Software Testing

Verification is an important activity in all development processes. Software development is no exception from this fact. It is widely accepted that verification activities often takes approximately 50% of the development resources (Myers 1979, Beizer 1990). The cost of verification goes up if the demands on the quality of service is high, e.g., for safety-critical systems. The purpose of verification is to gain sufficient confidence in the system behavior with respect to requirements and general software quality attributes such as reliability and safety (Avizienis, Laprie, Randell & Landwehr 2004). Laprie (1994) gives the following definitions:

**Definition 1.** *Verification: The process of determining whether a system adheres to properties (the verification conditions) which can be:*

*a) general, independent of the specification, or*

*b) specific, deduced from the specification.*

**Definition 2.** *Static verification: Verification conducted without exercising the system.*

**Definition 3.** *Dynamic verification: Verification involving exercising the system.*

**Definition 4.** *Testing: Dynamic verification performed with input values.*

These definitions are adopted in this dissertation and the focus in this dissertation is testing. Since testing is performed with input values, a central activity is to select which inputs to execute. It is therefore important to specify test requirements and select the test criteria to be used.

**Definition 5.** *A test requirement specifies one specific item that should be targeted during testing.*

**Definition 6.** *A test criterion specifies what tests should cover in terms of a class of test requirements.*

This means that the set of test requirements is defined by the test criterion. For example, the test criterion du-path coverage gives a set of test requirements where each individual requirement is a unique du-path. The relation between test criteria, test requirements and tests is described in the following definition from Ammann & Offutt (2008).

**Definition 7.** *Given a set of test requirements (TR) for coverage criterion C, a test set T satisfies C coverage if and only if for every test requirement tr in TR, there is at least one test t in T such that t satisfies tr.*

Software can be described by; (*i*) graphs representing the structure or data flow, (*ii*) logical expressions, e.g., decisions in a program, (*iii*) the input domain, and (*iv*) syntactic structures (Ammann & Offutt 2008). There is a variety of different test criteria that can be applied to each of these models.

Which criteria to choose depend on three things; (*i*) the type of faults that the tests aim to reveal, (*ii*) the requirements on the system with respect to failure intensity, and (*iii*) what the developer can afford. Although it can be more expensive not to test, the test budget imposes limitations to the choice of criteria since some criteria are more expensive than other.

Testing is performed at different levels and for different purposes. This dissertation uses the following definitions:

**Definition 8.** *Unit testing - A unit is the smallest testable piece of software. Unit testing is the testing we do to show that the unit does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure (Beizer 1990).*

**Definition 9.** *Module testing - A module is a collection of related units that are assembled in a file package or class. Module testing is designed to assess individual modules in isolation, including how the component units interact with each other and their associated data structures (Ammann & Offutt 2008).*

**Definition 10.** *Integration testing is designed to assess whether the interfaces between modules in a given subsystem have consistent assumptions and communicate correctly (Ammann & Offutt 2008).*

**Definition 11.** *System testing is concerned with issues and behaviors that can only be exposed by testing the entire system or major parts of it (Beizer 1990).*

Unit testing exercises the code on unit level e.g., a single procedure to assess the software with respect to the implementation. Module testing exercises a module, e.g., an object to assess the software with respect to detailed design. Integration testing exercises a set of modules e.g., a subsystem to assess software with respect to subsystem design. System testing assess software with respect to architectural design. System testing includes testing for performance, security, accountability, configuration sensitivity, start-up, and recovery.

### 2.1.1 Testing for Timeliness

A real-time system is a system where the correctness depends on when the system takes an action as well as what the system does. Each real-time task must meet a set of time constraints on its activation and completion.

**Definition 12.** *Deadline - A time constraint on the response time of a task is called a deadline (Ramamritham 1995).*

**Definition 13.** *Timeliness - A system in which all timing constrains are met is timely (Ramamritham 1995).*

Although software testing has matured during the last decades, much remains to do in the area of testing real-time systems. Time constraints, concurrency and the fact that many of these systems are embedded are factors that together make it hard for a tester to apply structured testing techniques on real-time systems. Some approaches are described in Hessel, Larsen, Nielsen, Pettersson & Skou (2003), Nilsson, Offutt & Andler (2004), Garousi (2008), and Thane & Hansson (1999*b*).

Testing for timeliness in a real-time system aims to determine whether the defined constraints on the timely behavior of individual tasks will be met or not. Whether or not the behavior of the real-time system is timely depends not only on the software application but also the execution environment and the hardware. For example, consider a test case targeting response time for a certain event, e.g., the pressure getting too high in a chemical control system. The response time depends not only on the execution of the corresponding task, it also depends on the frequency with which the system checks the sensor, communication delays, scheduling, etc. Hence, testing for timeliness is preferably done on the target system.

### 2.1.2 Test Effort and Testability

**Definition 14.** *Test effort is the effort, in terms of time, money, staff and other resources needed to test the system.*

**Definition 15.** *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 (IEEE 1990)*

The test effort depends on many different things such as:

*i* Testability, which concerns properties of the test object. Low testability implies that the test object is hard to test. For example, low predictability can lead to a need for running the same tests several times to gain statistical confidence for the resulting behavior.

*ii* Test process, a bad process can increase the workload. For example, if test cases are not traceable, it is hard know whether they are still needed after a change to the system. This can lead to extra effort due to redundant or invalid test cases.

*iii* Test method, some methods are easier to automate than others and different methods result in different size of the test suites.

*iv* Skill and experience, obviously a trained tester can do the work with less effort than an unexperienced tester.

*v* Specifications, a specification that lacks clear information with respect to expected functionality or correct (and incorrect) behavior is not sufficiently supportive for testing activities.

Testing activities represent a significant part of the development costs. At the same time, it can be even more expensive not to test. The cost of providing software with low quality is sometimes hard to estimate. For critical systems, there is a direct penalty in terms of e.g., money, injuries or damage associated with failures. For non-critical products such as computer games it is harder to estimate the damage to a trademark or the economical consequences of unsatisfied customers.

The trade-off between the cost for high reliability and the penalty for low reliability is often a difficult dilemma. It should therefore be of primary interest to investigate every possibility to decrease the test effort without decreasing the test quality. In this work, focus is on the possibility to decrease the necessary effort of testing for timeliness at a system level. The basic idea is to investigate the effect of the design of the real-time system on testability since system testability has a significant impact on the effort to test the system.

## 2.2 Real-Time System Design Paradigms

Real-time systems typically interact with other [sub-] systems and processes in the physical world, i.e., the environment of the real-time system. For example, the environment of a real-time system that controls a robot arm may consist of items coming down a conveyor belt and messages from other robot control systems along the same production line. The real-time system observes the state of the environment, e.g., via sensor signals, and responds to the situation in a timely manner, e.g., via actuators.

Figure 2.1 depicts an overview of a real-time system. Real-time applications are often modeled as a set of tasks (pieces of sequential code) that compete for system resources (for example, processor-time, memory and semaphores). The response times of such

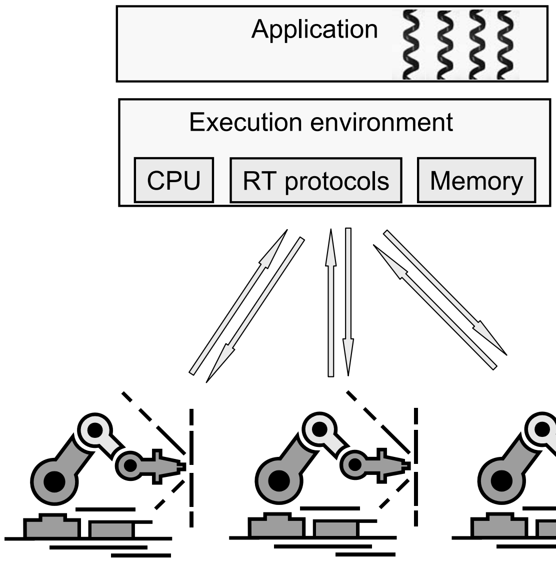


Figure 2.1: An overview of a real-time system with the controlled environment (i.e., robots), the application (i.e., real-time tasks), and the execution environment (i.e., processor, memory, and real-time protocols).

tasks depend on the order in which they are scheduled to execute. This, in turn, is controlled by real-time protocols (e.g., scheduling and concurrency control protocols) and properties of the execution environment (e.g., the real-time operating system, programming language and hardware).

The environment of a real-time system is observed periodically or in response to some triggering event. Tasks are usually periodic or sporadic.

**Definition 16.** *A periodic task is activated with fixed inter-arrival times, thus all the points in time when such tasks are activated are known.*

**Definition 17.** *A sporadic task is activated by events occurring in the environment, but assumptions about their activation patterns, such as minimum inter-arrival times, are used in analysis.*

Testing for timeliness typically tries to enforce the system to miss its deadlines. Therefore test cases include what the tester deems to be worst case situations that the system may have difficulty handling in a timely manner, e.g., bursts of sporadic events. This is further described in Garousi (2008).

The two main different ways to design real-time systems described in literature are the *time-triggered* and the *event-triggered* design (Kopetz 1991, Kopetz & Verissimo 1993).

**Definition 18.** *In the time-triggered approach all communication and processing activities are initiated periodically at predetermined points in time (Kopetz 1991).*

**Definition 19.** *In the event triggered approach all communication and processing activities are initiated whenever a significant change of state, is noticed (Kopetz 1991).*

In time-triggered systems a clock controls the execution. The system clock decides when to observe events, execute tasks, and deliver results. In event-triggered systems, the environment has control over the execution. Events are observed when they occur and a decision is made on how to react in response to the event. Such decisions are made dynamically by the system and are a major reason why these systems are so hard to test in comparison with time-triggered systems. The time-triggered design is usually preferred in hard critical systems where the consequence of missing a deadline may be catastrophic. The event-triggered design is usually preferred in less critical systems in which deadlines can be missed occasionally.

### 2.2.1 Time-Triggered Real-Time Systems

A pure time-triggered real-time system has a cyclic behavior. Clock interrupts trigger activities at predefined time points (see Figure 2.2). In time-triggered real-time systems, new inputs are observed at predefined, periodic, points in time, so called observation points. On a clock interrupt, the system reads all events that have occurred since the last observation point. Tasks that correspond to the occurred events are executed during the next time period in a pre-scheduled order. Computations scheduled in one period must finish before the next period starts regardless of which events have been observed. Execution time is assumed to be worst case and results are delivered at the next time point. This has several consequences:

*i* All tasks that can be executed in a period should be known beforehand

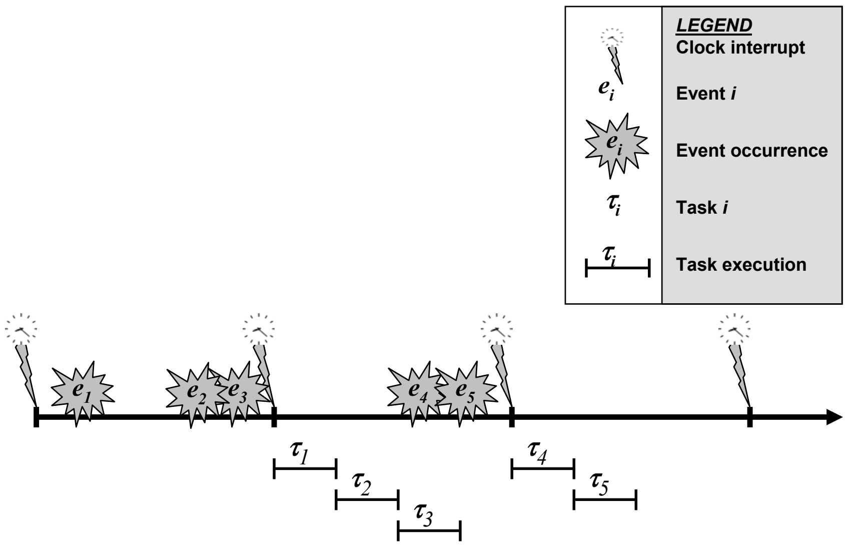


Figure 2.2: Event observation and task execution in a time-triggered system. On a clock interrupt, events are read and all triggered tasks finish their execution before next clock interrupt.

*ii* All resource requirements (e.g., execution time, communication, etc) should be known beforehand,

*iii* The observation granularity must be coarse enough to guarantee that all scheduled computations are finished before the next observation point,

*iv* All events occurring in the same interval are considered to have the same arrival time, and

*v* There is only one potential execution order for each possible input sequence

Together, these items explain why time-triggered systems have high testability. All tasks are assigned enough resources for their worst case execution times and the worst case situation is when all events occur at the same time. The predefined schedule guarantees that the behavior with respect to order and time is similar each time this situation occurs. A typical example of a time-triggered system is presented in Kopetz, Damm, Koza, Mulazzani, Schwabl, Senft & R.Zainlinger (1989).

The underlying assumption of resource adequacy in a time-triggered system makes changes and extensions difficult. Any unforeseen change to the system that increases the demand on resources (e.g., execution time or bandwidth) might necessitate a complete redesign of the system. Moreover, resource adequacy implies that the system is designed for the worst case demands. The static schedules are based on assumptions about worst cases that may lead to very low resource utilization. If the difference between worst case and average case with respect to resource demands is big, then a time-triggered solution will lead to low resource utilization. For example, assume a system with several sporadic events that seldom occur but have short, hard deadlines (such as alarm signals or sudden requests for evasive action). A time-triggered system would have to reserve resources and execute a periodic task that polls the environment frequently to be able to detect and respond to such event in a timely manner. Moreover, an unpredictable environment might require several operational modes. As the number of modes increases, so does the number of schedules and there is a potential risk for a combinatorial explosion. An event-triggered system, on the other hand, uses dynamic on-line scheduling and would only have to execute a sporadic task as a response to the event when it actually occurs. This leaves computation resources, as well as other resources, free to be used for other purposes. For these reasons, an event-triggered design is sometimes preferred.

### 2.2.2 Event-Triggered Real-Time Systems

In event-triggered systems, activities are triggered by events as they occur (see figure 2.3). The computer immediately reacts to an event by reconsidering the current schedule. A task corresponding to the occurred event is identified. A decision is made based on scheduling policy, current state of the system, resource requirements, and task priorities. The task may be dropped, scheduled for execution some time in the future, or started immediately by preempting the currently executing task. There are several consequences from this:

*i* Behavior is not cyclic

*ii* The schedule seldom repeats

*iii* There is no knowledge of future resource needs

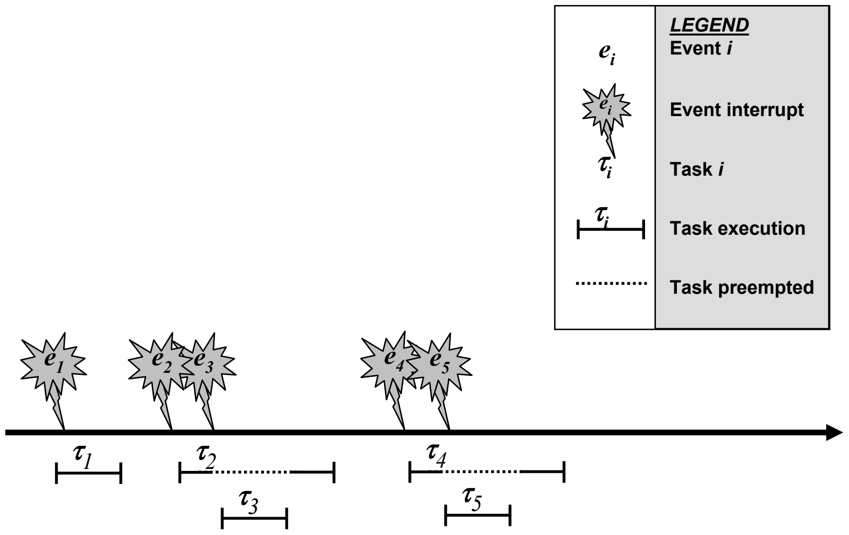


Figure 2.3: Event observation and task execution in an event-triggered system. Tasks are triggered when events occur and a new task may cause preemption of a current task.

*iv* There might be several potential execution orders for a single input sequence

Together, the above items explain why event-triggered systems have low testability. One reason is that the system does not await an observation point to read events or a communication point to deliver results. This means that the current state of the system, e.g., current schedule, blocked resources, program counter, etc., is part of the test case together with the event sequence. Moreover, small variations on e.g., execution time affect the behavior. Finally, predictability in the time domain is often low due to elements that are not controlled, e.g., the contents of a cache memory. Hence, it might be hard to repeat a test execution.

The priority of an individual task can be decided by its period, criticality or urgency. Results are delivered as soon as possible. This means that new tasks can be triggered during execution. Hence, dynamic scheduling and preemptions are necessary to guarantee timeliness of high priority tasks. Note that figures 2.2 and 2.3 give different behaviors although the event sequences are the same.

Event-triggered systems are flexible and can handle an unpredictable environment better than time-triggered systems because the event-triggered design does not require full knowledge of the resource demands. Hence, temporary overloads may occur and the event-triggered system cannot guarantee that all deadlines always will be met. Event-triggered systems are therefore often used for soft real-time systems or applications with a mixed task load where soft deadlines are tolerated to be missed occasionally under adverse circumstances. All hard deadlines must however be met and usually there is a limit for how long (or how often) a soft deadline can be missed. For example, garbage collection is usually a soft task as long as there is enough available memory. However, if the system runs out of memory, garbage collection becomes as critical as the most critical task in the system. The reason is of course that unless garbage collection is made, it is impossible to run any other task. A typical example of an event-triggered system is presented by Barrett, Hilborne, Verissimo, Rodrigues, Bond, Seaton & Speirs (1990).

### 2.2.3 Trade-off Decisions

An advantage with the time-triggered paradigm, in particular when designing safety-critical systems, is its predictability. Its cyclic behavior and static scheduling of CPU and other resources (e.g., communication) enhance predictability, especially in the time domain. The fact that it is designed for worst-case situations does, however, make these systems expensive in comparison with event-triggered systems. This illustrates the general design dilemma of finding suitable trade-offs when two or more system properties are in conflict with each other.

Unfortunately, there is little information about how real-time system design decisions and testability relate. Hence, it is difficult to find the optimal trade-off where testability and predictability as well as efficiency and performance are sufficient. Moreover, sometimes the static, time-triggered design is not an option and as long as dynamic, event-triggered systems are built, they should be tested. More information on how to make these systems easier to test is therefore needed.

Pure time-triggered or event-triggered systems are rare. Instead, a system may have characteristics from both types. A common approach is to separate critical parts of the system from non-critical and use a time-triggered design only on the critical parts where the cost is motivated by the criticality. Non-critical parts can then be designed according to the event-triggered design type, which is less expensive. Other solutions combine static scheduling with sporadic server tasks or a slack stealing algorithm that enables the system to handle a mixed task load to some extent, for example (Sprunt, Sha & Lehoczky 1989, Lehoczky & Ramos-Thuel 1992, Davis, Tindell & Burns 1993, Spuri & Buttazzo 1996, Isovic & Fohler 2000).

Many approaches to handle a mixed task load are based on the assumption that the task load consists of hard periodic tasks and soft sporadic tasks. This is not always the case. Emergency situations are usually not periodic, they often need attention immediately and the penalty of failing to handle the situation in a timely manner might be very high.

## 2.3 Efficient and Effective Testing

Since testing is an expensive but necessary activity, it is important that the methods and techniques used are effective and efficient with respect to their ability to reveal faults. In this dissertation the following definitions for effective and efficient are used.

**Definition 20.** *A test technique is considered effective if it has a high probability of revealing existing faults.*

**Definition 21.** *A test technique is considered* efficient *if it requires a small amount of effort to use it. For example, if it requires few test cases or is easy to automate.*

Effective testing is required to achieve a high level of quality and reliability in software (Mouchawrab, Briand & Labiche 2005). There are several reasons for this. Software applications tend to grow more complex and the more complex they get, the higher is the risk that faults are introduced into the software. Also, as complexity is increased there is a risk that faults get more complex and hard to reveal. Today, people are surrounded by and dependent on software in most of their daily activities and, as customers, they expect the software to have an adequate level of quality. As software testing techniques mature the excuse for poor software quality decreases and customers will eventually turn to those developers that can provide a satisfying level of quality for their products.

A large number of testing techniques has been invented and evaluated during the last decades. For example, techniques based on data-flow such as (Laski & Korel 1983, Rapps & Weyuker 1985), techniques based on the input domain such as (Myers 1979, Beizer 1990, Grindal, Lindström, Offutt & Andler 2006, Ostrand & Balcer 1988) techniques based on logical expressions such as (Chilenski & Miller 1994, Vilkomir & Bowen 2002, Zhu, Hall & May 1997) and techniques focusing on timeliness such as Nilsson et al. (2004) and Garousi (2008).

Whenever an evaluation of a testing technique is made, there are two questions that need an answer. How many faults were found and how many test cases were executed, compared to e.g., random testing? The first question focuses on effectiveness; whether the technique targets the faults more precisely than the other technique. The second question focuses on efficiency; whether the technique needs fewer test cases than the other technique.

Effective test execution, whether it is manual or automatic, requires the test object to satisfy some basic requirements. Most test cases have specific goals, e.g., execute a certain condition with a true outcome. Thus, it is important that the test case is executed exactly as described and that the behavior (internal as well as external) from the test object can be captured. This translates into the testability properties controllability and observability (Schütz 1993).

## 2.3.1 Controllability

**Definition 22.** *Controllability is the ability to (re)execute selected test cases.*

Given any ambition to use an effective test case selection method that targets the faults, it is important that the selected test cases are possible to execute. When it comes to testing the logical behavior of software, the main controllability issue is to identify the actual input that will lead to an execution which satisfies the test requirement. Consider the example above where the test requirement is to execute a certain condition with a true outcome. Suppose that the variables in the condition are internal and not among the input variables the tester can control. The problem is that it is seldom obvious which actual input that will take the execution to the specified location with the internal variables set to the intended values. The first part is a problem of reachability and the second part is known as the internal variable problem which is undecidable (Offutt 1988). The problem is frequently addressed by work on automated test data generation, for example (DeMillo & Offutt 1991), (Offutt, Jin & Pan 1999), (Korel 1990), (Gotlieb, Botella & Rueher 1998), (Chung & Bieman 2008), and (Ammann & Black 2002).

Controllability is hard to achieve in real-time systems. One reason is that an extra dimension, time, is added to the input/output domains. Moreover, the state of the system when the input is given might affect the behavior. The behavior does not only depend on which input is given but also when it is given, current schedule, program counters, blockings, etc. Time-triggered real-time systems approach the controllability issue in two ways. The cyclic behavior where the system completes execution during an activity interval and then returns to an initial state implies that there is no need to control the internal state as part of the test case input (Schütz 1993). Also, there is a coarse granularity with respect to observation points. To give an input that the system observes at time $t$, the input must be given in the interval $]t\text{-}o,t]$, where $o$ is equal to the elapsed time between two consecutive observation points (Schütz 1993). Event-triggered systems usually have a fine granularity with respect to observations and no cyclic behavior.

A minimum requirement on controllability, when testing for timeliness, is that it is possible to repeatedly inject a sequence of timed input events in the same way (Ammann & Offutt 2008). To use an effective test strategy with selected test cases (i.e., selected sequences of timed input events) it is necessary with a level of controllability that allows the tester to inject timed events at the exact points (or intervals) in time. Getting the time stamps right can be a very challenging task when the input events are e.g., items arriving at a sensor on a conveyor belt. Moreover, without a cyclic behavior, it is necessary to control the internal state as well as the time for the sensor signal (Schütz 1993, Birgisson et al. 1999, Thane 2000). This is a *very* hard challenge.

A property related to controllability is *reproducibility*. Reproducibility is the property that the system repeatedly exhibits identical behavior when stimulated with the same input. This property is especially important when it comes to regression testing and debugging (Schütz 1994, Thane 2000). Without reproducibility it might be difficult to activate the same error again during debugging. Moreover, the results from correction might be inconclusive whether the fault was removed or not.

