

Linköping Studies in Science and Technology

Dissertation No. 1030

A Mutation-based Framework for Automated Testing of Timeliness

Robert Nilsson

October 18, 2006



Linköping University
INSTITUTE OF TECHNOLOGY

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

Linköping 2006

ISBN 91-85523-35-6
ISSN 0345-7524

© Robert Nilsson, 2006. All rights reserved.
Printed in Sweden by LiU-Tryck.
Linköping 2006.

Abstract

A problem when testing timeliness of event-triggered real-time systems is that response times depend on the execution order of concurrent tasks. Conventional testing methods ignore task interleaving and timing and thus do not help determine which execution orders need to be exercised to gain confidence in temporal correctness. This thesis presents and evaluates a framework for testing of timeliness that is based on mutation testing theory. The framework includes two complementary approaches for mutation-based test case generation, testing criteria for timeliness, and tools for automating the test case generation process. A scheme for automated test case execution is also defined. The testing framework assumes that a structured notation is used to model the real-time applications and their execution environment. This real-time system model is subsequently mutated by operators that mimic potential errors that may lead to timeliness failures. Each mutated model is automatically analyzed to generate test cases that target execution orders that are likely to lead to timeliness failures. The validation of the theory and methods in the proposed testing framework is done iteratively through case-studies, experiments and proof-of-concept implementations. This research indicates that an adapted form of mutation-based testing can be used for effective and automated testing of timeliness and, thus, for increasing the confidence level in real-time systems that are designed according to the event-triggered paradigm.

Keywords: Automated Testing, Real-time systems, Time constraints, Timeliness, Model-based.

Acknowledgements

First and foremost, I would like to thank my supervisors. Sten F. Andler for encouragement, insightful comments on my work and for being a great boss. Jeff Offutt, for widening my horizon and giving me invaluable guidance in research and the art of writing papers. Jonas Mellin for some interesting discussions and for sharing his experiences of thesis writing. I would also like to thank Simin Nadjm-Tehrani for her comments on my work and for always being supportive and welcoming.

Next, I would like to thank the following (present and past) members of the DRTS research group (in alphabetical order), Alexander Karlsson, Ammi Ericsson, Bengt Eftring, Birgitta Lindström, Gunnar Mathiason, Henrik Grimm, Jörgen Hansson, Marcus Brohede, Mats Grindal, Ragnar Birgisson and Sanny Gustavsson. It has been a pleasure working with all of you and I'm grateful for all the procrastination and tilted discussions that we have shared. In particular, I would like to thank Alexander Karlsson for helping me with the attempts to seed timeliness errors.

I am also very grateful for all the contacts and opportunities given to me through the ARTES network and FlexCon project. It is probably not a coincidence that research in this thesis is inspired by ideas originating from Uppsala, Västerås and Lund. In particular, I want to thank Paul Pettersson, Hans Hansson, Anita Andler and Roland Grönros, for being good research catalysts and for organizing interesting summer schools and conferences. In the FlexCon project, I would specifically like to thank Karl-Erik Årzen and Klas Nilsson for valuable input and feedback. I also want to thank Dan Henriksson for introducing me to the intricacies of the truetime tool and for helping with the controller design of the experiments in our joint paper. In this context, I would also like to thank Anders Pettersson and Henrik Thane for interesting discussions about subtle issues of real-time system testing.

I want to thank the fellow employees at the former department of Computer Science and the School of Humanities and Informatics at the University of Skövde. I believe that there is great potential in a small academy being so dedicated to research. Keep up the good work!

Finally, I want to thank my family; Anders, Jane and Johan Nilsson and my grandmother Ulla Hjalmarsson for coping with me during my exile in academia. A very special thanks goes to my girlfriend Viola Bergman, for her cheerful support and warm encouragements. At last, I want to thank and tribute my grandfather, Helge Hjalmarsson, for being a great inspiration for me.

List of Publications

Some of the contributions of this Ph.D thesis has previously been published in the following papers and technical reports.

- **R. Nilsson**, S.F. Andler, and J.Mellin. Towards a Framework for Automated Testing of Transaction-Based Real-Time Systems. In *Proceedings of Eighth International Conference on Real-Time Computing Systems and Applications (RTCSA2002)*, pages 109–113, Tokyo, Japan, March 2002.
- **R. Nilsson**. Thesis proposal : Automated timeliness testing of dynamic real-time systems. Technical Report HS-IDA-TR-03-006, School of Humanities and Informatics, Univeristiy of Skövde, 2003.
- **R. Nilsson**, J. Offutt, and S. F. Andler. Mutation-based testing criteria for timeliness. In *Proceedings of the 28th Annual Computer Software and Applications Conference (COMPSAC)*, pages 306–312, Hong Kong, September 2004. IEEE Computer Society.
- **R. Nilsson** and D. Henriksson. Test case generation for flexible real-time control systems. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 723–721, Catania, Italy, September 2005.
- **R. Nilsson** J. Offutt and J. Mellin, Test case generation for Mutation-based testing of timeliness, In *Proceedings of the 2nd International Workshop on Model-Based Testing*, pages 102-121, Vienna, Austria, March 2006.

Contents

1	Introduction	1
1.1	Overview	1
1.2	Results and Contributions	2
1.3	Thesis Outline	3
I	Timeliness Demise	5
2	Dynamic Real-Time Systems	7
2.1	Real-time System Preliminaries	7
2.2	Tasks and Resources	8
2.3	Design and Scheduling	10
3	Testing Real-time Systems	13
3.1	Software Testing Preliminaries	13
3.2	Testing Criteria	15
3.3	Issues when Testing Real-Time Systems	16
3.4	Testing of Timeliness	17
3.5	Timeliness Faults, Errors and Failures	18
4	Problem Definition: Testing Dynamic Real-Time Systems	21
4.1	Purpose	21
4.2	Motivation for Automated Testing of Timeliness	21
4.3	Problem Definition	22
4.4	Approach and Delimitations	24
4.4.1	Thesis Statement	24
4.4.2	Objectives	25

II	Rise of the Mutated Models	27
5	A Framework for Testing Dynamic Real-Time Systems	29
5.1	Framework Overview	29
5.2	Timeliness Test Cases	31
5.3	Mutation-based Test Case Generation	32
5.4	Test Execution	34
6	Mutation-based Testing Criteria for Timeliness	37
6.1	Motivation	37
6.2	Adopted Modelling Formalism	38
6.3	Mutation Operators	41
6.3.1	Task Set Mutation Operators	42
6.3.2	Automata Mutation Operators	45
6.4	Mutation-based Testing Criteria	46
6.5	Generation of Test Cases Using Model-Checking	48
6.6	Validation of Mutation Operators	49
7	Generation of Test Cases Using Heuristic-driven Simulations	53
7.1	Motivation	53
7.2	Genetic Algorithms for Test case Generation	54
7.2.1	Genome Mapping Function	56
7.2.2	Fitness Function	57
7.2.3	Heuristic Cross-over Functions	57
7.3	Testing of Time Dependent Control Performance	59
7.4	Test Case Generation Experiments	61
7.4.1	Base-line Real-time System	62
7.4.2	Complex Dynamic Real-time System	64
7.4.3	Mixed Load Real-time Control System	66
III	Empiricism strikes back	71
8	T^3: A Tool for Mutation-based Testing of Timeliness	73
8.1	Overview	73
8.2	Flextime Simulator Extensions	74
8.2.1	TrueTime	75
8.2.2	Task Sets and Execution Patterns	75
8.2.3	Activation Patterns	78

8.3	Test Case Generation Support	78
8.3.1	Modes for Generating Test Cases	78
8.3.2	Support for Task Instance Modelling	81
9	Case Study: Testing using the Mutation-based Framework	83
9.1	Overview	83
9.2	A Flexcon Demonstrator Prototype	84
9.2.1	Real-time Design	84
9.2.2	Target Platform	87
9.2.3	Test Harness	89
9.2.4	Task Implementation	93
9.3	Applying the Framework for Testing of Timeliness	93
9.3.1	Constructing Task Input Data	94
9.3.2	Combining Task Input Data for System Level Testing	95
9.4	Demonstrator Testing Experiments	97
9.4.1	Experiment 1 - Framework Testing Effectiveness	98
9.4.2	Experiment 2 - Effectiveness with Seeded Errors	104
10	Discussion	109
10.1	Mutation Operators and Testing Criteria	109
10.1.1	System Specific Mutation Operators	111
10.1.2	Guidelines for Adding Mutation Operators	112
10.2	Test Case Generation Issues	115
10.2.1	Scalability of Test Case Generation	116
10.2.2	Genetic Algorithm Enhancements	116
10.3	Prefix-based Test Case Execution	117
10.3.1	Prefixed Test Cases for Testing of Timeliness	119
10.3.2	A Prefix-based Test Case Execution Scheme	120
10.4	Validation of Mutation-based Testing of Timeliness	121
10.4.1	Evaluating Timeliness Test Cases	121
10.4.2	Validation of Thesis Statement	122
11	Related Work	125
11.1	Test Case Generation	125
11.2	Testing of Real-Time Systems	129

12 Conclusions	131
12.1 Summary	131
12.2 Contributions	132
12.3 Future Work	133
References	137
A Demonstrator Task Code	145
B Demonstrator Tasks Testing	155
C Case-Study Test Execution Results	163
D Mutation Operators Overview	167

Introduction

The introduction chapter provides an overview of the contents covered by this thesis. The scientific problem and solution approach are briefly introduced and motivated in section 1.1. The results and contributions are summarized in section 1.2. Finally, section 1.3 presents the structure of the remainder of the thesis.

1.1 Overview

Real-time systems must be dependable as they often operate in tight interaction with human operators and valuable equipment. A trend is to increase the flexibility of such systems so that they can support more features while running on “off-the-shelf” hardware platforms. However, with the flexibility comes increased software complexity and non-deterministic temporal behavior. There is a need for verification methods to detect errors arising from temporal faults so that confidence can still be placed in the safety and reliability of such systems.

A problem associated with the testing of real-time applications is that their timeliness depends on the execution order of tasks. This is particularly problematic for event-triggered and dynamically scheduled real-time systems, in which events may influence the execution order at any time (Schütz 1994). Furthermore, tasks in real-time systems behave differently from one execution to the next, depending not only on the implementation of real-time kernels and program logic, but also on efficiency of acceleration hardware such as caches and branch-predicting pipelines. Non-deterministic temporal behavior necessitates methods and tools for effectively

detecting the situations when errors in temporal estimations can cause the failure of dependable applications.

The timeliness of embedded real-time systems is traditionally analyzed and maintained using scheduling analysis techniques or regulated online through admission control and contingency schemes (Burns & Wellings 2001). These techniques use assumptions about the tasks and load patterns that must be correct for timeliness to be maintained. Doing schedulability analysis of non-trivial system models is complicated and requires specific rules to be followed by the run-time support system. In contrast, timeliness testing is general in the sense that it applies to all system architectures and can be used to gain confidence in assumptions by systematically sampling among the execution orders that can lead to missed deadlines. Hence, from a real-time perspective, timeliness testing is a necessary complement to analysis.

It is difficult to construct effective sequences of test inputs for testing timeliness without considering the effect on the current set of active tasks and real-time protocols. However, existing testing techniques seldom use such information and they do not predict which execution orders may lead to timeliness failures (Nilsson, Andler & Mellin 2002).

In summary, problems with testing of timeliness arise from a dynamic environment and the vast number of potential execution orders of event-triggered real-time systems. Therefore, we need to be able to produce test inputs that exercise a meaningful subset of these execution orders. Within the field of software testing, several methods have been suggested for model-based test case generation but few of them capture the behavior that is relevant to generate effective timeliness test cases.

This thesis presents a mutation-based method for testing of timeliness that takes internal behaviors into consideration. The thesis also describes experiments for evaluating the effectiveness of this method.

1.2 Results and Contributions

The results of this thesis form a framework for automatic testing of timeliness for dynamic real-time systems. In this context, a framework means theory and methods as well as associated tools for performing automated testing of timeliness in a structured way¹. As a requirement for applying our proposed framework, the estimated temporal properties and resource requirements of real-time applications

¹The reason for providing a framework and not simply a method is that there are several ways to perform the steps in the chain from system modelling to test analysis.

must be specified in a model. As opposed to other approaches for model-based testing of timeliness, properties of the execution environment are part of the model which is used for generating test cases. This provides the advantage that the effects of, for example scheduling protocols and platform overheads, can be captured during automatic test case generation.

A set of basic mutation-based testing criteria for timeliness is defined and validated using model-checking techniques. Two methods for generating test cases that fulfill timeliness testing criteria are presented. One of the methods, based on model-checking, is suitable for dependable systems where a stringent test case generation method is preferred. For more complex target systems, the model-checking based approach may be impractical. A complementary method based on heuristic-driven simulations is presented and evaluated in a series of experiments. The thesis also presents a prototype tool that integrates with MATLAB/Simulink and support the heuristic-driven method for test-case generation. This tool allows control specific constraints and processes to be used as input to mutation-based test case generation.

A scheme for automated test case execution that uses the added information from mutation-based test case generation and has the potential to reduce non-determinism is discussed. This method exploits advantages inherent in transaction based systems and allows testing to be focused on the critical behaviors indicated by test cases.

The mutation-based testing criteria and test case generation approaches are evaluated in a series of experiments using model-checking and simulations. The overall testing framework is evaluated by testing a simplified robot-arm control application that runs on the Linux/RTAI platform. This study is an example of how the testing framework can be applied and demonstrates the effectiveness and limitations of the suggested approach.

1.3 Thesis Outline

Chapters 2 and 3 introduce relevant concepts from real-time systems development and software testing. Chapter 4 provides a more detailed description of the problem addressed by the thesis, motivates its importance, and presents our approach for addressing it.

Chapter 5 presents an overview of a framework for mutation-based testing of timeliness and defines concepts used in the remainder of the thesis, while the proposed methods for test case selection and automated test case generation are included in Chapters 6 and 7. The validation experiments of the proposed methods

are also presented in this context.

Chapter 8 describes the tool support proposed for the framework, and Chapter 9 contains the result of a case study in which the proposed testing framework is used for testing timeliness in a real system. In Chapter 10 the contributions of the thesis are discussed and the advantages and disadvantages of the proposed framework are described. Chapter 11 contains related work, Chapter 12 concludes the thesis and elaborates on future work.

EPISODE I
Timeliness Demise

Dynamic Real-Time Systems

This chapter presents real-time systems terminology as well as the relevant background to understand the contributions of this thesis and the systems and situations to which the results apply.

2.1 Real-time System Preliminaries

Real-time systems denote information processing systems which must respond to externally generated input stimuli within a finite and specified period (Young 1982). Typically, these kinds of systems are also embedded and operate in the context of a larger engineering system – designed for a dedicated platform and application.

Real-time systems typically interact with other sub-systems and processes in the physical world. This is called the *environment* of the real-time system. For example, the environment of a real-time system that controls a robot arm includes items coming down a conveyor belt and messages from other robot control systems along the same production line. Typically, there are explicit *time constraints*, associated with the response time and temporal behavior of real-time systems. For example, a time constraint for a flight monitoring system can be that once landing permission is requested, a response must be provided within 30 seconds (Ramamritham 1995). A time constraint on the response time of a request is called a *deadline*. Time constraints come from the dynamic characteristics of the environment (movement, acceleration, etc.) or from design and safety decisions imposed by a system developer. *Timeliness* refers to the ability of software to meet time

constraints (c.f. Ramamritham (1995)).

Real-time systems are sometimes called *reactive* since they react to changes in their environment, which is perceived through sensors, and influence it through different actuators. Dependable systems are computer systems where people's lives, environmental or economical value may depend on the continued service of the system (Laprie 1994). Since real-time systems control hardware that interacts closely with entities and people in the real world, they often need to be dependable.

2.2 Tasks and Resources

When designing real-time systems, software behavior is often described by a set of periodic and sporadic tasks that compete for system resources (for example, processor time, memory and semaphores) (Stankovic, Spuri, Ramamritham & Buttazzo 1998). When testing real-time software for timeliness, a similar view of software behavior is useful.

Tasks refers to pieces of sequential code that are activated each time a specific event occurs (for example, a timer signal or an external interrupt). While a task may be implemented by a single thread in a real-time operating system, a thread might also implement several different tasks¹. For simplicity, we assume a one-to-one mapping between real-time tasks and threads in this thesis. A particular execution of a task is called a *task instance*.

A *real-time application* is defined by a set of tasks that implements a particular functionality for the real-time system. The *execution environment* of a real-time application is all the other software and hardware needed to make the system behave as intended, for example, real-time operating systems and I/O devices (Burns & Wellings 2001).

There are basically two types of real-time tasks. *Periodic* tasks are activated at a fixed frequency, thus all the points in time when such tasks are activated are known beforehand. For example, a task with a period of 4 time units will be activated at times 0, 4, 8, etc. *Aperiodic* tasks can be activated at any point in time. To achieve timeliness in a real-time system, aperiodic tasks must be specified with constraints on their activation pattern. When such a constraint is present the tasks are called *sporadic*. A traditional constraint of this type is a *minimum inter-arrival time* between two consecutive task activations. In this thesis we treat all real-time tasks as being either periodic or sporadic, but constraints other than minimum inter-

¹Theoretically, a particular task may span over several threads, where each thread can be seen as a resource needed by the task

arrival times may be assumed in some cases. Tasks may also have an *offset* that denotes the time before any instance may be activated.

An assumption when analyzing timeliness of real-time systems is that the worst-case execution time (that is, the longest execution time) for each task is known beforehand. In this context, all the processor instructions which must be executed for a particular task instance contribute to the task execution time. This also includes execution time of non-blocking operating system calls and library functions used synchronously by the task. Furthermore, if tasks have critical sections, the longest execution time within such sections is assumed to be known. However, for many modern computer architectures, these figures are difficult to accurately estimate.

The reason for this difficulty is that hardware developers often optimize the performance of processors to be competitive for general computing. This is done by using mechanisms that optimize average execution times, such as multiple levels of caches and branch-predicting pipelines (Petters & Färber 1999). The combined effect of several such mechanisms increases system complexity so that the exact execution times appear non-deterministic with respect to control-flow and input data of tasks. Nevertheless, estimates based on measurements and assumptions are used during the analysis of dependable real-time systems, since no accurate and practical method exists to acquire the exact worst case execution times. These estimates may be the sources of timeliness failures.

The *response time* of a real-time task is the time it takes from when the task is activated until it finishes its execution². The response times of a set of concurrent tasks depend on the order in which they are scheduled to execute. We call this the *execution order* of tasks.

In this thesis, a *shared resource* is an entity needed by several tasks but that should only be accessed by one task at a time. Examples of such resources include data structures containing state variables that need to be internally consistent, and non-reentrant library functions. Mutual exclusion between tasks can be enforced by holding a semaphore or executing within a monitor.

Figure 2.1 exemplifies an execution order of tasks with shared resources on a single processor. The ‘||’ symbol denotes the point in time when a task is activated for execution, and the grey shading means that the task is executing with a shared resource. In the figure, task A has the highest priority (or is most urgent), and thus, begins to execute as soon as it is activated, preempting other tasks. Task B has a medium priority and shares a resource with the lower priority task C. In figure 2.1, the second instance of task B has a response time of 6 time units since it is activated

²This includes the time passing while a task is preempted or blocked

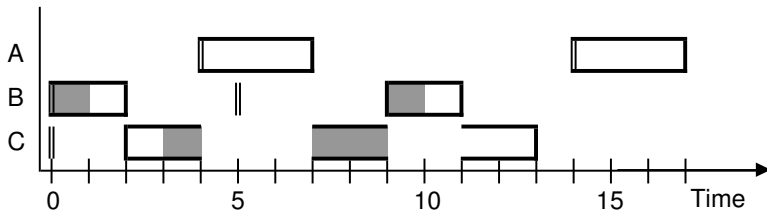


Figure 2.1: Task execution model

at time 5 and finishes its execution at time 11.

Blocking occurs when a real-time task scheduled for execution needs to use a shared resource already locked by another task. In figure 2.1 the second instance of task B is blocked from time 7 to time 9 because it has the highest priority, but it cannot execute since task C has locked a required resource. Incorrect assumptions about blocking are sources of timeliness failures.

Tasks may also be associated with criticality levels. One of the most simple classifications of criticality is that *hard* real-time tasks are expected to meet their time constraints, whereas *soft* real-time tasks can occasionally violate time constraints without causing failures.

2.3 Design and Scheduling

Since there are many different types of real-time systems this section presents classifications of real-time systems and clarifies in what context our problems and results are relevant.

Kopetz et al. describe two ways of designing real-time systems; time-triggered and event-triggered (Kopetz, Zainlinger, Fohler, Kantz, Puschner & Schütz 1991). The primary difference between them is that time-triggered systems observe the environment and perform actions at pre-specified points in time, whereas event-triggered systems detect and act on events immediately.

A pure *time-triggered* real-time computer system operates with a fixed period. At the end of each period the system observes all the events that have occurred since the period started and reacts to them by executing the corresponding tasks in the following period. The computations that are made in one period must be finished before the next period starts. Consequently, a time-triggered system design requires that information about events occurring in the environment is stored until the next period starts. The scheduling of time-triggered real-time systems is made before the system is put into operation to guarantee that all the tasks meet their deadlines,

given the assumed worst-case load. This means that a time-triggered real-time system may have to be completely redesigned if new features are added or if the load characteristics change.

In an *event-triggered* real-time system, the computer reacts to events by immediately activating a task that should service the request within a given time. The execution order of the active set of tasks is decided by a system scheduling policy and, for example, the resource requirements, criticality or urgency of the tasks.

It is also common to make a distinction between statically scheduled and dynamically scheduled systems. According to Stankovic et al. (Stankovic et al. 1998), the difference is that in statically scheduled systems all the future activations of tasks are known beforehand, whereas in dynamically scheduled systems, they are not. *Statically scheduled* systems are often scheduled before the system goes into operation, either by assigning static priorities to a set of periodic tasks or by explicitly constructing a schedule (for example, using a dispatch-table or cyclic executive). *Dynamically scheduled* systems perform scheduling decisions during operation and may change the order of task execution depending on the system state, locked resources, and new task activations. A time-triggered system is by definition statically scheduled, whereas an event-triggered system can be scheduled statically or dynamically. In this thesis, event-triggered real-time systems that are dynamically scheduled are referred to as *dynamic real-time systems*.