Reproducibility is not necessarily an issue for testing non-real-time software since software often is predictable. However, for real-time systems reproducibility is very difficult to achieve (Thane & Hansson 1999*a*, Schütz 1994). This is especially true for event-triggered systems. The reason is that the actual behavior of a system depends on elements that have not been expressed explicitly as an input to the system. This means that what is judged to be a repeated test case might lead to different behaviors due to elements that cannot be controlled. For example, the response time for a task in an event-triggered system depends not only on the given input event (including its parameter values and time). It also depends on the current load, blocking times, and varying efficiency of hardware acceleration components. Moreover, timing of internal events such as allocation and deallocation of shared resources varies for the same reasons. This means that the outcome of a race condition can differ between executions. It is therefore possible to get different execution orders when a test case is repeated (Thane & Hansson 1999*a*, Schütz 1994). Hence, the behavior in the time domain is non-deterministic and it is the behavior in the time domain that testing for timeliness tries to assess. Both Thane & Hansson (1999*a*) and Schütz (1994) identify the predictability with respect to the number of execution orders as a direct indicator to the level of testability. The more non-deterministic the behavior is the higher is the demand for controllability to effectively test the system (Ammann & Offutt 2008).

### 2.3.2 Observability

**Definition 23.** *Observability is the ability to observe internal and external system behavior during test execution.*

When the level of observability is low, it can be difficult to distinguish between correct and erroneous behavior (Schütz 1994). Consider the case when the only visible output is a boolean value. If the input domain is large and there is an equal probability for true and false output, then the resulting value might not be sufficient to let the tester decide whether the system behaved correct or not. The probability is high that the result from execution is the same as the expected result even if there is a fault present and the execution activated the fault with a resulting error. When the probability for the erroneous state to propagate to the interface is low so is observability and therefore also the testability (Voas & Miller 1995). Moreover, propagation of an erroneous state to an interface does not guarantee high observability. Software that affects hardware devises, databases or remote files are considered to have low testability. Consider a test case altering an item $x$ in a database. The test engineer will easily find out whether $x$ was altered as expected but what if the test case also altered something else? It is necessary to investigate the entire database to observe such a failure.

Another problem when observability is low is that it can be difficult to determine whether the execution reached the intended location, i.e., whether the test requirement that was intended to be covered by a certain test case was actually covered by the execution of the same test case. Although reachability is a controllability issue, determining the success of the attempt to reach a certain location is an observability issue.

Object-oriented software imposes extra challenges with respect to observability (Ammann & Offutt 2008). The main reason is that the data abstraction components typically hide state information from the test engineer. Sometimes part of the source code is not available for the test engineer, e.g., when testing a sub-class where part of the behavior is inherited from a, not available, super-class. The problem with observability in object-oriented software has been addressed by several authors such as (Binder 1994, Mouchawrab et al. 2005, Kansomkeat & Riveipiboon 2008).

When testing software in non-real-time systems, there are basically two things can be done to increase the observability. Additional outputs can be used to make internal states visible. For example, a simple addition of a write statement where the current location and the current value of an internal variable is displayed (or logged) increases the observability. Another way to increase observability is to throw an exception when an interesting state change is made. For object-oriented software a requirement for additional `get` methods can increase observability (Ammann & Offutt 2008).

The traditional techniques to achieve observability described above are unfortunately less useful for real-time systems. The reason is that these techniques introduce a probe effect (Gait 1986).

**Definition 24.** *The probe effect is a phenomenon where the behavior of a system may be affected due to the attempt of observing it.*

For example, alterations to the source code to produce additional output or throw exceptions under certain conditions will add execution time. This can in turn affect the response time as well as the time for synchronization and resource allocation. Finally, changing the timing behavior may change the execution order and this might affect the behavior in the value domain as well as the time domain.

Event-triggered systems are more prone to the probe effect than time-triggered systems (Schütz 1993). The event-triggered system delivers a result as soon as possible. Thus, even small changes can introduce probe effects. In time-triggered systems the time for delivery of a result is predefined. If the probe effect is small enough, the probe effect will not cause any detectable consequences (Schütz 1993).

Schütz (1993) identified three ways to handle the problem with the probe effect in real-time systems. The probe effect can be ignored, minimized or avoided. Ignoring the probe effect in the hope that it will in reality not or only rarely appear is a risky approach when it comes to testing for timeliness. The reason is that when a tester tests for timeliness that tester tries to stress the system as much as possible to determine whether a certain deadline will be met under the worst possible conditions. As an event-triggered system is stressed with a heavy load, the risk for race conditions increases and therefore the probability of a probe effect to have an impact on the behavior is increased (Schütz 1993).

Minimizing the probe effect by implementing "sufficiently" effective monitoring operations or by compensating the results for estimated probe effects is a less useful approach for event-triggered systems than for time-triggered. The reason is that in the event-triggered system, even the smallest change to the execution time can affect the resulting execution order and therefore the timely behavior (Schütz 1993). In the time-triggered system, on the other hand, small probes that can be executed within the allocated time frame will not affect the behavior (Schütz 1993).

Avoiding the probe effect can be done by employing dedicated hardware for monitoring or by leaving all the probes in the system after deployment (Thane 2000, Schütz 1993, Mellin 2004). By keeping the probes, it is possible to ensure that the tested system is no different from the deployed. Avoiding the probe effect is a feasible solution for event-triggered systems for example by using a predictable event monitor (Mellin 2004).

# Chapter 3

# Problem Statement

The work described in this thesis is motivated by and based on previous work in the area of testability in real-time systems. By constraining the execution environment, testability in event-triggered systems is said to be increased. This has however, never been shown so a major goal in this thesis is to determine whether the approach proposed in previous work will lead to higher testability for this type of systems. This problem has previously been discussed and motivated in Papers 7, 2, 5 and 6.

## 3.1 Previous Work

The most complete work in the area of testability in real-time systems is done by Schütz (1993). Schütz presents a formula for an upper bound of the test effort for time-triggered real-time systems. He also points out that the formula is a lower bound of the test effort of an event-triggered real-time system. The formula assigns a significantly higher bound for event-triggered systems than for time-triggered systems based on the frequency with which the system observes events in the environment. However, Schütz (1993) points out that the bound is a lower bound due to the fact that the formula does not consider preemptions. Schütz (1993) shows that testability is greatly improved by a time-triggered architecture. Based on these observations Schütz presents a methodology for testing time-triggered real-time systems.

In 1998, Mellin presented a method to define an upper bound on test effort for event-triggered real-time systems (Mellin 1998). The presented formula extended Schütz optimistic bound to include the system state with respect to preemptions and blockings. This formula was later refined to allow for more than one resource (Birgisson et al. 1999).

Birgisson et al. (1999) suggest that some constraints on the execution environment, such as predefined observation points, designated preemption points, and a maximum number of concurrently executing tasks, should be adopted in a constrained event-triggered design. The result would still be a dynamic, event-triggered system but with a level of testability which approaches that of time-triggered systems.

Birgisson et al. (1999) give the following formula for the upper bound on test effort for a system where the proposed constraints are applied. The upper bound gives all combinations of event sequences and internal states with respect to preemptions and blockings. Since the upper bound on test effort only reflects properties of the real-time system itself, it should give a bound on testability, assuming that the formula is correct.

$$FSTAT = ESTAT * BSTAT * PSTAT \tag{3.1}$$

$$ESTAT(s,n) = \sum_{k=0}^{n} \left( \begin{array}{c} n \\ k \end{array} \right) s^k = \sum_{k=0}^{n} \left( \begin{array}{c} n \\ k \end{array} \right) s^k 1^{n-k} = (s+1)^n \tag{3.2}$$

$$PSTAT(p,q,t) = \left( \sum_{k=0}^{q} (p+1)^k \right)^t = \left( \frac{(p+1)^{q+1} - 1}{p} \right)^t \;, where\ p > 0 \tag{3.3}$$

$$BSTAT(q,t,C) = \sum_{c \in C} \frac{\prod_{j=1}^{r} \binom{q*t-\sum_{k=1}^{j-1} c_k}{c_j}}{\prod_{n \in elem(c)} card(n,c)!} \tag{3.4}$$

*FSTAT:* Gives the upper bound for the number of combinations of event sequences and states with respect to experienced preemptions and current blockings for each task.

*ESTAT:* Gives the number of distinct event sequences for an interval with $s$ observation points and $n$ distinct events that may occur during the same interval. Each event is either not observed in the interval or it is observed at one of the $s$ observation points. Hence, there are $s + 1$ possibilities for each event and $(s + 1)^n$ possibilities for $n$ events. *ESTAT* is first presented in Schütz (1993).

*PSTAT:* Gives 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 allowed preemptions for a task. Example: PSTAT(*3,2,4*) gives that for each of the 4 task types, there might be 0, 1, or 2 current instances of that specific task type in any state. Each instance has been preempted 0, 1, 2 or 3 times, i.e., there are 4 possible preemption states for each task instance (in this example $p+1 = 4$). Hence, for each task type there is 1 (in case of zero instances) plus 4 (in case of one instance) plus $4^2$ (in case of two instances), i.e., 21 possible preemption states with respect to a single task type. 4 task types give $21^4$ possible preemption states with respect to all possible task sets.

*BSTAT:* Gives all possible blocking states for a maximum of $q$ concurrently executing instances of $t$ task types. $C$ is the set of blocking scenarios and $c$ is a blocking scenario such that $c \in C$. Assume that there is a blocking scenario $c$ such that $c = [5, 3, 3, 2]$. This blocking scenario contains four blockings, i.e., $|c| = 4$. The numbers indicate how many tasks that are involved in the different blockings. All blocking states that have 5, 3, 3 and 2 tasks blocked on four different resources belong to this scenario. Finally $c_i$ is the $i$th element in the scenario, i.e., $c_4$ in the given example is 2. BSTAT takes one scenario at a time from the set $C$ and enumerates the number of potential blocking states for that scenario. $card(h, c)$ is a function that calculates the number of $h$ in $c$. In the given example there is $card(5, c) = 1$ and $card(3, c) = 2$. The function $card(h, c)$ and an algorithm that generates $C$ is presented in Birgisson et al. (1999)

From this point on, these formulae is referred to as **FSTAT**.

## 3.2 Problem Definition

Even though time-triggered systems have higher testability than their event-triggered counterparts, it is sometimes necessary or at least preferable to choose an event-triggered design. One reason is the cost. Time-triggered systems are designed to meet the resource needs in worst-case situations. If there is a gap between the resource need in the worst and in the average case, there will be an expensive waste of resources. Another reason to choose an event-triggered design is unpredictable environments. Sometimes it is difficult to foresee changes in the environment or to determine the worst-case execution time. In such situations event-triggered systems are unavoidable.

As discussed in chapters 2.2 and 2.3, testability is hard to achieve in real-time systems and is especially low in event-triggered systems. Due to the low level of testability, it is difficult and expensive to apply effective test techniques to such systems. For example, both Schütz (1994) and Thane & Hansson (1999*b*) argue that each possible execution order should be tested adequately. Therefore the effort to test a system increases as the predictability with respect to the behavior in the time-domain decreases. The consequence is that the quality might not be sufficient. If it is possible to increase testability in event-triggered systems, then testers will be able to perform better testing and thereby increase quality of the delivered products.

Since event-triggered systems are common and hard to test, it would help to increase testability in such systems. A method for increasing testability in event-triggered systems by constraining the execution environment was proposed by Birgisson et al. (1999). Unfortunately, the usefulness of this method has never been shown. It is therefore necessary to determine whether the method is useful for obtaining increased testability in event-triggered systems or not.

**Aim:** This dissertation aims to determine whether applying the proposed set of constraints on the execution environment increases testability in event-triggered real-time systems while maintaining the event-triggered semantics, as claimed in previous work (Mellin 1998) (Birgisson et al. 1999).

A set of objectives are identified. The aim is considered met when all of the following objectives are met:

**Objective 1.** *To select a metric suitable for estimating the testability of a real-time system*

**Objective 2.** *To define a method for estimating the testability with the selected metric*

**Objective 3.** *To apply the proposed constraints to a system model and estimate the effect on testability*

**Objective 4.** *To compare the actual results with the results predicted by formula 3.1,* **FSTAT**

**Objective 5.** *To discuss the implications from the constraints on system semantics*

There are different views on testability and how testability can be estimated or measured. It is therefore necessary to discuss these views and clarify which view on testability this dissertation adopts. Without such clarification it is not possible to judge whether the constraints have an effect on testability or not. A survey of software testability and a discussion of some different testability definitions is therefore given in Chapter 4. Moreover, regardless of which testability view this dissertation adopts, it is not possible to measure testability directly since testability is not a property that can be directly quantified. Therefore a metric with which a reasonable approximation of testability can be made is needed. This approximation can be used to compare the estimated effect on testability with the upper bound given by Formula 3.1, **FSTAT**. Objective 1 is therefore to identify such metric. Moreover, the metric should go in line with previous work on testability in real-time systems and assign the highest level of testability to the time-triggered design type.

An event-triggered real-time system model is needed to investigate how the constraints affect testability. The constraints need to be included in the model in such way that they can be controlled and their effect on testability can be estimated. Objective 3 therefore includes applying the constraints into a real-time system model. This gives a model of a constrained event-triggered real-time system. The constraints are included in the model in such way that it is possible to vary the level of each constraint.

Objective 2 is to identify a method with which testability can be estimated in the constrained event-triggered real-time system model with the metric identified by objective 1. The comparison of the measured results and the expected results is reflected in objective 4.

Finally, objective 5 considers the impact from the constraints on the semantics of the system. Is a constrained event-triggered real-time system still an event-triggered system? The discussion includes a description of the characteristics of the event-triggered semantics and the effect the constraints may have on this.

The execution environment constraints included in this dissertation are:

C1) Predefined observation points. Observation of event occurrences is delayed until the next observation point. This is necessary to define an upper bound on the number of potential event sequences. The reason is that an event sequence contains both a set of events and their time stamps. With continuous time and arbitrary observations, the number of potential event sequences would, at least in theory, be infinite.

C2) A known upper bound on the number of preemptions a task can experience. Tasks are only allowed to be preempted at specific points in their execution. These points must be known beforehand. The reason is that the preemption points limit the number of potential interleavings.

C3) A known upper bound on concurrently executing tasks of the same task type. This constraint is necessary to define an upper bound on the number of states the system can enter.

## 3.3 Expected Results

For each execution environment constraint that this dissertation investigates, a hypothesis for the expected result is defined. The hypotheses are:

**Hypothesis 1.** *Given defined observation points, the number of observation points affects testability with $O(s^n)$ where $s$ is the number of observation points and $n$ is a fixed number of event types.*

**Hypothesis 2.** *Given designated preemption points in time, the number of allowed preemptions, $p$, for a task affects testability with $O(p^{tq})$ where $t$ is a fixed number of task types, and $q$ is a fixed maximum number of concurrently executing tasks of the same type.*

**Hypothesis 3.** *Given designated preemption points in time, the maximum number of concurrently executing tasks of the same type affects testability with $O(p^{tq})$ where $p$ is a fixed number of allowed preemptions for a task and $t$ is a fixed number of task types.*

**Hypothesis 4.** *Formula 3.1,* **FSTAT** *expresses an upper bound on test effort that is sufficiently tight to be used as an approximation of system testability. For example, given that Formula 3.1 suggests an exponential growth with respect to a variable v, the estimated effect on testability is exponential to* $k * v$*, where k is a constant and* $0 < k \leq 1$.

Hypotheses 1, 2 and 3 consider the question whether the formula gives a true upper bound or not while hypothesis 4 considers the question of whether the upper bound is true and tight enough to be useful as a metric for testability.

## 3.4 Research Methodology

The research problem is to investigate the relation between testability and a set of system properties. This is a problem that can best be addressed by an empirical study where quantitative measurements are collected and analyzed with respect to hypotheses 1 to 4.

According to Robson (1993) there are three types of research strategies that can be used when designing an empirical study; (*i*) a survey, (*ii*) a case study, and (*iii*) an experiment. In this dissertation the choice is to design an experiment. The reason is that the goal is to study the effect on testability while the value of three other variables are manipulated, i.e., the constraints on the execution environment. To do this, full control of the manipulated variables and full observability of the effect on testability are necessary. The level of control is higher in an experiment than it is in a case study (Wohlin, Runesson, Höst, Ohlsson, Regnell & Wesslén 2000).

The results from the experiment are objective quantitative measurements that either support or reject the above hypotheses 1 to 4.

Objective 5 is to discuss the effect on the event-triggered semantics when the execution environment constraints are applied to a system. This part of the dissertation work is better approached by a qualitative research strategy. Qualitative studies aim to discover and describe phenomena based on descriptions given by the subjects of study (Wohlin et al. 2000). The discussion in this dissertation is based on the semantical differences between event-triggered and time-triggered systems described in literature and on the observed behavior of the constrained event-triggered system used in the experiment.

# Chapter 4

# A Metric for Testability

Testability is approximated by the number of execution orders in this dissertation. This chapter explains why this metric is so suitable for estimating testability when the focus is testing for timeliness of event-triggered systems. This is the focus for this dissertation as explained in Section 4.2. The used metric has previously been discussed and motivated in Paper 3.

As discussed in Section 2.1.2, the test effort is affected by a number of different aspects. This includes the test process, the degree of automation, and the skills of the test team. One important such aspect is the testability of the test object. But what is testability? Recall the view on testability that is introduced in Section 2.1.2. Testability concerns properties of the test object. Low testability of a test object implies that the test object is hard to test. It is the author's opinion that testability of a test object is a property of the test object itself and therefore testability of a test object should not vary due to other factors such as a selected test strategy or the used test process.

Testability is a concept that is frequently used in literature and there is an overall agreement that low testability implies that the test object is hard to test. However, as mentioned previously in Chapter 2.3, different authors seem to have different interpretations of the concept. A survey of work in the area of software testability is therefore given in 4.1. The definition of system testability, which is used throughout this dissertation, is presented in 4.1.2.

Section 4.2 describes some of the special issues that are related to timeliness testability and event-triggered real-time systems. The issues motivate the metric with which timeliness testability is approximated in the experiment.

## 4.1 A Software Testability Survey

McCabe (1976) argues for a need of a mathematical technique that allows us to identify software modules which are difficult to test and maintain. This technique will also provide a basis for modularization. The approach is to measure and control the number of paths through the program. Cyclomatic complexity of a program is calculated by the number of nodes and vertices in a control graph where the nodes represent sequences and vertices represent decisions. It is shown that complexity is dependent on the decision structure of the graph. There is a discussion of how the cyclomatic complexity can be used to identify the minimal number of paths that should be tested. By reducing the cyclomatic complexity, it is therefore possible to increase testability.

Freedman (1991) states that a testable software component has the following properties:

*i* Small and easily generated test sets

*ii* Non-redundant test sets

*iii* Easily interpreted test outputs

*iv* Easily locatable software faults

In my view, these ideas are not good enough since only item three conforms to the view on testability that this dissertation adopts. Item *i* considers the selected test criteria, item *ii* considers the method for test generation, and item *iv* considers debugging. Even though all of these aspects clearly affect the resulting test effort, they are not properties of the software component itself.

Schütz (1993) shows how the distributed nature and the real-time characteristics add to the problems of testing software. The author describes six different fundamental problems that have to be considered when testing a distributed real-time system. The problems presented by Schütz (1993) are organization, reproducibility, observability, host/target approach, environment simulation and representativity. Schütz (1993) shows how different requirements affect testability and discusses the relations between the listed problems.

Binder (1994) states that the two key facets of testability are controllability and observability. Moreover, the author gives an argument that software testability is the result of six factors where three of the factors (item *i* to *iii*) are properties of the system while the other three (item *iv* to *vi*) refers to the test process:

*i* Characteristics of the representation

*ii* Characteristics of the implementation

*iii* Built-in test capabilities

*iv* The test suite (test cases and associated information)

*v* The test support environment

*vi* The software process in which testing is conducted

Another example is given by Voas & Miller (1995). The authors argue for design of software that has greater ability to fail when faults do exist. Observability can be poor due to implicit information loss, i.e., high domain/range ratio or explicit information loss, i.e., encapsulation of variables. The potential for implicit information loss can be predicted by functional descriptions or code inspections. Explicit information loss is less dependent on specification and more dependent on the design of the software. Several strategies for design of software with high testability are suggested; (*i*) Isolating modules that have a high information loss, (*ii*) minimizing reuse of variables, and (*iii*) increasing the number of output parameters, i.e., auxiliary output.

Vranken, Witteman & van Wuijtswinkel (1996) describe system testability as depending on complexity, state space, controllability, and observability. Testability is usually treated differently for hardware and software but, as the authors point out, when designing system level tests, it is often not yet decided which of the components that will be implemented as hardware or software. Hence, it is necessary to have methods that can handle both. The authors approach to tackle this problem is by partitioning the system into modules and inserting test functionality into the modules. The idea is that partitioning leads to improved testability. By making improved testability as the major criterion when partitioning, testability is further improved. Discussion is held about coupling and parallelism. The added test functionality allows control and observations of individual modules for testing purposes. Three kinds of test functions are described: (*i*) Transparent test mode (TTM) (*ii*) built-in self-test (BIST) and (*iii*) point of control and observation (PCO).

Byers (1997) discusses testability as the probability to execute a particular code location given some input distribution. A probability is associated with each edge in the control flow graph and finding the execution probability for a program is simply to solve a set of forward data-flow equations. Some initial work on the propagation probability is also presented in this paper. The described work is a static approach related to the PIE method presented in (Voas 1992) and the view of testability is also very similar to (Voas & Miller 1995).

Dssouli, Karoui, Saleh & Cherkaoui (1999) presents a method used for finite state machines (FSM). The authors propose a three-dimensional classification of FSMs, based on three testability properties: minimality, specifiedness and determinism where the highest testability is given by a reduced, complete, and deterministic FSM. Moreover, transformations of an FSM with testability in focus can move the FSM from one class to another class with higher testability.

Birgisson et al. (1999) present a method for reducing the test effort for event-triggered systems. The method uses a system architecture that inherits certain constraining properties of time-triggered systems but still maintains the flexibility of event-triggered systems. By applying such constraints on an event-triggered system the authors argue that it is possible to reduce the number of test cases required for full test coverage when testing for timeliness on a system level. Observability and controllability are regarded as prerequisites for testability and the authors do not distinguish between testability and test effort.

Thane & Hansson (1999*a*) present a method for deterministic and reproducible testing of distributed real-time systems. The main idea is to make it possible to use sequential test techniques for distributed real-time systems. The authors present a method that calculates all possible execution orders for a system with periodic tasks only and fixed priority scheduling. By doing this they claim that it is possible to apply traditional test techniques for sequential programs. The reason is that each identified execution order can be regarded as a sequential program and therefore tested as a sequential program. The authors present an algorithm that calculates the execution order graph (EOG). In addition to using the EOG for testing the authors argue that the EOG can be used to measure testability since the number of execution orders is a measure for the testability of the real-time system. This measure could be used as a scheduling criterion to generate static schedules of high testability. Hence, a schedule that gives a small EOG should be preferred over a schedule with a large EOG.

Wang, King & Wickburg (1999) describe how complete test functions can be placed into components. This is an approach that is referred to as the built-in test, BIT, approach. During maintenance the system can be executed in test-mode and the testing is conducted by calling the built-in test-functions.

Gao, Tsao, Wu & Jacob (2003) discuss three different approaches to increase software testability.

*i* Framework-based testing facility developed to allow engineers to add test code into the software components

*ii* Built-in tests that requires developers to include test code into the components to support self-testing

*iii* Automatic component wrapping for testing, which is a method to make a component testable by wrapping the component with code that supports testing

Mouchawrab et al. (2005) present a classification of testability attributes for object-oriented software. The attributes are classified according to size, cohesion, coupling and complexity with respect to different aspects such as state behavior, structure, scenarios and interfaces. The classification provides the reader with a set of testability measurements that can be applied to the object-oriented system to assess testability already during the design phase.

Kansomkeat & Riveipiboon (2008) present a technique that is used in order to improve testability of object-oriented components. The basic idea is to first perform an analysis on the Java component bytecode level and then use the analysis to gather information about control flow and dataflow. The information is then used to increase controllability and observability by instrumentation of the code.

### 4.1.1 Testability Definitions

As shown in Section 4.1, there are different views of testability but there seems to be some consensus among the authors. Most authors identify controllability as an important part of testability, e.g., (Voas & Miller 1995, Byers 1997, Schütz 1993, Vranken et al. 1996, Birgisson et al. 1999, Thane & Hansson 1999*a*, Dssouli et al. 1999, Wang et al. 1999, Binder 1994, Gao et al. 2003, Mouchawrab et al. 2005, Kansomkeat & Riveipiboon 2008). Most authors also identify observability to be a part of testability, e.g., (Freedman 1991, Schütz 1993, Vranken et al. 1996, Birgisson et al. 1999, Thane & Hansson 1999*a*, Dssouli et al. 1999, Wang et al. 1999, Binder 1994, Gao et al. 2003, Mouchawrab et al. 2005, Kansomkeat & Riveipiboon 2008). Other properties that are suggested to be included in the definition of testability are the size of the test set and the support for automation, e.g., (McCabe 1976, Freedman 1991, Schütz 1993, Vranken et al. 1996, Birgisson et al. 1999, Byers 1997, Dssouli et al. 1999, Vranken et al. 1996, Binder 1994). In my view, size of the test set and support for automation are properties that are tied to the test process and the methods used to test the software. The view in this dissertation is that the level of testability assigned to a piece of software should be independent of how it is tested.

As a consequence of the different interpretations, the concept of testability has proven to be hard to define. Several definitions exist. Not surprisingly, these definitions do not always agree. Three commonly used definitions are:

**Definition 25.** *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, and (2) the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met (IEEE 1990).*

**Definition 26.** *The probability that a piece of software will fail on its next execution during testing (with a particular input distribution) if the software includes a fault (Voas & Miller 1995).*

**Definition 27.** *Attributes of software that bear on the effort needed for validating the modified software (Standard ISO/IEC 9126 1991).*

The first definition is the IEEE standard definition. Some of the problems with this definition are:

*i* The testability of the software is mixed with the testability of the requirements. These are two separate things and should in my opinion, not be mixed.

*ii* The definition is open for different interpretations. One such interpretation is that testability has to do with whether the system provides test functionality or not. Another interpretation is that it has to do with whether a component is an off-the-shelf black box or a white box with available code. These are only two of many possible interpretations that are more related to test effort than testability.

*iii* Software testability should be a property of the software itself. It should not vary depending on the method used for testing the software. This is necessary to compare the level of testability given by different designs. It is in my opinion, not a good idea to have a software property (software testability) depending on anything apart from the software.

The second definition is a refinement of the IEEE standard definition made by Voas & Miller (1995). This definition is more precise than the IEEE definition, i.e., it is not open for different interpretations. Given this definition, testability can be measured (at least in theory) in probabilistic terms. The method calculates the probability that the test suit will; (*i*) execute the part of the software that contains the fault, (*ii*) have the right parameter values that activate the fault and (*iii*) propagate the resulting erroneous state so that a failure can be observed. The higher this probability is the higher is the assigned testability.

The view described by Voas & Miller (1995) is appealing since it clearly considers controllability and observability of the software. However, my opinion is that this definition suffers from being tied to the testing process (input distribution). With this definition testability varies with the input distribution as well as with the fault itself (different faults have different probabilities of being revealed). The consequence is that the same piece of software may have different testability depending on both the fault and the effectiveness of the used test method.

Finally, the third definition is Standard ISO/IEC 9126 (1991). Even though this definition does not suffer from being tied to the test process, it is somewhat vague and it is therefore not obvious what type of attributes it is referring to. Is it test functionality, software complexity, etc.? Moreover, it only concerns validation of software after a modification.

### 4.1.2 System Testability

Section 4.1.1 presents some definitions on software testability. However, when discussing testability in terms of system level testing, software testability as a concept is less appropriate. Instead of software testability a definition of system testability is needed.

When testing a real-time system, a point is reached where it is necessary to test for non-functional properties such as temporal correctness. For example, timeliness (which is the focus here) cannot be tested for the software in isolation from the execution environment. Instead, it should be tested at the system level and preferable on the target. The reason is that the execution environment affects timeliness. For example, it is not possible to state anything about the temporal behavior without considering the policy for scheduling. Moreover, it is not really relevant to discuss software or hardware testability in isolation when testing for timeliness. The temporal behavior is significantly affected by the hardware. It is therefore the system testability rather than the software testability that is of interest in this dissertation.

System testability is one of many contributing factors to test effort. In particular it is the factor that is based on properties of the system. As system testability increases, the test effort based on these properties decreases. However, it is the author's opinion that it is not possible to give a definition of system testability that is precise enough to be useful for a specific purpose and general enough to be useful from all aspects. 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 the tester is testing for, i.e., the test goal. Hence, a system may have high testability with respect to the purpose of some testing activities (for example, assessing the logical correctness in a software unit) and low with respect to the purpose of other (for example, assessing performance in a target system). It is therefore my 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 that is used here.

**Definition 28.** *System testability is the degree to which a system has a design or implementation that makes it easier to select, execute, observe, and analyze tests targeting verification of required system properties.*

The required property in focus for this dissertation is timeliness and testability is studied with respect to this property. This is referred to as *timeliness testability*. Note that even though the definition focuses on a specific property that the test activity tries to assess, the definition is independent of both the test process and the methods.

## 4.2 Testability in Event-triggered Systems

Testing an event-triggered system is harder than testing a corresponding time-triggered system (Schütz 1993). Schütz (1993) presents a comparison of the effort to test a system given the time-triggered and event-triggered design approaches for real-time systems. The comparison focuses on the number of possible event sequences that can enter the system. It is also noted that preemptive scheduling can further increase test effort due to the effect on the number of execution orders.

As described in Chapter 2.3, effective test execution requires the test object to satisfy some basic requirements with respect to controllability, observability, and reproducibility. Concurrency, resource allocation policy, and online scheduling are considered to be major factors that affect testability in dynamic, event-triggered real-time systems (Schütz 1993, Thane & Hansson 1999*a*, Birgisson et al. 1999). 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 (i.e., interleavings in a task set).

Controllability includes controlling the state to start test execution from and injecting sequences of events at specified points in time. Controlling the starting state is significantly easier with time-triggered designs because their cyclic behaviors ensure that they always return to their initial state before they accept any new input. Injecting events at specified points in time is also much easier with time-triggered solutions due to the coarse observation granularity. The finer the observation granularity is, the higher is the required time precision for the event injection.

In contrast to a time-triggered real-time system, the input space of an event-triggered real-time system does not have natural partitions of the temporal domain. An event may influence the behavior of the system at any point in time. The lack of natural partitions has several consequences for the tester.

*i* It produces a larger input space than in the time-triggered case. One reason is that the finer observation granularity implies a larger set of potential sequences of time-stamped events. Another reason is that the state of the system must be considered as part of the test case.

*ii* It makes controllability more difficult. Again, it is the fine granularity that puts higher demands on the precision. These demands affect the system both with respect to event injection and to the enforcement of a specified state from which to start the test execution.

*iii* It places higher demands on an environment simulator. The reason is that the simulator must have an observation granularity at least as fine as the target system. Also, the simulator must be tested.

As described in Chapter 2.3, event-triggered systems are more prone to the probe effect than time-triggered systems. The reason is that an event-triggered system will deliver the result as soon as it is ready. Thus, even small changes can introduce probe effects. In time-triggered systems the time when the final result is delivered is predefined. Given that the probe effect is smaller than the slack time that is available before the result should be delivered, the probe effect will not cause any detectable consequence (Schütz 1993). There are however, techniques to handle probe effects by e.g., predictable monitoring (Schütz 1993, Mellin 2004).

Reproducibility is hard to achieve in event-triggered systems since the system behavior partly depends on elements that have not been expressed explicitly as inputs to the system (Schütz 1994, Thane & Hansson 1999*a*, Birgisson et al. 1999). That is, the behavior is non-deterministic when just the software in concern is considered. Hence, what is judged to be a repeated test case might lead to different behaviors due to elements that testers cannot control, including hardware components. Both Schütz (1993) and Thane & Hansson (1999*a*) points out the non-determinism with respect to execution orders as an important testability factor since all execution orders must be adequately tested.

A common way to deal with non-deterministic behavior is to repeat test cases several times to get statistical confidence for the results. However, there is nothing that guarantees that the probability for a set of potentially different behaviors is equally distributed. Consider a concurrent update to a shared variable $x$, e.g., $x++$. $x++$ can be translated to three instructions by the compiler: (*i*) store the value of $x$ into the register, (*ii*) increase the value stored in the register, and (*iii*) store the register value into $x$. Suppose that $x$ is not properly protected by e.g., a semaphore. If a process is preempted between the first instruction (*i*) and the third instruction (*iii*), then all updates to $x$ from that point made by other processes will be overwritten when the preempted process is dispatched and continues its execution again. The chance to reveal this fault, i.e., the missing semaphore operation by running the test cases repeatedly is very small. The reason is of course that the chance of getting a preemption between instruction (*i*) and (*iii*) is small.

The problem is similar when testing an event-triggered system for timeliness. Consider a race condition between two tasks for a shared resource $R$, and assume that the deadline is met or missed depending on the outcome of the race condition. There is no guarantee that the two competing tasks have the same probability of winning the race for $R$. Hence, even if the same test case is executed several times and the deadline is met in all the executions, there might still be a possibility that the test case can lead to a missed deadline. This is different from time-triggered systems since such systems give one potential execution order for each possible input sequence (Schütz 1993).

In a pure time-triggered real-time system as described by Schütz (1993) and Kopetz (1991) new inputs are observed at the observation points. Events occurring in an interval between such points are observed by the system at the next observation point. At this point all triggered tasks are known by the system and executed according to an off-line schedule, i.e., a look-up table. Moreover all of these executions are finished before the system reaches the next observation point. The result will therefore be exactly the same regardless of the exact point in time when the involved events occurred, as long as they occur in the same observation interval between the same two consecutive observation points. The result will also be exactly the same regardless of any new event occurrences during their execution. These new events will not be observed or considered for execution until the next observation point.

The focus here is timeliness testing and one goal of timeliness testing is to provoke the system to miss a deadline. Hence, it is important to select test inputs that have high probability to reveal such behavior, i.e., the worst case scenarios with respect to timeliness.

In a time-triggered system, the worst case scenario with respect to timeliness is when the highest load and worst case execution time for all involved activities occur. In such systems, testing of timeliness becomes equivalent to finding the input conditions that maximize resource consumption of each task and then trigger all these tasks at the same time.

In an event-triggered system the problem is more complex. Occurrences of new events can affect the schedule at any point in time. The test case is therefore a combination of current state (e.g., current task load, program counter and other information in the process control blocks (PCB)) and the time and order of input events (i.e., interrupt signals). There is nothing that guarantees that the worst case is the test case where all involved tasks maximize their resource consumption. For example, a task finishing earlier than its worst case execution time might lead to a different outcome of a racing condition between two other tasks. This might in turn affect timeliness. Moreover, the test case where all tasks are triggered at the same time is probably not the worst case. The reason is that this situation implies that the scheduler has full information about the task load and is therefore likely to find a feasible schedule if such schedule exists. A test case where some of the urgent tasks arrive when resources already are allocated by other tasks may be more likely to make the system miss a deadline.

Introducing accelerating hardware such as caches and pipelines means that the number of variations in the time domain is increased (that is, the difference between best case and worst case with respect to elapsed time is increased). 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 same test is run.

The above described approach to get statistical confidence for the result by repeating the test execution several times is not appropriate for timeliness testing. 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, a goal is to increase the confidence that a deadline will be met under **all** circumstances. The average speed is of no concern for timeliness

(Stankovic 1988).

A final observation is that the problem with several potential execution orders tends to get worse when the system is stressed by event bursts (Schütz 1993). These are precisely the situations that a tester would use to provoke the system to miss a deadline. From a tester's point of view, it is the worst cases that should be executed, by for example, executing with overload or reduced capacity. Focusing on the worst cases and adverse circumstances distinguishes timeliness testing from e.g., testing for reliability, which usually focuses on test cases representative for an operational profile.

As the number of potential execution orders increases, it becomes harder to gain sufficient confidence for timeliness with a statistical approach. Moreover, since each execution order must be adequately tested, the test effort is increased when the number of execution orders is increased (Thane & Hansson 1999*a*, Schütz 1993). In this dissertation the number of potential execution orders therefore is considered to be a reasonable metric for estimation of timeliness testability. This metric of timeliness testability is used in the study of the execution environment constraints and their impact on testability. The metric goes in line with previous work on testability in real-time systems (Schütz 1993, Thane & Hansson 1999*a*) and assigns the highest testability to the time-triggered architecture.

# Chapter 5

# A Tool for Trace-Set Generation

In Chapter 4, the number of execution orders is identified as being a reasonable approximation of testability in a dynamic, event-triggered system. In this chapter, a method is defined with which the selected metric can be used to determine the level of testability of a real-time system. The algorithm presented here has previously been presented in Papers 3 and 4.

Enumerating all potential execution orders means that all possible behaviors with respect to the interleavings among a set of tasks must be explored. To do so on a real system or in a simulator is impractical and therefore a model checker is chosen. With a model checker it might be possible to explore the complete behavior of a real-time system model given the general limitations of model checking, i.e., consumption of memory and time due to a large state space.

The model checker used in this dissertation is UPPAAL (Larsen, Pettersson & Yi 1997, Amnell, Behrmann, Bengtsson, D'Argenio, David, Fehnker, Hune, Jeannet, Larsen, Müller, Pettersson, Weise & Yi. 2001). UPPAAL is a tool for modeling, simulation and verification of timed automata models (Alur & Dill 1994). The basic idea is to model the real-time system, including the execution environment, and then use a model checker to explore the model to enumerate all potential execution orders.

Two problems must be solved to estimate testability with this approach.

*i* The first problem is that a model checker cannot deliver more than a single trace per invocation; a tool is needed. This tool must keep track of the orders that are already found and force the model-checker to search for a trace with an execution order that differs from the ones already found. This gives a subset of all traces and this subset covers all execution orders.

*ii* The second problem is that the behavior of a dynamic real-time system model usually is too complex to explore exhaustively. This means that the state space explosion problem must be handled to guarantee that **all** potential orders are enumerated.

All execution orders cannot be enumerated by executing a real system or a simulator. Execution in a real system or a simulator will only give information about the orders that are covered. There is no information about any missed orders. Model checkers on the other hand, can sometimes prove properties of a system model by exploring its state space. Given a set of covered execution orders it might be possible to use a model checker for investigation of whether it is possible to reach an order different from the already covered orders. Model checking is therefore chosen for the experiment in this dissertation.

Model checking (Clarke & Emerson 1981, Queille & Sifakis 1982) has developed into a powerful technique for automatic formal verification of transition systems. A model checker can accept a state-based model and a property, and find a trace through the model that satisfies (or contradicts) that property if such a trace exists. Common properties to prove are global invariants, e.g., mutual exclusion, or showing that some state can be reached, e.g., a deadlock.

Model checking can also be used for job-shop scheduling; for example, to find a job schedule that gives high throughput and sufficient product quality. Another application is test case generation, where the model checker gives a trace that covers a test requirement. These applications all need individual traces, one for each property or requirement. A model checker therefore typically generates a single trace.

The addressed problem, however, requires **sets of traces** that collectively cover **all** execution orders. A key insight of this research is to generate sets of traces by iteratively invoking the model checker, where each new trace must differ from the previous traces with respect to these orders. Unfortunately iteratively invoking the model checker is sometimes less useful for a model of a real-time system since such an approach requires generation of the complete state space.

Real-time models tend to be complex, with many states, and the *state-space explosion problem* of model checkers (Holzmann 2003) means it is difficult to exhaustively analyze the models. The state-space explosion problem refers to the exponential size of state space with respect to the size of the input model, the number of clocks and the largest constant that is used in a clock constraint or guard. Several attempts have been made to reduce the memory usage of model-checking algorithms (Behrmann, Larsen, Pearson, Weise & Yi 1999, Bengtsson & Yi 2001, Bengtsson & Yi 2003). However, memory and time still remain bottlenecks in model checking.

When iteratively asking a model checker to find new and different traces, the model checker needs to explore more and more of the state space for each invocation until the generated state space is too large and the exploration fails due to memory consumption. Such an approach is therefore likely to fail. Instead, a tool is needed that can generate a set of traces, i.e., all execution orders, while at the same time mitigating the problem with excessive memory consumption. Such a tool is therefore, developed for the experimental study described in this thesis. Without this innovative method the included study would not have been possible to carry out. The tool is illustrated in Figure 5.1.

The input to the tool is a timed automata model and the output is a set of traces. Each trace is a list of edges in the model. The file RTS Model in Figure 5.1 contains a formal specification of a real-time system. The RTS Model file is manually transformed into the Modified Model, which contains special markers at each edge that should be included in the traces. The goal is to generate a set of execution orders, so a marker is included at each point where a dispatch can be made. A description of timed automata is given in Section 5.1 and the description of the model is given in Section 5.2.

The algorithm used by the calculator can force the model checker to generate all potential execution orders given a modified model (as in figure 5.1). The algorithm takes the modified model as input and generates the orders by repeated invocations of the model checker. To mitigate the state space explosion problem, each exploration is guided into those parts of the state space where a solution might be found. Section 5.3 describes how. The result is a file that contains all potential execution orders. An example and a performance evaluation of the method are given in 5.4 and 5.5.

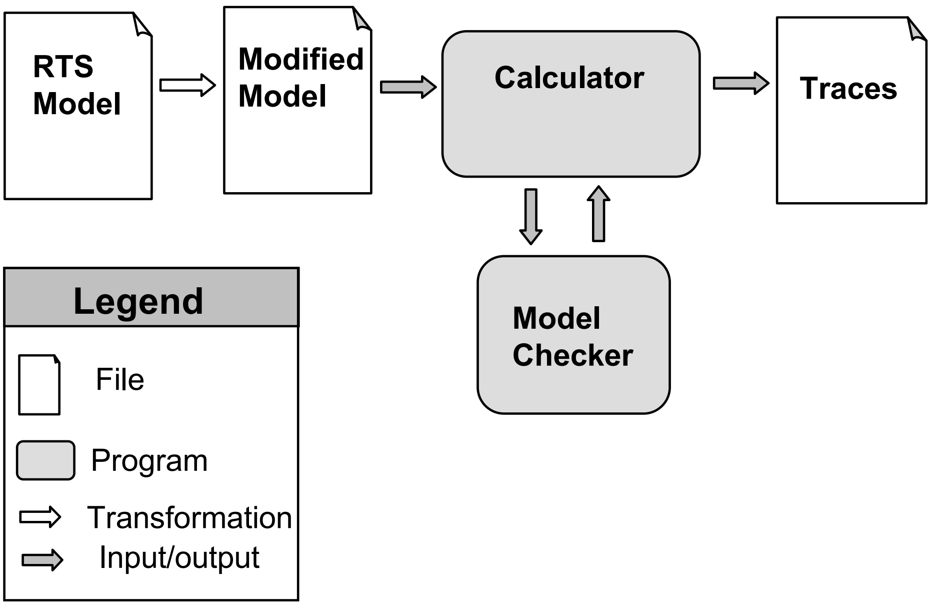


Figure 5.1: A tool for trace-set generation. A model of a real-time system is transformed into a modified model that includes markers on selected edges and a guide process. The calculator then saves all traces that are distinct with respect to the order with which the marked edges are traversed.

## 5.1 Timed Automata

To describe this work with a tool for generation of execution orders, a brief introduction to timed automata is needed. Engineers commonly use timed automata to specify and verify real-time systems. This section reviews definition used in this dissertation. Bengtsson & Yi (2004) and Hessel et al. (2003) have more details on these concepts.

*Clocks* are represented by a finite set of real-valued variables $\mathcal{C}$ and *actions* are represented by a finite alphabet $\Sigma$. Let $\mathcal{B}(\mathcal{C})$ denote the set of Boolean combinations of clock constraints of the form $x \sim n$ or $x - y \sim n$, where $x, y \in \mathcal{C}$, $n$ is a natural number and $\sim$ represents one of the relational operators $\{<, \leq, =, \geq, >\}$.

**Definition 29.** *A* ***timed automaton*** *($\mathcal{A}$) over ($\Sigma, C$) is a tuple* $\langle N, l_0, E, I \rangle$ where*:*

- *$N$ is a finite set of locations*

- *$l_0 \in N$ is the initial location*
- *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times N$ is the set of edges*
- *$I : N \rightarrow \mathcal{B}(\mathcal{C})$ assigns invariants to locations*

Consider the example in Figure 5.2. A bus is scheduled to leave a station at 10:05. The bus is however expected to wait for passengers arriving with a train. The bus is therefore required to stay at the station for at least two minutes after the arrival of the train. The set

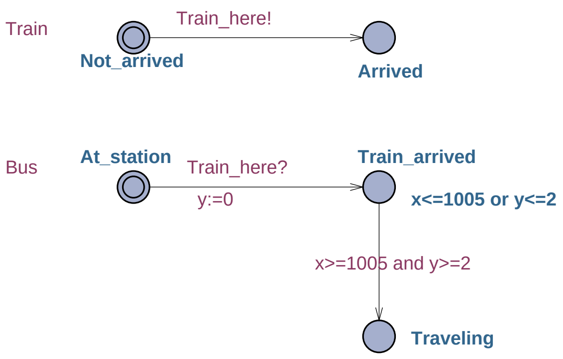


Figure 5.2: A timed automata model of a bus scheduled to leave at 10:05 but required to synchronize with a train before leaving the station.

of nodes for the bus is $N = [At\_station, Train\_arrived, Traveling]$. The initial node for the bus is $l_0 = At\_station$. The set of clocks for the bus is $\mathcal{C} = [x, y]$. There are also clock constraints in the form of a node invariant, $x <= 1005$ or $y <= 2$, and a guard $y >= 35$. The clock $y$ is reset when the train arrives while $x$ keeps track on real time.

**Definition 30.** *The **semantics of a timed automaton** is a timed transition system over states of the form $\langle l, u \rangle$, where $l \in N$ and $u$ is a clock assignment of all clocks in $\mathcal{C}$ to non-negative real-numbers. Transitions are defined by the two rules:*

- *(discrete transitions) $\langle l, u \rangle \xrightarrow{a} \langle l', u' \rangle$ if $\langle l, g, a, r, l' \rangle \in E$, $u \in g$, $u' = [r \mapsto 0]u$ and $u' \in I(l')$*
- *(delay transitions) $\langle l, u \rangle \xrightarrow{d} \langle l, u \oplus d \rangle$ if $u \in I(l)$ and $(u \oplus d) \in I(l)$ for a non-negative real $d \in \Re_+$*

where $u \oplus d$ denotes the clock assignment, which maps each clock $x$ in $\mathcal{C}$ to the value $u(x) + d$, and $[r \mapsto 0]u$ is the clock assignment $u$ with each clock in $r$ reset to zero. Basically there are two rules for transitions because there are two kinds of transitions; the discrete transition, which is an instant move from one node to another, and the delay transition, which is an incrementation of the clocks but includes no move to another node.

**Definition 31.** *A* ***run*** *of a timed automaton $\mathcal{A} = \langle N, l_0, E, I\rangle$ with initial state $\langle l_0, u_0\rangle$ over a timed trace $\xi = (t_1, a_1)(t_2, a_2)(t_3, a_3)...$ is a sequence of transitions:*

$$\langle l_0, u_0\rangle \overset{d_1}{\rightarrow}\overset{a_1}{\rightarrow} \langle l_1, u_1\rangle \overset{d_2}{\rightarrow}\overset{a_2}{\rightarrow} \langle l_2, u_2\rangle \overset{d_3}{\rightarrow}\overset{a_3}{\rightarrow} \langle l_3, u_3\rangle ...$$

*satisfying the condition $t_1 = d_1$ and $t_i = t_{i-1} + d_i$ for all $i \geq 1$. The* timed language $L(\mathcal{A})$ *is the set of all timed traces $\xi$ for which there exists a run of $\mathcal{A}$ over $\xi$.*

**Definition 32.** *A network of timed automata $\mathcal{A}_1 \| ... \| \mathcal{A}_n$ over $(\Sigma, C)$ is the parallel composition of $n$ timed automata over $(\Sigma, C)$.*

where components are required to synchronize on delay transitions and discrete transitions are required to be synchronized on complementary actions. An action a? is complementary to a!.

## 5.2 Creating the Input Model

This section describes how the input model, needed to generate the sets of traces, is created.

A trace through a timed automaton is a sequence of discrete or delay transitions that are traversed in a given order, where a discrete transition is derived from edges while a delay transition includes no edge. The basic idea with the approach presented here is to annotate selected edges and then generate a trace-set with respect to these edges. The final set of traces gives all unique sequences with which these selected edges can be traversed in the given model. Since the goal is to generate execution orders, the focus is on edges where tasks can be dispatched, e.g., preemption points. Each trace is generated and extended step-wise by guiding the model checker, so that the search for an extension stays within the partition where the extension can be found.