One advantage of dynamic real-time systems is that they do not waste resources “looking” for events in the environment. In particular, if such events seldom occur and require a very short response time (such as an alarm signal or a sudden request for an evasive action), a dynamic real-time system would waste less resources than a statically scheduled real-time system. In a statically scheduled system a periodic task would frequently have to “look” for events in the environment to be able to detect them and respond before the deadlines. In this context, resources mean, for example, processor-time, network bandwidth and electric power.

However, both paradigms have benefits and drawbacks. Statically scheduled systems offer more determinism and are therefore easier to analyze and test. One reason for this is that the known activation pattern of periodic tasks repeats after a certain period of time. This period is called a *hyper-period* and is calculated using the least common multiplier of all the inter-arrival times of periodic tasks (Stankovic et al. 1998). There are many useful results for analyzing and testing statically scheduled real-time systems (Schütz 1993, Thane 2000).

This thesis addresses problems associated with the testing of dynamic real-time systems, because such systems are suitable for many application domains but lack effective methods for testing of timeliness.

As an example of a system to which our testing methods can be applied, consider an onboard control system for a high-speed train. The system is used in many different types of rail-cars and should operate in changing environments that incorporate people, therefore it is desirable to use an event-triggered design for some parts of this system. Each car in the train set has a cluster of sensors, actuators and dedicated real-time computer nodes for the safety critical control of processes, such as the brakes and the tilt of the cars. These components are interconnected with a real-time network. On each rail car there is also a number of event-triggered real-time system nodes. The dynamic real-time system is used for monitoring and adjusting the operation of the underlying safety critical real-time control system, performing data-collection for maintenance, and for communicating with nodes in other rail cars and in the engine cockpit. The event-triggered nodes may also run other real-time applications such as controllers for air conditioners, staircases, and cabin doors. These kinds of applications need to be timely, but they are not critical for the safety of the passengers of the train.

Testing Real-time Systems

This chapter presents a relevant background to software testing, in particular concepts relating to automated and model-based testing. Furthermore, issues related to testing real-time systems are included.

3.1 Software Testing Preliminaries

Software testing means exercising a software system with valued inputs (Laprie 1994). In this thesis, the tested software system is called the *target system*. A large fraction¹ of the development cost of software is typically spent on testing. Testing is important for development of dependable and real-time software, due to stringent reliability requirements and to avoid high maintenance costs. Two purposes of testing are conformance testing and fault-finding testing (Laprie 1994). *Conformance testing* is performed to verify that an implementation corresponds to its specification while *fault-finding testing* tries to find special classes of implementation or design faults. Common to these purposes is the underlying desire to gain confidence in the system's behavior.

However, a fundamental limitation of software testing is that it is generally impossible to test even a small program exhaustively. The reason is that the number of program paths grows quickly with the number of nested loops and branch state-

¹varying from 30 up to 80 percent in the literature, depending on the type of development project and what activities are considered testing

ments (Beizer 1990). For concurrent and real-time software, the number of possible behaviors are even higher due to the interactions between concurrent tasks. Consequently, testing cannot show the absence of faults (to prove system correctness), it can only show the presence of faults (Dahl, Dijkstra & Hoare 1972).

According to Laprie et al. (Laprie 1994), *faults* are mistakes manifested in software implementations. Examples of faults are the incorrect use of conditions in if-then-else and the misuse of a binary semaphore in a concurrent program. An *error* is an incorrect internal state resulting from reaching a software fault. If no error handling is present, an error may lead to an externally observable system failure. A *failure* is a deviation from the system's expected behavior.

Testing is traditionally performed at different levels during a software development process (Beizer 1990). This thesis focuses on *system level testing* that is done on the implemented system as a whole; this typically requires that all parts of the system are implemented and integrated (Beizer 1990). The reason for this is that some timeliness faults only can be revealed at the system level (Schütz 1994). Other levels are *unit testing*, where a module or function with a specified interface is tested, and *integration testing* where the interaction between modules is tested.

A distinction is commonly made between structural and functional testing methods (Laprie 1994). In *structural* testing methods, test cases are based on system design, code structure and the systems internal architecture. In *functional* testing methods, the system under test is considered a black-box with an unknown internal structure; test cases are created based on the knowledge of system requirements and high level specification models of the system.

This thesis presents a model-based testing method where both requirements and structural knowledge are used. In particular, the method is an adapted version of mutation-based testing.

One advantage of model-based testing is that it allows test cases to be automatically generated, hence, moving effort from generating specific test cases to building a model. Models can be built before the system is implemented and used as a reference or part of a specification of the system. It is also possible to build the model directly for testing purposes after or concurrently with development. According to, for example, Beizer (1990), errors and ambiguities in the specification may be detected while building models for testing.

Mutation testing is a method in which a program (DeMillo, Lipton & Sayward 1978) or a specification model (Ammann & Black 1999) is copied and mutated so that each copy contains an artificially created fault of a particular type. Each mutated copy is called a *mutant*. Test cases are specially generated that can reveal the faults in the copies of the program or model (manually, or using reachability

tools). For example, if one occurrence of the operator ‘>’ is mutated into the operator ‘<’, then a test case must be selected that causes the mutated code to be executed and the corresponding failure to be detected. If such a test case is found, then the mutant is *killed*.

An underlying hypothesis of these methods is that the test cases generated should be able to find not only the mutations, but also many faults that are similar, hence, achieving effective coverage of the tested program or system. If a specification model is used as a source for mutation-based testing, then the implementation may actually contain some fault that was added some of the mutant models. It is thus useful that such faults correspond to ones likely to occur.

3.2 Testing Criteria

A desirable property of any testing method is to have an associated metric in which the completeness of testing can be expressed. It is not trivial to formulate such a metric, since it typically requires bounding the number of possible tests. To achieve this, different test requirements and testing criteria are used.

Test requirements are specific goals that must be reached or investigated during testing (Offutt, Xiong & Liu 1999). For example, a test requirement can be to execute a specific source code statement, to observe a specific execution order of tasks, or to cover a transition in a state-machine model.

A *testing criterion* is a way to express some class of testing requirements. Hence, examples of a test criterion can be execute all source code statements containing the letter x, execute all possible execution orders of tasks that share data, or cover all transitions in a state machine model.

Once testing criteria have been established, they can be used in two ways. They can be used to measure *test coverage* of a specific set of test cases, or they can be used during test case generation so that the constructed set of test cases implicitly fulfills an associated test criterion. A set of test cases that has been generated with the purpose of fulfilling a specific testing criterion is called a *test suite* in this thesis.

Test coverage and testing criteria are also used to express the level of ambition when testing software. That is, testing criteria sets a threshold for when an application has been sufficiently tested, and the test coverage denotes the minimum fraction of test requirements that should be covered.

Testing criteria for mutation testing are typically formulated with respect to the number of mutated programs or models. That is, the standard testing criterion is “kill all mutants”, but the thoroughness and test effort can be controlled by how

many different types of mutant are created. For example, a test suite can be required to kill all mutants where the variable ‘i’ has been replaced by the variable ‘j’.

When testing concurrent real-time systems, testing criteria for sequential software can be used on each of the possible execution orders in the system (Thane 2000). However, even for a statically scheduled real-time system the number of execution orders that need to be tested grows very quickly with the number of task activations. For dynamic real-time systems, the problem is elevated by the non-deterministic times between sporadic task activations. Hence, it is necessary to develop testing criteria for selecting relevant sub-sets of all possible execution orders.

3.3 Issues when Testing Real-Time Systems

Schütz (Schütz 1994) describes issues that need to be considered when testing real-time systems. In particular, some issues impose requirements on the test case generation methods investigated and the experiments conducted in this thesis.

Schütz uses the term *observability* for the ability to monitor or log the behavior of a tested system. Observability is usually achieved by inserting probes that reveal information about the current state and internal state-changes in the system. A problem in this context is that by introducing probes into the real-time software you actually influence the temporal behavior of the system. Hence, you cannot generally remove the probes once testing is complete without invalidating the test results. This problem is usually referred to as *probe-effect* (Gait 1986). The most common way to avoid the probe-effect problem is to leave the probes in the system, but direct their output to a channel that consumes the same amount of resources but is inaccessible during operation (Schütz 1993). A special version of this is to have a built-in component (software or hardware) that monitors the activity in the system and then leave that component in the system, or compensate for the activity of such a component during operation. In systems with scarce computing resources, the probe-effect makes it desirable to keep the amount of logging to a minimum.

Two related concepts in this context are reproducibility and controllability. *Reproducibility* refers to the property of the system repeatedly exhibiting identical behavior when stimulated with the same test case. Reproducibility is a very desirable property for testing, particularly it is useful during regression testing and debugging. *Debugging* is the activity of localizing faults and the conditions under which they occur so that the faults can be corrected. *Regression testing* is done after a fault is corrected to ensure the error is no longer present and that the repair did

not introduce new faults.

In real-time systems, and especially in event-triggered and dynamically scheduled systems, it is very difficult to achieve reproducibility. This is because the actual (temporal and causal) behavior of a system depends on elements that have not been expressed explicitly as part of the system's input. For example, the response time of a task depends on the current load of the system, varying efficiency of hardware acceleration components, etc. In this thesis, systems with this property are called *non-deterministic*.

Controllability refers to the amount of influence the tester has over the system when performing a test execution. A high degree of controllability is typically required to achieve effective testing of systems that are non-deterministic. It is also useful to have high controllability to be able to reach maximum coverage of a particular test suite when testing a non-deterministic system (see section 10.3.2 for a discussion of this).

If the target system is non-deterministic and controllability is low, testers must resort to statistical methods to ensure the validity of test results, which in turn requires that the same test case may have to be executed many times to achieve statistically significant results. A minimum requirement on controllability is that a sequence of timed inputs can be repeatedly injected in the same way.

3.4 Testing of Timeliness

The purpose of testing of timeliness is to gain confidence that an implementation of a real-time system complies to the temporal behavior assumed during the design and analysis. In particular, these assumptions must hold in situations where deviations would cause timeliness failures. It is also useful to test timeliness when the behavior of the environment deviates slightly from assumptions.

In some cases, the generation of test cases for testing of timeliness is trivial. Some scheduling analysis methods present algorithmic ways to derive worst-case situations for their assumed system models. For example, a set of periodic tasks, without shared resources using rate monotonic priority assignment, experience their worst case response time when all tasks are released simultaneously and execute their longest time (Liu & Layland 1973). Hence, releasing all the tasks simultaneously at a critical instant would be the only meaningful test case for testing timeliness for such a system. However, not all real-time systems meet the assumptions in such a simple model. For example, the worst-case response times for a set of tasks sharing mutual exclusive resources are harder to derive analyti-

cally, especially if dynamically scheduled tasks and advanced concurrency control protocols are used (see analysis of worst case response times using the EDF algorithm (Stankovic et al. 1998)). In fact, many scheduling problems with these kind of characteristics are NP-hard or NP-complete (Stankovic, Spuri, Di Natale & Buttazzo 1995). Other aspects that complicate derivation of worst-cases are sporadic tasks, tasks with multiple criticality levels sharing resources, different types of precedence constraints and arbitrary offsets. Since there is a plethora of real-time system scheduling and concurrency control protocols, it is useful to have general methods and theory for testing of timeliness.

Scheduling models often neglect inherent application semantics and causality constraints in the environment. In particular, sporadic tasks could have more complex constraints on consecutive arrivals than minimum inter-arrival times. For example, it might be known that up to three requests for a particular task activation can occur within 5 milliseconds, but after that interval, no new requests can occur for half a second. By allowing environment models to be specified more accurately, the derived situations will be more likely to correspond with the operational worst cases instead of the theoretical worst cases defined by the generic task models.

Conversely, there are several model-based test case generation methods that focus on covering a model of the environment, or a model of a real-time application that abstract away from real-time design paradigms and interactions between concurrently executing tasks (see section 11.1). Such approaches have little or no knowledge of what type of inputs cause timeliness to be violated in an event-triggered real-time system (Nilsson 2000). In this thesis, a method that is capable of exploiting both knowledge about the internal behavior of event-triggered real-time systems and complex temporal and causal relations in the environment is proposed and evaluated. The method specially focuses on finding faults that cause timeliness violations and is complementary to more general test methods that aim to cover models of the system and its input domain.

3.5 Timeliness Faults, Errors and Failures

This thesis specializes the definitions of Laprie et al. (see section 3.1) for testing of timeliness. The relation between the concepts are preserved, so that a timeliness fault may lead to timeliness error which in turn, may lead to timeliness failure.

The term *timeliness fault* denotes a mistake in the implementation or configuration² of a real-time application that may result in unanticipated temporal behaviors.

²In this context, configuration is when an application is adjusted for a specific platform.

In particular, this can become a problem if another temporal behavior is assumed during analysis and design. For example, a timeliness fault can be that a condition in a branch statement is wrong and causes a loop in a task to iterate more times than expected for a particular input. Another example is when two tasks disturb each other (for example, via unprotected shared hardware and software resources) in a unanticipated way. Both these examples of timeliness faults may lead to the timeliness error of some part of a task executing longer than expected. Another type of timeliness fault is that the environment (or sensors and actuators) behave differently than expected. For example, if an interrupt handling mechanism is subject to an unforeseen delay, then the internal inter-arrival time may become shorter than expected.

A *timeliness error* is when the system internally deviates from assumptions about its temporal behavior. This is similar to a situation where a sequential program internally violates a logical state invariant. Timeliness errors are difficult to detect without extensive logging and precise knowledge about the internal behavior of the system. In addition, timeliness errors might only be detectable and lead to system level timeliness failures for specific execution orders.

A *timeliness failure* is an externally observable violation of a time constraint. In a hard real-time system, this typically has an associated penalty or consequence for the continued operation of the overall system. Since time constraints typically are expressed with respect to the externally observable behavior of a system (or component), timeliness failures are often easy to detect once they occur.

Problem Definition: Testing Dynamic Real-Time Systems

This chapter motivates and describes the scientific problem addressed by this thesis. Furthermore, the chapter presents the thesis statement and the scientific approach taken to evaluate it.

4.1 Purpose

The purpose of this thesis is to investigate how automated testing of timeliness can be performed in a structured and effective way for real-time systems that are scheduled dynamically and have both periodic and sporadic tasks that compete for shared resources. In particular, a framework for testing of timeliness based on mutation testing is proposed and evaluated.

4.2 Motivation for Automated Testing of Timeliness

Real-time requirements are prevalent in commercial systems and there are strong reasons to believe that the need for new products with real-time requirements is increasing while the time-to-market for such systems remains short. For example, dependable real-time systems such as autonomous vehicle control systems, sensor network applications and ubiquitous computing devices with multimedia applications are being developed. Unfortunately, dynamic real-time system designs, that may be suitable for applications of this type, are difficult to analyze using existing

methods. Hence, contributions to the verification and testing of such systems are important.

A large proportion of the effort of developing software is spent on testing and verification activities (see section 3.1). However, Schütz (1994) points out that the testing phase generally is less mature than other phases of the development cycle and that the typical testing methods do not address the issues specific for testing real-time systems. In particular, industrial practitioners generally do not have access to specific methods for testing real-time properties, and the methods used for this are often case-specific or ad-hoc. Defining new practical methods for testing real-time systems is an important area of research. For example, the issue of specific test case generation methods for real-time systems is mentioned by Schütz (Schütz 1993), and there are still few approaches for doing this in a structured and effective way (see section 3.4).

The software development industry has been advocating automated testing for a long time. The main emphasis has been on automating test execution for regression testing (Rothermel & Harrold 1997). Furthermore, while various methods for automatic test case generation have been presented in the literature (c.f., DeMillo & Offutt (1993)), few reports exist of methods that are being used for development of commercial real-time systems. The advantage of automation is that it reduces the risk of human mistakes during testing, and also potentially decreases the associated time required for generating and executing test cases.

Furthermore, formal methods researchers advocate that software testing (and other structured software engineering methods) are complementary to, and should be integrated with, formal refinement for developing dependable software (Bowen & Hinchey 1995). For instance, testing of the integration between formally developed components and standard libraries, which have not been part of the formal development, may be necessary.

4.3 Problem Definition

Timeliness is one of the most important properties of real-time systems. Formal proofs, static analysis and scheduling analysis that aim to guarantee timeliness in dependable real-time systems typically require full knowledge of worst-case execution times, task dependencies, and maximum arrival rates of requests. Reliable information of this kind is difficult to acquire, and if analysis techniques are applied, they must often be based on estimations or measurements that cannot be guaranteed correct. For example, it has become increasingly complex to model a

state-of-the-art processor in order to predict timing characteristics of tasks (Petters & Färber 1999).

Further, many analysis techniques require that the tested system is designed according to specific rules. For example, all tasks may be required to be independent, or to have fixed priorities based on their periodicity or relative deadline.

In contrast to schedulability analysis and formal proofs, testing of timeliness is general in the sense that it applies to all system architectures and does not rely blindly on the accuracy of models and estimations; instead the target system is executed and monitored so that faults leading to timing failures can be detected. Hence, testing of timeliness is complementary to analysis and formal verification.

The problem in this context is the huge number of possible execution orders in dynamically scheduled and event-triggered systems (Schütz 1993). In these kinds of systems, schedules do not repeat in the same way as in statically scheduled systems and the number of execution orders grows exponentially with the number of sporadic tasks in the system (Birgisson, Mellin & Andler 1999). This makes it important to develop test selection methods that focus testing on exposing situations where timeliness failures are most likely to occur. However, it is generally not trivial to derive test cases that exercise critical execution orders (that is, the worst case interleaving of tasks) in systems that have internal resource dependencies and are dynamically scheduled. Existing methods for testing of timeliness typically only model the environment of the real-time system or cover abstract models of real-time applications. Such test case generation methods miss the impact of varying execution orders and competition for shared resources, which definitely has an impact on system timeliness (Nilsson, Offutt & Mellin 2006). A related problem when performing testing of timeliness is associated with the lack of reproducibility on dynamic real-time systems platforms (Schütz 1994). In such a system it is desirable to be able to control the execution so that a potentially critical, but rarely occurring interleaving of tasks can be tested.

These problems can be summarized:

Analysis of timeliness for dynamic real-time systems relies on assumptions that are hard to verify analytically. Hence, automated testing techniques are needed as a complement to build confidence in temporal correctness. However, existing testing methods, tools or testing criteria neglect the internal behavior of real-time systems and the vast number of possible execution orders resulting from non-deterministic platforms and dynamic environments. Consequently, the test cases that are executed seldom verify the execution orders most likely to reveal timeliness failures.

4.4 Approach and Delimitations

There are several ways of addressing the problem outlined in section 4.3. The approach taken in this thesis is to refine an existing testing method so that it can exploit the kind of models and assumptions used during schedulability analysis. Such a method enables systematic sampling of the execution orders that can lead to violated time constraints and, thus, can be used to assess the assumptions expressed in the real-time system model. In particular, a method based on mutation testing (see section 3.1) is proposed and evaluated for testing of timeliness.

We conjecture that mutation-based testing is suitable for refinement since it is a mature, strong and adaptable testing method. In this context, mature means that the original mutation test method has existed and evolved for over twenty years (DeMillo et al. 1978). It is a strong testing method in the sense that it is often used for evaluating the efficiency of other, less stringent, testing methods (Andrews, Briand & Labiche 2005). Furthermore, many previous adoptions of mutation-based testing, for example, for safety-critical (Ammann, Ding & Xu 2001) and object-oriented software exist (Ma, Offutt & Kwon 2005).

However, it is not possible to directly apply mutation-based testing for the timeliness testing problem. For example, a specification notation that captures the relevant design properties of dynamic real-time systems must be adopted and the test case generation method must be modified so that it copes with dynamic real-time systems. This kind of issues are addressed by this thesis. Our hypothesis is outlined in subsection 4.4.1 and our objectives are in subsection 4.4.2.

4.4.1 Thesis Statement

Based on the above observations, the following thesis statement is proposed:

Mutation-based testing can be adapted for automated testing of timeliness in a way that takes internal system behaviors into consideration and generates effective test cases for dynamic real-time systems

This thesis statement can be divided into the following sub-hypotheses:

- **H1:** There is a real-time specification notation which captures the relevant internal behavior of dynamic real-time systems so that meaningful mutation operators can be defined.

- **H2:** Test cases for testing of timeliness can automatically be generated using mutation-based testing and models of dynamic real-time systems.
- **H3:** Test cases generated using mutation-based testing are more effective than random test cases¹ for revealing timeliness errors on a realistic target platform.

4.4.2 Objectives

The following objectives are formulated as steps in addressing the outlined problem. For each objective, the thesis chapter that describes related efforts is given within parentheses.

1. Identify the requirements on test cases and target system modelling notations so that structured and automated testing of timeliness can be supported (Chapter 5).
2. Adopt a notation for modelling dynamic real-time systems that enable meaningful testing criteria for timeliness to be formulated (Chapter 6).
3. Propose an automated and practical approach for generation of test cases which exploits models of dynamic real-time systems (Chapter 7).
4. Implement tool prototypes for supporting the automatic test case generation approach and for evaluating its feasibility (Chapter 8).
5. Investigate the applicability of the proposed testing method and its relative effectiveness by using it for testing timeliness on a real-time target platform (Chapter 9).

¹This could potentially be generalized to test cases generated by any method that do not consider internal state and execution orders. However, due to problems with the validation of such hypothesis, random testing is used as a base-line method.

EPISODE II
Rise of the Mutated Models

A Framework for Testing Dynamic Real-Time Systems

This chapter provides an overview of the proposed framework for testing timeliness and introduces solution specific concepts that are used in the remainder of this thesis.

5.1 Framework Overview

This section introduces central concepts that are used within the proposed framework for testing of timeliness and presents a high-level introduction of how the test framework can be applied.

Estimates of the temporal behavior of application tasks that run on the target system are expressed in a *real-time applications model*. This model also contains assumptions about the behavior of the system's environment, for example, physical laws or causality that limit certain task activations from happening simultaneously.

An *execution environment model* reflects the policies and real-time protocols that are implemented in the target system. For example, the execution environment model may express that application tasks are scheduled using EDF-scheduling, share data using monitors with FIFO semantics, and that context switches impose an overhead of two time units. The execution environment model can potentially be reused for several variations of real-time systems using the same type of platform and protocols.

When both these models are integrated, we refer to the combined model as the *real-time system model*.

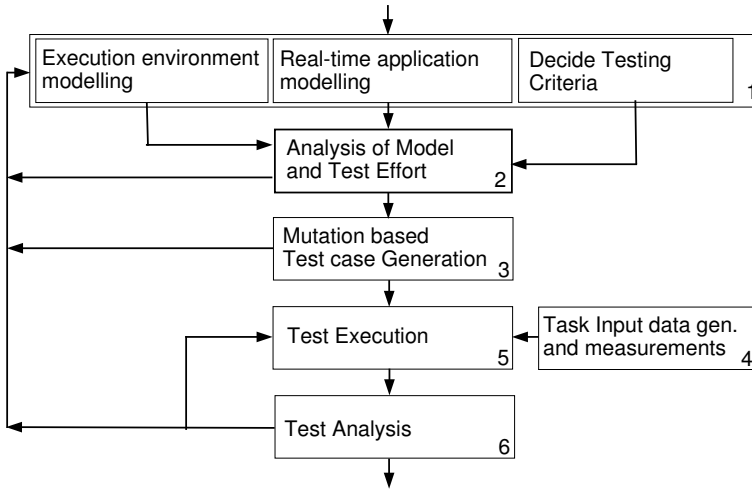


Figure 5.1: Overview of Framework Activities

Test case generation tools help testers to automatically generate test cases according to particular testing criteria (see section 3.2). The real-time application model and execution environment model are used as inputs for this.

Figure 5.1 depicts the flow of activities in the proposed testing framework. In summary, testing of timeliness according to the proposed testing framework is performed in the following way:

1. First, a real-time system model is built and testing criteria are decided upon. These activities can be in any order
 - (a) An execution environment model is configured to correspond with the architectural properties and protocols that are present in the target system.
 - (b) The temporal behaviors of real-time tasks that make up the tested real-time applications and the corresponding triggering environment entities are modelled.
 - (c) Suitable mutation-based testing criteria are selected based on the required levels of thoroughness and the allowed test effort.
2. At this phase it is possible to perform an analysis of the system model and refine the application models or testing criteria. In particular, the maximum number of test cases produced with a specific testing criterion primarily de-

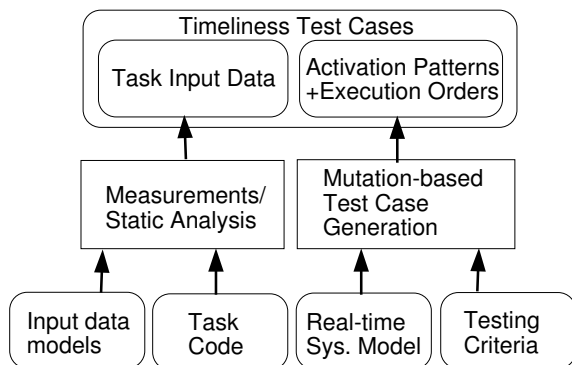


Figure 5.2: Timeliness tests overview

depends on the size of the system. The analysis of the model and testing criteria may result in refinements before proceeding with test generation.

3. Test cases are generated automatically from the model in accordance with the selected testing criterion. Based on the result of the automated test case generation, the testing criterion may also be changed.
4. Sets of input data for individual tasks are acquired using compiler based methods or temporal unit testing with measurements.
5. Each generated test case is executed repeatedly to capture different behaviors of the non-deterministic platform. Prefix-based test execution techniques can be used if the test harness and target platform supports it.
6. During test execution, the test harness produces logs that can be analyzed off-line to further refine the model or isolate timeliness faults. If a particular execution order has not been sufficiently covered more test-runs can be performed. The real-time system model, the implementation or testing criteria can be refined based on the test results and a new iteration of timeliness testing can be started.

5.2 Timeliness Test Cases

Figure 5.2 shows an overview of the data flow when generating timeliness test cases. Information from the real-time application model and execution environment

model, as explained in section 5.1, is used for generating activation patterns and execution orders.

Activation patterns are time-stamped sequences of requests for task activation. For example, an activation pattern may express that task A should be activated at times 5, 10, and 14 while task B should be activated at times 12, 17 and 23.

The execution order part of test cases predicts how tasks are interleaved in a situation where timeliness may be violated for a particular activation pattern. This is sometimes referred to as a *critical execution order*. The execution orders can be used to derive a test prefix (see section 5.4) for test execution; it can also be used during test analysis to determine if a test run has revealed a dangerous behavior.

Timeliness test cases should also specify relevant input data for the various tasks so that their execution behavior can be, at least partially, controlled during timeliness testing. *Task input data*, in this context, are the values that are read by tasks throughout their execution. For example, a task for controlling the temperature in a chemical process might read the current temperature from a memory mapped I/O port and the desired temperature from a shared data structure. Both these values influence the control flow of the task and, thus, decide the execution behavior of the task. Typically, it is interesting to use input data that cause long execution times or cause shared resources to be used a long time.

There are several ways of obtaining such input data for tasks running undisturbed. For example, Wegener, StHammer, Jones & Eyres (1997) applied a method based on genetic algorithms to acquire test data for real-time tasks. Petters & Färber (1999) used compiler based analysis for the same purpose. Further, deriving task input data is similar to deriving input data for unit testing of sequential software; hence, methods from that domain can be adapted to ensure a wide range of execution behaviors is covered. Common to all such approaches is that they require the actual implementation and information about the input domain of tasks. Hence, these requirements are inherited by the framework for testing of timeliness. In this thesis, we assume that suitable input data for the various tasks are available and focus on generating activation patterns and execution orders for testing system timeliness when several tasks execute concurrently.

5.3 Mutation-based Test Case Generation

Mutation-based testing of timeliness is inspired by a specification-based method for automatic test case generation presented by Ammann, Black and Majurski (Ammann, Black & Majurski. 1998). The main idea behind the method is to sys-

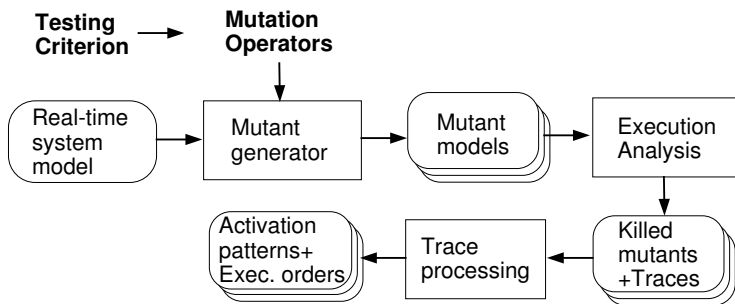


Figure 5.3: Mutation-based test case generation

tematically “guess” what faults a system contains and then evaluate what the effect of such a fault would be. Once faults with severe consequences are identified, specialized test cases are constructed that aim to reveal such faults in the system implementation.

Figure 5.3 illustrates the automated test generation process. In this figure, rounded boxes denote artifacts whereas rectangles denote some type of processing or transformation. The inputs to mutation-based testing of timeliness are a real-time system model and a testing criterion. As mentioned in section 5.1, the real-time system model contains assumptions about the temporal behavior of the target system and its environment.

A *mutation-based testing criterion* specifies what mutation operators to use, and thus, determines the level of thoroughness of testing and what kinds of test cases are produced. *Mutation operators* are defined to change some property of the real-time system model, to mimic faults and potential deviations from assumptions that may lead to timeliness violations. For example, a mutation operator for timeliness testing may change the execution time of a critical section.

A mutant generator tool applies the specified mutation operators to the real-time model and sends each mutated copy of the model for execution order analysis (marked “execution analysis” in figure 5.3). Execution order analysis determines if and how a specific mutation can lead to a timeliness failure. Execution order analysis can be performed in different ways. In this thesis, two complementary approaches are proposed, model-checking (section 6.5) and heuristic-driven simulation (section 7.2). If execution analysis reveals non-schedulability (or some other timeliness failure) in a mutated model, it is marked as killed. A mutated model containing a fault that can lead to a timeliness failure is called a *malignant mutant* whereas one containing a fault that cannot lead to a timeliness violation is called a

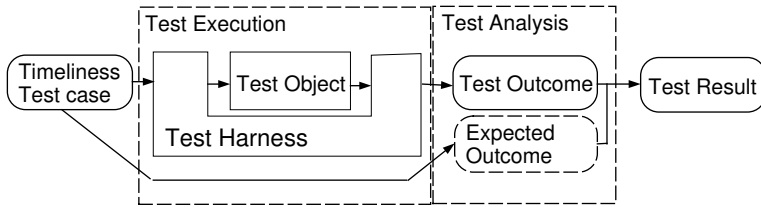


Figure 5.4: Timeliness test execution overview

benign mutant. Ideally, an execution order analyzer should always be able to kill all malignant mutants. Traces from killed mutated models are used to extract an activation pattern with the ability to reveal faults similar to the malignant mutant model in the system under test. It is also possible to extract the corresponding execution order of tasks that leads to deadline violations from such traces.

5.4 Test Execution

Figure 5.4 shows the components and artifacts associated with test execution and test analysis.

Test execution is the process of running the target system and injecting stimuli according to the activation pattern part of the timeliness test case. Since the target platform contains sources of non-determinism, several execution orders may occur in the real system for the same activation pattern. Consequently, a minimal requirement for applying the testing framework is that each activation pattern automatically can be injected repeatedly (see section 3.3). Each single execution of a test case is called a *test run* (that is, a test case execution may consist of one or more test runs). The outputs collected from the system during test execution are collectively called a *test outcome*.

The execution orders from test cases provide the ability to determine when a potentially critical execution order has been reached. Optionally, more advanced test execution mechanisms, such as prefix-based testing (Hwang, Tai & Hunag 1995), can be used to increase the controllability during test execution. In that case, the system is initialized in a specified prefix state before each test run starts, increasing the probability that a particular execution order can be observed. A discussion of how this kind of test execution can be supported for timeliness testing, is available in section 10.3.2.

The *test harness* incorporates all the software needed for controlling and ob-

servicing a test execution of the target system. The design of the test harness is critical for real-time systems since parts of it must typically be left in the operational system to avoid probe effects (see section 3.3). To be specific, the test object in figure 5.4 is the part of the target system that is being tested.

During *test analysis*, the test outcome produced by the system during test execution is analyzed with respect to the expected outcome. A *test result* indicates whether a test execution succeeded in revealing an error or not. For timeliness testing, the most relevant test outcomes are the execution orders and temporal behavior observed during a test run. Expected outcomes are that the specified timing constraints are met and, optionally, that no critical execution order is observed. If mutation-based testing has been used to generate a test case, then both the mutated and original models can be used for test analysis.

Test analysis can be made both off-line or on-line. One motivation for off-line test analysis is to avoid probe-effects, another is that more resources are typically available for the algorithms which process the test outcome. It might also be easier to predict the expected correct behavior after certain variables have been instantiated during the test run. The advantage of on-line test analysis is that test results can be acquired at run-time, making it possible to perform error handling or capture additional data for debugging. The software that performs test analysis is sometimes called a *test oracle*.

Mutation-based Testing Criteria for Timeliness

This chapter introduces mutation-based testing criteria for testing of timeliness. In addition, a method for automated test case generation using these criteria is described.

6.1 Motivation

As mentioned in section 3.2, testing criteria are used to specify goals for testing. Further, when a testing activity is limited in time, testing criteria are useful for reporting the achieved level of coverage for each tested module or property. For example, a test report may state that a module has been tested up to 78 percent statement coverage, or that 85 percent of all pairwise combinations of startup parameters have been tested.

In the same way, it is useful to be able to state to what degree system timeliness has been tested. However, if conventional testing criteria for sequential software are applied on all possible execution orders of a dynamic real-time system (as suggested for statically scheduled systems (Thane 2000)), then it becomes practically impossible to reach full coverage. The reason for this is the exponential growth of execution orders in event-triggered systems (Schütz 1993). Hence, covering all execution orders in dynamic real-time systems is generally unfeasible and comparable to full path coverage of sequential software.

Less ambitious and yet effective test criteria are needed for testing dynamic real-time systems. Further, it is desirable to focus testing efforts on the execution

orders where small deviations in assumptions may cause timeliness violations.

To formulate such testing criteria and generate test cases automatically, a notation for modelling dynamic real-time systems is needed, as described in section 6.2. Based on this notation, mutation operators and corresponding testing criteria are defined in sections 6.3 and 6.4 respectively. Section 6.5 describes our method for generating timeliness test cases using model-checking; this method is subsequently used in a evaluation experiment described in section 6.6.

6.2 Adopted Modelling Formalism

The Timed Automata (TA) (Alur & Dill 1994) notation has been used to model different aspects of real-time systems and to generate test cases that do not take execution orders and platform information into consideration (e.g., Petitjean & Fochal (1999)). An extension of timed automata, Timed Automata with Tasks (TAT) was presented by Norström, Wall and Yi (Norström, A.Wall & Yi 1999) and refined by Fersman et al. (Fersman, Pettersson & Yi 2002) (Fersman 2003). TAT includes explicit means of modelling scheduling and execution behavior of concurrent real-time tasks, so it is suitable as a source for mutation-based testing of timeliness.

A timed automaton (TA) is a finite state machine extended with a collection of real-valued clocks. Each transition can have a guard, an action and a number of clock resets. A guard is a condition on clocks and variables, such as a time constraint. An action makes calculations, resets clocks and assigns values to variables. The clocks increase uniformly from zero until they are individually reset by an action. When a clock is reset, it is instantaneously set to zero and then starts to increase uniformly with the other clocks. TAT extends the TA notation with a set of real-time tasks P . The elements of the set P represent tasks that perform computations in response to an activation request. Formally, we use definition 6.1 from Fersman (2003). In her definition, Act is a finite set of actions and ζ a finite set of real-valued variables for clocks. $B(\zeta)$ denotes the set of conjunctive formulae of atomic constraints in the form $x_i \sim C$ or $x_i - x_j \sim D$ where $x_i, x_j \in \zeta$ are clocks, $\sim \in \{\leq, <, =, \geq, >\}$ and C & D are natural numbers.

Definition 6.1 *A timed automaton extended with tasks over actions Act , clocks ζ and tasks P is a tuple $\langle N, l_0, E, I, M \rangle$ where*

- $\langle N, l_0, E, I \rangle$ is a timed automaton where
 - N is a finite set of locations
 - $l_0 \in N$ is the initial location
 - $E \subseteq N \times B(\zeta) \times Act \times 2^\zeta \times N$

ID	c	d	SEM	PREC
A	4	12	{}	{}
B	2	8	{}	{}

Table 6.1: Task set for example TAT

- $I : N \mapsto B(\zeta)$ is a function assigning each location with a clock constraint
- $M : N \mapsto P$ is a partial function assigning locations to tasks in P

A state in TAT is defined by a triple consisting of the current location of the automaton, instantiations for all data variables and clocks, and a task queue. The task queue contains tuples with the remaining execution time as well as the time remaining until the deadline is reached for each task instance. The first task instance in the queue is assumed to be executing and its remaining execution time decreases when the clocks progress. A task instance is placed into the task queue when a location associated with the task is reached. If the remaining execution time reaches 0 for the first task in the queue, it is removed, and the next task starts to execute. Preemptions occur when the first task in the queue is replaced before its remaining execution time is zero. Parallel decomposition of multiple TA and TAT automata can be done with synchronization functions (Fersman 2003).

Shared resources are modelled by a set of system-wide semaphores, R , where each semaphore $s \in R$ can be locked and unlocked by tasks at fixed points in time, $0 \leq t_1 < t_2 \leq c_i$, relative to the task's start time. Elements in P express information about task types as quadruples $(c, d, SEM, PREC)$. The variable c is the required execution time and d is the relative deadline. These values are used to initialize the remaining execution time and time until deadline when a new instance is put in the task queue. The set SEM contain tuples of the form (s, t_1, t_2) where t_1 and t_2 are the lock and unlock times of semaphore $s \in R$ relative to the start time of the task. In this notation, precedence constraints are modelled as relations between pairs of tasks, hence, $PREC$ is a subset of P that specifies which tasks must precede a task of this type.

The precedence constraints and semaphore requirements impact the task queue so that only tasks with their execution conditions fulfilled can be put in the first position of the schedule (Fersman et al. 2002).

As an example, consider the TAT model in figure 6.1, and the simple task set in table 6.1. All clocks start at zero, the task queue starts empty and the automaton is in the Initial location. Two types of transitions may occur: (i) time may progress an arbitrary time period (delay transitions), increasing all clock variables, or (ii) a transition to location Alpha can be taken (an action transition). In this example, two

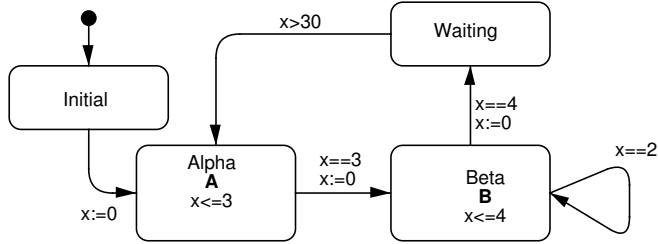


Figure 6.1: Timed automaton with tasks

Transition	State < location;clocks;queue >
Delay: 0 → 10	< Initial; x = 10; q[] >
Initial → Alpha	< Alpha; x = 0; q[(4, 12)] >
Delay: 10 → 13	< Alpha; x = 3; q[(1, 9)] >
Alpha → Beta	< Beta; x = 0; q[(2, 8), (1, 9)] >
Delay: 13 → 15	< Beta; x = 2; q[(1, 7)] >
Beta → Beta	< Beta; x = 2; q[(1, 7), (2, 8)] >
Delay: 15 → 17	< Beta; x = 4; q[(1, 6)] >
Beta → Waiting	< Waiting; x = 0; q[(1, 6)] >

Table 6.2: Trace of possible execution order

things happen on a transition to Alpha. First, Alpha is associated with task type A, so an instance of task A is added to the task queue. Second, the transition to Alpha contains a reset, so clock x is set to zero. Location Alpha has an invariant of $x \leq 3$ and a transition to Beta is taken when x is 3, so the automata waits exactly 3 time units. However, table 6.1 defines an execution time of 4 time units for A, so it does not complete at this point if preemptive EDF scheduling is assumed. When location Beta is reached, an instance of task B is added to the task queue. Tasks of type B have a shorter deadline than tasks of type A, so the task queue sorts on the earliest deadline and preempts the task of type A. The automaton stays in location Beta for 4 time units. After 2 time units, task B finishes its execution (before its deadline), and task A executes to completion. If the transition on $x == 2$ is taken (allowed but not required), another task of type B is released. This allows multiple task execution patterns. Table 6.2 provides a trace over the state changes for this simple example.

In the TAT notation, task execution times are fixed. This may appear unrealistic if the input data to a task is allowed to vary. However, to divide and conquer the

testing problem, the test case generation step assumes that each task is associated with a particular (typical or worst-case) equivalence class of input data (see the case-study in chapter 9 for an example of how this can be handled).

6.3 Mutation Operators

As mentioned in section 5.3, the basis for mutation testing is the set of operators. Mutation operators usually model possible faults, therefore the first step in designing mutation operators is to understand the types of faults and errors that can lead to timeliness failures. We have identified two categories of timeliness faults.

The first category represents incorrect assumptions about the system behavior during schedulability analysis and design. This includes assumptions about the longest execution paths, use of shared resources, precedence relations, overhead times and cache efficiency. If the wrong assumptions are used then unforeseen execution orders may cause deadlines to be missed. Furthermore, incorrect assumptions about task attributes may result in incorrect behavior of admission control and overload management protocols used in more complex real-time system architectures (for example, as described by Hansson, Son, Stankovic & Andler (1998)).

The second category represents the system's ability to cope with unanticipated discontinuities and changes in the load, for example, disturbances in the sampling or in the constraints on the activation patterns of sporadic tasks. Faults of this type may come either from the implementation of the mechanism that trigger task execution, or from incorrect modelling of the operational environment. If the operational environment differs from the assumptions, even an extensively tested system may fail during operation.

Mutation operators based on these two categories are described in this section. For six operators, a parameter is used to indicate constant differences in time. This time can be tuned by the tester to yield mutants that are easier or harder to kill, hence, Δ is used to denote the size of the change. In the following descriptions, the variable n denotes the number of tasks in the system's task set and r is the number of shared resources requiring mutual exclusion. In some systems, the same resource is locked and released multiple times by the same task (for example, to decrease blocking times), therefore l denotes the maximum number of times a resource is locked by the same task.

6.3.1 Task Set Mutation Operators

For each operator described below, the points in time when a resource is locked and unlocked are bound by zero and the assumed execution time of the task. If the mutation operator causes the lock or unlock times of a resource to fall outside of this interval, it is adjusted to the appropriate endpoint.

Execution time mutation

Execution time operators change the assumed execution time of a task by a constant time delta. These mutants represent an overly optimistic estimation of the worst-case (longest) execution time of a task or an overly pessimistic estimation of the best-case (shortest) possible execution time. For each task in a task set, two mutants are created, one increases the execution time by delta time units, and one decreases the execution time. The number of mutants created is $2n$.

Estimating worst-case execution times is known to be a difficult problem (Burns & Wellings 2001, Petters & Färber 1999). In addition, the execution time when running a single task concurrently with other tasks may be longer than running the task uninterrupted if there are caches and pipelines in the target platform (due to cache misses and resets of the pipeline). The $\Delta-$ execution time operator is relevant when multiple active tasks share resources in the system. A shorter than expected best-case execution time of a task may lead to a scenario where a lower priority task locks a resource and blocks a higher priority task so that it misses its deadline.

Mutation Operator 1 : $\Delta+$ execution time

Given a TAT model with task set P , for every task $(c_i, d_i, SEM_i, PREC_i) \in P$, create one mutant in which the execution time c_i is changed to $c_i + \Delta$.

Mutation Operator 2 : $\Delta-$ execution time

Given a TAT model with task set P , for some task $(c_i, d_i, SEM_i, PREC_i) \in P$, create one mutant in which the execution time c_i is changed to $c_i - \Delta$.

Hold time shift mutation:

The delta +/- hold time shift mutation operator changes the interval of time a resource is locked. For example, if a semaphore is to be locked at time 2 and held until time 4 in the original model, a $\Delta+$ hold time shift mutant (with $\Delta = 1$) would cause the resource to be locked from time 3 until time 5. The maximum number of mutants of this type created is $2nrl$.

Execution times differ and external factors vary in how they disturb the execution times, making it hard to accurately predict when a resource is acquired and

released relative to the start of the task. The execution time before a resource is allocated may also take more or less time than expected. This introduces new chains of possible blocking.

Mutation Operator 3 : $\Delta+$ hold time shift

Given a TAT model with task set P , for every task $(c_i, d_i, SEM_i, PREC_i) \in P$ and every semaphore use $(s, t_1, t_2) \in SEM_i$, create one mutant in which t_1 is changed to $\min(t_1 + \Delta, c_i)$ and t_2 is changed to $\min(t_2 + \Delta, c_i)$.