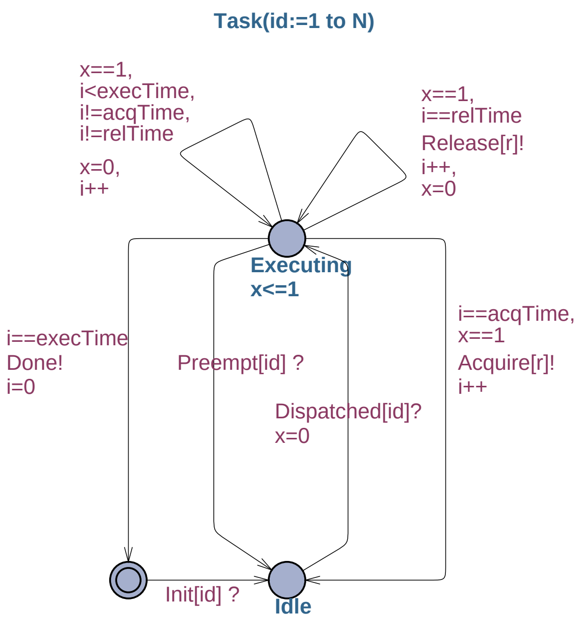


Figure 5.3: A simplified task model. An initiated task alters between the states Idle and Executing until it is done. Clock constraints implement execution time and allocation time.

Consider the following example. The automaton in Figure 5.3 specifies a simple model of a task. The task is executed by a process that is part of a concurrent system with $N$ processes executing similar tasks. When initiated, the task alters between the states Idle and Executing until it has been in state Executing for $execTime$ time steps. At time $acqTime$ a request for resource $r$ is made and at time $relTime$ the resource is released. It is assumed that there is a scheduler with which the automaton synchronizes via the channels Dispatch, Preempt and Done. The automaton synchronizes with a resource handler via the channels Acquire and Release.

The problem addressed when generating sets of traces is to get a subset of all traces that is complete and non-redundant with respect to the execution orders. A model of a real-time system includes more than a task set, e.g., scheduler and environment and it is only the interleavings among the timed automata processes that model real-time tasks that should be collected. Moreover, only the interleavings with respect to the access of the processor should be collected. Therefore a subset of the edges in the automaton is selected and the global order of the transitions is derived from these edges only. All edges where tasks synchronize with the scheduler are therefore selected. These edges are labeled *Dispatched[id]?* in figure 5.3 on the edge from Idle to Executing. By generating all global orderings with which these edges can be traversed, all execution orders are collected.

An interesting note here is that this approach can be used to generate trace-sets that cover test criteria based on orders. Depending on which edges that are selected, the set of traces will have different properties. For example by selecting all edges where a shared resource $r$ is acquired and released it is possible to generate all orders with which the resource is used by the different tasks.

An innovative aspect of the research presented here is that all edges that dispatch tasks are marked with special markers called *p-points*. A *p-point* is denoted p-point[$id$,$j$], where $id$ is the process identity, and $j$ is a number associated with that p-point (e.g., the edge and the point in execution). This is shown in Figure 5.4 on the edge from Idle to Executing.

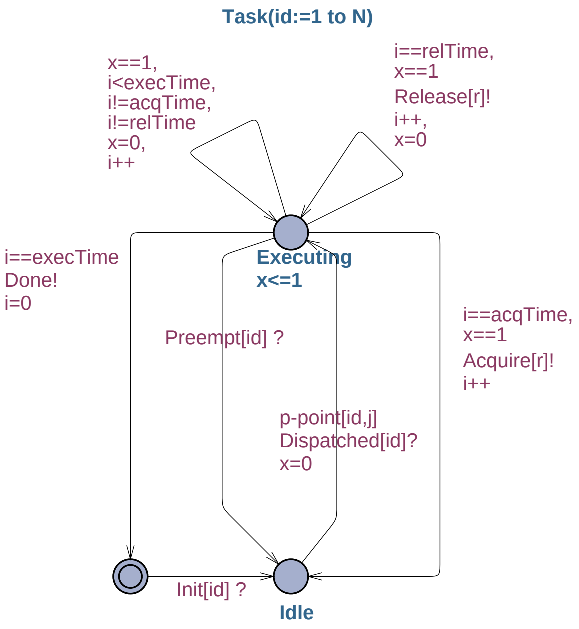


Figure 5.4: The simplified task model annotated with a p-point, *p-point[id,j]*, on the edge from Idle to Executing. *id* is the process identity, and *j* is an enumeration of the p-point.

Any trace $\xi \in L(\mathcal{A})$ will traverse a subset of the p-points in some order. Such sequence of traversed p-points is referred to as a *p-path*. Figure 5.4 contains a p-point on the edge labeled *Dispatched[id]?* on the edge from Idle to Executing. Section 5.3 shows how p-paths are collected using the existing model checker UPPAAL (Larsen et al. 1997) and dynamic manipulation of the analyzed model.

## 5.3 Partitioning and Model-Checking

If the model checker can detect loops[1], then it can generate a finite set of p-paths through the model in Figure 5.4. The set of p-paths in this example describes the set of execution orders since the *p-point* is placed on the edge from Idle to Executing where the task synchronizes with the scheduler.

**Definition 33.** *Let $pp_\sigma$ be the sequence of p-points that a trace $\sigma \in L(\mathcal{A})$ traverses (possibly the empty sequence). Let $PP$ be the set of all p-paths pp of an automaton $\mathcal{A}$. A trace $\sigma$ is said to follow a p-path $pp \in PP$ if $pp_\sigma$ is a prefix of pp.*

The p-paths are generated by iteratively invoking the model checker with queries that extend an existing p-path prefix by one step. An extra, *guiding automaton* is used to guide the model checker in its search for the one-step extensions. The basic idea is that the original automaton will synchronize with the guiding automaton at each p-point (as shown in figure 5.5). The guiding automaton uses an array, Ppath, to store the current p-path prefix and an integer, Length, to store the length of the p-path prefix. Synchronization at p-point $j$ as the $(i+1)$th traversed p-point is successful only if the following is true:

*i* $Ppath[i] = j \wedge i < Length$, *or*

*ii* $i = Length$,

where $Ppath[i-1]$ is the $i$th element of the array *Ppath*. Intuitively, (1) holds if the execution follows the p-path prefix specified by *Ppath*, and (2) holds if an extension to the p-path prefix is reached.

[1]A reasonable requirement since many model checkers do, including UPPAAL (Larsen et al. 1997) and SPIN (Holzmann 1997).

Figure 5.5 shows a guiding automaton (on the right). The left automaton is the simple task model extended with two new edges (labeled *Allow!* and *Allowed?*), which implement the p-point on the edge from Idle to Executing in Figure 5.4. Note how the left automaton synchronizes with the guiding automaton whenever a p-point is reached (i.e., when the left automaton is given access to the processor). Note also that when the end of current p-path is reached (the edge guarded $i == Length$ in the guiding automaton), and hence a potential extension of the p-path is found, the identity of the current p-point extending the p-path is stored in a variable Next and the system deadlocks.

The guiding automaton will prevent the state-space exploration from exploring states where extensions cannot be found. Any attempt to abandon the current p-path during exploration is bound to fail. The approach thereby mitigates the problem of memory consumption. By following the p-path, the process will find an extension if one exists. The use of p-points and a guiding automaton allow the current

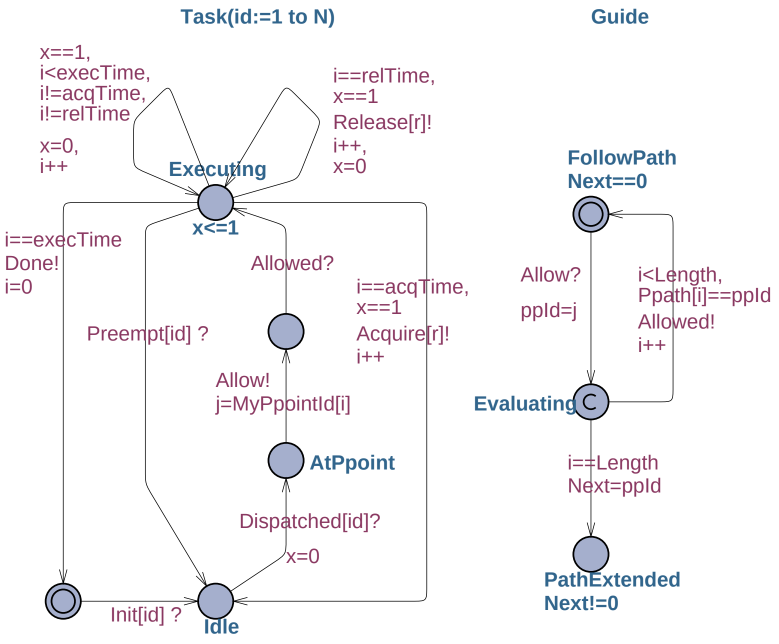


Figure 5.5: The simplified task model extended with a guiding automaton, which guides the search for next p-point. A dispatch immediately leads to a synchronization with the guide process.

p-path to be extended if possible. It also mitigates the state-space explosion problem since the search follows the p-path.

To generate all complete p-paths without modifying the applied model checker, it is necessary to dynamically manipulate both the verified safety property and the model files. The algorithm for generating p-paths is shown in Figure 5.6.

The algorithm uses a stack, $S$, to store information about the current set of generated p-paths. Each stack item is a pair $\langle pp, n\rangle$, where $pp$ is a p-path prefix and $n$ is its length. $\epsilon$ is used to represent the empty p-path, and $pp :: q$ to represent the result of appending the path $q$ to $pp$. The algorithm also uses files that contain the model of the system, $M$, and the property to be verified, $Q$. Properties of the form $\exists \diamond \phi$ is used to specify that a state satisfying $\phi$ is reachable in the model.

The stack starts with one element $\langle pp_0, n_0\rangle$, where $pp_0 = \epsilon$ and $n_0 = 0$, the query in the property file is initialized to $\exists \diamond Next \neq 0$, and the model file is modified by setting the values for the constant array $\mathsf{Ppath}$ to $pp_0$ and $\mathsf{Length}$ to $n_0$ with values from the stack (i.e. $\epsilon$ and 0). The initial property is satisfied as soon as a possible continuation is found, i.e. when a process synchronizes with the guiding automaton at a p-point. From the diagnostic trace generated by the model checker, a possible continuation of $pp_0$ can be extracted from the value of $\mathsf{Next}$. This value is denoted $pp_1^1$. At this point, two things are done:

*i* Push $\langle pp_0 :: pp_1^1, n_0 + 1\rangle$, onto the stack, and

*ii* Call the model checker with the extended query $\exists \diamond Next \neq 0 \wedge Next \neq pp_1^1$

This is repeated until the query $\exists \diamond (Next \neq 0 \wedge Next \neq pp_1^1 \wedge ... \wedge Next \neq pp_1^n)$ cannot be satisfied. At this point, there are $n$ p-paths on the stack: $\langle pp_1^1, 1\rangle, ..., \langle pp_1^n, 1\rangle$, each with length one.

A new pair $\langle pp_i^j, n_i\rangle$ is popped in each iteration. These values are used to set $\mathsf{Ppath}= pp_i^j$ and $\mathsf{Length}= n_i$ in the model file. The query in the property file is re-initiated to $\exists \diamond \mathsf{N}ext \neq 0$. The above procedure is then repeated until all possible continuations of $pp_i^j$ with length $n_i + 1$ are pushed onto the stack.

The algorithm finds a complete p-path when the initial query $\exists \diamond Next \neq 0$ returns false, meaning that there is no continuation of the current p-path. The result is a unique sequence of p-points

```
while $S$ DO →
        pop $S$ for $pp$ and $n$
        $Q$ = "$\exists \diamond Next \neq 0$"
        copy original model file to $M$
        find and replace in $M$ →
              $Ppath[0] = \{\}$ with $Ppath[n] = \{pp\}$
              $Length = 0$ with $Length = n$
        $satisfied = true$
        $alternatives = 0$
        while $satisfied$ DO →
              invoke model checker with $M$ and $Q$
              if $satisfied$ DO →
                 $alternatives++$
                 find $pp_i^j$ in generated trace
                 push $< pp :: pp_i^j, n+1 >$ onto $S$
                 $Q = Q ::$ "$\wedge\ Next \neq pp_i^j$"
              else if $alternatives == 0$ DO →
                 save $pp$                       # This is a complete p-path
              end
        end                                      # All continuations for $pp$ found
end                                              # All p-paths found
```

Figure 5.6: P-path generating algorithm. *pp* is the current sub p-path, $n$ is its length, $S$ is the stack, $Q$ is the current query, and $M$ is the file containing the model.

defining a p-path. The algorithm terminates when the stack is empty. At that point all p-paths are identified.

## 5.4 A Small Example

The approach to generate trace sets can be used to generate all orders with respect to any kind of edges of interest. For example, it might be interesting from a testing perspective to generate a set of orders with respect to synchronization or resource usage.

This section illustrates the algorithm with a step-by-step example, using the automata model in Figure 5.7. This model is simplified to

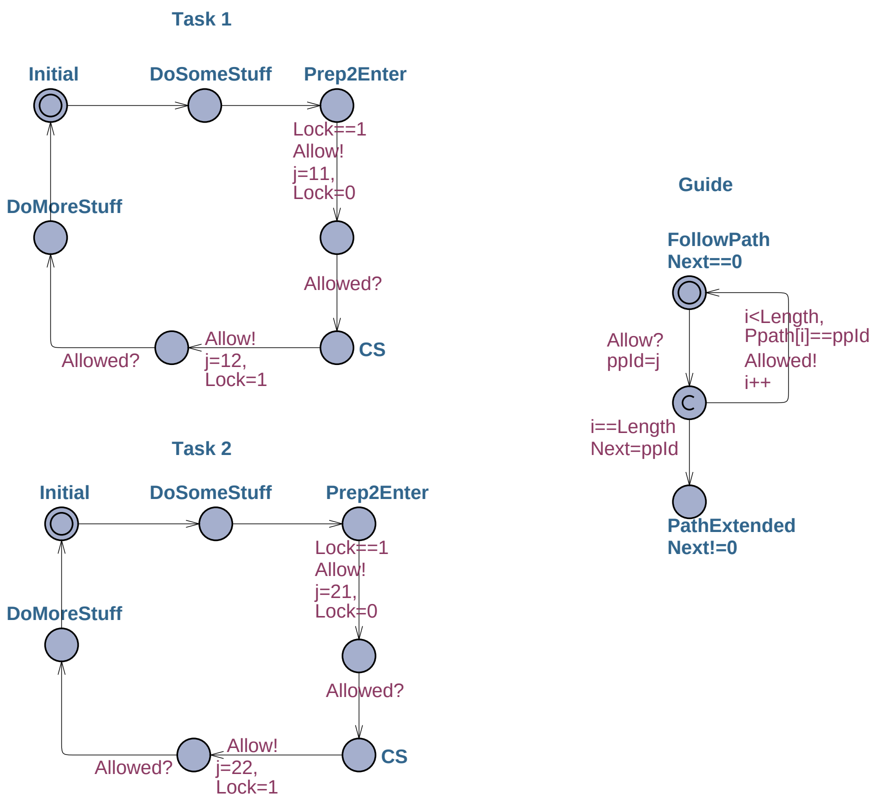


Figure 5.7: A simplified model using a binary semaphore Lock to protect the critical section CS.

focus on how this algorithm works on the example. The model in this small example does therefore, not consider time, scheduling or the behavior of the tasks. The only detail left is a critical section (CS) and the protocol for *entry* and *exit*.

The algorithm is used to generate all orders with which the tasks may enter and exit CS. Hence, the edges where semaphore Lock is used are selected and a p-point is inserted at each selected edge. An empty sequence and a 0 are pushed onto the stack to begin the first iteration. Each iteration begins by modifying the model file, $M$, with values fetched from the stack, $S$. The query, $Q$, is modified before each invocation of the model checker.

In the first iteration (see Table 5.1) two alternatives for the first p-point are identified, 11 and 21. The third query is falsified. The next iteration searches for extensions to the last saved p-path, $\{21\}$.

The second iteration (see Table 5.2) gives only one possible extension to p-path {21}, 22. The reason is that Task 2 has entered the critical section CS and the entry is locked for Task 1. The search is continued from p-path {21, 22}.

There are two possible extensions to p-path {21, 22}, 11 and 21 (see Table 5.3). The search continues from the p-path {21, 22, 21} in iteration 4.

The model checker returns false in response to the first query (see table 5.4). This means that it is not possible to extend the p-path {21, 22, 21} with an additional p-point without re-entering a state that has already been visited, i.e., a loop. This p-path is complete and therefore saved. The search is continued for extensions to {21, 22, 11} in iteration 5.

There is only one possible extension to p-path {21, 22, 11}, 12 (see Table 5.5). The search is continued from p-path {21, 22, 11, 12}.

There are two possible extensions to p-path {21, 22, 11, 12}, 21 and 11 (see Table 5.6). The search continues from the p-path {21, 22, 11, 12, 11}.

No extensions to {21, 22, 11, 12, 11} were found (see Table 5.7). This is the second complete p-path in this example and the p-path is therefore saved. The search continues for extensions to p-path {21, 22, 11, 12, 21}.

There is only one possible extension to p-path {21, 22, 11, 12, 21}, 22 (see Table 5.8). The search is continued from p-path {21, 22, 11, 12, 21, 22}.

There is only one possible extension to p-path {21, 22, 11, 12, 21, 22}, 21 (see Table 5.9). The search continues from p-path {21, 22, 11, 12, 21, 22, 21}.

No extensions to {21, 22, 11, 12, 21, 22, 21} were found (see Table 5.10). This is the third complete p-path in this example and it is therefore saved. The search continues for extensions to p-path {11}.

At this point, all orders that begin with p-point 21 are covered. The saved orders so far are {21, 22, 21}, {21, 22, 11, 12, 11}, and {21, 22, 11, 12, 21, 22, 21}. Continuing the search will give the additional orders {11, 12, 11}, {11, 12, 21, 22, 21}, and {11, 12, 21, 22, 11, 12, 11}, as shown in Figure 5.8.

Table 5.1: Iteration 1: Extensions to p-path {}.

| | | |
|---|---|---|
| *Stack*: | [] | Result |
| *Query*: | $\exists\Diamond Next \neq 0$ | 11 |
| *Stack*: | [{11}1] | |
| *Query*: | $\exists\Diamond Next \neq 0 \land Next \neq 11$ | 21 |
| *Stack*: | [{11}1,{21}1] | |
| *Query*: | $\exists\Diamond Next \neq 0 \land Next \neq 11 \land Next \neq 21$ | False |

Table 5.2: Iteration 2: Extensions to {21}.

| | | |
|---|---|---|
| *Stack*: | [{11}1] | Result |
| *Query*: | $\exists\Diamond Next \neq 0$ | 22 |
| *Stack*: | [{11}1,{21,22}2] | |
| *Query*: | $\exists\Diamond Next \neq 0 \land Next \neq 22$ | False |

Table 5.3: Iteration 3: Extensions to {21,22}.

| | | |
|---|---|---|
| *Stack*: | [{11}1] | Result |
| *Query*: | $\exists\Diamond Next \neq 0$ | 11 |
| *Stack*: | [{11}1,{21,22,11}3] | |
| *Query*: | $\exists\Diamond Next \neq 0 \land Next \neq 11$ | 21 |
| *Stack*: | [{11}1,{21,22,11}3,{21,22,21}3] | |
| *Query*: | $\exists\Diamond Next \neq 0 \land Next \neq 11 \land Next \neq 21$ | False |

Table 5.4: Iteration 4: Extensions to {21,22,21}.

| | | |
|---|---|---|
| *Stack*: | [{11}1,{21,22,11}3] | Result |
| *Query*: | $\exists\Diamond Next \neq 0$ | False |
| | $\rightarrow$ **Save {21,22,21}** | |

Table 5.5: Iteration 5: Extensions to {21,22,11}.

| | | |
|---|---|---|
| *Stack*: | [{11}1] | Result |
| *Query*: | $\exists\Diamond Next \neq 0$ | 12 |
| *Stack*: | [{11}1,{21,22,11,12}4] | |
| *Query*: | $\exists\Diamond Next \neq 0 \land Next \neq 12$ | False |

## 5.5 Performance Evaluation

To evaluate the performance of the algorithm, it is compared to ordinary model checking where the model checker is asked to return

Table 5.6: Iteration 6: Extensions to {21,22,11,12}.

| | | |
|---|---|---|
| *Stack*: | [{11}1] | Result |
| *Query*: | $\exists\Diamond Next \neq 0$ | 21 |
| *Stack*: | [{11}1,{21,22,11,12,21}5] | |
| *Query*: | $\exists\Diamond Next \neq 0 \wedge Next \neq 21$ | 11 |
| *Stack*: | [{11}1,{21,22,11,12,21}5,{21,22,11,12,11}5] | |
| *Query*: | $\exists\Diamond Next \neq 0 \wedge Next \neq 21 \wedge Next \neq 11$ | False |

Table 5.7: Iteration 7: Extensions to {21,22,11,12,11}.

| | | |
|---|---|---|
| *Stack*: | [{11}1,{21,22,11,12,21}5] | Result |
| *Query*: | $\exists\Diamond Next \neq 0$ | False |
| | → **Save {21,22,11,12,11}** | |

Table 5.8: Iteration 8: Extensions to {21,22,11,12,21}.

| | | |
|---|---|---|
| *Stack*: | [{11}1] | Result |
| *Query*: | $\exists\Diamond Next \neq 0$ | 22 |
| *Stack*: | [{11}1,{21,22,11,12,21,22}6] | |
| *Query*: | $\exists\Diamond Next \neq 0 \wedge Next \neq 22$ | False |

Table 5.9: Iteration 9: Extensions to {21,22,11,12,21,22}.

| | | |
|---|---|---|
| *Stack*: | [{11}1] | Result |
| *Query*: | $\exists\Diamond Next \neq 0$ | 21 |
| *Stack*: | [{11}1,{21,22,11,12,21,22,21}7] | |
| *Query*: | $\exists\Diamond Next \neq 0 \wedge Next \neq 21$ | False |

Table 5.10: Iteration 10: Extensions to {21,22,11,12,21,22,21}.

| | | |
|---|---|---|
| *Stack*: | [{11}1] | Result |
| *Query*: | $\exists\Diamond Next \neq 0$ | False |
| | → **Save {21,22,11,12,21,22,21}** | |

complete p-paths, one at a time. This approach requires a defined final state $S$. The initial query asks the model checker if $S$ is reachable ($\exists \Diamond S$). The resulting trace gives a complete p-path, $pp_1$. The query is then extended to $\exists \Diamond S \wedge \neg pp_1$. This is repeated until the query $\exists \Diamond S \wedge \neg pp_1 \wedge ... \wedge \neg pp_n$ is falsified, meaning there are no more p-paths. This procedure is automated and is referred to as the *base-line*

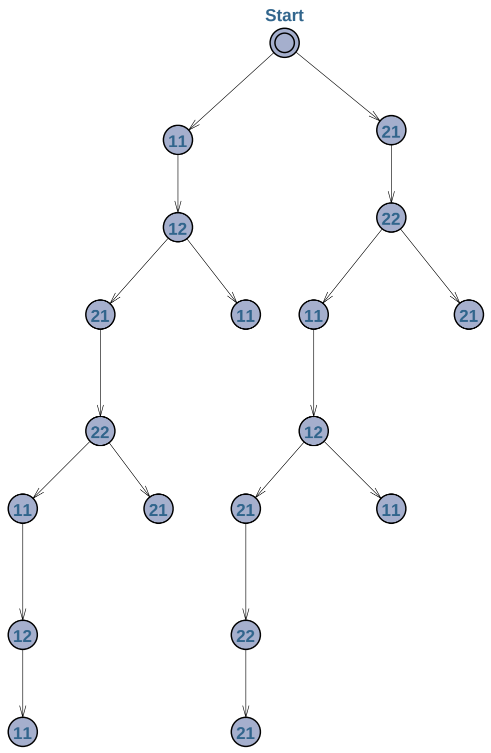


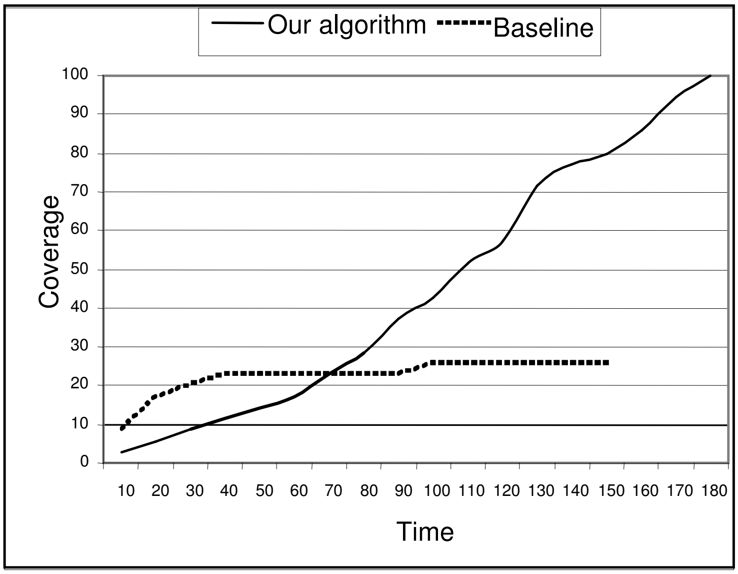
Figure 5.8: A tree showing all orders with which the p-points can be traversed in the model given in Figure 5.7.