Mutation Operator 4 : $\Delta-$ hold time shift

Given a TAT model with task set P , for every task $(c_i, d_i, SEM_i, PREC_i) \in P$ and every semaphore use $(s, t_1, t_2) \in SEM_i$, create one mutant in which t_1 is changed to $\max(0, t_1 - \Delta)$ and t_2 is changed to $\max(0, t_2 - \Delta)$.

Lock time mutation

The delta +/- lock time mutation operator increases or decreases the time when a particular resource is locked. For instance, if a resource is to be locked from time 2 until 4 in the original model, a $\Delta+$ lock time mutant (with $\Delta = 1$) causes the resource to be locked from time 3 to 4. The maximum number of mutants created is $2nrl$.

An increase in the time a resource is locked may increase the maximum blocking time experienced by a higher priority task. This mutation operator requires test cases that can distinguish an implementation where a resource is held too long from one where it is not. Furthermore, if a resource is held for less time than expected, the system may get different execution orders that can result in timeliness violations.

Mutation Operator 5 : $\Delta+$ lock time

Given a TAT model with task set P , for every task $(c_i, d_i, SEM_i, PREC_i) \in P$ and every semaphore use $(s, t_1, t_2) \in SEM_i$, create one mutant in which t_1 is changed to $\min(t_1 + \Delta, t_2)$.

Mutation Operator 6 : $\Delta-$ lock time

Given a TAT model with task set P , for every task $(c_i, d_i, SEM_i, PREC_i) \in P$ and every semaphore use $(s, t_1, t_2) \in SEM_i$, create one mutant in which t_1 is changed to $\max(0, t_1 - \Delta)$.

Unlock time mutation

Unlock time mutation operators change when a resource is unlocked. For example, if a resource is to be locked from time 2 until 4 in the original model, a $\Delta+$

unlock time mutant (with $\Delta = 1$) causes the resource to be held from time 2 to 5. The maximum number of mutants created is $2nrl$.

Mutation Operator 7 : $\Delta+$ unlock time

Given a TAT model with task set P , for every task $(c_i, d_i, SEM_i, PREC_i) \in P$ and every semaphore use $(s, t_1, t_2) \in SEM_i$, create one mutant in which t_2 is changed to $\min(t_2 + \Delta, c_i)$.

Mutation Operator 8 : $\Delta-$ unlock time

Given a TAT model with task set P , for every task $(c_i, d_i, SEM_i, PREC_i) \in P$ and every semaphore use $(s, t_1, t_2) \in SEM_i$, create one mutant in which t_2 is changed to $\max(t_1, t_2 - \Delta)$.

Precedence constraint mutation

Precedence constraint mutation operators add or remove precedence constraint relations between pairs of tasks. This represents situations where a precedence relation exists in the implementation that is not modelled during the analysis of the system, or a precedence constraint was not implemented correctly. An additional or missed precedence constraint may cause two tasks to be executed in the wrong order, causing a task to break its deadline. These faults are not necessarily found by other types of testing, since the program's logical behavior may still be correct.

Precedence constraints can be modelled as relations between pairs of tasks. The mutation operators add a precedence relation between pairs of tasks in the task set. If there already is a precedence constraint between the pair, the relation is removed. Hence the total number of mutants generated by these mutation operators is $n^2 - n$.

Mutation Operator 9 : Precedence constraint -

Given a TAT model with task set P and task $p_i \in P$, for each task $p_j \in P$, if $p_j \in PREC_i$, create a mutant by removing p_j from $PREC_i$.

Mutation Operator 10 : Precedence constraint +

Given a TAT model with task set P and task $p_i \in P$, for each task $p_j \in P$, if $p_j \notin PREC_i$, create a mutant by adding p_j to $PREC_i$.

6.3.2 Automata Mutation Operators

The following mutation operators test the consequences of faults in the load hypothesis or model of the system environment. These mutation operators assume that each task is controlled by a separate, parallel decomposed TAT. Mutants are created from the sub-automata processes in isolation and are then composed with the rest of the model to form the TAT specifying the whole system. The parallel decomposition works in the same way as in ordinary timed automata (Larsen, Pettersson & Yi 1995).

Inter-arrival time mutation

This operator decreases or increases the inter-arrival time between requests for a task execution by a constant time Δ . This reflects a change in the system's environment that causes requests to come more or less frequently than expected. The inter-arrival time operator is important in event-triggered systems when the temporal behavior of the environment is unpredictable and cannot be completely known during design. The resulting test cases stress the system to reveal its sensitivity to higher frequencies of requests. For periodic tasks, a decrease in invocation frequency might also result in potential errors, for example if the system has been built using harmonic frequency assumptions and offsets. The maximum number of created mutants is $2n$.

Mutation Operator 11 : Δ - inter-arrival time

Given a TAT automaton $\langle N, l_0, E, I, M \rangle$ with clock set ζ and task set P , for every location $l_i \in M$ with clock constraint $(x < C) \in I(l_i)$ and outgoing transitions from l_i with guards $(x \geq C) \in B(\zeta)$, create a mutant that changes the natural number constant C to $C - \Delta$ in both the clock constraint and guards on all outgoing transitions from l_i .

Mutation Operator 12 : Δ + inter-arrival time

Given a TAT automaton $\langle N, l_0, E, I, M \rangle$ with clock set ζ and task set P , for every location $l_i \in M$ with clock constraint $(x < C) \in I(l_i)$ and outgoing transitions from l_i with guards $(x \geq C) \in B(\zeta)$, create a mutant that changes the natural number constant C to $C + \Delta$ in both the clock constraint and guards on all outgoing transitions from l_i .

Pattern offset mutation:

Recurring environment requests can have default request patterns that have offsets relative to each other. This operator changes the offset between two such patterns by Δ time units. This operator is relevant when the system assumes that two

recurring event patterns have a relative offset so that their associated tasks cannot disturb each other. An example of a fault causing this type of deviation is when periodic tasks are implemented using relative delay primitives that do not compensate for cumulative drifts (Burns & Wellings 2001). Since this operator type changes offsets on a per-task basis the maximum number of created mutants is $2n$.

Mutation Operator 13 : $\Delta+$ pattern offset

Given a TAT automaton process $\langle N, l_0, E, I, M \rangle$ with clock set ζ , task set P and initial location l_0 with clock constraint $(x < C) \in I(l_0)$ and outgoing transitions leading from l_0 with guards $(x \geq C) \in B(\zeta)$, create one mutant by changing the natural number constant C to $C + \Delta$ in both the clock constraint and guards on all outgoing transitions from l_0 .

Mutation Operator 14 : $\Delta-$ pattern offset

Given a TAT automaton process $\langle N, l_0, E, I, M \rangle$ with clock set ζ , task set P and initial location l_0 with clock constraint $(x < C) \in I(l_0)$ and outgoing transitions leading from l_0 with guards $(x \geq C) \in B(\zeta)$, create one mutant by changing the natural number constant C to $C - \Delta$ in both the clock constraint and guards on all outgoing transitions from l_0 .

6.4 Mutation-based Testing Criteria

Within the testing framework proposed in this thesis, testing criteria for timeliness is formed by combining mutation operators, such as defined in section 6.3. For example, a testing criterion may require that a set of test cases which causes all malignant “2+ Execution time” and all “4- Inter-arrival time” mutants to reveal a missed deadline is generated and run on the target system.

This thesis proposes a set of basic testing criteria that corresponds with the mutation operators formally defined in section 6.3. However, it is the users of the testing framework who to decide which of these mutation operators should be used for a particular testing project. Furthermore, it is possible to add customized mutation operators to the framework by adding, removing or changing entities and relations in the system model. This is discussed in section 10.1.2.

For the basic testing criteria proposed in this thesis, the number of test requirements grows with the (structural) size of the model from which mutants are generated. In particular, the maximum number of mutants grows with the number of tasks (n), resources (r), and mutual exclusive segments (l), as described in section 6.3 and summarized in table 6.3. These testing criteria are not directly affected by

Mutation operator type	Max. Mutants
Execution time	n
Hold time shift	nrl
Lock time	nrl
Unlock time	nrl
Precedence constraint	n^2
Inter-arrival time	n
Pattern offset	n

Table 6.3: Maximum number of mutants generated by each operator

the size of the reachable state space¹ of the real-time system model or the implementation and, therefore, an exponential increase of test requirements (things to cover) is avoided. An effect of this is that mutation-based testing criteria are more scalable for dynamic real-time systems than testing criteria which require a fraction of all execution orders to be covered.

Since the model used for test case generation is based on assumptions about the system under test, and not the structure or implementation of the real system, it is possible that the actual mutations of the model correspond with real differences between the model and the implementation (timeliness errors). Hence, if mutation operators correspond to deviations that are likely to occur and might lead to timeliness violations, then test cases are generated that target these types of faults. This relation between mutants and implementation faults makes it desirable to have a comprehensible set of mutation operators. *Comprehensiveness* means that for all types of errors which might lead to timeliness violations in a particular system, there is a corresponding mutation operator. However, it is difficult to reach a comprehensive set of meaningful mutation operators, especially since the combined application of mutation operators that individually cannot create malignant mutants may become meaningful. In this context, a mutation operator is *meaningful* if it creates at least one malignant mutant. Further, there might be several fault types that are specific for a particular execution environment mechanism or design style. Examples of such mutation operators, and guidelines for how to extend the testing framework with new mutation operators are presented in section 10.1.

Similar to other types of mutation-based testing, mutation-based testing of timeliness relies on the assumption that the generated test cases are also useful

¹The reachable state space of a state machine (or program) is comprised of all states that might be reached from the initial state, given any possible input (Fersman 2003)

for detecting faults that do not exactly correspond to any mutant (see section 3.1).

6.5 Generation of Test Cases Using Model-Checking

Given a real-time system specified in TAT and an implementation of the aforementioned mutation operators, model-checking can be used to kill mutants, that is, finding a trace that shows how a task in the mutated model violates a time constraint. The mechanism that makes this possible is the generic schedulability analysis using TAT model-checking described by Fersman et al (Fersman et al. 2002). This mechanism is implemented in the Times tool, developed at Uppsala University (Amnell, Fersman, Mokrushin, Pettersson & Yi 2002).

Once a real-time system has been modelled in TAT, an implementation of the mutation operators, defined in section 6.3, automatically produce each mutant and send it to the model checker for schedulability analysis. If the mutant is malignant the schedulability analysis results in a trace of transitions over the automata model that leads to a violated timing constraint.

From a timeliness testing perspective, the points in time in which different sporadic tasks are requested are interesting to use as activation patterns (a part of timeliness test cases, as defined in section 5.2). Hence, by parsing the automata trace it is possible to extract the points in time where stimuli should be injected in order to reveal a fault in a system represented by the mutated model.

In the trace from TAT model-checker, such points are characterized by transitions to task activation locations (that is, a location l_i where $M(l_i)$ maps to an element in P). For each such transition, the task associated with the activation location, $M(l_i)$, together with the current global time, are added to the activation pattern part of the derived test case.

The same trace contains information of how the system is assumed to behave internally for the timeliness failure to occur. This can be extracted by monitoring the state changes over the set of active tasks in the ready queue (see table 6.2 for an example). This information can be used during test case execution and test analysis, as described in section 5.2.

The main advantage with this approach is that, theoretically, if a mutant is malignant it is killed by the model-checker, given sufficient time and resources for the analysis.

The model-checking approach for generating test cases can be used with arbitrary complex automata modelling constraints on task activation patterns (in the environment of the real-time system) without changing the modelling notation or

model-checker implementation. This makes it possible to model systems where some sporadic tasks always arrive in clusters and others never occur at the same time. Hence, the test cases generated with this approach comply to or are close to the operational worst-cases of a system instead of artificial constraints on activation patterns such as minimum inter-arrival times.

However, when system models increase in size and a larger fraction of triggering automata becomes more non-deterministic, the analysis performed by the model-checker suffers from reachable state space explosion. A further limitation of the model-checking approach is the assumption that the target system complies with a limited set of execution environment models supported by the model-checker tool. If the execution environment model must be changed, that change must be done in the “back-end” of the model-checker². It is unclear how such changes affect the effectiveness of schedulability analysis.

6.6 Validation of Mutation Operators

To validate that the proposed mutation operators are meaningful a test case generation experiment was performed. If mutants are killed with this approach it means the faults introduced in the model indeed represent faults that may cause timeliness violation in a real-time system.

This case study uses a small task set that has a simple environment model but relatively complex interactions between the tasks. Static priorities were assigned to the tasks using the *deadline monotonic scheme*, that is, the highest priority is given to the task with the earliest relative deadline. Arbitrary preemption is allowed.

The model-checker implements a variant of the *immediate priority ceiling* protocol to avoid priority inversion. That is, if a task locks a resource then its priority becomes equal to the priority of the highest priority task which might use that resource, and is always scheduled before lower prioritized tasks. The FCFS policy is used if several tasks have the same priority.

The model has five tasks. Two are controlled by TAT models such as the one in figure 6.2. The automata are sporadic in nature but have a fixed observation grid (denoted OG in the figure), which limits when a new task instance can be released (for example, such a limitation may exist if requests may only arrive at certain slots of a time-triggered network). These experiments use an observation grid of 10 time units. The three remaining tasks are strictly periodic and controlled by generic

²In the Times tool, some changes of this type can be done by exporting the execution environment model to the Uppaal model-checker

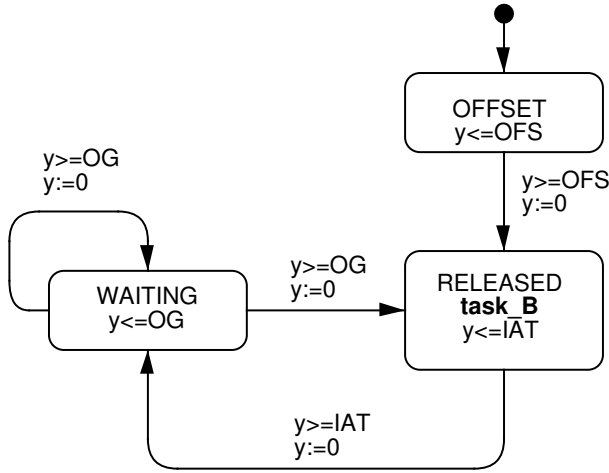


Figure 6.2: TAT for case study task

automata, with defined periods and offsets.

The system has two shared resources modelled by semaphores, and one precedence relation between tasks D and A, which specifies that a new instance of task A cannot start unless task D has executed after the last instance of A.

Table 6.4 describes the task set. The first column ('ID') indicates task identifiers, column 'c' provides execution times and column 'd' relative deadlines. Column 'IAT' provides the inter-arrival times for periodic tasks which are activated with fixed periods and the associated constraint on the sporadic task's activation pattern as exemplified by the TAT template in figure 6.2. Column 'OFS' describes the initial offsets, or the delay before the first task may be activated, column 'SEM' specifies the set of resources used and in which interval they are needed, and column 'PREC' specifies which tasks have precedence over tasks of this type.

Mutants are automatically generated according to the mutation operators with two different values for Δ . Table 6.5 divides the results into two groups, based on the Δ values. The number of mutants for each type is in column ' μ ', and the number of killed mutants of that type is in column " K_{MC} ".

The number of killed mutants corresponds to the number of counter example traces that were converted to test cases for the actual system.

ID	c	d	SEM	PREC	IAT	OFS
A	3	7	{(S1, 0, 2)}	{D}	≥ 28	10
B	5	13	{(S1, 0, 4), (S2, 0, 5)}	{}	≥ 30	18
C	7	17	{(S1, 2, 6), (S2, 0, 4)}	{}	40	6
D	7	29	{}	{}	20	0
E	3	48	{(S1, 0, 3), (S2, 0, 3)}	{}	40	4

Table 6.4: Task set for base-line experiment

Mutation operator	Δ	μ	K_{MC}	Δ	μ	K_{MC}
Execution time	1	10	6	2	10	5
Hold time shift	1	14	1	2	14	0
Lock time	1	8	1	2	8	1
Unlock time	1	11	2	2	11	2
Precedence constraint	-	20	15	-	-	-
Inter-arrival time	1	5	4	4	5	4
Pattern offset	1	10	5	4	10	4
Total	-	78	34	-	58	16

Table 6.5: Mutation and model-checking results

As revealed in table 6.5, the model checker killed at least one mutant generated for each testing criterion. This means that the errors corresponding to each mutation operator potentially can lead to a timeliness violation.

Hence, to reveal timeliness faults, testing can focus on demonstrating that these traces can occur in the system, when subject to the activation pattern extracted from the model-checker trace. A perhaps counter-intuitive observation is that some mutants that were malignant with small deltas become benign when the delta is increased.

Generation of Test Cases Using Heuristic-driven Simulations

This section presents an alternative method for generating mutation-based test cases that are suitable for flexible real-time systems.

7.1 Motivation

In chapter 6 model-checking of TAT specifications is used for the execution order analysis step of test case generation. This is possible since the counter-examples from schedulability analysis shows the execution orders that can lead to missed deadlines. Model-checking is a reliable method in the sense that timeliness violations are guaranteed to be revealed if they exist in the mutated model and the model-checker terminates. However, for large task sets the computational complexity of this type of analysis often becomes too great. In particular, the complexity of analysis increases if many tasks use shared resources and are released in response to sporadic requests¹.

In dynamic real-time systems, there is potential for many sporadic tasks and, thus model-checking may be impractical. We propose an approach for testing timeliness of such systems where a simulation of the mutant model is iteratively run and evaluated using heuristics. By using a simulation-based method instead of model-checking for execution order analysis, the combinatorial explosion of full

¹In TAT-models, each activation automaton for a sporadic task would require at least one local clock, this is known to increase analysis complexity.

state exploration is avoided.

Further, we assume that it is less time consuming to modify a system simulation to correspond with the execution environment model, compared to changing the model-checker in such a way that schedulability analysis remains feasible.

The transition from model-checking to simulation can be compared with a similar trend in automatic test data generation for sequential software, where constraint-solving (DeMillo & Offutt 1991) has been successfully complemented with dynamic approaches in which actual software executions guide the search for input values (Offutt, Jin & Pan 1999).

When simulation is used for execution order analysis, the TAT model task set is mapped to corresponding task and resource entities in a real-time simulator. The activation patterns of periodic tasks are known beforehand and are included into the static configuration of the simulator. The activation pattern for sporadic (automata controlled) tasks is varied for each simulation to find the execution orders that can lead to timeliness failures.

Consequently, a necessary input to the simulation of a particular system (corresponding to a TAT model) is an activation pattern of the sporadic tasks. The relevant output from the simulation is an execution order trace over a hyper-period (see section 3.4) of the periodic tasks where the sporadic requests have been injected according to the activation pattern. A desired output from a mutation testing perspective is an execution order that leads to a timing constraint being violated.

By treating test case generation as an heuristic search problem, different heuristic methods can be applied to find a feasible solution. In the following sections, we present and evaluate an approach based on genetic algorithms.

7.2 Genetic Algorithms for Test case Generation

Genetic algorithms (also known as evolutionary algorithms (Michalewicz & Fogel 1998)) are suggested for the heuristic search of TAT models, since they are configurable and are known for coping well with search spaces containing local optima (Michalewicz & Fogel 1998).

Genetic algorithms operate by iteratively refining a set of solutions to an optimization problem through random changes and by combining features from existing solutions. In the context of genetic algorithms, the solutions are called *individuals* and the set of solutions is called the *population*. Each individual has a *genome* that represents its unique features in a standardized and compact format. Common formats for genomes are bit-strings and arrays of real values. Consequently, users

of a genetic algorithm tool-box must supply a problem specific mapping function from a genome in any of the standard formats to a particular candidate solution for the problem. It has been argued that the mapping is important for the success of the genetic algorithm. For example, it is desirable that all possible genomes represent a valid solution (Michalewicz & Fogel 1998).

The role of the fitness function in genetic algorithms is to evaluate the optimality or *fitness* of a particular individual. The individuals with the highest level of fitness in a population and are more likely to be selected as input to cross-over functions and of being moved to the next generation.

Cross-over functions are applied on the selected individuals to create new individuals with higher fitness in the next generation. This means either combining properties from several individuals, or modifying a single individual according to heuristics. Traditionally, a function applied on a single individual is called a “mutation” but to avoid ambiguity² we use the term *cross-over functions* for all the functions that result in new genomes for the next generation.

There are generic cross-over functions that operate on arbitrary genomes expressed in the standard formats. For example, a cross-over function can exchange two sub-strings in a binary string genome or increase some random real value in an array. However, depending on the encoding of genomes, the standard cross-over functions may be more or less successful in enhancing individuals. Using knowledge of the problem domain and the mapping function, it is possible to customize cross-over functions in a way that increases the probability of creating individuals with high fitness. On the other hand, some cross-over functions must remain stochastic to prevent the search from getting stuck in local optima. A genetic algorithm search typically continues for a predetermined number of generations, or until an individual that meets some stopping criteria has been found.

In general, three types of functions need to be defined to apply genetic algorithms to a specific search problem: (i) a genome mapping function, (ii) heuristic cross-over functions, and (iii) a fitness function. The following subsections suggest such functions for mutation-based timeliness test case generation.

The performance of a genetic algorithm can also be enhanced by preparing the initial population with solutions that are assumed to have high fitness based on application knowledge. Since this type of initialization is system specific, it is not fully investigated in this thesis. However, some example enhancements of this type are discussed in section 10.2.2.

²In this thesis, the term mutation refers to changes to a real-time system model that is done before execution order analysis begins.

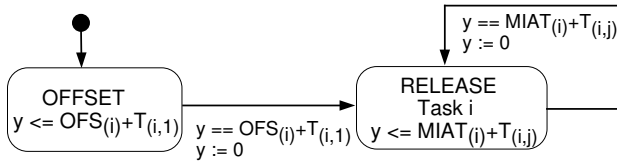


Figure 7.1: Annotated TAT template

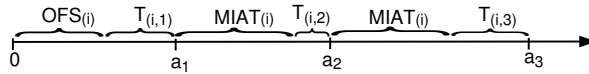


Figure 7.2: Activation pattern of sporadic task i

7.2.1 Genome Mapping Function

For the test case generation problem, only the activation patterns of non-periodic tasks vary between consecutive simulations of the same TAT model. Thus, it is sufficient that a genome can be mapped to such an activation pattern. Each activation pattern deterministically results in a particular execution order trace in the simulation. The execution order traces are the individuals for this search problem.

Figure 7.1 contains an annotated TAT-automata for describing activation patterns of sporadic tasks. Generally, activation patterns can be expressed by any timed automata, but this mapping function uses sporadic task templates, since event-triggered real-time tasks are often modelled as being sporadic (Burns & Wellings 2001). The template has two parameters that are constant for each mutant. The parameter OFS defines the assumed offset, that is, the minimum delay before any instance of this task is assumed to be requested. The parameter MIAT defines the assumed minimum inter-arrival time between instances of the sporadic task.

An array of real values $T_{(i,1..m)}$ defines the duration of the *variable delay interval* between consecutive requests of a sporadic task i . Here m is the maximum number of activations that can occur during the simulation. Figure 7.2 depicts the first three activations of a sporadic task and exemplifies the relation between the automata constants and the values in T .

By combining the arrays for n sporadic tasks in the mutant task set P we form a matrix $T_{(1..n,1..m)}$ of real values, where each row corresponds to an activation pattern of a sporadic task. The matrix T can be used as a genome representation of all valid activation patterns for the mutant model.

7.2.2 Fitness Function

Since our genome representation of valid activation patterns is meaningless without a particular TAT-mutant model, the simulation must be run with the activation pattern matrix as input before fitness can be calculated. The fitness for each individual in a population is computed in three steps. First, a genome is translated to an activation pattern of sporadic tasks. Second, a simulation of the mutant TAT model is run in which the sporadic tasks are requested according to the activation pattern.

Third, the resulting execution order trace from the simulation is used to calculate fitness. A suitable fitness for the generation of timeliness test cases should express how close a mutant is to violating a time constraint. The *slack* is the time between the response time of a task instance and its deadline (see section 2.2). Hence, measuring the minimum slack observed during a simulation is one of the simplest ways of calculating fitness. The highest fitness is given to the activation pattern that results in the simulated execution order trace with the least minimum slack. More elaborate fitness functions (for example, using weighting based on diversity or average response times) can also be evaluated for improving the performance of the genetic algorithm.

7.2.3 Heuristic Cross-over Functions

For timeliness testing, there are intuitive heuristics for what kind of activation patterns are likely to stress the mutant model. For example, it seems possible that releasing many different types of sporadic requests in a burst-like fashion is more likely to reveal timeliness violations than an even distribution of requests.

The variable M is a TAT model that contains n sporadic tasks controlled by automata templates such as in figure 7.1. As described in section 7.2.1, a genome matrix of size $n * m$ is denoted by the variable T . The integer variable i is used to index over the rows in a genome matrix T . The rows in such matrices correspond to sporadic tasks, hence it is bounded by 1 and n . The integer variable j and k is used to index over the columns in genome matrices and is bounded by 1 and m . The variable ϵ is used to denote a small positive real number. The expression $[a_{beg}, a_{end}] \sqsubseteq [b_{beg}, b_{end}]$ means that the left hand interval $[a_{beg}, a_{end}]$ is a sub-interval of the right hand interval $[b_{beg}, b_{end}]$. Formally, this can be expressed $([a_{beg}, a_{end}] \sqsubseteq [b_{beg}, b_{end}]) \iff (a_{beg} \geq b_{beg} \wedge a_{end} \leq b_{end})$.

Definition 7.1 : *Critical task instance*

The task instance with the least slack in an execution order trace.

Definition 7.2 : *Idle point*

A point in time where no real-time task executes or is queued for immediate execution on the processor.

Definition 7.3 : *Critical interval* $[ci_{beg}, ci_{end}]$

The interval between the activation time and response time of a critical task instance.

Definition 7.4 : *Loading interval* $[li_{beg}, li_{end}]$

The interval between the latest idle point and the activation time of the critical task instance. Note that $li_{end} = ci_{beg}$.

Definition 7.5 : *Delay interval matrix* D

A matrix of size $n * m$ containing the variable delay intervals associated with each sporadic task activation in T such that:

$$D_{(i,j)} = [epat(i, j), epat(i, j) + T_{(i,j)}]$$

Where $epat(i, j)$ is the earliest possible arrival time of the j 'th instance of sporadic task i given a TAT model M and a genome matrix T . For tasks activated according to the automata template in figure 7.1, $epat(i, j) = OFS_{(i)} + ((j - 1) * MIAT_{(i)} + \sum_{k=1}^j T_{(i,k)})$.

Focus Critical interval

This cross-over function analyzes the logs from the simulation to find the critical interval. A sporadic tasks activation pattern is chosen by random and changed so that requests become more likely to occur within or close to the critical interval.

Cross-over Function 1 : *Focus critical interval left*

Select any index i . Let j be the largest index such that $D_{(i,j)} \sqsubseteq [0, ci_{beg}]$ then increase $T_{(i,j)}$ with ϵ time units and decrease $T_{(i,j+1)}$ with ϵ time units.

Cross-over Function 2 : *Focus critical interval right*

Select any index i . Let j be any index such that $D_{(i,j)} \sqsubseteq [ci_{beg}, ci_{end}]$ and modify T so that $T_{(i,j)} = 0$.

Critical interval move:

All sporadic tasks activation patterns are shifted a small random period so that the sequence of sporadic activations leading up to a critical interval occurs at some other point relative to the activations of periodic tasks.

Cross-over Function 3 : *Critical interval move right*

For every index i such that $D_{(i,1)} \sqsubseteq [0, ci_{beg}]$ increase $T_{(i,1)}$ with ϵ time units.

Cross-over Function 4 : *Critical interval move left*

For every index i such that $D_{(i,1)} \sqsubseteq [0, ci_{beg}]$ decrease $T_{(i,1)}$ with ϵ time units.

New interval focus

This cross-over function generates new candidate critical intervals to keep the optimization from getting stuck in local optima. A new point in time is chosen by random and all the closest sporadic releases are shifted towards the selected point in time.

Cross-over Function 5 : *New interval focus*

Let t_{new} be a new random instant within the simulation interval. For every index i select the largest index j such that $D_{(i,j)} \sqsubseteq [0, t_{new}]$ and increase $T_{(i,j)}$ with ϵ time units. Also decrease $T_{(i,j+1)}$ with ϵ time units.

Loading interval perturbation:

Theoretically, all task activations in the loading interval may influence response time through the state in the system when the request of the critical task instance occurs (Mellin 1998). In practice, it is more likely that changes at the end of the loading interval have a direct effect on the timeliness of the critical task instance. This cross-over function changes the activation pattern at the end of the loading interval.

Cross-over Function 6 : *Loading interval perturbation*

Select any index i and a positive value ϵ . Let j be the the largest index such that $D_{(i,j)} \sqsubseteq [li_{beg}, li_{end}]$ and $j > 1$, modify T so that $T_{(i,j-1)} = \epsilon$ time units.

7.3 Testing of Time Dependent Control Performance

The only failure type considered in the previous examples and experiments is the violation of time constraints, and in particular, response time deadlines. With a heuristic driven approach it is possible to use mutation-based testing to test more complex time constraints (for example, expressed using the constraint graph notation used by Berkenkötter & Kirner (2005)) and other properties that indirectly depend on execution orders and temporal behavior of tasks.

Further, a real-time system simulator can be extended to take more aspects of the controlled system into consideration than those expressed in the real-time application model. In particular, by changing the set of variables being traced during simulation and by updating the fitness function and termination criteria, test cases can be generated that test new types of time constraints and time dependent properties. One such property is control performance.

Many real-time systems have tasks that implement control applications. Such applications interact with physical processes through sensors and actuators to achieve a control goal. For example, a painting robot may have a control application that periodically samples joint angles and sets different motor torques so that the robot movement becomes smooth and aligned with the painted object.

Control algorithms contain built-in time constraints that are more subtle than response time deadlines. The delay between sensor readings (typically done in the beginning of task execution) and the actuator control (typically done at the end of task execution) is known as *input-output latency*. This includes the time passing while the task is preempted and blocked. Hence, this interval is different from response times and execution times, as defined in section 2.2.

Excessive input-output latency compromises the performance of a control system, and may even cause instability. This means that an application may fail its control goal causing a *control failure* to occur. Since the same types of errors that cause timeliness failures may result in increased input-output latency (or a larger difference between maximum and minimum input-output latency than expected), it is possible to use the mutation-based testing criteria presented in section 6.3 for this purpose.

By co-simulating the real-time kernel execution with continuous process models (for example, using Simulink) it is possible to measure the expected control performance for a particular execution order of tasks. This means that it is possible to use mutation-based testing to automatically generate test cases that can stress a real-time control system to reveal control failures.

To support this type of test case generation, the closed-loop behavior of both the controlled process and its controller logic (as implemented by real-time tasks) must be included in the simulation. Further, control specific variables must be used for calculating the fitness for the heuristic search problem.

In order to evaluate control performance it is common to use weighted quadratic cost functions (Åström & Wittenmark 1997). For a scalar system, with one output y and one input u , the cost can be written

$$J = \int_0^{T_{sim}} \left(y^2(t) + \rho u^2(t) \right) dt \quad (1)$$

where the weight factor, ρ , expresses the relation between the two counteracting objectives of control design (*i*) to keep the regulated output close to zero and (*ii*) to keep the control effort small. The higher the cost measured during a simulation run, the worse the control performance. Therefore, we assume that $1/J$ is proportional to the control performance when calculating the fitness of a particular execution order trace³.

The fitness of a simulation trace where both timeliness of hard tasks and control performance of soft, adaptive controllers should be considered can be defined as

$$F = \sum_k \frac{1}{J_k} - S_{min} * w \quad (2)$$

where S_{min} denotes the least slack observed for any hard real-time task. The variable J_k denotes the value of J for adaptive controller k at the end of the simulation. A weight variable w is used to adjust the minimum slack so that the timeliness factor is of the same magnitude as the control quality factor.

Apart from calculating the general fitness that drives the genetic algorithm heuristics toward evaluating more fit solutions, the fitness function are also used to detect failures and halt the search. Relevant failure conditions are that (*i*) a hard deadline is missed, (*ii*) the control system becomes unstable (the cost, J , exceeds some threshold value), or (*iii*) a control constraint is violated, for example, the motion of a robot arm becomes too irregular. Failure condition (*i*) and (*ii*) can easily be detected by checking the minimum slack of hard tasks and the value of the cost function for the controller tasks. Failure condition (*iii*) is application-specific and might require checking one or several process variables.

7.4 Test Case Generation Experiments

To evaluate the proposed test case generation method, three experiments were conducted. First, we establish basic confidence in the method by applying it to a system of the same type as used in the model-checking experiment. This allows us to evaluate the reliability of the proposed test case generation method in a base-line experiment and detect if it has difficulties to kill some types of malignant mutants.

³This way of measuring control fitness was developed together with Dan Henriksson for the experiment presented in Nilsson & Henriksson (2005)

Second, we test the hypothesis that the approach based on genetic algorithms can remain effective for larger and more complex system models where the model-checking approach fails (because of the size of the reachable state space). In this experiment we automatically generate test cases from a real-time system model with a larger task set and a considerable fraction of sporadic tasks under complex real-time system protocols.

Finally, we demonstrate the flexibility of the genetic algorithm based approach by using it for generating test cases for both timeliness and control performance from an extended real-time system model. In particular, this real-time system model has a mixed load of soft and hard real-time tasks that share resources. The periodic tasks are soft and implement adaptive controllers for three inverted pendulums and the sporadic tasks are modelled to have hard deadlines.

To conduct the experiments, we extended the real-time and control co-simulation tool *TrueTime* to simulate the execution of TAT-models (see section 8.3 for a description of the added features). *TrueTime* was developed at the department of automatic control at the University of Lund to support the integrated design of controllers and real-time schedulers (Henriksson, Cervin & Årzén 2003). We also configured and extended a genetic algorithm tool-box (Houck, Joines & Kay 1995) to interact with our simulated model. For model-checking experiments we used the *Times* tool, developed at Uppsala University (Amnell et al. 2002).

7.4.1 Base-line Real-time System

This experiment uses a small task set that has simple environment models but complex interactions between the tasks. Static priorities were assigned to the tasks using the *deadline monotonic* scheme, that is, the highest priority was given to the task with the earliest relative deadline. Arbitrary preemption was allowed.

The system used the *immediate ceiling priority* protocol to avoid priority inversion (Sha, Rajkumar & Lechzky 1990). That is, if a task locks a semaphore then its priority becomes equal to the priority of the highest priority task that might use that semaphore, and is always scheduled before lower prioritized tasks. The “first come first served” policy is used if several tasks have the same priority.

The base-line setup has five tasks (denoted A-D, and listed in table 7.1). Two tasks are sporadic and the three remaining tasks are strictly periodic. The system has two shared resources and one precedence relation between tasks D and A. The precedence relation specifies that a new instance of task A cannot start unless task D has executed after the last instance of A.

Table 7.1 describes the assumptions of the task set. The first column (“ID”)

ID	c	d	SEM	PREC	MIAT	OFS
A	3	7	{(S1, 0, 2)}	{D}	≥ 28	10
B	5	13	{(S1, 0, 4), (S2, 0, 5)}	{}	≥ 30	18
C	7	17	{(S1, 2, 6), (S2, 0, 4)}	{}	40	6
D	7	29	{}	{}	20	0
E	3	48	{(S1, 0, 3), (S2, 0, 3)}	{}	40	4

Table 7.1: Task set for base-line experiment

indicates task identifiers, column ‘c’ gives execution times, and column ‘d’ provides relative deadlines. The “SEM” column specifies the set of semaphores used and in which interval they are required. Column “PREC” reveals what other tasks have precedence over tasks of this type. For sporadic tasks, the “MIAT” column contains the minimum inter-arrival time assumptions. For periodic tasks the same column contains the fixed inter-arrival time. Column “OFS” denotes the initial offset constant.

Table 7.2 contains the results from mutation testing the task set in table 7.1. A Δ value of 1 time unit was used to generate the mutants. The number of mutants generated for each operator type is listed in column ‘ μ ’ and the number of mutants killed by model-checking is contained in column “ K_{MC} ”. For the genetic algorithm setup, we used a population of 20 individuals per generation and searched each mutant for 100 generations before terminating. We used the heuristic cross-over functions described in section 7.2.3 as well as three generic cross-over functions. The first generic cross-over function changed a random value in the genome representation, the second created a new random individual in the population and the third replaced a random value in the genome with 0. To gain confidence in the results, each experiment was repeated in eight trials, each with a different random seed. The number of mutants killed using genetic algorithms in any of the trials is listed in column “ K_{GA} ”. Column “ $\overline{K_{GA}}$ ” lists the average number of malignant mutants that were killed per trial. The average number of generations required to kill malignant mutants of this type is contained in column “ \overline{GEN} ”.

As revealed in table 7.2, both the simulation-based and model-checking approaches killed all the malignant mutants. Strangely, the model-checking approach also killed some mutants that was benign using the simulator implementation. By comparing the execution orders of the benign mutants that were killed, we observed that tasks sometimes inherited ceiling priorities before they started executing in the traces produced by the model-checker. The number of mutants that were found

Mutation operator	μ	μ_M	K_{MC}	K_{GA}	$\overline{K_{GA}}$	\overline{GEN}
Execution time	10	6	6	6	5.8	7.6
Lock time	8	1	1	1	1.0	2.2
Unlock time	11	2	2	2	2.0	1.3
Hold time shift	14	0	1	0	-	-
Precedence	20	14	15	14	14.0	1.2
Inter-arrival time	10	3	4	3	3.0	5.7
Pattern offset	10	3	5	3	3.0	2.5
Total	83	29	33	29	28.8	-

Table 7.2: Results from base-line system experiment

malignant after this comparison is listed in column “ μ_M ”. We conjecture that the model-checker tool implements a different version of the immediate priority ceiling protocol than originally defined (Sha et al. 1990). Since we do not know the exact semantics and properties of the model-checker’s implementation of the protocol, we use the original definition⁴.

An interesting observation is that all the malignant mutants were killed within 10 generations on average (see column “ \overline{GEN} ” of table 7.2). Furthermore, all the malignant mutants were killed in 7 of the 8 experiments.

7.4.2 Complex Dynamic Real-time System

The purpose of this experiment was to evaluate how well the genetic algorithm based method generates test cases for a system consisting of more sporadic tasks as well as complex scheduling and concurrency control protocols. In this setup we use the *earliest deadline first (EDF)* dynamic scheduling algorithm together with the *stack resource protocol (SRP)*. The EDF protocol dynamically reassigns priorities of tasks so that the task with the current earliest deadline gets the highest priority. The SRP protocol is a concurrency control protocol that limits chains of priority inversion and prevents deadlocks under dynamic priority scheduling. This is done by not allowing tasks to start their execution until they can complete without becoming blocked (Baker 1991).

This system consists of 12 hard real-time tasks, seven of which are sporadic and only five periodic. The system has three shared resources but no precedence

⁴Given this discrepancy, the relevance of the comparison can be discussed. However, the systems are very similar so the analysis problem should be at the same level of complexity.

ID	c	d	SEM	PREC	MIAT	OFS
A	3	20	{(S1,0,2), (S2,0,2)}	{}	≥ 28	10
B	4	24	{(S1,0,3)}	{}	≥ 30	4
C	5	35	{(S2,2,5)}	{}	≥ 38	6
D	6	57	{(S2,0,6), (S3,2,5)}	{}	≥ 48	0
E	5	51	{}	{}	≥ 52	7
F	6	39	{(S3,3,6)}	{}	≥ 44	0
G	3	52	{}	{}	≥ 52	2
H	3	38	{(S3,0,2)}	{}	40	5
I	3	35	{(S1,1,2)}	{}	48	2
J	4	52	{}	{}	60	2
K	2	70	{(S2,0,2)}	{}	80	10
L	3	59	{}	{}	60	12

Table 7.3: Task set for complex real-time system

constraints. The complete task characteristics are listed in table 7.3, using the same notation as in table 7.1.

For this system we found it to be very time consuming and complicated to manually derive the number of malignant mutants. Moreover, model-checking cannot be used for comparison since the reachable state-space becomes too large⁵. Since we could not find an alternative way to efficiently and reliably analyze mutants, we cannot guarantee that the method killed all the malignant mutants.

To increase confidence in the correctness of the original specification model, every generated test case was also run on the original (un-mutated) TAT specification. In a previous iteration of experiments, this kind of test actually revealed a fault in a specification that was assumed to be correct.

For this experiment, we used a delta size of two time units for the mutations of the execution patterns, and a delta size of six time units on the mutations on automata template constants. We ran the genetic algorithm and searched each mutant for 200 generations or until a failure was encountered. Each experiment was performed in five trials with different initial random seeds to guard against stochastic variance. For each simulation performed during the heuristic search, a random test was also conducted. This provides an indication of the relative efficiency between

⁵The Times tool does not currently support the SRP protocol and a sporadic task set of this size was refused by the model-checker (the analysis was terminated when no more memory could be allocated).

Mutation operator	μ	K_R	K_{GG}	K_{GA}	$\overline{K_{GA}}$	\overline{GEN}
Execution time	24	0	0	12	9.2	62
Unlock time	16	0	0	0	-	-
Inter-arrival time	24	0	0	8	3.8	90
Offset time	22	0	0	0	-	-
Total	86	0	0	20	13.0	-

Table 7.4: Results from complex system experiment

random search and genetic algorithms with our heuristic. Furthermore, we ran a genetic algorithm experiment using only the three generic cross-over functions, described in section 7.4.1, to assess the added performance of our heuristics.

Since every mutant operator generated more mutants for this system model we decided to use a subset of mutation operator types. Table 7.4 uses the same column notation as table 7.2, but the additional columns “ K_R ” and “ K_{GG} ” include the results from random testing and non-customized genetic algorithms respectively.

As table 7.4 indicates, no malignant unlock time or offset time mutants were found for this particular system. The average number of generations required to kill a mutant was higher for this system specification model, which indicates that the search problem is much more difficult than for the more static system presented in section 7.4.1.

The low average number of mutants killed in each experiment suggests that the genetic algorithms may have to run longer on each mutant to achieve reliable results. A possible explanation for this is that the genetic algorithm has trouble finding comparable candidates without the iterative refinement from the heuristic operators. Hence, it would often prematurely discard partially refined candidates.

A possible remedy for these problems is to redo the search multiple times using a fresh initial population. Since searching the mutant models is a fully automated process, the additional cost of multiple searches may be acceptable. If no new mutants are killed after a specified number of searches, the process can be halted.

The comparison with random testing and generic genetic algorithms shows that the heuristic cross-over functions are vital for the performance of the method.

7.4.3 Mixed Load Real-time Control System

The purpose of this experiment was to investigate if a mutation-based testing technique can generate test cases to reveal timeliness in flexible control systems where there is a mixed load of soft, adaptive controllers and reactive real-time tasks with

hard deadlines. To evaluate the control performance testing, described in section 7.3, test cases which target execution orders that result in unacceptable control performance are also generated. Hence, this experiment evaluates whether the mutation operators can create malignant mutants in this kind of system as well as how effective our genetic algorithm based approach is in killing such mutants.

For this experiment, we simulated a real-time system with fixed priorities and shared resources under the immediate priority ceiling protocol (Sha et al. 1990). The task set consists of three soft periodic tasks that implement flexible controllers for balancing three inverted pendulums. The linearized equations of the pendulums are given as

$$\ddot{\theta} = \omega_0^2 \theta + \omega_0^2 u \quad (3)$$

where θ denotes the pendulum angle and u is the control signal. ω_0 is the natural frequency of the pendulum. The controllers were designed using LQ-theory with the objective of minimizing the cost function⁶

$$J = \int \left(\theta^2(t) + 0.002u^2(t) \right) dt. \quad (4)$$

Further, the system has four sporadic real-time tasks with hard deadlines, assumed to implement logic for responding to frequent but irregular events, for example, external interrupts or network messages. The system also includes two resources that must be shared with mutual exclusion between tasks.

Table 7.5 lists the exact properties of the simulated task set using the same notation as in the previous experiments.

Three continuous-time blocks, modelling the inverted pendulums, was included in the simulation and connected in a feedback loop to the TrueTime block that simulated the flexible control system. Each of the modelled pendulums have a slightly different natural frequency, ω_0 , and the goal of the control application was to balance the pendulums to an upright position. The pendulums had an initial angle of 0.1 radians from the upright position when the simulation started. An application-specific control failure was assumed to occur when the angle of a pendulum became greater than or equal to $\pi/8$ (~ 0.39) radians.

A set of mutants was generated by applying the mutation operators described in section 6.3 on the extended task set in Table 7.5 using a Δ of two time units for the first three mutation operator types and four time units for the last two. The

⁶The controller design for the inverted pendulums was provided by Dan Henriksson (Nilsson & Henriksson 2005).