*algorithm.*

The baseline approach only invokes the model checker once per p-path and it does not require the model to be altered during the search. The consequence is that the first p-paths are found rather quickly compared with the new algorithm described in this chapter. However, the search is not guided and the model checker has to explore a larger part of the state-space for each additional p-path. The final invocation, which falsifies the query for an additional p-path, forces the model checker to explore the complete state-space. Hence, if the model is too complex for exhaustive search, then at some point in the search, the baseline algorithm will run out of memory. This leaves part of the state space unexplored and the set of p-paths found incomplete.

We applied the new algorithm and the baseline algorithm to the

Table 5.11: Comparison of p-path coverage.



same model[2]. The complete set of p-paths contained 35 different orders. Table 5.11 shows that the new algorithm covered all p-paths in the model within 3 hours and the p-paths were found with a constant rate. The baseline algorithm delivered the first p-paths quickly but the performance dropped severely after 20% coverage. It only managed to find nine p-paths, giving less than 26% coverage. At that point the model checker stopped with an "out of memory" error message.

The new trace-finding algorithm split the search space into partitions along the p-paths. The algorithm finds one p-path at a time, using a stack to remember where to continue the search. However, it is important to note that, due to the tree structure and the fact that the search begins at the root node, each item on the stack contains enough information for an independent search for all continuations of

[2]The model included a total of 34 TA processes, eight of them modeling tasks annotated with p-points as shown in Figure 5.5. The experiment was performed on US-II sparc 12 CPU multiprocessor with 400MHz, 4Mb cache, and 6144Mb memory. The search order option in the model checker was set to depth-first.

that particular p-path prefix. Hence, this algorithm is particularly suitable for parallelization. It is possible to distribute the search over several processors, thereby performing a more time-efficient search.

The experiments was conducted on a 12 CPU multiprocessor, which increased efficiency by an order of magnitude. The limitation of this approach is, of course, the size of the partitions. Threading on a multiprocessor implies shared memory and might create a memory consumption problem. If this happens, the partitions can easily be distributed over separate processors. The cost for distribution in terms of communication is bounded by the number of partitions since there is only need for one message per p-path.

## 5.6 Termination and Correctness

Introducing the iterator i in the guiding automaton in figure 5.5 might cause loops to be unfolded in the state space. The reason is that the model checker cannot recognize a previously explored state (and thereby avoid exploring it again) if i is incremented. Variable i is therefore declared as a *hidden* (or *meta*) variable. Such variables are used to annotate a model, but are not considered when states are compared during the state-space generation. This means that states that only differ by the value of i will be considered to be equivalent by the model checker. Using a hidden variable guarantees that the number of p-paths is finite and that the algorithm will terminate. Hidden variables are common in model checking tools like SPIN (Holzmann 1997) and UPPAAL (Larsen et al. 1997).

The described algorithm for generation of trace sets is sound since it does not introduce any new states. The Guide process does not change any global variables and the only global variable that is touched by the Task process at the p-point (i.e., at the transitions between the two states AtPpoint and Executing) is the integer $j$ which is introduced to communicate the p-point identity with the Guide process. $j$ is not used anywhere else in the model. Moreover, there are no delay transitions introduced by the p-point; (*i*) channel Allow is an urgent channel and since the Guide process is ready to synchronize, the synchronization will occur before there is a change to the clock values, (*ii*) state Evaluating in the Guide process is a committed state meaning that the Guide process is not allowed to delay in that state. The only effect the Guide process and the p-points can have on the state space is a reduction since transitions that abandon the current p-path immediately leads to a deadlock.

The described algorithm for generation of trace sets is complete since it generates all p-paths. Consider Figure 5.8, there are three ways to miss a p-path:

*i* the algorithm has missed a branch somewhere in the tree

*ii* the algorithm has missed a continuation of a leaf of the tree

*iii* the algorithm stops before all p-path prefixes are explored.

Case *i* cannot happen for the following reason. Given an arbitrary node in the tree, the p-path prefix is used to find continuations. All traces that traverse the p-points in the order specified by the p-path prefix are allowed. The query that is given to the model checker is an open reachability query, i.e., is there **any** trace that reaches a consecutive p-point, which is different from the ones already found. As long as the answer is yes, the traversed p-points (including the continuation) is saved as a p-path prefix on the stack and the query repeated to find other continuations. When the answer is no this implies that it is **not** possible for **any** trace to traverse the p-points in the specified order and continue to any other p-point than the ones that are already found, i.e., no such branch exists.

The argument for case *ii* not to happen is very similar. There is a leaf node only if the model checker returns no to the first query for the current p-path prefix. This means that it is not possible for **any** trace to traverse the p-points in the specified order and continue to **any** consecutive p-point. Hence, if the model checker itself is correct, Case *i* and *ii* cannot occur.

Case *iii* differs since it concerns the stack and how it is used. The reason that case *iii* cannot occur is that each identified continuation to a p-path prefix results in a new p-path prefix, which is pushed onto the stack. The algorithm does not halt until the stack is empty. It is therefore not possible for the algorithm to stop until all p-path prefixes are explored.

# Chapter 6

# Impact on Testability

This chapter describes an investigation of the impact on testability from a set of execution environment constraints when these are applied to a system model. The theory (see Chapter 3.1) predicts a certain impact on testability. The purpose of the work described in this dissertation is to evaluate the usefulness of previous theoretical work. The experiment and its results has been presented and discussed in Paper 1. The experimental setup with the steel plant is used as a case study in Paper 4. Some of the material has also been presented as work in progress in Papers 5 and 7.

The equations (**FSTAT**, see Equations 3.1 to 3.4) provide appealing models, but they must be verified empirically to be useful. Specifically, this dissertation seeks the answer to three questions: (*i*) do the proposed constraints (the values of *s, q* and *p*) affect testability in the same way as suggested by the formula **FSTAT**, (*ii*) does the **FSTAT** give a true upper bound for testability, and (*iii*) is the **FSTAT** an appropriate approximation of testability as it is measured in this dissertation (is the bound sufficiently tight).

This chapter presents an empirical study where testability is estimated as the degree of the constraints is varied in a model of a dynamic real-time system and its execution environment. The results show that some of the factors, previously identified as possibly impacting testability, do not have an impact, while others do. The innovative idea to use *p-points* together with the algorithm and the guide process described in Chapter 5 is what makes this investigation of testability possible.

Aside from constraints on the execution environment, constraints might also arise from other sources, e.g., internal dependencies such as cause-effect relations. For example, a task that is triggered by an object arriving on a moving conveyor belt will not be triggered unless something is placed on the conveyor belt and the conveyor belt is moving. Moreover, the exact time for a sensor registration of an arrival is decided by the speed of the conveyor belt and the exact point in time when the item was placed. Such time dependencies among tasks and also between tasks and the controlled environment are of course common and do affect the number of potential execution orders. Other dependencies arise from the fact that there are shared resources. Exclusive access to a shared resource implies that some interleavings are prohibited. Hence, it is likely that the actual impact on testability is much lower than the theoretical upper bound suggests.

The main idea for the experimental setup is to model an application as a dynamic real-time system. A set of triggering events and task types that respond to such events are therefore identified. Each task is assumed to be executed by a process. Internal dependencies and the execution environment together with the proposed constraints are included in the model.

The model is created in a number of variants. These differ only in the values of the parameters representing the investigated constraints. Everything else is kept fixed. A set of test scenarios (i.e., input sequences) is selected and applied to all variants. For each test scenario, testability is estimated by counting the potential execution orders in each variant with the model checker UPPAAL (Larsen et al. 1997).

The number of execution orders is likely to grow exponentially as the parameter values are varied. Hence, this experiment has to be limited to a small model. A subset of the identified tasks is therefore selected for the study. However, even though the study is conducted on a subset of all tasks, the results are still useful for showing the relation between the investigated constraints and testability since adding more tasks is not likely to reduce the impact.

## 6.1 Subject of Study

The subject of the study presented here is a control application of a steel plant, SIDMAR. The steel plant has been previously described in detail and used in several studies, e.g., by Behrmann, Hune & Vaandrager (2000), Hune, Larsen & Pettersson (2000), Boel

(2000), Boel & Montoya (2000) and Fehnker (1999). This application suits the purposes of this study well in the sense that the detailed description of the plant behavior makes it easy to derive a subset of tasks with natural dependencies among them with respect to timing and shared resources. This means that the real-time tasks in the subset have built-in constraints on their timely behavior.

### 6.1.1 Plant Layout

The physical SIDMAR plant is located in Gent, Belgium. The steel plant consists of two cranes, two convertor vessels, five machines, two normal tracks, a buffer, a storage place, a holding place, and a continuous casting machine (figure 6.1). The plant is used in case studies of scheduling problems where the goal is to find an optimal job schedule or to verify properties (Fehnker 1999). The steel plant has a number of physical components (see Figure 6.1).

*i* Convertor vessels: This is where the pig iron (raw cast iron) enters the system. It is poured portion-wise into steel ladles.

*ii* Tracks: The ladles can move autonomously along two normal tracks. Moving from one track to another requires a crane.

*iii* 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.

*iv* 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.

*v* 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.

*vi* Buffer: The buffer can hold at most five ladles and can be used to pass ladles between the cranes.

*vii* 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.

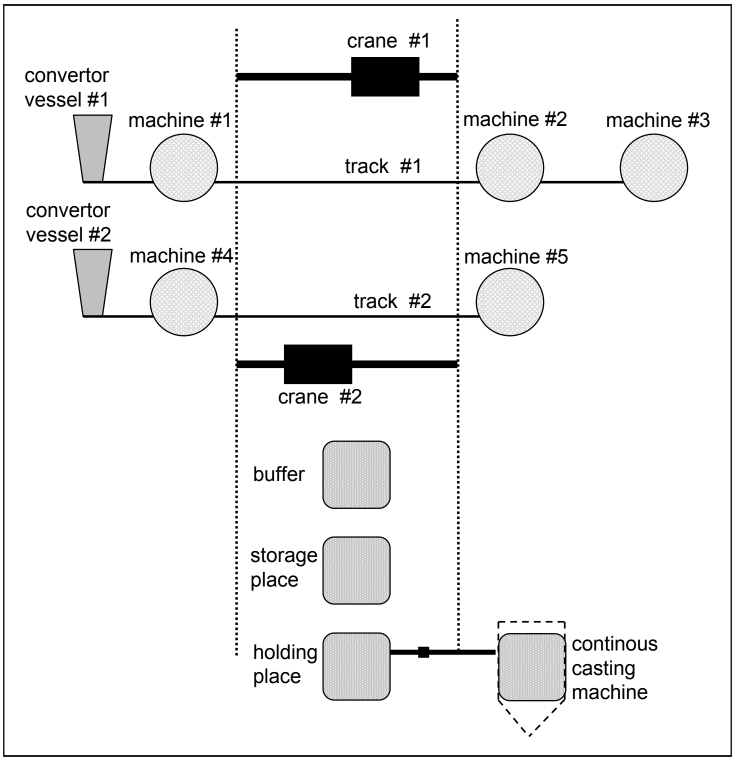


Figure 6.1: Layout of the steel production plant. The Figure is based on Fehnker (1999).

The steel plant is a typical example of a real-time system. The behavior of the controlled environment, where steel ladles arrive on conveyor belts to be handled by machines and cranes, does not differ much from e.g., any factory where components arrive on conveyor belts to be assembled by industrial robots.

## 6.2 The Timed Automata Model

A formal model of the system is developed in timed automata (TA). The model, summarized in Figure 6.2, has three parts:

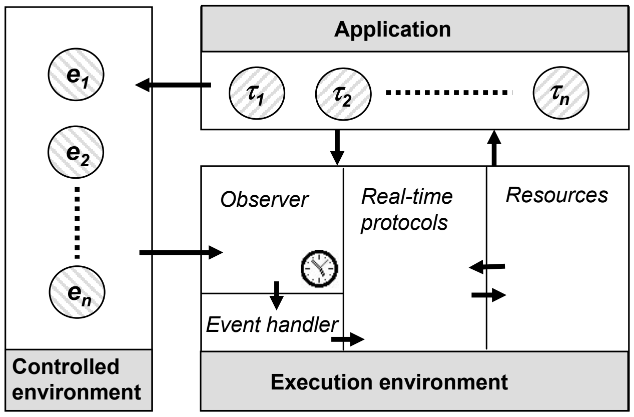


Figure 6.2: An overview of the modeled system including controlled environment, application and execution environment.

*i* Application: All tasks that are involved in the execution orders are part of the application $\tau_1$, ..., $\tau_m$, where $\tau_j$ is a timed automata process corresponding to a real-time task.

*ii* Controlled environment: This part is restricted to a set of events: $e_1$, ..., $e_n$, where $e_i$ is a timed automata process corresponding to an event (e.g., a sensor signal).

*iii* Execution environment: This part consists of an observer, a scheduler, and a resource handler. The observer observes events in the controlled environment and communicates with the scheduler.

### 6.2.1 Application

A dynamic approach is assumed for a steel plant application with small real-time tasks triggered by sensor signals. It is easy to identify a large number of task types in this application. However, a subset of these is selected to be included in the model. The reason is that the tasks are modeled in timed automata and it must be possible to elaborate with the number of concurrently executing tasks of the same type. With a large number of task types this would soon lead to an infeasible large model. The model must be sufficiently small to enable verification. Therefore, a small subset of task types are selected, i.e., four types. This means that the number of potential execution orders that this study gives is with respect to this subset of task types only. Hence, each order in the study can be viewed as a class of execution orders, where members of a class have the same order among the subset of tasks but differ with respect to interleavings with other tasks not included in this study.

The selected task types are:

*i* Start_machine. This task is triggered by a sensor signaling the arrival of a steel ladle at the end of a conveyor belt. The task starts the processing of steel by the current machine.

*ii* Stop_machine. This task is triggered when processing in a machine is finished. The steel ladle is removed from the machine and placed on the conveyor belt ready to continue to the next position.

*iii* Between_machines. This task is triggered when a steel ladle arrives at the center of track #1 or #2. A decision is taken whether the ladle should continue to next machine on the same track or the other track, or whether it should be placed on the buffer.

*iv* On_buffer. This task is triggered when a steel ladle is placed on the buffer. As soon resources are available (i.e., crane, conveyor belt, and machine) the ladle is removed from the buffer and placed on the corresponding track.

Each task type is modeled as a timed automata template.

### 6.2.2 Controlled Environment

The controlled environment in the used model is simply a set of event occurrences, i.e., sensor signals. Each event type is modeled as a timed automata process that synchronizes with the observer included in the execution environment.

In previous work with the steel plant, physical parts and their movements were modeled (Fehnker 1999). This is of no interest in this study and therefore such processes are replaced by timers. The reason for using a timer rather than ignoring the physical processes is that they take time and this imposes a time constraint on some of the tasks. For example, suppose a task *Start_machine* is triggered by an event *At_machine*. That event can only occur a certain time after a ladle has been placed on a track and the conveyor belt started to

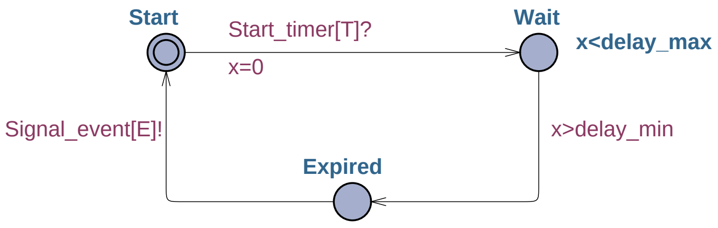


Figure 6.3: A timer that ensures timing constraints among two tasks. A local clock $x$ is reset when the timer is started. The timer expires after a minimum of $delay_{min}$ and $delay_{max}$ time units.

move. Modeling movements with timers makes it possible to maintain such constraints. Moreover, the timers make it possible to simulate a more unpredictable environment since the time constraint is likely to be a short time interval rather than an exact point in time. For example, consider a ladle placed on a conveyor belt at time $t$. The corresponding event *At_machine* will occur when the ladle has been transported by the conveyor belt to the end of the corresponding track where a sensor is placed. The sensor signal informing the system of the arrival will be given sometime in the interval $[t + delay_{min}, t + delay_{max}]$. A TA process implementing such timer is shown in figure 6.3.

A test scenario in this application is a sequence of steel ladles that enter the system at certain points in time. The ladles are placed on one of the tracks leading to machine #1 or #4. In actual operation, when a ladle reaches one of the machines an event is generated. In the model, this arrival is simulated by starting a timer (start node in Figure 6.3) when a ladle is placed on the track. The timer expires after a time interval that corresponds to the time it would take for the ladle to reach the machine.

### 6.2.3 Execution Environment

In addition to the application and the controlled environment, the execution environment is included in the model. To investigate how the execution environment affects testability, the selected execution environment constraints must be implemented. Other constraints that arise naturally due to scheduling, resource handling, precedence, etc must also be considered. This is therefore included in the model.

A simple scheduler is added to the model. The scheduler keeps all active tasks in a ready queue and permission to proceed in execution is only given to the task that is first in line of that queue. The currently used scheduling policy is earliest deadline first (EDF). The goal of this experiment is not to compare different scheduling policies and evaluate them or to guarantee deadlines. Hence, a sophisticated scheduler is not needed. It is sufficient with a simple scheduler that selects tasks according to priorities derived from their deadlines since a deadline driven policy is likely to be used in a dynamic real-time system. A task is inserted into the ready queue when it is triggered or when it is unblocked. A task is removed from the queue when it is done or blocked. If the currently executing task has a later deadline than a task entering the queue, the current task is preempted at the next preemption point. Preemption points are equally distributed over the time of task execution. This means that there is a known maximum time delay associated with each preemption point.

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. Observation points are equally distributed over time. This means that there is a known maximum time delay from an event occurrence to the point in time when a corresponding task is triggered.

Tasks are triggered on event observations and scheduled according to their deadlines in an earliest deadline first manner. Execution of tasks is conducted in non-preemptive intervals separated by the designated preemption points. If there is a task in the ready queue with higher priority than the task that is currently executing, the current task is preempted on arrival at next preemption point as long as the number of preemptions a task can experience is unbounded. However, if the number is bounded to $MAX\_P$ and the task already has been preempted $MAX\_P$ times, other actions must be made to maintain the constraints. For example the task can be aborted or continue its execution in a non-preemptive mode.

To simulate execution of tasks that share resources, a resource handler is implemented. The resource handler manages a queue for each shared resource and the queue policy currently used is first-in-first-out (FIFO). Hence, this resource handler implements the same behavior as a set of FIFO semaphores, one for each shared resource. The policy can easily be altered to give priority according to deadline or criticality.

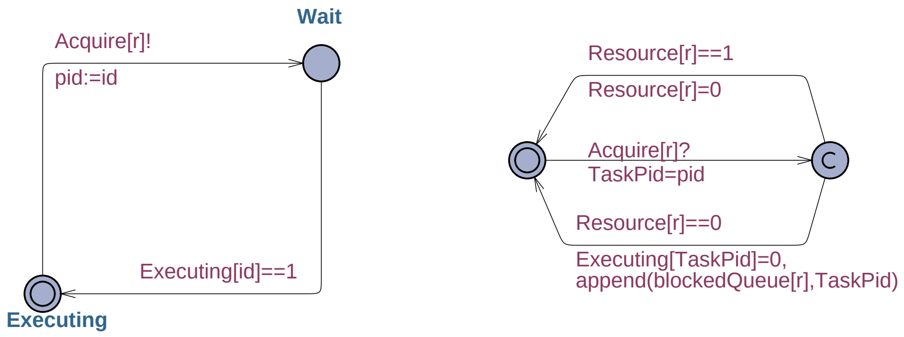


Figure 6.4: This pattern occurs when a task tries to allocate a shared resource. The task synchronizes with the resource handler on a channel associated with the resource. No response is required. If the task is blocked, the condition for executing is falsified until the task has allocated both the resource and the CPU.

A task that tries to allocate a shared resource is blocked if the resource is not free (i.e., the semaphore is 0). The resource handler gives the resource to a blocked task, if any, when the resource is freed. If no tasks are blocked on the freed resource, the semaphore is set to 1.

The patterns that occur in a task when allocating or de-allocating resources are shown in figures 6.4 and 6.5. Figure 6.6 shows the simplified task model from Chapter 5 extended with these patterns. Full descriptions of all variables, channels and functions used in the figures are given in tables 6.1 to 6.4.

The basic semantics in these designated preemption points is that when the scheduler decides to preempt a task, the task is not preempted immediately. Instead, the preemption occurs when the task reaches its next preemption point. This means that execution of the task can be viewed as execution of a number of non-preemptive segments. The longer these non-preemptive segments are and the lower the number of allowed preemptions is, the lower the number of potential interleavings is.

The pattern for preemption points used in the model is shown in figure 6.7. The point consists of three states, $NonPreemptable$, $Preemptable$ and $Wait$. The condition to continue from $Wait$ is set to true when the task is dispatched and set to false when the task is preempted, blocked or finished. The task stays in $NonPreemptable$ for $ExecutionInterval$ time units and then moves

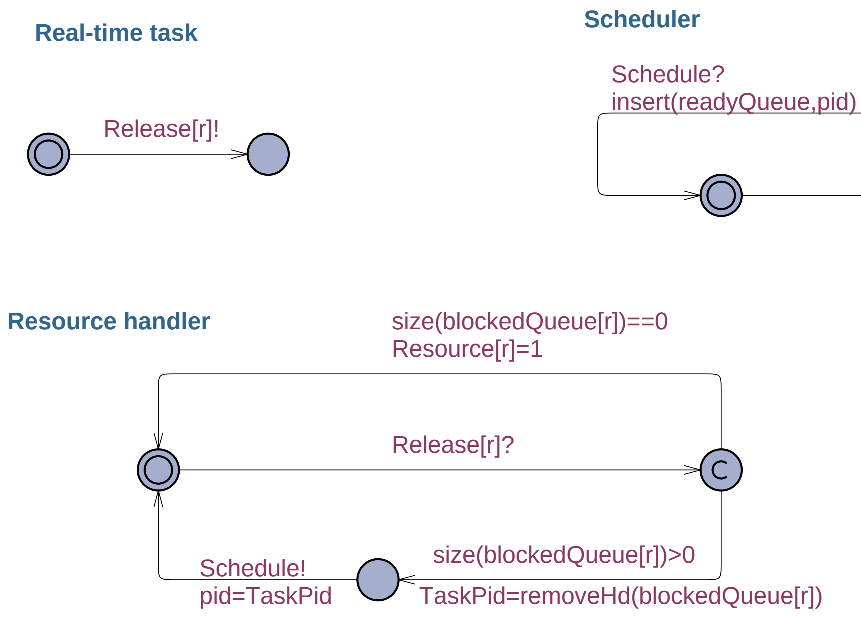


Figure 6.5: This pattern occurs when a task is deallocating a resource. The task synchronizes with the resource handler on a channel associated to the resource. No response is required. If tasks are blocked, the first task is removed from the queue and the scheduler is informed.

to state $Preemptable$. A preemption will occur when the task is in state $Preemptable$ if two conditions are fulfilled;

*(i)* The scheduler tries to synchronize with the task on channel $Preempt[id]$

*(ii)* The task has not yet been preempted $MAX\_P$ times

If a preemption occurs, the task moves to state $Wait$ until it is scheduled again. If the task is not preempted, the task will move to state $NonPreemptable$ after one time unit and continue its execution for another $ExecutionInterval$ time units. Figure 6.8 shows the simplified task model extended with a designated preemption point.