ID	c	d	SEM	PREC	MIAT	OFS
A	3	7	{(R2, 0, 2)}	{}	≥ 30	10
B	5	15	{(R1, 0, 3), (R2, 2, 5)}	{}	≥ 40	20
C	4	20	{(R1, 0, 3)}	{}	≥ 40	0
D	5	26	{(R2, 2, 5)}	{}	≥ 50	28
E	5	20	{(R1, 4, 5)}	{}	20	1
F	5	29	{(R2, 0, 4)}	{}	29	1
G	5	35	{(R1, 0, 3)}	{}	35	1

Table 7.5: Case study task set

total number of mutants generated for each operator type is listed in column ' μ ' of table 7.6.

For the genetic algorithm setup, we used a population size of 25 activation pattern matrices. The heuristic cross-over functions as presented in section 7.2.3 were used for stochastically changing activation patterns. The fitness function defined in section 7.3 was used when analyzing the simulation traces.

First, the unmodified system was simulated for 200 generations to gain confidence in the assumed correct specification. This was repeated five times with different random seeds to protect against stochastic variance. No failures were detected in the original model.

Second, each mutant was searched for 100 generations or until a timeliness or control failure was detected. When a mutant was killed, the same activation pattern was used as input for a simulation of the assumed correct system model. The motivation for this extra step was to further increase confidence in the safety of the original model.

The experiment was repeated in five trials to assess the reliability of the approach. Table 7.6 summarizes the results for each mutation operator and failure type using the same notation as in table 7.2.

As indicated in table 7.6, our mutation-based approach using the Flextime tool automatically generates test cases for revealing both timeliness and control failures. In addition, the malignant mutants that cause timeliness failures were killed in all of the experiments. This result increases confidence in the genetic algorithm being effective in revealing critical execution orders with respect to timeliness in models of mixed load control systems of this size. The relatively low average of killed mutants causing control failures indicates that finding a critical scenario with respect to control is difficult. A possible explanation is that the optimization problem contains

Failure type \Rightarrow		Timeliness			Control		
Mutation operator	μ	K_{GA}	$\overline{K_{GA}}$	\overline{GEN}	K_{GA}	$\overline{K_{GA}}$	\overline{GEN}
Execution time	14	1	1.0	2	6	5.2	29
Lock time	13	2	2.0	3	0	0	-
Unlock time	15	1	1.0	2	0	0	-
Inter-arrival time	14	0	0	-	5	2.1	16
Pattern offset	13	0	0	-	0	0	-
Total	69	4	4.0	-	11	7.3	-

Table 7.6: Mutants killed in case study

local optima with respect to control performance fitness. Another possible explanation is that the controlled process only is sensitive to controller disturbance in combination with some other disturbance that was not controlled in the simulation.

In this experiment we actually observed a number of malignant mutants with control failures. This result support the claim that some mutation operators for testing of timeliness are indeed useful for testing control performance.

This experiment demonstrated that the mutation-based testing framework can be customized for non-trivial system models as well as for testing properties that are similar to timeliness.

EPISODE III
Empiricism strikes back

T^3 : A Tool for Mutation-based Testing of Timeliness

This chapter describes different parts of a prototype tool that support the framework for mutation-based testing and implements the heuristic driven simulation approach for test case generation described in chapter 7.

8.1 Overview

The Timeliness Testing Tool (T^3) consists of the real-time simulator extensions provided by the Flextime add-on, as well as implementations of the defined mutation operators, parametrization of the genetic algorithm, and the scripts that automate the test case generation process.

Figure 8.1 provides an overview of how the different parts of this prototype tool are used to perform automated test case generation. As the figure indicates, a real-time system model and a testing criterion must be supplied as input to the tool (arrow 1a and 1b in figure 8.1). In the current prototype, the real-time application model must be manually translated to a native tabular format. However, this information could be retrieved from an existing TAT modelling tool (such as Times (Amnell et al. 2002)) or a graphical user interface. The execution environment model can be modified by configuring the simulator using the Flextime add-on.

The mutation operators corresponding to the given testing criterion create a set of mutant models. These operators have been implemented to operate on the tabular representation of real-time application models. For each generated mutant in the set, the corresponding model is given as input to the Flextime simulator add-on

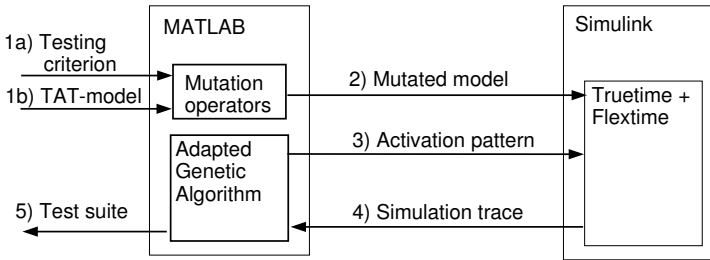


Figure 8.1: Test case generation tool

running on Simulink and TrueTime (arrow 2 in figure 8.1).

A genetic algorithm search is then initiated for each mutant. First, a population of activation patterns is randomly generated by the adapted genetic algorithm and supplied one by one to the simulator (arrow 3 in figure 8.1). For each simulation, a trace is returned to the genetic algorithm (arrow 4 in figure 8.1) so that fitness can be calculated and the activation patterns refined according to heuristics. The new activation patterns are fed back to the simulator (arrow 3 in figure 8.1) and the process is repeated until the mutant is killed or the search is terminated. After each mutant has been searched for violated time constraints, a test suite is generated containing one test case for each of the killed mutants (arrow 4 in figure 8.1).

8.2 Flexitime Simulator Extensions

Flexitime is an add-on to the real-time control systems simulator TrueTime (Henriksson et al. 2003). The purpose of the Flexitime add-on is primarily to support automated analysis and mutation-based test case generation. For this purpose TrueTime must be adapted to (i) do efficient simulation of *TAT* system models, (ii) support structured parametrization of simulations, and (iii) simplify extensions that are consistent with *TAT* specifications.

When Flexitime is used for mutation-based test case generation a mutant *TAT* model should be mapped to simulation entities. The following subsections describe the TrueTime tool and how *TAT* task sets and activation patterns are mapped to simulations by the Flexitime extension.

8.2.1 TrueTime

TrueTime is an open source real-time kernel simulator based on MATLAB/Simulink (Henriksson et al. 2003). The main feature of the simulator is that it offers the possibility of co-simulation of task execution in a real-time kernel and continuous dynamics modelling. The simulator is mainly used for the integrated design of controllers and schedulers, and can be used to analyze the effects of timeliness errors on the control performance of such systems.

The TrueTime kernel support the simulation of periodic and aperiodic tasks, and the attributes of the tasks (such as periods and deadlines) may be changed dynamically during simulation. The scheduling algorithm used by the kernel is configurable and can be defined by the user. Synchronization between tasks is supported by events and the protection of shared resources is simulated as mutual exclusive monitors¹.

Each task in TrueTime is implemented in a separate code function that defines the execution behavior. The code function includes everything from interaction with resources, I/O ports and networks to the specification of execution time of different segments. The TrueTime code functions may be written as either C++ functions or as MATLAB scripts (called m-files).

8.2.2 Task Sets and Execution Patterns

For automated analysis and mutation-based test case generation, we find it useful to distinguish between application functionality and execution behavior. Therefore, in the Flextime extension, execution times, resource requirements, and precedence constraints are specified separately from code functions. This design makes it possible to specify execution patterns of large task sets without having to generate a specific code function for each type. The role of code functions in Flextime is specialized to perform control related calculations and to interact with external Simulink blocks.

Figure 8.2 shows a subset of the class diagram for Flextime. The class *ftTask* is an abstract class that maps down to the TrueTime tasks. This means that when objects of any of the sub-classes to this class are created, a TrueTime task is also created and initialized. The abstract *ftTask* class contains basic information about tasks, such as periods, deadlines and offsets. Moreover, the *ftTask* class extends TrueTime tasks with a list of execution items that define the execution pattern for each instance of this task.

¹These primitives can be used to implement other, more complex, synchronization mechanisms.

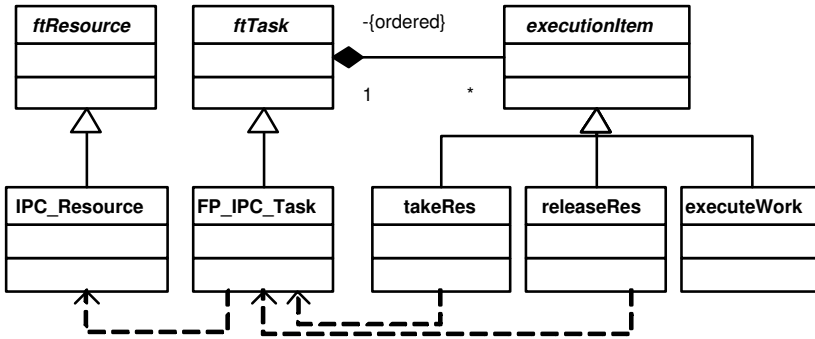


Figure 8.2: Flextime classes

The sub-classes of *ftTask* and *ftResource* are primarily used for supporting different concurrency control protocols, but other types of execution environment extensions are also supported. For example, one pair of subclasses can be used to simulate tasks and resources under the immediate priority ceiling protocol (Sha et al. 1990), whereas another pair may be used for simulation of tasks under EDF scheduling and the stack resource protocol (Baker 1991). The reason sub-classes are needed for both types of entities is that such protocols often require specific data to be kept with task and resource representatives.

When an *ftTask* begins its execution, a virtual *do_seg* method is called sequentially on each item in the execution item list. Execution items of type *takeRes* and *releaseRes* specify that a particular resource is to be locked or unlocked. The *do_seg* function in these execution items simply invokes a corresponding virtual *take* and *release* function in the *ftTask* class with the resource identifier as a parameter. In this way, the logic associated with acquiring and releasing resources can be implemented in the protocol-specific sub-classes of *ftTask*, and execution item classes remain protocol independent. Execution items of the type *executeWork* are generic and specify that execution of code should be simulated for some duration, and optionally, that a segment of a Flextime code function should be executed. All the aforementioned objects can be created using the factory method pattern to support extendability and configuration (Gamma, Helm, Johnson & Vlissides 1994).

In the Flextime tool, task sets can be initialized in two different ways. One is to create the Flextime tasks and populate execution item lists statically in the TrueTime initialization code function. Figure 8.3 includes an example² of the C++

²For simplicity, the task and execution pattern instances are explicitly constructed without the factory method pattern in this example.

```

void init(){
  // String ID
  static IPC_Resource R1("State_Sem");

  // TYPE, StringID, Priority , MIAT, OFS, Deadline
  static FP_IPC_Task A(SPOR,"Safety_check", 1 , 0.040,0.0,0.020);
  A << 0.001 <<+R1<< 0.005 << -R1 << 0.002 << FINISHED;

  static FP_IPC_Task B(PER,"Pendulum", 2 , 0.040, 0.020, 0.040);
  B << 0.002 << +R1 << 0.006 << -R1 << 0.001 << FINISHED;
}

```

Figure 8.3: C++ Syntax for initializing task set

```

// Specify task and resource sub-types
RFactory<IPC_Resource> RCreator;
TFactory<FP_IC_Task> TCreator;

void init(){

  // Import task set from MATLAB variables TS and XP
  TasksetImporter TSI("TS","XP");
  TSI.importRes(RCreator);
  TSI.importTasks(TCreator);
  TSI.importPrecs();
}

```

Figure 8.4: C++ Syntax for importing task set

syntax required for initializing a task set. The second way, which is assumed if no specific C++ initialization file is given, is that the task set specification for a simulation is provided through matrices in the global MATLAB workspace. See figure 8.4 for an example setup in which a task set is imported.

The matrix representation of task sets has the advantage that different mutation operators directly can be applied to create new mutants. However, if new task types, concurrency control mechanisms or scheduling protocols are used, then the C++ initialization file must be customized accordingly.

New tasks types may have additional constraints on activation patterns corresponding to timed automata models. For such task types, a C++ method that

performs the mapping from the genome representation to a valid activation pattern must be implemented, this is further discussed in section 10.2.2.

8.2.3 Activation Patterns

As previously described in section 7.2.1, the activation patterns from periodic tasks are deterministic and can be initialized as usual with the TrueTime kernel block. The activation pattern for sporadic tasks should be varied for each simulation to find execution orders that can lead to timeliness failures. Consequently, a new activation pattern for the sporadic tasks must be imported for each simulation run. The FlexTime tool is capable of importing activation patterns in the genome format described in section 7.2.1 or as a matrix with absolute activation times for each task instance. In particular, activation pattern matrices of this type are also imported from the global MATLAB workspace at the start of each simulation run.

Traces from each simulation run are stored and exported to MATLAB after the simulation run has ended. For example, task activation times, start times³ as well as response times are stored from each simulation. In the global MATLAB workspace, these simulation logs can be visualized, analyzed, filtered and converted to timeliness test cases (see section 5.2).

8.3 Test Case Generation Support

This section presents some different ways that the prototype tool can be used to support modelling and test case generation.

8.3.1 Modes for Generating Test Cases

The prototype tool currently support two different modes for generating and analyzing mutant models, fixed delta mode and iterative delta reduction mode.

In the fixed delta mode, a mutation operator is invoked with a specified delta value and mutants are created by changing some aspect of the original specification with a constant value, delta. The heuristic search is run on each of the mutated models and if a deadline violation is detected, a test case is generated and the search is terminated.

In the iterative delta reduction mode, a maximum delta is specified together with the mutation operator. This mode is a feature that supports generation of

³These are the points in time when tasks actually start executing in response to activations

test suites with different delta parameters rather than an extension to the theory of mutation-based testing of timeliness. Recalling the framework overview figure 5.1 this supports an automated feedback loop from activity 3 to activity 1.

In this mode, the tool starts by applying the mutation operator using the maximum delta on the original real-time system model to create a set of mutants. This step is similar to the fixed delta generation mode. If a deadline violation is found in a mutant, then delta is decreased with one time unit and all the individuals in the test population are re-evaluated using the new model. A new heuristic search is started on the new mutant using the end population of the previous search as an initial population. This process is iterated until no deadline violation is found or delta has been decreased to zero.

If the maximum delta is set high it is likely that the maximum number of test cases is produced for each mutation operator. As the delta is decreased it becomes harder to kill the malignant mutants, since the difference between the mutants and the feasible model becomes smaller. Using the iterative delta reduction mode, the user does not need to manually try different delta sizes before deciding which one would give the best trade-off between test quality and test suite size. Instead, when the test case generation terminates the user may choose between generated test suites for different sizes of the parameter delta. It is also possible to create a test suite containing only the test cases that can kill the smallest mutation of a particular entity. For example, one of the test cases might kill all Δ - interarrival time mutations of task A where $\Delta > 2$ milliseconds, whereas another test case in the same test-suite may kill all Δ - interarrival time mutations of task B where $\Delta > 5$ milliseconds. Figure 8.5 shows the pseudo code for the iterative delta reduction mode where only one test case for each mutated entity is included in a test suite.

The effectiveness of this mode relies on the hypothesis that there is a subsumption relation between mutants of the same type but with different delta sizes. This means that a test case generated using a smaller delta should be effective for killing all mutants of the same entity with larger deltas. Since such subsumption relation does not hold for all system designs (for example, see section 6.6), the tool supports an extra validation step where the final test case is simulated on all previously killed mutants with larger delta sizes. If a mutant that is generated using a large delta cannot be killed using that test case, then the anomaly is logged and both test cases are stored.

```
Iterative_delta_reduction(TAT,operator,delta_max)

MSET=generate_mutant_set(TAT,operator,delta_max)

for each mutant M in MSET
begin

    delta=delta_max;
    trace=heuristic_search(M,Random_test());

    while (killed(trace)==true) and (delta>0)
    begin
        killed_once=true;
        critical_trace=trace;
        delta=delta-1;
        T = make_test(M,critical_trace);
        M = remake_mutant(M,delta, operator);
        trace = heuristic_search(M,T);
    end

    if (killed_once==true)
    begin
        T=make_test(M,critical_trace);
        TSET.Add(T);
        report(M, 'killed', delta);
    else
        report(M,'not killed',delta_max);
    end
end
return TSET;
end
```

Figure 8.5: Pseudo-code for iterative delta reduction mode

8.3.2 Support for Task Instance Modelling

The Flextime add-on and the mutant generator programs that implement the mutation operators also support an extended form of task modelling that enables a unique task execution pattern to be associated with each task instance.

This addition does not require an extension of the TAT notation since it can be modelled as a TAT automata template that releases one different task for each stimuli in an activation pattern. If task mutation operators are applied on systems modelled in this way, each task instance may be mutated independently. However, to maintain the semantics of the mutation operators and to keep the number of mutants bounded by the maximum in table 8.2, all execution patterns of a particular task must be mutated in the same way.

The advantage with this extension is that deterministic variations in task execution patterns can be modelled in a simple way. For example, if we know that every third release of a particular task always executes longer or locks a different set of resources, the behavior of that task can be modelled as a chain of task execution patterns instead of one static task execution pattern. An example illustrating the usefulness of this modelling style is presented in section 9.4.1.

To keep the simulation deterministic with respect to activation pattern, each execution pattern for a particular task must be statically ordered so that the n 'th activation of task A always has the same execution pattern. A possible future extension would be to associate each task type with a “bag” of possible execution patterns (behaviors) and let the heuristic search algorithm dynamically decide which should be used for a particular task activation. However, this kind of extension would further increase the domain of the search problem (see section 10.2.1).

Case Study: Testing using the Mutation-based Framework

This chapter demonstrates how the framework described in chapter 5 can be applied for testing of timeliness and presents some experiments that evaluate the generated test cases.

9.1 Overview

The purpose of this case study is to assess the applicability of the approach for mutation-based testing presented in chapter 5 (and the associated tool, presented in chapter 8) for the automated testing of timeliness.

This case-study can be classified as a *controlled case-study*, which means that an idea is used in a limited situation with realistic assumptions. This does not necessarily generalize to a broad range of cases, but can give indications that the idea is useful. The idea in this context is using the mutation-based framework to test timeliness. In particular, the proposed framework is used to test a small control application running on an open source real-time operating system. The configuration of the target system and the real-time system model is changed over a set of experiments to identify under which conditions the framework is useful.

The effectiveness of test cases generated using the mutation-based testing framework is compared with timeliness test cases generated using random testing. If the applied framework is more effective than random testing for finding timeliness faults, confidence is increased in the feasibility of the approach (also see the thesis statement in section 4.4.1). Further, application on a realistic system platform may

reveal if the method does contain any unforeseen or inherent limitations.

9.2 A Flexcon Demonstrator Prototype

This section describes the design and implementation of a prototype of the Flexcon demonstrator used to evaluate the testing framework presented in chapter 5.

Unfortunately, we have not found any publicly available benchmark implementations for this kind of testing research. Libraries and support systems for implementing different kinds of real-time applications exists; but the applications themselves often require specific hardware platforms and are therefore difficult to access for software experimentation purposes. In addition, time constraints are seldom available, since they are typically closely connected with the target platform and the controlled process. Hence, a prototype control system inspired by the Flexcon robot demonstrator was specially built for this research. This section presents the application and describes the design of the system and the test harness.

The application used as the Flexcon demonstrator is a control system for a robot arm, hereafter referred to as the slave robot. The slave robot has the ability to move in a fixed operational plane, using three servo motors for adjusting angles of joints (figure 9.1). The purpose of the slave robot in this application is to balance a ball on a beam while the other end of the beam is moved by an independent master (robot). A camera is used to capture images of the beam angle as well as the ball's position on the beam. A force sensor on the arm of the slave robot is used to help determine the movement of the other robot together with the camera images. This case study focuses on a sub-part of the Flexcon demonstrator that controls the movement of the slave robot. That is, the processing of camera images and movement of the master robot arm are assumed to be handled by other nodes in the system and commands from this control are sent to the slave robot in the form of trajectories.

9.2.1 Real-time Design

The system prototype consists of 7 tasks; of which 4 are periodic and 3 sporadic. The terms used for these tasks as well as an overview of their relationships are depicted in figure 9.2.

The *TrajectoryMgr* is a sporadic task that continuously maintains and updates the current trajectory based on orders and adjustments received from a remote node. This task updates a cyclic buffer containing the current trajectory and desired path for the end-point of the beam, hereafter referred to as *the hand*. New updates from the network may require recalculation or extension of the planned trajectory. This

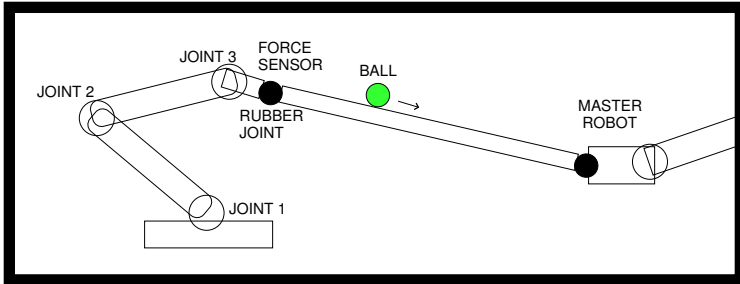


Figure 9.1: Robot demonstrator overview

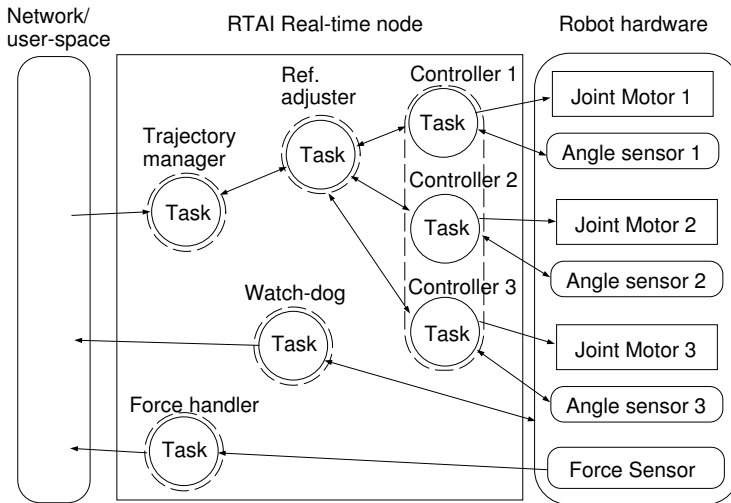


Figure 9.2: Tasks and their relation to each other

task also translates trajectory coordinates in the form $\langle x, y, angle \rangle$ to the corresponding set points for the three joints. Here x and y are the position of the hand relative to the base point of the robot arm; $angle$ is the desired angle of the hand, relative to the ground.

The *RefAdjuster* task is coordinating the controllers by adjusting the reference signal (set-points) to follow the precalculated trajectory. This sporadic task is activated when the robot arm is within the bounds of certain intermediate via-points along the trajectory.