In the model, the remaining execution is considered non-preemptive when the task has reached a maximum number of preemptions. This is probably not desired for all applications, which means that when such situations occur it might be necessary to abort the task in favor

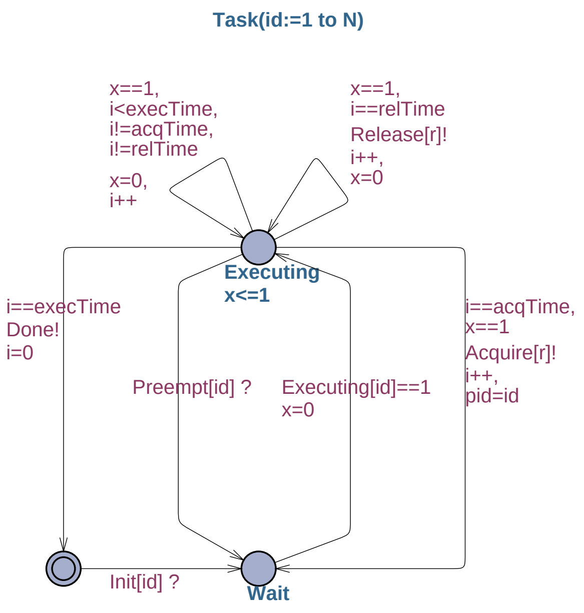


Figure 6.6: The simple task model extended with the pattern for resource handling.

of executing a more critical task. Another possibility, if the worst case execution time is known, is to ensure that the number of preemption points equals the maximum number of allowed preemptions.

For each kind of event $E$ and associated task type, there is a timed automata process, Observer, shown in Figure 6.9. When an event occurs, the observer process receives it and waits for condition $O\_flag == 1$, which is satisfied as soon as the next observation point is reached. As soon as the condition is true, the time for observation is set. Thereafter, the task is triggered and the scheduler notified. A new event of the same event type cannot occur until the local clock $x$ is incremented. This means that two events of the same type can never have the same time stamp. This agrees with the assumptions made by Birgisson et al. (1999). The process setting the flag for observations is shown in Figure 6.10.

Each task type is modeled as a template in timed automata. The number of instances of each template is determined by the system assignments and definition. Hence, there's no need for a special design pattern for this constraint. All that is needed is to instantiate the number of tasks.

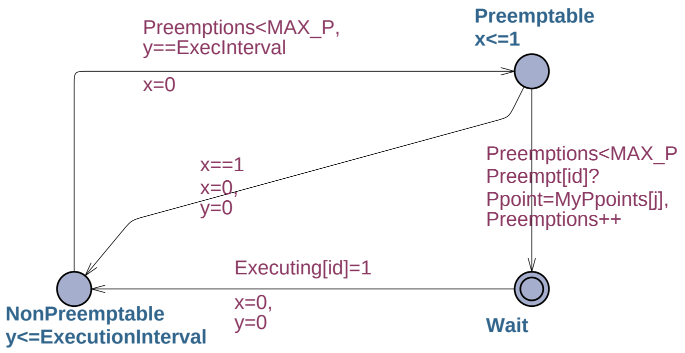


Figure 6.7: This is the pattern for a preemption point in a timed automata process.

Table 6.1: Global variables used for communication.

| Name | Description |
|---|---|
| pid | Parameter used to communicate a task identity. |
| r | Parameter used to communicate a resource identity |
| Ppoint | Parameter used to communicate a preemption point identity |
| Executing | Executing[$i$]==1 if task $i$ has access to the CPU. Executing[$i$] is set to 0 whenever task $i$ is preempted, blocked or finished. |
| ObsInterval | Constant defining the observation granularity |
| O_flag | Flag that is set to 1 when arriving at an observation point and to 0 when leaving the observation point. |
| T | Integer that keeps track of global time |

## 6.3 Creating Variants

A timed automata model of the set of tasks from the steel plant application was implemented as described in 6.2. A number of variants were then created from this model. Each variant differs from the original model only with respect to a set of controlled

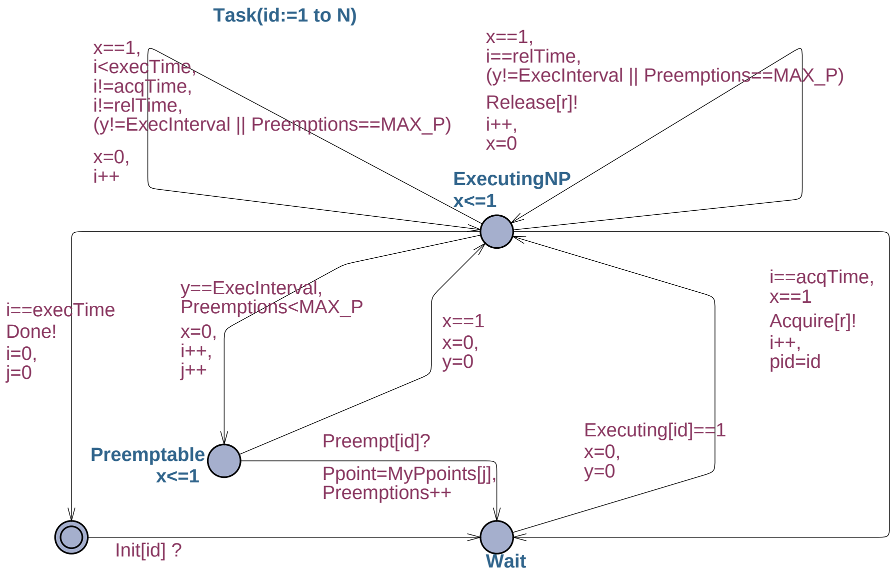


Figure 6.8: The simple task model extended with designated preemption points and resource allocation.

variables. These variables represent the constraints on the execution environment that the experiment aims to investigate, i.e., $p$, $s$ and $q$ in Formulae 3.1 to 3.4. Inherited constraints from the application (e.g., causal ordering, execution time, resource blocking time, etc.) remain fixed.

The first controlled variable defines an upper bound on the number of preemptions that a task can experience. As described above, preemptions are restricted to designated preemption points. Execution is performed in non-preemptable intervals between these points. The constraint gives a defined maximum on the number of preemptions for each task. Preemption points are included in the model according to the pattern shown in Figure 6.7.

Preemptions are only allowed in certain states. Preemption will only occur at next preemption point if the dispatcher tries to preempt the task and the condition for preemption is true. The condition ensures that there may be at most $MAX\_P$ number of preemptions, where $MAX\_P$ is a constant that can be set to different values to study the effect of constraining preemptions as suggested by Birgisson et al. (1999).

The second controlled variable defines the observation granularity.

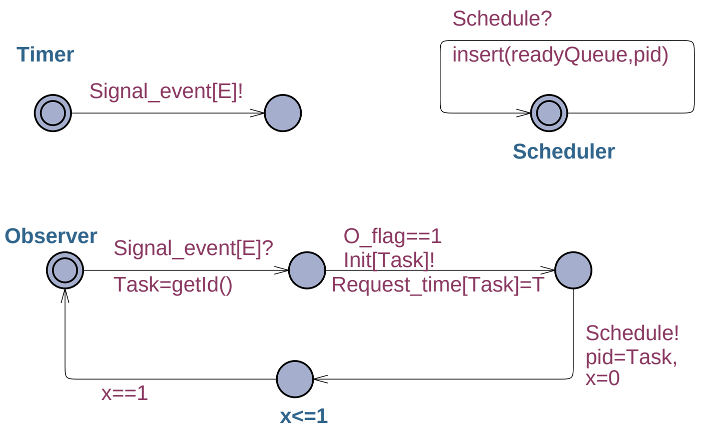


Figure 6.9: The timed automata process Observer ensures that observations are delayed until the next predefined observation point where the Observer synchronizes with the Scheduler.

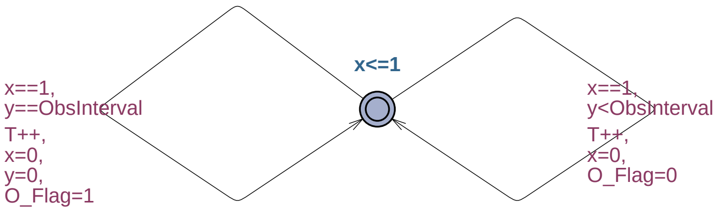


Figure 6.10: A timed automata process which keeps track of time pacing and sets the flag O_Flag, which is true at observation points and false otherwise.

Whenever an event occurs, the observation of that event is delayed until the condition $O\_Flag == 1$ is satisfied (i.e., an observation point). The request time for the triggered task is set according to the time of observation, not the time of occurrence (figure 6.9). Hence, the potential request time for tasks is limited to the set of observation points. The flag *O_Flag* is set to 0 after 1 time interval.

The distance between the observation points is decided by the value of a constant *ObsInterval* (see Figure 6.10). By setting *ObsInterval* to different values, it is possible study the effect of constraining observations as suggested by Birgisson et al. (1999).

The third controlled variable defines the maximum number of concurrently executing tasks of the same task type. Each task type is

Table 6.2: Local variables and clocks.

| **Name** | **Description** |
|---|---|
| x and y | Clocks |
| i and j | Counters |
| id | Task identity, local to the task |
| MAX_P | Constant that defines the maximum number of preemptions allowed |
| execTime | Constant that defines the execution time for the task |
| acqTime | Constant that defines the point in execution where a request for a resource is made |
| relTime | Constant that defines the point in execution where an allocated resource is released |
| Preemptions | Counter for the number of preemptions the task has experienced |
| ExecInterval | Constant defining the length between two consecutive preemption points |
| MyPpoints | Array containing preemption point identities |
| TaskPid | Task identity, local to the resource handler |
| Resource | Array of resource identities |
| readyQueue | Queue of tasks ready for execution, local to the scheduler |
| blockedQueue | Array of queues. blockedQueue[$r$] contains tasks blocked on resource $r$ |
| delay_min, delay_max | Constants that define the bounds of a delay, i.e., the timer expires after a delay that is at least delay_min and at most delay_max time units long. |
| Request_time | Array that keeps track of the request time for each task. This is used to derive the task deadline |

modeled as a template in timed automata. The number of instances of each template is determined by the system assignments and definition. An instance of a template simulates the behavior of a specific task type in a loop. The loop starts when the task is triggered and ends

Table 6.3: Channels used for synchronization.

| **Name** | **Description** |
|---|---|
| Start_Timer | Timer $T$ starts when it receives a signal on channel Start_Timer[$T$] |
| Signal_event | Synchronization on this channel simulates a sensor signal. The observer process associated with event $E$ will trigger a task at the next observation point whenever receiving a signal on channel Signal_event[$E$] |
| Init | Task $i$ is triggered when it receives a signal on channel Init[$i$] |
| Done | A signal is sent on this channel when a task finishes its execution and the schedule must be updated |
| Preempt | Task $i$ is preempted when it receives a signal on channel Preempt[$i$] |
| Release | Resource $r$ is released when the resource handler receives a signal on channel Release[$r$] |
| Acquire | Resource $r$ is requested when the resource handler receives a signal on channel Acquire[$r$] |
| Schedule | The schedule is updated whenever the scheduler receives a signal on this channel |

Table 6.4: Functions used in figures to reduce complexity.

| **Name** | **Description** |
|---|---|
| append | Appends a task identity at the end of a queue. |
| removeHd | Removes the head of a queue and returns it. |
| size | Returns the size of a queue. |
| insert | Inserts a task identity in a queue according to the priority |

when the task is done.

It is not possible to have more instances running than defined. A new task cannot start unless there is a template of correct type, which is ready to start. Hence, the number of template instances defined in the system decides the upper bound for concurrently executing tasks of that type. Hence, if the upper bound is reached, the task cannot be triggered until one of the running tasks is finished. Important to note is that this approach is not necessarily suitable for all applications. For instance, it might be better to reject the task or to abort a running task to trigger a new one. This is however, considered out of scope for this study.

## 6.4 Measured Effect on Testability

This section presents the experimental results. The effect on testability from each of the investigated constraints is estimated by varying the controlled variables and measuring the effect on the number of potential execution orders. Section 6.4.1 describes the effect of observation granularity on the size of the input domain while sections 6.4.2 to 6.4.4 present the results from the experiments.

### 6.4.1 Time-triggered Observations and the Input Domain

In the study, it is assumed that the size of the input domain is of less interest from a testability perspective compared to predictable execution orders. However, previous work on the relation between observation granularity and testability focused on the size of the input domain (i.e., the number of potential event sequences). It is clear from the following example that the size of the input domain is affected by the observation granularity.

This example assumes time-triggered observations and shows the effect on the number of potential event sequences when varying the granularity of the observation interval. Consider the following two input sequences:

ES 1 The set of input events is $\{e_1, e_2, e_3\}$. The set of triggered tasks is $\{\tau_1, \tau_2, \tau_3\}$. The actual time of occurrence for the events are $time(e_1) = 2.5$, $time(e_2) = 13.5$ and $time(e_3) = 19$.

ES 2 The set of input events is $\{e_1, e_2, e_3\}$. The set of triggered tasks is $\{\tau_1, \tau_2, \tau_3\}$. The actual time of occurrence for the events are $time(e_1) = 4$, $time(e_2) = 16$ and $time(e_3) = 18.5$.

The only difference between ES 1 and ES 2 is the time of the actual occurrence of the three events. The time stamps that are assigned to these events are the time of observation. With the fine observation granularity that is used in Figure 6.11 there are two significantly different set of time stamps. For ES 1 there are $obs\_time(e_1) = 5$, $obs\_time(e_2) = 15$ and $obs\_time(e_3) = 20$. For ES 2 there are $obs\_time(e_1) = 5$, $obs\_time(e_2) = 20$ and $obs\_time(e_3) = 20$.

As shown in Figure 6.11 the difference with respect to observation time affects the time when task $\tau_2$ is initiated. Task $\tau_2$ is initiated later in the second case. Hence, it will have a later deadline. This might lead to a lower priority and therefore possibly another placement in the priority ordering of the tasks. A different priority ordering implies a different execution order.

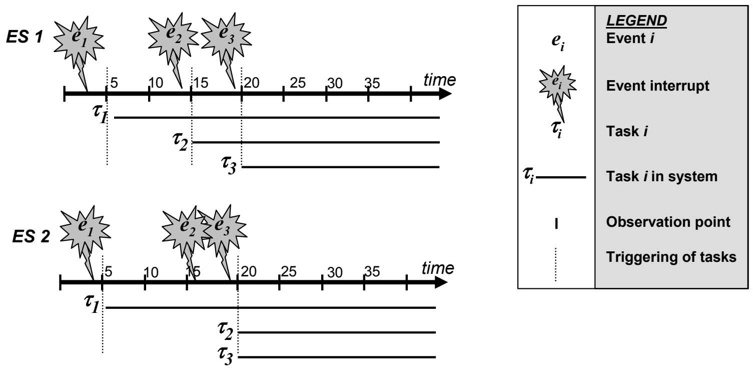


Figure 6.11: Two different input sequences, ES 1 and ES 2. The fine observation granularity gives two different behaviors for ES 1 and ES 2.

Now assume that the time granule for observations is doubled, that is, instead of observing the environment every fifth time unit it is now observed every tenth time unit. As shown in Figure 6.12, the sets of time stamps for ES 1 and ES 2 are now identical, $obs\_time(e_1) = 10$, $obs\_time(e_2) = 20$ and $obs\_time(e_3) = 20$. The time when task $\tau_2$ is initiated is the same for ES 1 and ES 2 and so is the deadline.

Using fixed observation points thus leads to potential input event sequences being separated into equivalence classes. Each event sequence from such an equivalence class will be observed by the system and reacted upon as being the same event sequence.

The magnitude of this impact depends on the granularity of the

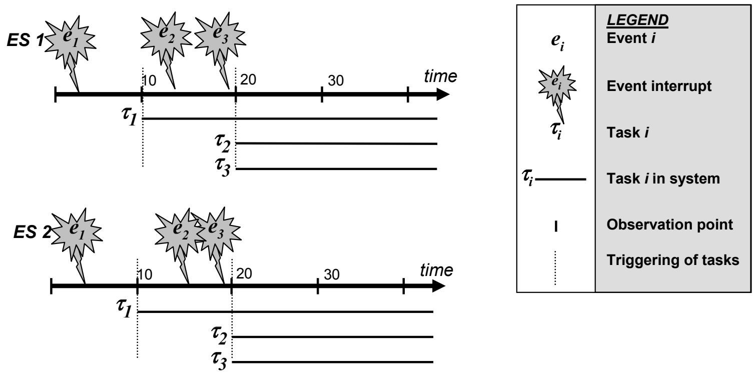


Figure 6.12: The same input sequences, ES 1 and ES 2, as in Figure 6.11. The coarse observation granularity gives identical behavior for ES 1 and ES 2.

observation interval. Hence, to limit the size of the input domain, observation granularity should be as coarse as possible. There is, of course, a trade-off decision between testability and flexibility. However, it is important to choose a granularity that is as coarse as possible, given all other considerations.

### 6.4.2 Time-triggered Observations and Execution Orders

If the goal is exhaustive testing, the large input domain implies low testability. If the goal is to execute a set of selected test cases for testing timeliness, then the relation is less obvious. The investigation described here does not focus on the impact on number of potential input event sequences (and thereby the size of the input domain for test cases). This impact should be clear from the previous discussion in Section 6.4.1.

The focus in this investigation is the impact on the number of resulting execution orders for some given event sequences, selected for testing. We refer to such event sequences as *scenarios*. The selection of scenarios is made by compromising the interest of finding "difficult" test cases to stress the system as much as possible with the limitations given by the state space explosion problem.

The impact on testability from parameter $s$ from Formulae 3.2 has been explored in four different scenarios by setting the observation granularity *ExecInterval* to 1, 2, 4, and 5 given a fixed time interval

(2500 time units). This gives the measure points $s == 500$, $s == 625$, $s == 1250$, and $s == 2500$. The results show that testability is affected by $s$. Figures 6.13 and 6.14 show the effect on the number of potential execution orders for the four scenarios.

Each scenario give a linear growth of the number of execution orders as $s$ is increased. The fourth scenario gives a higher number of execution orders, which is explained by the fact that this scenario contains four input events while the other three only contain three, i.e., $n$ is assigned a higher value. However, $n$ was never fully investigated in this study and therefore, no further results with respect to $n$ is presented here.

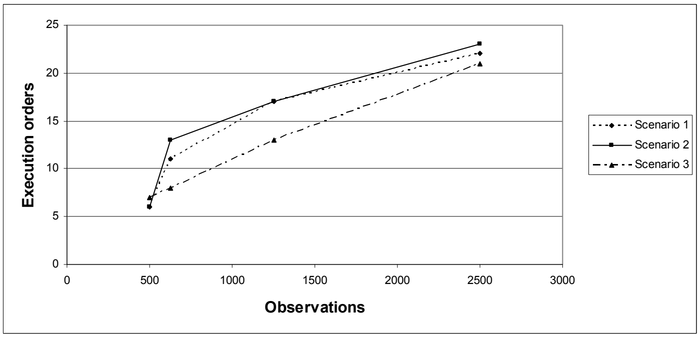


Figure 6.13: Measured effect on number of execution orders for three different scenarios with four measure points each, i.e., $s == 500$, $s == 625$, $s == 1250$, and $s == 2500$.

The results in figures 6.13 and 6.14 suggest that parameter $s$ has a linear impact on testability. Since the input domain grows exponentially and it is not likely that the number of interesting scenarios decreases with a growing input domain, the conclusion is that this parameter does affect testability and that predefined observation points with a coarse granularity therefore is a good candidate for increasing testability.

### 6.4.3 Designated Preemption Points

Schütz (1993) highlighted preemptions as being one of the factors leading to low testability of event-triggered systems in comparison with time-triggered systems. Moreover, the approach Birgisson et al. (1999) suggested to increase testability includes the use of *designated*

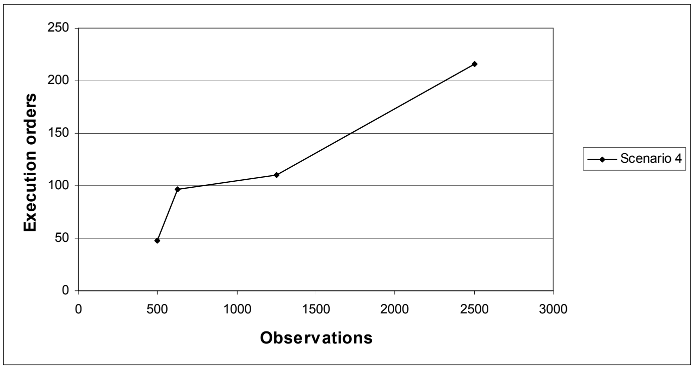


Figure 6.14: Measured effect on number of execution orders for the fourth scenario with the same four measure points as in Figure 6.13.

*preemption points* and an upper bound on the number of preemptions a task can experience during its execution. A designated preemption point is a predefined point where a task can be preempted from the CPU. These preemption points can be predefined in time, space, or both.

This section describes the results concerning the effect of the number of allowed preemptions ($MAX\_P$, or parameter $p$ in Formulae 3.3) on testability. The expected impact on testability according to the formulae is exponential $(p+1)^{\#tasks}$.

The scenarios used in previous experiments were investigated with little result. Test results from the initial four scenarios showed no impact on testability from this parameter. A closer analysis of the execution showed that none of the selected scenarios were appropriate to investigate this parameter. The reason was that in these scenarios, with their small number of events, there were very few preemptions even when an unlimited number was allowed. The scenarios were selected to implement a "nice" behavior to keep a limited size of the state space. One of the drawbacks with these scenarios was, however, that none of the executions reached the specified maximum of preemptions. Hence, a variation of the value of this parameter did not affect the behavior for the used scenarios.

To study the effect of the constraint a scenario with a potentially high number of preemptions was needed , 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. Reaching the limit requires frequent interrupts of short urgent tasks. Therefore a different type of test scenario was selected to study the effect on testability from this parameter. In these scenarios there are two long-running low-priority tasks and a burst of short high-priority tasks causing frequent interrupts. Three scenarios of this type were examined. The results from varying parameter $p$ in this scenario are shown in figures 6.15 through 6.17.

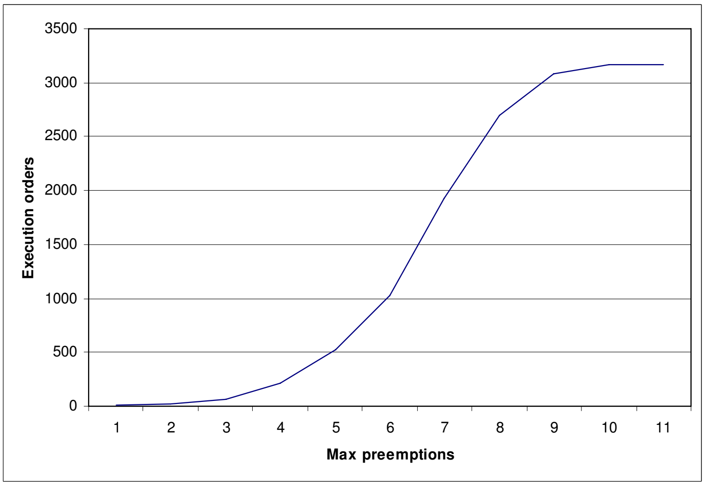


Figure 6.15: Result from scenario 1 when varying $MAX\_P$ between 1 and 11.

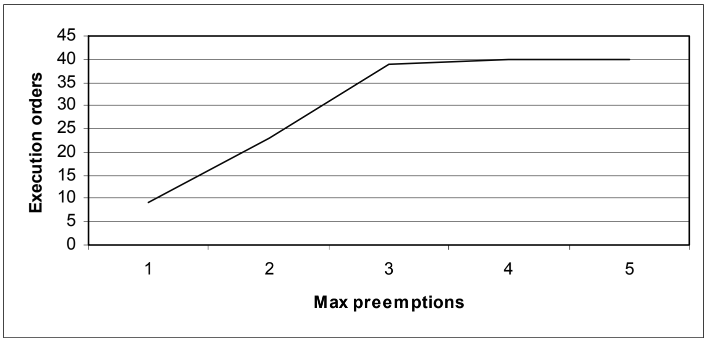


Figure 6.16: Result from scenario 2 when varying $MAX\_P$. In this scenario it is not possible to get more than 3 preemptions per involved task.

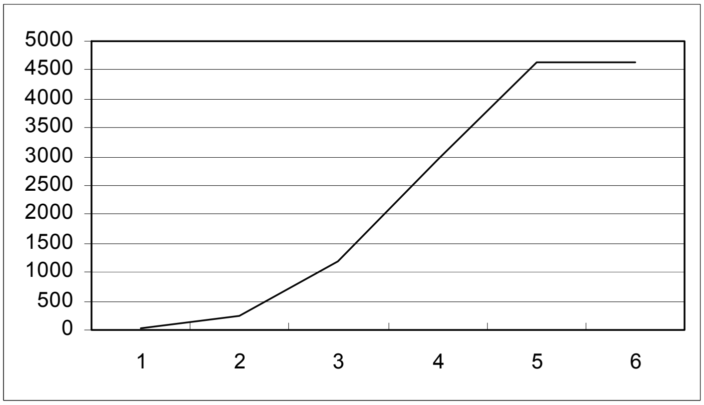


Figure 6.17: Result from scenario 3 when varying $MAX\_P$. In this scenario it is not possible to get more than 5 preemptions per involved task.

The maximum number of preemptions is varied from 1 to 11 and as shown in Figure 6.15 through 6.17, the number of execution orders increase rapidly until it stabilizes. The point of saturation is determined by the arrival pattern of the different tasks involved in the burst. For each such scenario, there is an upper limit on the number of preemptions that the involved tasks may experience even without an explicit constraint.

It is clear from the results that the maximum number of preemptions has a significant impact on the number of execution orders. The increase is exponential at first until a point of saturation (see Figure 6.18). In the study, this point of saturation occurs after a few preemptions, implying that constraining the number of allowed preemptions might give little impact on testability. However, this heavily depends on the choice of scenarios. There are a number of things to consider here:

*i* Many scenarios do not lead to race conditions and will only give one single execution order disregarding the value of $p$

*ii* Selection of scenarios in the experiment is a trade-off between worst case scenarios with many race conditions and what is feasible to verify with a model checker

*iii* Timeliness testing however, selects worst case scenarios that are likely to be more affected by the value of $p$ than can be shown in

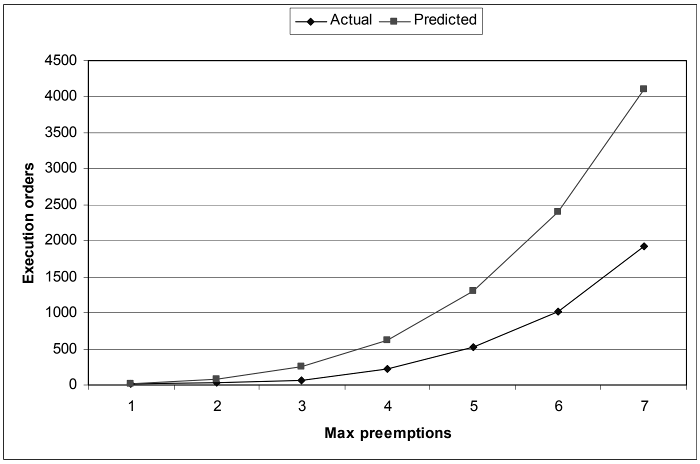


Figure 6.18: Zoomed in result from scenario 1 when varying $MAX\_P$ between 1 and 7. The actual result is compared with the predicted result given the parameter settings used.

the limited experiments

The first observation is that the behavior of the system with respect to execution orders is not completely arbitrary. There is EDF scheduling, which means that the scheduler is predictable unless two tasks have the same deadline. There are FIFO queues on resources, which means that resource allocation is predictable as long as there is no race condition, i.e., two tasks trying to allocate a resource at the exact same point in time. Hence, a random selection of test scenarios would, to a high extent, generate scenarios with only one potential execution order and few (if any) preemptions per task. For these scenarios an upper bound on the number of allowed preemptions would have little or no impact at all.

The second observation is that the selected scenarios are far from worst case scenarios. The choice of test scenarios in the study is a trade-off between the intention to stress the system with a burst of events and the limitation imposed by model checking. Therefore the test scenarios were selected to be 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.

Only two types of tasks are involved in the scenarios used and the load is never higher than four tasks executing concurrently in these scenarios. The number of preemptions a task can experience is therefore low. This is also the reason for why the point of stabilization is reached so early. The difference between the three scenarios with respect to execution orders is due to different arrival patterns of the involved tasks.

The final observation is that when the purpose of testing is verification of timeliness, the tester is likely to select test scenarios that will stress the system as much as possible. This means that there will be a high load of concurrently executing tasks and there will be bursts of arriving tasks. For each of these scenarios, the number of potential preemptions (if not limited by a constraint) will be significantly higher than those shown in the study.

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 (Birgisson et al. 1999, Lindström, Mellin & Andler 2002) and suggests that an upper bound on the number of preemptions a task can experience is a good candidate for increasing testability.

### 6.4.4 Tasks of Same Task Type

This section presents the results concerning the effect of the number of concurrently executing tasks of the same type (parameter $q$ in Formulae 3.3 and 3.4) on testability. The same four scenarios used to investigate the impact from observation granularity were checked to investigate the impact from concurrency. Parameter $q$ was varied between 1 and 3. The experiments found no impact from this parameter. Several explanations for this are possible:

*i* The selected scenarios might not be appropriate to provoke multiple execution orders.

*ii* The selected policies for scheduling and resource handling might not be appropriate for investigating this parameter.

*iii* The parameter has little or no impact on testability.

It is hard to argue that the parameter has no effect on testability just because no scenario with a shown impact was found. However, there are intuitive reasons to believe that this parameter is less interesting from a testability perspective:

*i* The timed automata model uses an EDF scheduling policy. A real system might use another policy but a dynamic real-time system bases its scheduling decisions on the tightness of the deadlines and/or the importance (criticality) of the tasks.

*ii* Importance of a task is generally decided by the task type and its associated value function (i.e., the penalty of a missed deadline).

*iii* Deadline of a task is decided by the type of triggering event and the time when the event was observed by the system.

*iv* Two events of the same type cannot be observed at the same point in time according to the underlying assumptions for Formula 3.2 (Birgisson et al. 1999).

Hence, if several concurrently executing tasks of the same type are present, it is reasonable to assume that 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. Maybe it is possible to provoke an execution where a task preempts a previously triggered task of the same type, e.g., with another locking protocol and specific blocking scenarios. However, locking protocols, such as the Stack resource protocol (Baker 1991), 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.

## 6.5 Threats to Validity

When conducting an empirical study like the one described in this chapter, there are always a number of threats to the validity of the study. If these threats are left unaddressed, the results may not be valid. Threats to validity can be of different types (Cook & Campbell 1979). This section identifies and discusses how these threats are addressed by the study.

*Construct validity* concerns whether or not the experiment measures what is believed to be measured. Testability is a property that cannot be measured directly. It is therefore necessary to make an approximation of testability using some other metric. A decision is made to use the number of execution orders as such a metric. Chapter 4 includes a survey of different views on testability and motivates the choice of using the number of execution orders as an approximation for testability when the system is dynamic. Since we cannot directly measure testability, the answer to another question, whether we actually measure the number of execution orders becomes crucial.

The execution orders are collected using the technique described in Chapter 5. Each time a task is dispatched, the scheduler synchronizes with a guide automaton. Dispatches are made when the task starts execution and when a blocked or preempted task resumes its execution. An execution order that is found will therefore contain a series of preemption points that indicates all points in execution where tasks are dispatched. All preemption points have **unique identities**. For example, assume two tasks $p_1$ and $p_2$, where $p_1$ has preemption point identities $\{p_1^1,p_1^2,...,p_1^m\}$ and $p_2$ has preemption point identities $\{p_2^1,p_2^2,...,p_2^n\}$. A generated execution order, e.g., $\{p_1^1, p_2^1, p_1^3, p_2^5, p_1^7\}$, can be interpreted as $p_1$ executes from start to $p_1^3$, followed by $p_2$, which executes from start to $p_2^5$, followed by $p_1$, which executes from $p_1^3$ to $p_1^7$, followed by $p_2$, which executes from $p_2^5$ to the end, followed by $p_1$, which executes from $p_1^7$ to the end. Extensions to the orders during the search for unique execution orders are only made as a result of a dispatch. Hence, it is not possible that preemption point identities are inserted into the order unless it is a point where the task gets access to the CPU.

Correctness of the algorithm is discussed in Section 5.6. The algorithm was verified by hand for a set of small models and automatically for a set of larger models. The automatic verification was made by using an oracle, i.e., a redundant implementation. We ran both the original program and the oracle on nine different models and compared the sets of generated execution orders. The generated sets were identical in all cases.

The main problem of using redundant implementations as described is that failures might not be independent. Hence, there is always a certain risk that both the implementations contain the same fault and therefore agree on an erroneous result. In such case the failure would be masked and not possible to detect by comparing the results. To minimize this risk, the oracle generates the execution orders using a different algorithm, $\beta$. $\beta$ uses p-points and iterative invocations of the model checker but the similarity between the two algorithms stops there. $\beta$ uses another search mechanism, *potentially always* (E[] $\phi$ in UPPAAL). This property holds if there is a path where $\phi$ is true in every state. The search is therefore guided by the query instead of a special guide process. Hence, $\beta$ does not need any guide process nor any modification of the model between the invocations.

*Internal validity* concerns whether or not the results are affected by additional factors that are not accounted for, i.e., factors other than the manipulated variables. The experiment is controlled in the sense that the models are identical apart from the manipulated variables and all settings for the model checker is the same for each invocation. Hence, if there is a difference, this can only depend on the manipulated variable. However, due to the state space explosion problem, there is always a risk that the model checker halts before the state space is fully explored. If this happens, there might be unidentified orders. This problem is handled by the algorithm described in Chapter 5. The algorithm mitigates the state space explosion problem by guiding the search into those parts of the state space where the search may be successful. Moreover, the state space tends to grow as the unpredictability increases. Hence, the probability that the algorithm would occasionally miss an additional extension to a found order is higher when the manipulated variable is set to a higher (more allowing) value. Since the experiment shows that the number of orders increases as the manipulated variable is set to a higher value, the conclusion is that the actual increase in number of orders is at least as high as the observed increase.

*External validity* concerns the generalization of the results, i.e., whether the investigated object is representative enough to allow general conclusions. The steel plant used in the experiment is a typical environment for a real-time control application in industry. Steel ladles arriving on conveyor belts and handled by machines and cranes is to a large extent similar to objects arriving on conveyor belts in a factory and assembled by industrial robots. The steel plant only defines the controlled environment of the real-time system. This means that all information about the timing and ordering among different events and actions are taken from the steel plant itself. Another environment would give other test scenarios, which might affect the results with respect to number of execution orders. However, the steel plant provides a rather predictable environment compared to, for example a telephone switch. Since an unpredictable environment increases the probability of having a race condition, the author concludes that the effect of constraining the execution environment shown by the experiment, is likely to be even bigger in a more unpredictable environment.

The choice of policies for scheduling and resource handling might affect the results. Dynamic scheduling is necessary in dynamic systems and dynamic priorities should be set based on deadline, remaining slack time, or some value function. Policies for resource handling can be based on task priorities or e.g., a simple FIFO queue. For the experiment, the choice is to keep it simple using earliest deadline first for the processor and a FIFO queue for shared resources. These choices should not be controversial since FIFO semaphores are commonly used for shared resources. However, a future direction in this work should be to investigate the impact on testability from different policies.

# Chapter 7

# Constrained Dynamic Systems

In this chapter the impact from the execution environment constraints on the system semantics is discussed to determine whether the dynamic semantics of the event-triggered system is maintained when the constraints on the execution environment is applied.

The design types, event-triggered and time-triggered, for real-time systems are described and compared above in Chapter 2.2. Instead of comparing the types, the focus now lies on the discussion about the semantic characteristics of the event-triggered type.

The first step is to establish a set of significant properties of an event-triggered system from a semantic perspective. Each listed property is then analyzed with respect to the behavior of the model used in the study, described in Chapter 6, and the applied constraints on the execution environment in the same model.

All of the properties that are listed below are known from previous work and can be found in different sources. For example Kopetz (1991) gives a thorough description of the event-triggered design and its implications on the semantics.

S1 The arrival pattern of tasks is often unpredictable and the computer system does therefore, not necessarily have any knowledge about future arrivals

  S1a The resource needs, in terms of processor, communication media, data items, etc., for a task is not necessarily known before the task is triggered

- S1b There are no pre-allocated slots for the tasks to use
- S1c Support for dynamic task creation possible
- S1d Conflicts concerning competing tasks is solved dynamically

S2 The execution is not performed in cycles

- S2a The system does not necessarily return to an initial state before new event occurrences
- S2b Dynamic preemptive scheduling is required

S3 Response is given as soon as possible

- S3a Variations in execution time for a task type is visible
- S3b The start point for execution of a waiting task depends on actual execution time of other tasks

S4 An extension of the system does not require a recalculation of the static schedules

## 7.1 Resource Requirements, S1

A common motivation for the event-triggered design is an unpredictable environment. When the environment is unpredictable, so is the arrival pattern of tasks. There might be bursts of events which must be handled in a timely manner. One of the typical properties of the event-triggered design is that knowledge about future arrivals is not necessary. The introduction of the execution environment constraints in the model did not change that. The scheduler that is used in the model had no knowledge about future arrivals.

For scheduling the study described in Chapter 6 used an earliest deadline first policy on the processor and a first-in first-out policy for other resources. Hence, there were no pre-allocated slots for the tasks to use. Neither the scheduler nor the resource handler had any information about future needs. Also, variations with respect to the resource needs in terms of processor and items were to some extent included in the model. Conflicts with respect to race conditions were resolved dynamically according to the policy for scheduling and resource handling. The properties S1a, S1b and S1d are therefore met.

The answer to whether S1c is met in the study must however be negative. The reason is that the model checker does not support dynamic task creation. Dynamic task creation means that the tasks are created at run-time and not at compile time. Timed automata templates were used to model the different task types and the maximum number of concurrently executing tasks were defined by the number of instances of each template. It is not possible to create a new instance at run-time. This is exactly the same semantics as having a pool of threads created at compile time.

S1c was not shown in the study and the question is whether it is possible to maintain S1c in a constrained event-triggered system. It is the authors opinion that one of the execution environment constraints does not support S1c. There are arguments for why it is likely that this property cannot be met in a system where all of the investigated execution environment constraints are applied.

There are intuitive reasons to believe that two of the investigated constraints will have no effect on S1c. A bound on the number of preemptions should not have any effect on S1c. The reason is that the task is created on occurrence and not necessarily at a preemption point. The bound on number of preemptions will affect the time when the new task can start execution but it will not affect the possibility to create the task. There is a delay of the creation with respect to the observation points. This delay is however bounded by the observation granularity and imposes no hinder for dynamic task creation.

However, the third constraint, an upper bound on the number of tasks, has an obvious impact on the dynamic creation of tasks. Whenever there is an event occurrence and the number of executing task of the corresponding task type has reached the limit then the situation must be handled.

Deciding what to do when an event occurs and the upper bound on the corresponding task type is already reached is basically the same problem as performing admission control during an overload. There are different approaches to handle such a situation. For example the event occurrence can be ignored (similar to a rejection of a new task) or an admitted task can be aborted in favor of the new one. However, both these methods bypass the need for dynamic task creation since the methods make it possible to use a pool of statically created tasks instead of creating them dynamically. If the requirement on dynamic task creation is based on a true need for an unbounded task set, then this constraint cannot be applied to the system.

## 7.2 Non-cyclic Schedule, S2

One of the significant characteristics of the pure time-triggered design is the cyclic behavior, where the schedule repeats. At the time-triggered observation point tasks are initiated and scheduled according to a static schedule, i.e., a look-up table. Execution takes place during an activation interval. All tasks are finished during the activation interval and results are delivered at the end point of the same interval. Thereafter the cycle repeats with a new observation point.

There are two implications from the cyclic behavior. The first implication is that the number of potential schedules is limited by the potential combinations of events that can be observed simultaneously. The second implication is that the system returns to the initial state after each activation interval. Hence, the set of input events is the only factor to be considered when selecting the schedule.

The event-triggered system does not have this cyclic behavior. New tasks are triggered while other tasks are executing and results are delivered as soon as possible. The scheduler therefore must adapt to new situations and find a new schedule whenever a task arrives. If a new task has a higher priority than the currently executing task, the current task will be preempted in favor of the new one.

Finding an optimal schedule is usually an NP-hard problem and due to heuristics, the scheduler might be a source of unpredictability. The implications of this behavior are that the number of schedules is much higher in the event-triggered system and that it is not certain that the system ever will return to an initial state.

The constrained system used in the study has non-cyclic schedules. Time-triggered observations are applied but this simply means a short delay before triggering of the task. Delivery of results is not delayed. They are delivered as soon as possible. There are no specified activation intervals. Hence, the system is not necessarily in an initial state when tasks arrive and therefore the property S2a is met.

Finally, dynamic preemptive scheduling is required in dynamic event-triggered systems. The reason is that the scheduler has no information about future tasks or their resource requirements. On arrival of an urgent task, it must therefore be possible to preempt an executing task that is less urgent. This is not always true for the study. The reason is that whenever the maximum number of preemptions was reached for a task, that task continued its execution in a non-preempted mode. Hence, the property S2 was not met in the study.

To meet S2 another policy for handling such situations is needed. One possible solution is to ensure that the bound on the number of experienced preemptions is equal to the number of preemption points. This can lead to longer time intervals between the preemption points to keep a low number. On the other hand, such solution can give a predictable length of the non-preemptive intervals and therefore a shorter maximum delay for the next preemption point.

Although S2 was not met in the study, it is reasonable to believe that this property can be met in a constrained system as long as the maximum delay for a preemption plus the maximum delay for an observation does not exceed available slack time for a waiting high-priority task.

## 7.3 Response Time, S3

One of the most significant semantic differences between time-triggered and event-triggered systems is the predictability with respect to response time. In a time-triggered system, as described by Kopetz (1991), the response is usually not given until the end of an activation interval.

An activation interval starts with an observation point and ends with a point where all responses are given, i.e., a *communication point*. The allocated time slots for each task assume worst case execution time. It does not matter whether the actual execution time is shorter. The response will not be visible until the communication point at the end of the activation interval.

In the event-triggered system, tasks do not have pre-allocated execution slots and responses are given as soon as possible. If the result of an execution is a response sent to the controlled environment small variations of the response time will have an immediate impact on the system behavior. If the result is an internal response to e.g., a calling process, it will affect the internal state in some way e.g., the calling process is unblocked and/or internal data is updated, etc.

A change of the internal state can lead to a changed task set and an updated schedule. If the state change includes that a task is released, finished, blocked or un-blocked, the schedule is affected. Hence, variations in time for delivery of a result can affect the time to start execution of other waiting tasks, if such exists. Small variations in execution time for single tasks can therefore add up to large variations in the system behavior with respect to time.

As mentioned above (see Section 7.2) the results from execution are delivered as soon as possible in the study. Moreover, tasks do not have pre-allocated time slots for execution. As soon as one task is done, the next scheduled task is dispatched. Hence, the properties S3a and S3b are met in the study.

## 7.4 Extendability, S4

From a temporal point of view, tasks in a time-triggered system are encapsulated. As long as the already allocated slots for execution and communication are sufficiently large to include the changes, the modification does not require any recalculation of the schedules. If the modification changes the resource requirements, then it might require a recalculation of the static schedules.

The event-triggered system does not have the same knowledge of future arrivals and resource needs as the time-triggered. Instead the system adapts to the situation by dynamic scheduling. Hence, a modification or extension can easily be done. However, since the tasks are not encapsulated from a temporal point of view, a modification might change the timely behavior of the system.

Even very small changes can potentially propagate to the complete system and thereby affect the behavior in other nodes as well as the node containing the modified task. For example, changing the execution time of a task in one node might delay the access to the LAN for a task in another node. Hence, even though it is easier to implement the changes in event-triggered systems, such modifications require thorough regression testing for timeliness.

Dynamic scheduling and resource handling is used in the study (see Chapter 6. Adding tasks or prolonging their execution time was therefore easy and no cause to recalculation of any schedule and property S4 is therefore met. However, the changes did have a direct impact on the behavior from a temporal point of view. Hence, the semantics of the constrained system conforms with the event-triggered system semantics when it comes to extendability.

# Chapter 8

# Conclusions

This chapter is arranged as follows. Section 8.1 contains a retrospect of the thesis aim and objectives and presents the conclusions made with respect to each objective. Section 8.2 presents related work and discusses how the work in this thesis differs from other work. The contributions of this thesis are then stated in Section 8.3. Finally, ideas of future directions are given in Section 8.4. This chapter is based on the conclusion sections from Papers 1, 3, 4 and 7.

## 8.1 Discussion

The overall goal in this dissertation is to determine whether the set of execution environment constraints proposed by Birgisson et al. (1999) increases testability while maintaining the semantics of an event-triggered real-time system. Chapter 3.2 lists a set of objectives that must be met to fulfill this goal. In this section each objective is discussed and the conclusions are presented with respect to the objective and the results.

Objective 1 concerns the selection of a testability metric suitable for the study. Chapter 4 presents a survey of testability and discusses testability for real-time systems. The survey shows that the main testability issues concern controllability and observability. The survey also shows that for real-time systems the predictability with respect to execution orders is an important testability factor. A dynamic real-time system 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 real-time system than in a corresponding static real-time system and this is one of the reasons why it is so hard to test event-triggered real-time systems. The number of potential execution orders has previously been proposed as a testability metric by Thane (2000). The number of execution orders is therefore chosen to be used as a testability metric for the study. This metric goes in line with previous work on testability of real-time systems and assigns the highest testability to the time-triggered design (Schütz 1993, Thane 2000).

Objective 2 concerns the establishment of a method with which the selected testability metric can be used on a real-time system. Chapter 5 presents a method for trace-set generation. The presented algorithm enumerates all orders with which specified edges in a timed automata model are traversed. By specifying the edges where dispatches are made, all potential execution orders can be enumerated.

Objective 3 concerns estimation of the testability level in a real-time system model. Chapter 6 shows how the constraints from Section 3.1 are applied to a model of an event-triggered system, and presents a testability experiment. In this experiment the constraints are included in the model, and varied. The effect on testability is then estimated for each variation by enumerating the execution orders.

Objective 4 concerns the comparison of the measured effect on testability against the impact on testability predicted by (Birgisson et al. 1999). Chapter 6 presents the results from the testability experiment. For each of the investigated execution environment constraints, there is a comparison between the predicted impact and the impact demonstrated by experiment. The comparison shows a significant effect for two of the constraints, the number of allowed preemptions, $p$, and the number of observation points, $s$. Hypothesis 2 predicted an impact from $p$ of $O(p^{tq})$ for a fixed number of task types, $t$, and a fixed number of concurrently executing tasks of the same type, $q$. The demonstrated impact from $p$, was exponential just as expected. The demonstrated impact from $s$ was however linear instead of exponential. Hypothesis 1 predicted an impact of $O(s^n)$ for a fixed number of event types, $n$, while the demonstrated effect suggests an impact of $O(s)$. Hypothesis 2 is thus supported whereas Hypothesis 1 is not.

For the third constraint, the number of concurrently executing tasks of the same task type, no effect on testability is shown.

Hypothesis 3 is therefore also not supported by the results from the experiments. The discussion in Chapter 6 concludes that this constraint probably is of less interest from a testability perspective since it is not likely to affect the number of execution orders.

The fourth hypothesis concerns the tightness of the formulae. Hypothesis 4 is not supported since only one of the parameters shows an actual effect on testability similar to the effect predicted by the formula. This means that even though two of the suggested constraints, a maximum number of preemptions and observations, do affect testability, the formula is not sufficiently tight to be useful as a testability metric.