In this prototype, the robot arm is assumed to be equipped with a sub-system that sends interrupts when the force applied to the robot hand is changed. The interrupts from the force sensor are serviced by the sporadic *ForceHandler* task that filters the information and sends notifications to a remote node if changes are above a certain threshold.

Three tasks (*Controller 1, 2 and 3*) are periodic controllers with a relatively high sampling rate for accurately controlling the joints of the robot. These controllers get feedback through angle sensors that are read at the beginning of each task execution. The implementation of controllers are of PID-type, configured according to the dynamics of the controlled system. The three controller tasks are implemented using a single thread similar to a cyclic executive (c.f., Burns & Wellings (2001)). This means that the periods of the controllers harmonize and that the relative activation order of these periodic tasks is known. This implementation style is advocated by Klas Nilsson, the domain expert within the Flexcon project (Nilsson 2005). For test case generation purposes this thread can be modelled as a shared mutual exclusive resource between the controller tasks.

A high priority *Watch-dog* task is used to ensure that the hardware is always serviced within a certain interval of time. The reason for this is that the robot hardware requires the hardware interface to be polled with a certain minimum rate (Nilsson 2005). To monitor the polling, a shared time stamp structure is updated every time any task polls the hardware interface. If the watchdog detects that the hardware has not been polled within the interval then actions are taken to service the interface. Furthermore, the watchdog task is responsible for checking the integrity of the control system and regularly sends messages to remote nodes indicating that the controllers are up and running.

This design of the robot control prototype has four resources that require mutual exclusive access, which are listed in table 9.1. Table 9.2 summarizes the application task set and lists the various shared resources used by the different tasks. For an application such as a robot arm controller, there is a trade-off between control performance and the periods (sampling frequency), hence, it is the designer who

Semaphore	Protected resource
Q	HW interface + Timestamps
R	Current setpoint
S	Trajectory buffer + indices
T	Network interface

Table 9.1: Shared system resources

Task	Resources			
TrajectoryMgr			S	T
RefAdjuster		R	S	
ForceHandler	Q			T
Controller 1	Q	R		
Controller 2	Q	R		
Controller 3	Q	R		
Watch-dog	Q			T

Table 9.2: Task set of prototype with shared resources

sets suitable periods. In other applications involving the Flexcon robots, the sampling frequency of the joint controllers varies between 200 Hz (which is the lower limit from the closed-loop dynamics of the system) up to 4 KHz to improve the disturbance rejection of the system (Henriksson 2006). This prototype is designed for a sampling frequency at 2.5 KHz, to get disturbance rejection while not overloading the system.

For sporadic tasks, the offsets, periods and timing constraints are typically decided by properties of the environment, network protocols, or set by system designers to meet end-to-end deadlines in distributed real-time applications. For the testing experiments in this chapter, the attributes of sporadic tasks are chosen so that the system remains timely in worst-case situations without wasting resources. Hence, these attributes can be altered during experimentation. The reason for this experiment design is that it is easier to change the deadlines and periods rather than the system implementation or properties of the platform.

9.2.2 Target Platform

The target system for this application is a x86 based pc (a 450 MHz Pentium III) running Linux with RTAI (Real-Time Applications Interface). The operating system Linux/RTAI (Bianchi, Dozio, Ghiringhelli & Mantegazza 1999) has several

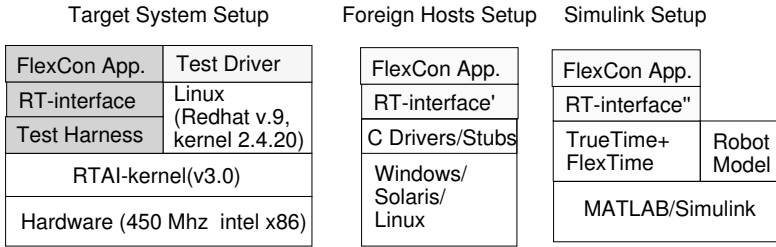


Figure 9.3: Overview of configurations

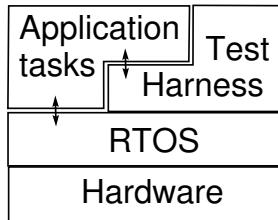


Figure 9.4: Role of the test harness

advantages as a platform for this research. First and foremost, it is possible to run hard real-time applications using an ordinary desktop computer. This avoids issues with initialization and cross-compiling that often arise when developing embedded software. Linux/RTAI has been used for implementing real-time applications and is designed so that the Linux kernel and all the user-level processes can be fully preempted when a real-time task is activated. This also means that the Linux operating system runs in the background reclaiming “wasted” cpu resources for its desktop applications. Theoretically, the only consequence this should have on the tested real-time system is that shared hardware caches may be flushed when the real-time system is assumed to be idle. However, this could also be the case if a RTAI-based system would be used for hard real-time control while user-space threads are used for independent, soft real-time (or non real-time) services.

An overview of the system is depicted in figure 9.3. As can be seen in this figure, the real-time application (shaded gray) has a layered structure where the code of the real-time tasks are separated from the test harness and real-time operating system using a layer called rt-interface. The conceptual relations between the test harness, the application tasks and the real-time operating system is further depicted in figure 9.4. In this figure, the rt-interface is denoted by double-headed arrows.

The motivation for the layered structure is to hide the test-instrumentation from the code of the application tasks and to make the robot application portable to other real-time platforms. Obviously the timing and order of tasks may be completely different on non-real time hosts, but applications can be compiled and tested for logical and syntactical correctness on any supported platform. The MATLAB/simulink target also makes it possible to test how the control application would interact with a simulated hardware robot, given that the assumptions about temporal behavior are correct. In this context, the test driver in figure 9.3 runs in linux user-space and reads data files containing timeliness test cases into memory, so that it is available as needed when a test run starts.

9.2.3 Test Harness

The test harness incorporates all the software needed for controlling and observing a test-run on the target system. The design of the test harness is critical for real-time systems since it often must be left in the operational system to avoid probe effects. An important issue in this context is that the instrumentation code must not change the execution behavior in other ways than increasing the execution time. For example, the application behavior could be significantly altered if calls to logging functions were blocking.

According to Schütz (1993), the following significant events need to be instrumented to capture an execution order of event-triggered real-time systems.

- Access to time
- Context switches
- Asynchronous interrupts

Apart from this, when testing timeliness it is necessary to log the start and end time of each task instance so that it is possible to check that it executes within its time constraints.

The instrumentation of interrupts can be added directly to the interrupt handler functions. Access to time can be instrumented by logging the system calls used to access the system clock.

The order of context switches can be indirectly monitored by logging the invocation and completion of task instances. However, to obtain full information of the timing when a context switch occurs it is necessary to be able to insert probes in the system dispatcher. According to Schütz (1993), it is application dependent whether the timing of these events is necessary or not.

Injecting Stimuli

During a test run sporadic stimuli should be injected at the points in time specified by the activation pattern part of timeliness test cases. To realize this, the test harness for the Flexcon application uses a sporadic task that emulates the interrupt-handlers in the system. This task reads an activation pattern specification from a FIFO-queue that has been populated by the test driver before testing starts. The activation pattern specification contains the type of the next sporadic task to be activated and an absolute point in time when it should be activated.

Since the test harness task for injecting stimuli executes on the highest priority and does not share any resources with the application tasks, the execution disturbance from this task is assumed to be short and similar to that of the emulated interrupt handlers. Further, the same mechanism is used to evaluate all the test suites compared and therefore should not introduce any bias.

The activation of periodic tasks is handled by using the facilities provided by the RTAI native interface. Hence, the periods and start times of tasks are specified at system initialization and activation (stimuli) are generated by the operating system.

Monitoring and Logging

In the current prototype implementation, all tasks log their own events in separate main-memory buffers. Thus, no synchronization between real-time tasks is necessary for this purpose. The events in the log are time-stamped so they can be ordered and analyzed after the completion of a test run. The main-memory buffers are also accessible from user space (as devices in the Linux file system), hence it is possible to perform on-line analysis of test logs concurrently. In the following experiments, information from these main-memory buffers is collected after each test run, to avoid unnecessary competition for resources.

Only partial information about context switches is acquired in this application (through logging of task activation and completion times); the reason for this is to make sure that the test execution harness does not depend on features of the Linux/RTAI infrastructure. However, the information acquired is sufficient for determining the execution order of the application tasks and checking that tasks execute within their time constraints. If necessary, context switch overhead times can be approximated by the time a higher priority task (than the one currently running) is requested and the logged time when the task begins its execution.

```
static void Spor_Thread2(int dummy) {  
  
    const int MyID = ID_FSSENSORH;  
    RT_TASK* ME = rt_whoami();  
  
    SPT2_data A;  
    SPT2_init(&A,MyID);  
  
    rt_task_suspend(ME); // Init done waiting for test scope to start  
  
    while (SYS_STATE==Operative) {  
        log(MyID,TS_Start);  
        SPT2_work(&A,MyID);  
        log(MyID,TS_End);  
        rt_task_suspend(ME);  
        // Sporadic tasks remain suspended until invoked by  
        // a new interrupt  
    }  
}
```

Figure 9.5: RTAI-thread implementing sporadic task in test harness

```
static void Per_ThreadALLContr(int dummy) {  
  
    PT0_data A;    // Local Data (think C++ object state data)  
    PT1_data B;  
    PT2_data C;  
  
    PT0_init(&A, ID_CONTR1);  
    PT1_init(&B, ID_CONTR2);  
    PT2_init(&C, ID_CONTR3);  
  
    rt_task_suspend(rt_whoami());    // Init done  
  
    while (rt_get_time() < end_time) {  
  
        PT0_work(&A, ID_CONTR1); // Invoke controller 1  
        PT2_work(&C, ID_CONTR3); // Invoke controller 3  
        rt_task_wait_period(); // wait until (P*n)+(P/4)  
  
        PT1_work(&B, ID_CONTR2); // Invoke controller 2  
        rt_task_wait_period(); // wait until (P*n)+(P/2)  
  
        PT0_work(&A, ID_CONTR1); // Invoke controller  
        rt_task_wait_period(); // wait until (P*n)+(P*3/4)  
  
        PT1_work(&B, ID_CONTR2); // Invoke controller 2  
        rt_task_wait_period(); // wait until (P*n+1)  
  
    }  
    rt_task_suspend(rt_whoami());  
}
```

Figure 9.6: RTAI-thread implementing controller tasks in test harness

Task Support

The real-time tasks are implemented as functions that are free from semantics associated with the target platform. The reason for this is to be able to support the layered architecture suggested in section 9.2.2.

Hence, each real time task is assumed to have a data structure containing its local context (state variables), an “Init” function called once at the start-up of the system, and a “Work” function called at each invocation of the task. To support concurrency, each task is typically implemented using a dedicated operating system thread that is responsible for logging and calling the task-functions at the right time. See figure 9.5 for an example of how a sporadic task is implemented. A similar thread is needed for each sporadic task and it is possible to parameterize the setup so that this indirection becomes transparent to the user of the test harness¹.

It is also possible that tasks are implemented by more advanced threads, for example, periodic tasks that share a common major cycle time can be implemented using a single thread (see figure 9.6).

9.2.4 Task Implementation

The implementation of the robot application tasks is quite small, only approximately 1000 lines of C code (excluding code in library function calls etc).

As previously mentioned, the implementation of this prototype contains some simplifications, but is assumed to be detailed enough to be representative of small control applications. One simplification is that some parts are omitted, for example no hardware device drivers or networks stacks are implemented. The parts not implemented are replaced with stubs (function calls) that make dummy operations using a busy delay primitive. A description of the implementation of the application tasks including source code extracts is available in Appendix A.

9.3 Applying the Framework for Testing of Timeliness

When performing system level testing of timeliness (according to the framework outlined in chapter 5), tasks are requested as specified by the activation pattern part of timeliness test cases to exercise a particular execution order. During such a test run the individual tasks are fed with input data that are good candidates for causing long execution times or blocking times. This should stress the system to reveal potential timeliness failures.

¹In this way the test harness can be adapted for other applications and target platforms

This section describes in detail how input data for individual tasks were derived for this case-study and how these input data are combined to form system level test cases (this is activity number 4 in figure 5.1). The system level input data constructed by this method are used in the test execution experiments presented in section 9.4. The models and mutation-based testing criteria required to automatically generate activation patterns (activity number 1 in figure 5.1) are varied between the testing experiments, and hence, these are presented in section 9.4.

9.3.1 Constructing Task Input Data

In this case study, the same task input data are used for all the experiments that compare methods for testing of timeliness. Thus, the thoroughness of this step is important for the success of any of the methods.

To do this systematically, a set of input data that yields a high degree of coverage with respect to temporal behavior and control flow was constructed for each task. The execution time of the task running undisturbed was measured using each task input datum.

In particular, a black-box testing technique based on equivalence partitioning and pair-wise coverage was used to create test suites for each task. The methods discussed in section 5.2 that are developed for deriving this type of input data (for example, the compiler based method developed by Puschner & Schedl (1991)) could unfortunately not be used, since no version of the required tools was available for the target platform.

Instead, equivalence partitioning was used as a basis for input data generation and applied manually in order to exercise a wide range of temporal behaviors of tasks. This means that parameters which change the execution time of tasks were deliberately included.

For example, the behavior of the TrajectoryMgr task directly depends on coordinates received from the network and the state of a circular trajectory buffer. Three parameters were identified that might influence the behavior of this task; these parameters and their associated classes of inputs are listed in table 9.3. One of these parameters is the size of the coordinate buffer when the task is invoked. This parameter is marked “Size of buffer” in table 9.3. Another parameter is the actual type of operation sent to the TrajectoryMgr task (this parameter is denoted “Operation” in table 9.3). The last parameter is called “Coordinate” and refers to the structure of the actual coordinate data sent in the request. The variables N1 and N2 used in table 9.3 are random positive real value between 200 and 1000. The values with dagger symbols denote values outside the assumed operational profile.

Size of buffer	Operation	Coordinate
0	Insert	(N1,N2,1.57)
15	Move All	(-N1,N2,0.0)
99	Adjust (10)	(N1, -N2, 3.15) †
100	Adjust (99)	(0,0,0.0)†

Table 9.3: Equivalence partitioning of TrajectoryMgr

From this equivalence partitioning, each parameter is pair-wise combined with each of the other parameters. Table 9.4 lists an example suite that provides pair-wise coverage of the parameter values in table 9.3.

After input data were created, each task was run on the target system without any disturbances from other application tasks. For each input datum the task was activated periodically 20 times, and the procedure was repeated 5 times. The statistics of the tasks execution behavior for a particular input datum, over these 100 invocations, were collected. Analysis tools were developed to automatically extract worst-case, best-case, mean, median and standard deviation of the execution times from the logs. Based on these measurements it is possible to rank each task input datum with respect to the resulting execution time and execution pattern of the task.

These lists of input data for the different system tasks are then combined to create sets of test data for system level testing of timeliness where all the tasks are run concurrently. The equivalence partitioning, input data specifications and the results from the measurements of the tasks in the Flexcon demonstrator application are available in Appendix B. From the measurements it can be concluded that hardware caching has a significant impact on the execution times for this application and platform.

9.3.2 Combining Task Input Data for System Level Testing

Before starting system level testing on the target system it is necessary to choose which input data for the different tasks should be combined when tasks run concurrently. An intuitive choice is to combine all the input data that cause the longest execution time into one single set of input data to use for all the test runs. However, if only one set of input data is used then it is possible that only a small fraction of the actual code is executed during system level testing. Furthermore, anomalies might exist so that input data which cause a short execution time during measurements (when the task is executing undisturbed) cause a disproportionately longer execution time when a task is executed concurrently with other tasks.

#	Size of buffer	Operation	Coordinate
1	0	Insert	(N,N,1.57)
2	0	Move All	(N, -N, 3.15)
3	0	Adjust(10)	(0,0,0.0)
4	0	Adjust (99)	(-N,N,0.0)
5	15	Insert	(0,0,0.0)
6	15	Move All	(-N,N,0.0)
7	15	Adjust(10)	(N,N,1.57)
8	15	Adjust(99)	(N, -N, 3.15)
9	99	Insert	(N, -N, 3.15)
10	99	Move All	(N,N,1.57)
11	99	Adjust(10)	(-N,N,0.0)
12	99	Adjust(99)	(0,0,0.0)
13	100	Insert	(-N,N,0.0)
14	100	Move All	(0,0,0.0)
15	100	Adjust(10)	(N, -N, 3.15)
16	100	Adjust(99)	(N,N,1.57)

Table 9.4: Pair-wise combinations of parameters

Another extreme is to test each combination of input data derived in the previous step (described in section 9.3.1). For example, if the equivalence partitioning and pair-wise combination resulted in 20 input data for task A and a similar process resulted in 12 input data for task B, then this would give us 240 different sets of input data when tasks A and B are running concurrently. These 240 input data are then combined with each of the activation patterns (from mutation-based test case generation) to form test cases for testing of timeliness. Clearly, this quickly becomes an expensive test method. Again, it is possible to adopt existing testing techniques to reduce the test effort. One way is to use the input data causing the longest execution times as a “base-choice” (Ammann & Offutt 1994). Using this test method, all the tasks except one execute with the input data causing the longest execution time. The inputs to the last task are varied among the other candidates for the longest execution time.

For the case study we selected three candidate input data for each sporadic task and two input data for each periodic controller task and combined them using the input data causing longest execution time as the base-choice. Except the base choice, task input data candidates were chosen based on the diversity of the control flow paths covered, with bias for such input data that cause long execution time.

IDS	TrajectoryMgr	RefAdjuster	ForceHandler	Controllers		
1	10	2	1	1	1	3
2	1	2	1	1	1	3
3	12	2	1	1	1	3
4	10	3	1	1	1	3
5	10	4	1	1	1	3
6	10	2	5	1	1	3
7	10	2	6	1	1	3
8	10	2	1	2	1	3
9	10	2	1	1	4	3
10	10	2	1	1	1	1

Table 9.5: Base choice and combinations of task input data

Table 9.5 lists the different sets of task input data, the bold numbers represent deviations from the base choice. The column denoted “IDS” enumerate the different combinations and the other columns contain the identifiers of the input data used for each task. These identifiers correspond to the task input data constructed for each task (described in section 9.3.1). For example, the input data in row numbers 1, 10 and 12 from table 9.4 are used for the TrajectoryMgr task during system level testing. The reason that only two input data were selected for the periodic tasks is that the control flow is identical for these three tasks (except for constant value differences), so in effect all four classes of input data are tested. The test harness was setup so that each activation pattern was run at least once with each of the input data sets. We assume that this approach provides a high coverage of the candidate paths causing long execution times.

9.4 Demonstrator Testing Experiments

There are different decisions associated with applying the framework for testing timeliness. One such decision is how data from measurements are used for modelling. This section evaluates testing of timeliness of the Flexcon demonstrator prototype in a number of experiments.

Two kinds of experiments are presented here. First, two different modelling styles are evaluated. The purpose of the first experiment is to determine what modelling style best captures the behavior of the target system and which model is the one most suitable for this particular target system. This experiment also demon-

strates the relative effectiveness between test suites generated using different kinds of mutation-based testing and random testing.

The mutation-based testing framework is subsequently used to test a number of variants of the Flexcon demonstrator prototype with seeded errors. The purpose of this experiment is to investigate the relative effectiveness between test cases generated with mutation-based testing and random test cases for finding the type of errors seeded. This experiment also investigates the difference in response times caused by the different kinds of test cases.

In this context, an effective test case is one that causes a time constraint to be violated in at least one test run. One test suite is said to be more effective than another if it contains a significantly larger fraction of effective test cases. As previously mentioned, the same combinations of task input data are used for all the generated test suites, thus, the activation patterns generated using mutation testing and random testing is the only parameters that vary between the compared testing methods.

9.4.1 Experiment 1 - Framework Testing Effectiveness

In the first experiment, two different types of real-time models are evaluated as a base for test case generation and execution.

In the first model, all task instances are conservatively modelled to behave as the task instance with the longest execution time. That is, the longest execution time observed for a particular task is used to construct a task execution pattern that is used for all task instances.

In the second model, the n 'th task instance is modelled to behave as the worst case behavior observed for that particular task instance. One effect of this is that if, for example, the third invocation of a task always executes longer in all measurements, then it is modelled to have a longer execution time.

From both models, test cases were generated using the heuristic driven simulation approach described in chapter 7. Hence, this is similar to the setup in previous test case generation experiments (see section 7.4). The execution environment model encoded in the tool was configured to correspond to the fixed priority scheduling and priority inheritance protocols defined by the RTAI documentation.

The same testing criteria and mutation operators were used for generating test cases for both approaches. The testing criteria was based on the “+/-20 Execution time”, “+/-20 Lock time” and “+/-20 Unlock time” mutation operators. To be able to keep these testing criteria fixed, some time constraints and system attributes had to be changed between experiments. The reason for this difference is that one of the

Task	Worst-case execution pattern								
TrajectoryMgr	10,5	+T	5,5	-T	4,5	+S	399,9	-S	3,5
RefAdjuster	14,7	+S	6,7	-S	4,8	+R	1,9	-R	4
ForceHandler	7,0	+Q	5,4	+T	4,1	-T	2,7	-Q	3,1
Controller 1	11,5	+Q	8	+R	12,3	-R	6,7	-Q	3,5
Controller 2	12,2	+Q	12,4	+R	13,1	-R	14,6	-Q	11,9
Controller 3	9,6	+Q	7,7	+R	10,2	-R	7,7	-Q	3,4
Watch Dog	7,5	+Q	4,5	-Q	3,9	+T	4,1	-T	3,1

Table 9.6: Worst-case execution measurements (in micro-seconds)

modelling styles is more pessimistic with respect to resource consumption than the other. Hence, if exactly the same system is modelled with both approaches, then the more pessimistic approach results in a model that is unfeasible (that is, deadlines can be missed in the unmutated model) whereas the other model contains so much slack that no generated mutant would be malignant. This difference makes it impossible to compare the results from the different approaches directly. However, since random testing is used as a base-line in both experiments, it is possible to assess the relative effectiveness of the different models for this particular platform.

Setup 1

The motivation for the first modelling approach is that if the system is designed to be timely it must cope with the longest execution times of tasks. Even if a “worst-case” execution time of a particular task seldom occurs, it may eventually occur together with other worst-cases.

Table 9.6 lists the execution pattern used for the different tasks. In the execution pattern table, +X denotes that the task locks resource X and -X means that the task unlocks resource X.

The task set attributes for this system are listed in table 9.7. The deadlines for the sporadic tasks are set so that the model is feasible. Hence, for this unmutated system model a genetic algorithm search did not find any deadline violations (running 10 times for 200 generations).