Finally, objective 5 concerns the implications from the execution environment constraints on the system semantics. Chapter 7 gives a discussion on the event-triggered semantics and the effects given by the constraints. The discussion concludes that it is possible to maintain the event-triggered semantics while using two of the constraints, an upper bound on number of preemptions and an upper bound on number of observation points. However, the upper bound on the number of concurrently executing tasks of the same task type implies that it is possible to implement the solution with, for example, a pool of threads. This is a limitation that affects the event-triggered semantics, because the ability to handle applications with a need for an unbounded task set is one of the semantic differences between time-triggered and event-triggered systems (Kopetz 1991).

## 8.2 Related Work

This section presents related work and points out the differences between this dissertation and other work.

Software testability for non-real-time systems is addressed by several authors, e.g., (Voas & Miller 1995, Byers 1997, Vranken et al. 1996, Wang et al. 1999, Binder 1994, Gao et al. 2003, Mouchawrab et al. 2005, Kansomkeat & Riveipiboon 2008) and the issue of how to measure testability is addressed by some of them. For example, Voas and Miller have defined and used metrics that are based on the probability that a test case would reach, activate and propagate an existing fault (Voas 1992, Voas & Miller 1993, Voas & Miller 1995, Voas & Miller 1996). Mouchawrab et al. (2005) present a set of testability measurements that can be applied to object-oriented software. However, the focus in this dissertation is testability in real-time systems when testing for timeliness. Since the addressed faults are non-functional and not necessarily depending on software only, the metrics for software testability presented in their work are less useful for this dissertation.

Mouchawrab et al. (2005) and Kansomkeat & Riveipiboon (2008) survey the area of software testability. The survey presented in Chapter 4 also includes the work on testability of real-time systems.

Thane & Hansson (1999*a*) and Schütz (1993) address testability in real-time systems. However, Thane & Hansson (1999*a*) assume static scheduling and Schütz (1993) assumes a time-triggered design. The focus in this dissertation is event-triggered real-time systems, which are required to have dynamic scheduling.

Schütz (1993) describes the relation between execution orders and testability for distributed real-time systems. Schütz (1993) gives a formula for testability given by different design paradigms. The formula is based on the number of observations that the system makes during a specified interval and allows a comparison of testability between event-triggered and time-triggered solutions. However, while the formula gives an upper bound on execution orders for time-triggered systems, it only gives a lower bound for event-triggered systems. Schütz (1993) points out preemptions as one of the reasons for why this formula is a lower bound for event-triggered systems. Mellin (1998) defined an upper bound on test effort based on the work by Schütz. Mellin (1998) includes preemptions and task load in the formula. This formula is improved by Birgisson et al. (1999) to allow for shared resources. The actual effect that the execution environment constraints have on testability has never been studied before. This dissertation contains an investigation that studies the effect on testability when applying the execution environment constraints proposed by Birgisson et al. (1999) to an event-triggered real-time system.

Thane and Hansson (Thane & Hansson 1999*a*) present a method to generate all execution orders (i.e., an EOG graph) for static real-time systems, arguing that there are not enough test methods for concurrent programs. They also discuss testability related to the number of execution orders in the sense that each order is similar to a sequential program. Hence, test methods for sequential programs can be applied to concurrent programs by testing each execution order as a sequential program. This dissertation also presents a method to generate all execution orders. However, the method presented in this dissertation does not depend on a static schedule. The execution orders were generated for an event-triggered system with dynamic scheduling, where we used earliest deadline first.

The idea to use a model checker to generate test cases is not new. Whenever a test criterion can be expressed as a property verifiable by a model checker, such techniques for test case generation are useful. For example, Visser et al. (Visser, Pasareanu & Khurshid 2004) presented a framework built on top of the Java PathFinder (JPF) to automatically generate test inputs for Java programs. The basic idea is that the test criterion is expressed as a safety property $\phi$ and symbolic execution of a path that satisfies $\phi$ generates a set of constraints. A constraint solver then gives the input that fulfills the constraints, i.e., executes the path where $\phi$ is satisfied. Garganti and Heitmeyer (Gargantini & Heitmeyer 1999) present a method for obtaining a test input sequence from a system property and a software cost reduction requirements specification. Ammann et al. (Ammann, Black & Majurski 1998) present a mutation analysis approach for test case generation with a model checker. A model checker typically returns a single trace. With the method presented in this dissertation it is shown that it is possible to generate a set of traces. The generated trace set is a subset of all traces. All traces included in the generated set are distinct with respect to the order with which they traverse specified edges in an automaton.

## 8.3 Contributions

**Testability experiment:** The results from the experiment described in Chapter 6 give a deeper understanding of the relation between properties of the execution environment and system testability. The actual impact on testability from three system properties, i.e., execution environment constraints, is investigated. These properties have been discussed in previous work as being important factors for the level of testability of event-triggered real-time systems. This is however the first time that the actual impact from these properties has been investigated in an experiment. The results show that one of the properties, the number of allowed preemptions, have an impact on testability as expected. One of the properties, the observation granularity, has less impact than expected but still significant. The results from the experiment show no impact from the third property, the number of concurrently executing tasks of the same type. The conclusions drawn here are that the third property is less interesting from a testability perspective and Formulae 3.1 FSTAT is not sufficiently tight to use as a testability approximation.

**Dynamic trace set generation:** The algorithm presented in Chapter 5 is a new and innovative approach that can be used to generate sets of traces from a model checker. Together these traces cover all orderings with respect to how some of the edges in an automaton are traversed. This algorithm was developed as a necessary step in order to carry out the experiment described in Chapter 6. However, the algorithm can also be useful in testing, where sets of related traces are needed to satisfy coverage criteria. For example, both Thane & Hansson (1999*a*) and Schütz (1994) argue that coverage of execution orders is important for real-time systems. In a complex model, such orders are not known beforehand. The algorithm presented in Chapter 5 therefore collects them iteratively. By specifying certain transitions in timed automata as p-points, the algorithm generates all potential orders with which these transitions can be traversed. If a test method focuses on execution orders, each transition that contains a context switch should be specified as a p-point. In this case, the set of traces generated would cover all execution orders. Other orders that might be interesting from a test perspective are communication and access to shared data.

**Dynamic partitioning of the state space:** Memory consumption is a major problem when verifying complex timed automata models. The algorithm presented in Chapter 5 dynamically divides the state space into smaller partitions along the traces as they are generated. These partitions are independent, so the memory needed by the traces is not increased. In addition, the trace generation can easily be distributed over several computers.

**Testability survey:** This dissertation presents a comprehensive survey over testability. The survey includes work in the area of testability for both non-real-time and real-time systems.

## 8.4 Future directions

This section suggests future work in directions that concern extensions of the experiments and how to use the trace-set generation technique for verification.

- The first suggestion for future work is to investigate whether the results would be the same if the policies for scheduling and resource handling were changed. There is a large variety of policies to schedule real-time tasks and it is possible that the choice of policy has an impact on testability.

- Three of the parameters in Formulae 3.1, FSTAT, are investigated in our study and a natural suggestion for future work is to study the impact on testability from the two remaining parameters, the number of task types, $t$, and the number of event types, $n$.

- The results from the experiment suggest that two of the investigated properties are interesting from a testability perspective. However, only the properties suggested by Birgisson et al. (1999) were investigated. A future direction that is particularly interesting is to identify other system properties that might affect testability and investigate them.

- The fourth direction for future work suggested here is to define test criteria based on orders, e.g., all execution order coverage. A method for trace-set generation is defined and the orders might be very useful for model-based testing but it is not yet shown how effective such a test strategy is with respect to other strategies. A next step would therefore be to compare, e.g., all execution order coverage to e.g., random testing. These are important steps to determine the practical relevance of the novel technique presented in Chapter 5.

- The final suggestion for future work presented here is to investigate the practical use in general model checking problems of the algorithm presented in Chapter 5. The algorithm mitigates the problem of space limitations in model-checking algorithms based on state-space exploration. The state space is partitioned by the algorithm and the analysis problem can thereby be divided into sub-problems, which can be analyzed independently. To use this approach in model-checking algorithms, a formal proof of correctness with respect to completeness is however, needed.

# Bibliography

Alur, R. & Dill, D. L. (1994), ‘A Theory of Timed Automata’, *Theoretical Computer Science* **126**(2), 183–235.

Ammann, P. & Black, P. E. (2002), Model Checkers in Software Testing, Technical Report NIST-IR 6777, National Institute of Standards and Technology.

Ammann, P. & Offutt, J. (2008), *Introduction to Software Testing*, Cambridge University Press.

Ammann, P. E., Black, P. E. & Majurski, W. (1998), Using Model Checking to Generate Tests from Specifications, *in* ‘Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM’98)’, pp. 46–54.

Amnell, T., Behrmann, G., Bengtsson, J., D’Argenio, P. R., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K. G., Müller, M. O., Pettersson, P., Weise, C. & Yi., W. (2001), UPPAAL - Now, Next, and Future, *in* ‘Modelling and Verification of Parallel Processes (MOVEP’2k)’, pp. 100–125.

Avizienis, A., Laprie, J. C., Randell, B. & Landwehr, C. (2004), ‘Basic Concepts and Taxonomy of Dependable and Secure Computing’, *IEEE Transactions on Dependable and Secure Computing* **1**(1), 11–33.

Baker, T. P. (1991), ‘Stack-Based Scheduling of Realtime Processes’, *The Journal of Real-Time Systems* **3**, 67–99.

Barrett, P. A., Hilborne, A. M., Verissimo, P., Rodrigues, L., Bond, P., Seaton, D. & Speirs, N. (1990), The Delta-4 Extra Performance Architechture (XPA), *in* ‘International Symposium on Fault-Tolerant Computing’, pp. 481–488.

Behrmann, G., Hune, T. & Vaandrager, F. (2000), Distributed Timed Model Checking - How the Search Order Matters, *in* 'Proc. of 12th International Conference on Computer Aided Verification', Lecture Notes in Computer Science, Springer-Verlag, Chicago.

Behrmann, G., Larsen, K., Pearson, J., Weise, C. & Yi, W. (1999), Efficient Timed Reachability Analysis Using Clock Difference Diagrams, *in* 'Eleventh International Conference on Computer Aided Verification', Vol. 1633 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 341–353.

Beizer, B. (1990), *Software Testing Techniques*, 2nd edn, Van Nostrand Reinhold.

Bengtsson, J. & Yi, W. (2001), Reducing Memory Usage in Symbolic State-Space Exploration for Timed Systems, Technical report 2001-009, Department of Information Technology, Uppsala University.

Bengtsson, J. & Yi, W. (2003), On Clock Difference and Termination in Reachability Analysis of Timed Automata, *in* 'IEEE International Conference on Formal Engineering Methods, ICFEM 2003', Vol. 2885 of *Lecture Notes in Computer Science*, Springer-Verlag.

Bengtsson, J. & Yi, W. (2004), Timed Automata: Semantics, Algorithms and Tools, *in* W. Reisig & G. Rozenberg, eds, 'Lecture Notes on Concurrency and Petri Nets', Springer-Verlag.

Binder, R. V. (1994), 'Design for Testability in Object-Oriented Systems', *Communications of the ACM* **37**(9), 87–101.

Birgisson, R., Mellin, J. & Andler, S. (1999), Bounds on Test Effort for Event-Triggered Real-Time Systems, *in* 'The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)'.

Boel, R. K. (2000), Automatic Synthesis of Schedules in a Timed Discrete Event Plant, *in* 'Proceedings of International Conference on Automation of Mixed Processes: Hybrid Dynamic SystemsADPM 2000'.

Boel, R. K. & Montoya, F. J. (2000), Modular Synthesis of Efficient Schedules in a Timed Discrete Event Plant, *in* 'Proceedings of the 39th IEEE Conference on Decision and Control', Vol. 1, pp. 16–21.

Byers, D. (1997), 'Towards Estimating Software Testability Using Static Analysis', Licentiate thesis No 626. Linköping University.

Chilenski, J. J. & Miller, S. P. (1994), 'Applicability of Modified Condition/Decision Coverage to Software Testing', *Software Engineering Journal* **9**(5), 193–200.

Chung, I. & Bieman, J. M. (2008), 'Generating Input Data Structures for Automated Program Testing', *Software Testing, Verification and Reliability* **n/a**(n/a), n/a.

Clarke, E. M. & Emerson, A. (1981), 'Synthesis of Synchronization Skeletons for Branching Time Temporal Logic', *In Logic of Programs: Workshop, Lecture Notes in Computer Science* **131**, 52–71.

Cook, T. & Campbell, D. (1979), *Quasi-Experimentation Design and Analysis Issues for Field Settings*, Houghton Mifflin Company.

Davis, R. I., Tindell, K. W. & Burns, A. (1993), Scheduling Slack Time in Fixed Priority Preemptive Systems, *in* 'Proceedings of Real-Time Systems Symposium', pp. 222–231.

DeMillo, R. A. & Offutt, A. J. (1991), 'Constraint-Based Automatic Test Data Generation', *Transactions of Software Engineering* **17**(9), 900–910.

Dssouli, R., Karoui, K., Saleh, K. & Cherkaoui, O. (1999), 'Communications Software Design for Testability: Specification Transformation and Testability Measures', *Information and Software Technology* **41**(11-12), 729–743.

Fehnker, A. (1999), Scheduling a Steel Plant with Timed Automata, *in* 'Real-Time Computing Systems and Applications (RTCSA)', pp. 280–286.

Freedman, R. S. (1991), 'Testability of Software Components', *IEEE Transactions on Software Engineering* **17**(6), 553–564.

Gait, J. (1986), 'A Probe Effect in Concurrent Programs', *Software Practice and Experience* **16**(3), 225–233.

Gao, J. Z., Tsao, J., Wu, Y. & Jacob, T. H. S. (2003), *Testing and Quality Assurance for Component-Based Software*, Artech House, Inc., Norwood, MA, USA.

Gargantini, A. & Heitmeyer, C. L. (1999), Using Model Checking to Generate Tests from Requirements Specifications, *in* 'ESEC / SIGSOFT Foundations of Software Engineering', pp. 146–162.

Garousi, V. (2008), Traffic-aware Stress Testing of Distributed Real-Time Systems based on UML Models in the presence of Time Uncertainty, *in* '1st International Conference on Software Testing, Verification, and Validation', IEEE Computer Society, Lillehammer, pp. 92–101.

Gotlieb, A., Botella, B. & Rueher, M. (1998), 'Automatic test data generation using constraint solving techniques', *SIGSOFT Software Engineering Notes* **23**(2), 53–62.

Grindal, M., Lindström, B., Offutt, A. J. & Andler, S. F. (2006), 'An Evaluation of Combination Strategies for Test Case Selection', *Empirical Software Engineering* **11**(4), 583–611.

Hessel, A., Larsen, K. G., Nielsen, B., Pettersson, P. & Skou, A. (2003), Time-Optimal Real-Time Test Case Generation using Uppaal, *in* 'The 3rd International Workshop on Formal Approaches to Testing of Software', number 2931, pp. 136–151.

Holzmann, G. J. (1997), 'The Model Checker SPIN', *IEEE Transactions on Software Engineering* **23**(5), 279–295.

Holzmann, G. J. (2003), *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley.

Hune, T., Larsen, K. & Pettersson, P. (2000), Guided Synthesis of Control Programs using Uppaal, *in* I. Society Press, ed., 'Proceedings of the IEEE ICDS International Workshop on Distributed Systems Verification and Validation', pp. E15–E22.

IEEE (1990), *IEEE Standard Glossary of Software Engineering Terminology*.

Isovic, D. & Fohler, G. (2000), Online Handling of Aperiodic Tasks in Time-Triggered Systems, Technical report, Mälardalen Real-Time Research Center, MDH-MRTC-22/2000-1-SE.

Kansomkeat, S. & Riveipiboon, W. (2008), 'An Analysis Technique to Increase Testability of Object-Oriented Components', *Software Testing Verification and Reliability* **n/a**(n/a), n/a.

Kopetz, H. (1991), *Operating Systems of the 90s and Beyond*, Vol. 563 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, chapter Event-Triggered Versus Time-Triggered Real-Time Systems, pp. 86–101.

Kopetz, H. & Verissimo, P. (1993), *Distributed Systems*, Addison Wesley, chapter 16, pp. 411–446.

Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C. & R.Zainlinger (1989), 'Distributed Fault-Tolerant Real-Time Systems: The MARS approach', *IEEE Micro* **9**(1), 25–40.

Korel, B. (1990), 'Automated Software Test Data Generation', *Transactions of Software Engineering* **16**(8), 870–879.

Laprie, J. C. (1994), *Dependability: Basic Concepts and Terminology*, Springer Verlag for IFIP WG 10.4.

Larsen, K. G., Pettersson, P. & Yi, W. (1997), 'UPPAAL in a Nutshell', *Int. Journal on Software Tools for Technology Transfer* **1**(1–2), 134–152.

Laski, J. & Korel, B. (1983), 'A Data Flow Oriented Program Testing Strategy', *IEEE Transactions on Software Engineering* **9**(3), 347–354.

Lehoczky, J. P. & Ramos-Thuel, S. (1992), An Optmal Algorithm for Scheduling Soft Aperiodic Tasks in Fixed Priority Preemptive Systems, *in* 'Proceedings of Real-Time Systems Symposium', pp. 110–123.

Lindström, B., Mellin, J. & Andler, S. (2002), Testability of Dynamic Real-Time Systems, *in* 'Proceedings of Eight International Conference on Real-Time Computing Systems and Applications (RTCSA2002)', Tokyo, Japan, pp. 93–97.

McCabe, T. J. (1976), 'A Complexity Measure', *IEEE Transactions on Software Engineering* **2**(4), 308–320.

Mellin, J. (1998), 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’, Vol. 1553 of *Lecture Notes in Computer Science*, Springer, pp. 194–211.

Mellin, J. (2004), Resource-Predictable and Efficient Monitoring of Events, Phd thesis no. 876, Linköping University, Sweden.

Mouchawrab, S., Briand, L. C. & Labiche, Y. (2005), ‘A Measurement Framework for Object-Oriented Software Testability’, *Information and Software Technology* **47**(15), 979–997.

Myers, G. J. (1979), *The Art of Software Testing*, John Wiley and Sons.

Nilsson, R., Offutt, J. & Andler, S. F. (2004), Mutation-based testing criteria for timeliness, *in* ‘IEEE Computer Software and Applications Conference’, Hong Kong, China, pp. 306–311.

Offutt, J. (1988), Automatic Test Data Generation, Technical report GIT-ICS88/28, Georgia Institute of Technology.

Offutt, J., Jin, Z. & Pan, J. (1999), ‘The Dynamic Domain Reduction Approach to Test Data Generation’, *Software–Practice and Experience* **29**(2), 167–193.

Ostrand, T. J. & Balcer, M. J. (1988), ‘The Category-Partition Method for Specifying and Generating Functional Tests’, *Communications of the ACM* **31**(6), 676–686.

Queille, J. P. & Sifakis, J. (1982), Specification and Verification of Concurrent Programs in CESAR , *in* ‘Proc. 5*th* Int. Symp. on Programming’, number 137, Springer–Verlag, Berlin, pp. 195–220.

Ramamritham, K. (1995), The Origin of TCs, *in* M. Berndtsson & J. Hansson, eds, ‘Proceedings of the First International Workshop on Active and Real-Time Database Systems (ARTDB 1995)’, Springer, Skovde, Sweden, pp. 50–62.

Rapps, S. & Weyuker, E. J. (1985), ‘Selecting Software Test Data Using Data Flow Information’, *IEEE Transactions on Software Engineering* **11**(4), 367–375.

Robson, C. (1993), *Real World Research*, Blackwell, Oxford, UK.

Schütz, W. (1993), *The Testability of Distributed Real-Time Systems*, Kluwer Academic Publishers.

Schütz, W. (1994), 'Fundamental Issues in Testing Distributed Real-Time Systems', *Real-Time Systems* **7**(2), 129–157.

Sprunt, B., Sha, L. & Lehoczky, J. (1989), 'Aperiodic task scheduling for hard real-time systems', *The Journal of Real-Time Systems* **1**, 27–60.

Spuri, M. & Buttazzo, G. (1996), 'Scheduling Aperiodic Tasks in Dynamic Priority Systems', *The Journal of Real-Time Systems.*

Standard ISO/IEC 9126, I. (1991), 'Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their use', International Organization for Standardization, International Electrotechnical Commission, Geneva.

Stankovic, J. A. (1988), 'Misconceptions About Real-Time Computing', *IEEE Computer* **21**(10), 10–19.

Thane, H. (2000), Monitoring, Testing and Debugging of Distributed Real-Time Systems, PhD thesis, Royal Institute of Technology, KTH.

Thane, H. & Hansson, H. (1999*a*), Towards Deterministic Testing of Distributed Real-Time Systems, *in* 'Swedish National Real-Time Conference (SNART'99)'.

Thane, H. & Hansson, H. (1999*b*), Towards Systematic Testing of Distributed Real-Time Systems, *in* 'RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium', IEEE Computer Society, Washington, DC, USA, p. 360.

Vilkomir, S. A. & Bowen, J. P. (2002), Reinforced Condition/Decision Coverage (RC/DC): A New Criterion for Software Testing, *in* 'ZB2002: 2nd International Conference of Z and B Users', Springer-Verlag, Grenoble, France, pp. 295–313.

Visser, W., Pasareanu, C. & Khurshid, S. (2004), Test Input Generation with Java PathFinder, *in* 'International Symposium on Software Testing and Analysis', ACM Press, Boston, Massachusetts, USA, pp. 97–107.

Voas, J. M. (1992), 'PIE: A Dynamic Failure-Based Technique', *IEEE Transactions on Software Engineering* **18**(8), 717–727.

Voas, J. M. & Miller, K. W. (1993), 'Semantic Metrics For Software Testability', *Journal of Systems and Software* **20**(3), 207–216.

Voas, J. M. & Miller, K. W. (1995), 'Software Testability: The New Verification', *IEEE Software* **12**(3), 17–28.

Voas, J. M. & Miller, K. W. (1996), 'Dynamic testability analysis for assessing fault tolerance', *High Integrity Systems*.

Vranken, H. P. E., Witteman, M. F. & van Wuijtswinkel, R. C. (1996), 'Design for Testability in Hardware-Software Systems', *IEEE Design and Test of Computers* **13**(3), 79–87.

Wang, Y., King, G. & Wickburg, H. (1999), A method for Built-in Tests in Component-based Software Maintenance, *in* '3rd European Conference on Software Maintenance and Reengineering (CSMR '99)', pp. 186–189.

Wohlin, C., Runesson, P., Höst, M., Ohlsson, M. C., Regnell, B. & Wesslén, A. (2000), *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Press.

Zhu, H., Hall, P. A. V. & May, J. H. R. (1997), 'Software Unit Test Coverage and Adequacy', *ACM Computing Surveys* **29**(4), 366–427.

of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and

Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer**: Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-suppor-ted Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-
318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002,
ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented
Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the
Computational Complexity of Temporal
Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for
Enabling Interoperability of Structured and
Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informa-

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.

No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.

No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.

No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.

No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.

No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.

No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.

No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.

No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.

No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.

No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.

No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.

No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.

No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.

No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.

No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.

No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.

No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.

No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.

No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.

No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.

No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.

No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.

No 1156 **Artur Wilk**: Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.

No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.

No 1185 **Jörgen Skågeby:** Gifting Technologies - Ethnographic Studies of End-users and Social Media Sharing, 2008, ISBN 978-91-7393-892-1.

No 1187 **Imad-Eldin Ali Abugessaisa:** Analytical tools and information-sharing methods supporting road safety organizations, 2008, ISBN 978-91-7393-887-7.

No 1204 **H. Joe Steinhauer:** A Representation Scheme for Description and Reconstruction of Object Configurations Based on Qualitative Relations, 2008, ISBN 978-91-7393-823-5.

No 1222 **Anders Larsson:** Test Optimization for Core-based System-on-Chip, 2008, ISBN 978-91-7393-768-9.

No 1240 **Fredrik Heintz:** DyKnow: A Stream-Based Knowledge Processing Middleware Framework, 2009, ISBN 978-91-7393-696-5.

No 1241 **Birgitta Lindström:** Testability of Dynamic Real-Time Systems, 2009, ISBN 978-91-7393-695-8.

**Linköping Studies in Statistics**

No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.

No 10 **Karl Wahlin:** Roadmap for Trend Detection and Assessment of Data Quality, 2008, ISBN: 978-91-7393-792-4

**Linköping Studies in Information Science**

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informa-tionssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.