Using the tool described in chapter 8, 70 mutants were automatically created and analyzed. The system model defined by tables 9.7 and 9.6 as well as the mutation operator types listed in table 9.8 were used as input (The columns headings in table 9.8 have the same meaning as in section 7.4). Only 6 mutants were killed for this system model. None of the mutation operators that changed the locking

Task	Period/MIAT	Offset	Deadline
TrajectoryMgr	1000	600	800
RefAdjuster	700	1500	650
ForceHandler	600	300	600
Controller 1	400	0	700
Controller 2	400	200	700
Controller 3	800	0	1500
Watchdog	600	0	500

Table 9.7: Task set attributes, setup 1

Mutation operator	Δ	μ	K_{GA}	$\overline{K_{GA}}$
Execution time	20	14	6	6.0
Lock time	20	28	0	0
Unlock time	20	28	0	0
Total	-	70	6	-

Table 9.8: Results from mutation-based test generation

behavior actually caused any timeliness failures. Five test-suites were generated to be able to repeat our experiment. Consequently, five test suites with 6 generated activation patterns each were run on the target system to test its timeliness. Each activation pattern was 10 times (once for each set of task input data).

For comparison, five test suites with 10 random activation patterns each were also created and run with all sets of input data. All the random activation patterns for this experiment were generated with uniform distribution so that the specified minimum interarrival time was maintained while the average system load was at the same level as when the mutation-based tests were run.

The result of the test execution is presented in table 9.9. In this table, the column marked ‘#’ contains the number of random test cases in each test-suite. The columns marked ‘F’ contain the number of test cases in each test suite that revealed timeliness failures. Since no “+/-20 Lock time” or “+/-20 Unlock time” mutants were killed, no tests of this kind could be run on the target system.

From table 9.9, it can be seen that two of the 50 random activation patterns revealed a timeliness violation in at least one test run. Furthermore, 3 of the 30 activation patterns generated using mutation testing revealed timeliness failures. The difference in effectiveness is negligible and consequently much smaller than expected. An comment on this result is presented in the analysis part of this section.

Trial	Random		Exec.	
	#	F	K_{GA}	F
1	10	0	6	1
2	10	1	6	0
3	10	0	6	1
4	10	1	6	0
5	10	0	6	1
Total	50	2	30	3

Table 9.9: Results from initial comparison with random testing

Task	Period/MIAT	Offset	Deadline
TrajectoryMgr	600	600	800
RefAdjuster	700	1500	650
ForceHandler	600	300	600
Controller 1	400	0	400
Controller 2	400	200	400
Controller 3	800	0	800
Watchdog	600	0	500

Table 9.10: Task set attributes, setup 2

Setup 2

The hypothesis for the model used in this setup is that, for small applications a large part of the task code and data fit in the system cache, and hence, the execution time differs significantly between the first and the remainder of the task instances. Furthermore, state changes in the tested tasks and variations of series of input data read by the tasks may cause deterministic differences in execution behavior from one task invocation to the next. Using the less pessimistic modelling style, the model was feasible using the time constraints specified in table 9.10. Five test suites of each kind were generated for this system configuration. Table 9.11 lists the number of malignant mutants detected during the generation of these test suites, and the number of mutants killed on average for each type.

Five test suites with ten random activation patterns were generated for comparison. These were of the same type as the random tests used in setup 1.

To further check the relative effectiveness between mutation-based testing and other naive testing approaches, we manually generated a test suite with two acti-

Mutation operator	Δ	μ	K_{GA}	$\overline{K_{GA}}$
Execution time	20	14	7	7.0
Lock time	20	28	8	6.0
Unlock time	20	28	6	4.0
Total	-	70	21	17.0

Table 9.11: Results from mutation-based test generation

Trial	Random		Manual		Exec		Lock		Unlock	
	#	F	#	F	K_{GA}	F	K_{GA}	F	K_{GA}	F
1	10	0	4	0	7	4	7	5	4	1
2	10	0	4	0	7	4	7	4	5	1
3	10	0	4	0	7	4	5	4	5	0
4	10	0	4	0	7	5	5	1	4	1
5	10	0	4	0	7	3	6	2	2	0
Total	50	0	20	0	35	20	30	16	20	3

Table 9.12: Results from Test Execution

vation pattern types that we believed would reveal timeliness failures better than random testing. The activation patterns were (i) to activate all sporadic tasks with their minimal interarrival time, and (ii) let the first activation of all sporadic tasks occur at the same instant and then with their minimum interarrival time. Each manual test suite contained two activation patterns of each type.

Each test suite was executed on the target system and each test case was repeated 10 times using a different set of input data. The results are listed in table 9.12 using the same column headings as in table 9.9. As this table illustrates, all test suites generated with mutation testing found timeliness violations in the real system. A surprising result was that neither the manual nor the randomly generated tests cases found any timing faults in any of the trials. Since all the random tests cases were unique activation patterns, it can be concluded that any one of the five test suites complying to “+/-20 execution time” coverage is more effective than 50 random activation patterns for this system setup.

Analysis of experiment 1

When comparing test logs from the first setup and simulation traces from killed mutants it can be seen that most of the mutants are killed late in the simulation

trace whereas almost all deadline violations in the real system are detected when tasks are activated for the first time. It is also evident that the simulated task instances execute much longer than the real software tasks on average. This effect is particularly evident late in the simulation trace.

We believe that this behavior is due to the large size of hardware caches and the relatively complex caching architecture of the target system. After the first invocation of a task all the source code and local data for a task may have been copied to the cache memory (first or second level). Since each real-time application task is activated with a high frequency (relative to other threads running in Linux user-space) the probability that the data used by the the real-time tasks remain in the cache is high.

The effect of this discrepancy between the model and the real system's behavior is that the heuristic search algorithm often creates an activation pattern to stress a critical interval late in the simulation. However, on the target system, the real execution time is much shorter than in the simulation in this interval, and hence, the generated activation pattern is unlikely to reveal any timeliness failures.

This effect is compensated by the modelling style applied in experiment setup 2. By modelling the task instances separately, the simulated model is more similar to the system behavior. Consequently, the heuristic search algorithm has a better chance of identifying the activation patterns that also stress the real system implementation. The improvement in the relative effectiveness of the mutation-based test suites for the second setup indicates that the mutation-based framework is sensitive to large mistakes in execution pattern modelling. This implies that it is useful to exploit feedback from measurements collected during unit testing and not be overly pessimistic when creating models for system level testing purposes.

Another observation is that for both experiment setups there exist intervals in the test logs in which no real-time task is allowed to run even if several tasks have been requested for execution. We assume this is caused by platform overheads and disturbances that are not part of the model.

If overheads of this type occur deterministically for certain execution orders or task executions, then they can be treated as a difference between the model and the implementation and included in the execution environment model for a new iteration of analysis and test case generation. However, transient faults that cannot be controlled or predicted are difficult to model in a meaningful way.

Once transient faults have been identified as causing timeliness failures, there are two ways of proceeding. The preferred way when building a real-time system is trying to identify the causes of the transient timeliness faults (unpredictability) and removing them from the system. If that is not possible (which may be the case when

building real-time systems with off-the-shelf hardware and software components), extra computing resources can be included, or features removed so there is slack in the system to cope with the “noise” introduced by the transient faults.

A problem in this context is estimating the amount of slack needed to avoid timeliness failures in the new system. The only way to gain confidence in a such system may be testing since traditional analysis methods for real-time systems typically do not cope well with unpredictable disturbances (Burns & Wellings 2001). Further, by allowing this types of timeliness faults in the system, the effectiveness of timeliness testing methods is also negatively affected since test cases must be run more times to obtain reliable results. In addition, it is generally impossible to know if the worst case has been observed for a particular test case.

9.4.2 Experiment 2 - Effectiveness with Seeded Errors

The purpose of this experiment is to investigate the effectiveness of mutation-based testing when the target system is designed to cope with some transient timeliness faults. In particular, we investigated the relative effectiveness of the test suites in testing system variants with both seeded errors and transient faults². We compared the results for different levels of slack to discover how much the different test suites stress the non-deterministic system.

For this experiment, we created a number of variants of the Flexcon demonstrator application with seeded errors by adding a busy delay at a random point in the tasks code. The error seeding was controlled so that each task was affected by one error. The basic block in the task code where the error was injected was chosen by random. This means that the error can occur within a condition statement or in a code segment where a shared resource is locked. The size of the busy delay added to the source code was 80 microseconds. Table 9.13 summarizes the result of the errors seeding for different variants of the system.

Since this experiment required running 7 times as many tests, we only did a comparison between random testing and one of the mutation-based testing criteria (approximately 6000 test runs were executed for this experiment). In particular, five “+/-20 execution time” test suites and five random test suites from experiment one were rerun on each of the 7 variants of the target system.

As an example, figure 9.7 shows the number of effective test cases for two test suites executed on variant 7. In this figure, the solid line is the test cases generated with mutation-based testing and the dashed line represents the random test cases.

²These remains in the seeded variants since we could not easily fix the transient faults detected in the first experiment.

Version	Task	Resources
1	TrajectoryMgr	S
2	Refadjuster	-
3	Force handler	Q&T
4	Controller 1	Q
5	Controller 2	Q&R
6	Controller 3	-
7	Watch-dog	T

Table 9.13: Software variants and affected entities of seeded errors

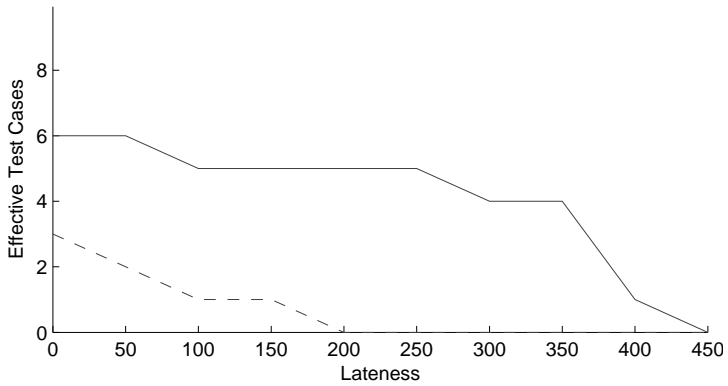


Figure 9.7: Failures detected in variant 7 of the system

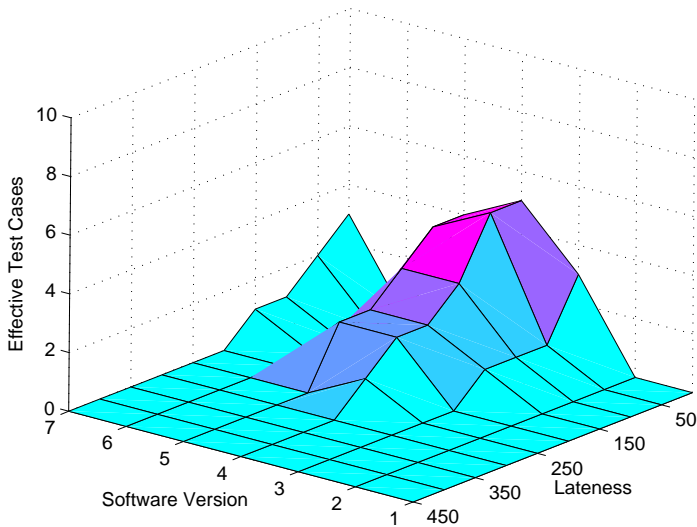


Figure 9.8: Failures detected in seeded variants with random testing

Point 0 on the x-axis indicates the number of effective test cases (using the y-axis), assuming the deadlines listed in table 9.10. The values along the x-axis represent an iterative increase (in steps of 50 microseconds) of all the deadlines in table 9.10. Since there is 7 mutation-based test cases and 10 random test cases in a test suite, the maximum of the solid line is 7 whereas the maximum of the dashed line is 10. The graph is not normalized to maintain visibility of the number of test cases leading to failures. This type of graph indicates how many test cases in each test suite cause deadlines to be missed and how late the system responds when tested with different kind of test suites.

An overview of the results for all the software variants is presented as a surface plot in figures 9.8 and 9.9. These figures contains the results of the most effective test suites for each software variant with seeded faults. The graphs contain the same information as in figure 9.7, but each of the variants is plotted on the z-axis. Similar plots for the least effective test suites and the mean effectiveness is given in Appendix C.

From these surface plots it can be seen that the test suites generated with mutation-based testing contain a larger percentage of effective test cases, and consistently reveal behaviors with longer response times than random testing.

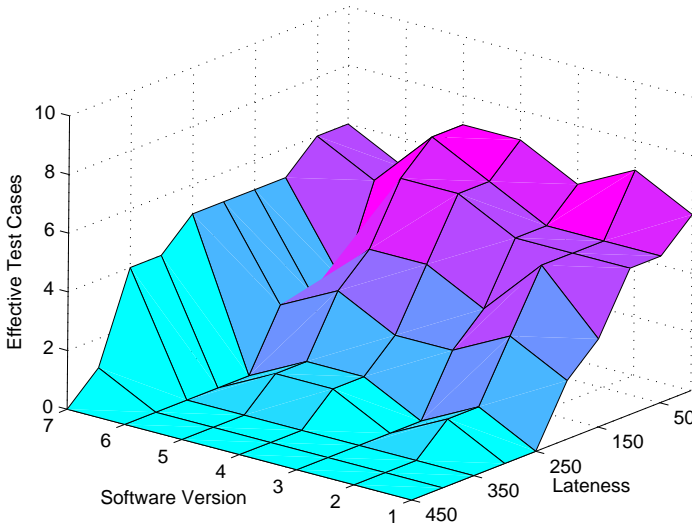


Figure 9.9: Failures detected in seeded variants with mutation testing

Analysis of experiment 2

This experiment indicates that a large fraction of the mutation-based test cases remains effective for finding seeded errors when the slack in the system is increased to compensate for some transient errors.

A remarkable observation is that even after the software has been seeded with deterministic timeliness errors, test cases with random activation patterns are often incapable of causing any deadlines to be missed. This indicates that a system tested only with random activation patterns can contain many potential timeliness errors that are never revealed.

For example, if variant two of the system implementation was tested using 50 random activation patterns (that is, 500 test runs with different combinations of task input data and activation patterns) no deadline violations would be detected³. In the trial with the least effective mutation-based tests (see figure C.2 in Appendix C), five of the timeliness test cases reveal missed deadlines in this system and the

³If we assume that the performance of executing random activation patterns is comparable to the simulation experiments with random search, described in section 7.4, then it is likely that random testing will perform badly even if the number of random tests are significantly increased.

system was stressed so that a deadline was missed with over 200 microseconds (this trial consisted of 70 test runs). These results support the claim from section 3.4 that methods for testing of timeliness benefit from exploiting information about the internal behavior of real-time systems.

There are some issues concerning the setup of this experiment, that need to be discussed. One such issue is that the errors seeded during experiment two are big in relation to the measured execution behavior of tasks. The reason for this is that the increase in system slack adds a “dampening” effect which makes small errors difficult to detect. By adding big timeliness errors we increase the probability of a deadline being missed in a system with additional slack and this makes it easier to compare the testing approaches. A more thorough discussion related to seeding timeliness errors is available in section 10.4.1.

Another important note is that execution orders are independent of the amount of slack in the system when using fixed priority scheduling protocols. This is not the case when using other execution environments. It is, thus possible that the effectiveness of mutation-based test suites for such systems is more sensitive to increasing the slack. Hence, the comparison presented in this section may not necessarily generalize to systems with dynamic priority schedulers.

Discussion

This chapter contains a discussion of the methods and experiments presented in this thesis. Furthermore, some related ideas that remain to be evaluated are presented.

Referring to the framework overview in figure 5.1, issues relating to testing criteria and model analysis (activity 1 and 2) are discussed in section 10.1, aspects of test case generation (activity 3) are discussed in section 10.2 and a discussion of how test execution (activity 5) can be performed in a way that exploits mutation-based test case generation is found in section 10.3. Task input data generation and test analysis (activities 4 and 6) are not addressed in this chapter, since no significant contributions have been made for these activities.

Finally, section 10.4 contains a discussion of issues relating to validation and the research presented in this thesis.

10.1 Mutation Operators and Testing Criteria

When discussing mutation-based testing criteria, three issues need to be addressed. These are *(i)* comprehensiveness, *(ii)* meaningfulness, and *(iii)* cost effectiveness.

In this context, comprehensiveness means, that for all errors that might lead to timeliness failures of a tested system, a corresponding mutation operator is defined. A mutation operator is considered meaningful if it creates at least one malignant mutant. It could also be argued that a meaningful mutation operator should generate mutants which represent some error type that actually can occur during the design or implementation of real-time systems. This property is more difficult to

evaluate, but if it is possible to come up with examples of faults that would cause such an error (such as described in section 6.3), then it is likely that the operator is meaningful even under that definition. A mutation operator is not cost effective if it creates a huge number of rarely malignant mutants.

As previously mentioned, it is generally impossible to show that a set of mutation operators is comprehensive. Further, it is possible that some mutation operators are not meaningful or too expensive for a specific system (for example, due to properties enforced by real-time protocols).

These issues are encountered for most types of mutation testing, and knowledge of comprehensiveness, meaningfulness and cost effectiveness typically develop as a testing method matures and is used in different situations.

For comparison, a set of mutation operators for class testing (object-oriented testing) were first presented by Kim, Clark & McDermid (2000). These were used and refined by Chevalley & Thévenod-Fosse (2002) two years later. At the same time Offutt et al. developed a categorization for object oriented programming faults (Offutt, Alexander, Y.Wu, Xiao & Hutchinson 2001). This resulted in a more comprehensive set of mutation operators for the Mujava tool (Ma et al. 2005). However, subsequent experimental studies revealed that some of the mutation operators were expensive compared to their fault finding capability when used on a set of open source systems, whereas other operators were determined to be meaningless since they only produced benign mutants (Offutt, Ma & Kwon 2006).

With this analogy in mind, our current impression is that the mutation operator types in section 6.3 provide a comprehensive set of mutations which are meaningful for real-time applications that can be modelled with the TAT notation. Subsection 10.1.1 also presents some additional variants of mutation operators and subsection 10.1.2 describes how new mutation operators could be integrated to the proposed testing framework.

In this thesis, we generated mutation-based test cases for a small real-time application (see section 9.3) and some different types of hypothetical real-time system models (see section 7.4). It is thus too early to make general statements about the overall cost effectiveness of the mutation operators in section 6.3. However, our first impression is that the mutation operators which indirectly increase the load in potentially critical intervals (for example, $\Delta+$ execution time mutants or $\Delta-$ inter-arrival time mutants) are more cost effective than the corresponding mutations which decrease the system load. This is intuitive, since the load of real-time systems is directly related to timeliness, whereas other types of mutations typically cause timeliness failures via increased blocking times and race conditions. On the other hand, timeliness faults leading to dangerous race conditions may be

hard to reveal using other testing methods (for example, brute force random testing on a non-deterministic platform), and hence, the low cost effectiveness and the increased computation effort might be acceptable in some situations.

10.1.1 System Specific Mutation Operators

As previously discussed, the mutation operators evaluated in this thesis are only comprehensive for applications that comply with the TAT real-time application model notation. This excludes some fault types that might lead to timeliness failures in some specific systems.

For example, the TAT notation presented in section 6.2 assumes that each resource is locked and unlocked by the same task instance before it terminates. In practice, this means that the handling of semaphores must be encapsulated and used only for protecting critical sections. This is typically the case if concurrency control is provided to the user in the form of mechanisms such as condition critical regions (where a compile-time warning is issued if a semaphore is incorrectly used) or monitors (where locking and unlocking is handled implicitly when entering and exiting monitor functions).

Unfortunately, not all execution environments for real-time applications enforce programmers to use such well behaved concurrency control mechanisms¹.

Further, in some systems, low level mechanisms such as semaphores might be used for synchronization between tasks, or for modelling some condition in the systems environment. For such situations, the basic TAT notation is not sufficient to accurately model the possible execution orders in the target system.

However, the simulation based approach for test-case generation and the flex-time tool is designed in such a way that it can still be used to support many extensions. System specific extensions to the notation for real-time application models and new types of mutation operators might therefore be needed to automatically generate effective test cases. It is unclear if these kinds of extensions can be added to a model checker tool so that the schedulability analysis used for killing mutants can still be performed. If the simulation based approach is used, it should be straightforward to add such extensions to the execution order analysis.

An example of such a mutation operator and the associated extensions follow. However, we have not conducted any experiments with these extensions.

¹Despite many efforts in academia to raise the abstraction of real-time system design and implementation, many real-time operating systems and platforms still assume that low-level languages and synchronization mechanisms are used directly.

Lock omission mutant

In a system where semaphores must be manually set for protecting critical sections it might be the case that unlocking a resource is not only delayed, but completely omitted. Hence, the “lock omission” mutation operator removes the operations² that lock or unlock resources.

Omitting the operation where a resource is locked does not have a direct effect on timeliness. However, since a mutual exclusion constraint is violated, it is possible that a logical error eventually occurs which changes the control flow so that tasks behave differently than expected. The effect of omitting the operation in which a resource is unlocked is much more severe, since any other real-time tasks which need to lock that resource are blocked indefinitely.

In addition, elusive race conditions might occur if a system contains errors in which both lock and unlock operations for a particular resource are omitted by different tasks.

To support this type of mutation operator, the notation for resource locking would have to be extended with a special symbol \dagger , representing “never”. With this extended notation, the set SEM could contain tuples in any of the forms (s, t_1, t_2) , (s, t_1, \dagger) , (s, \dagger, t_2) . Given this extension, the lock point omission operators can be defined.

Mutation Operator 15 *unlock omission:*

Given a TAT model with task set P , for every task $(c_i, d_i, SEM_i, PREC_i) \in P$ and every semaphore use $(s, t_1, t_2) \in SEM_i$, create one mutant in which t_2 is changed to \dagger .

Mutation Operator 16 *lock omission:*

Given a TAT model with task set P , for every task $(c_i, d_i, SEM_i, PREC_i) \in P$ and every semaphore use $(s, t_1, t_2) \in SEM_i$, create one mutant in which t_1 is changed to \dagger .

10.1.2 Guidelines for Adding Mutation Operators

The mutation-based framework for testing of timeliness proposed in this thesis is intended to be general and support as many types of target systems as possible. However, it is not feasible to predict all possible target systems and all the possible system specific mutation operators that might be desirable for comprehensiveness. Therefore, this section describes some guidelines for adding mutation operators.

²These are sometimes called the entry or exit protocols of a critical section.

In principle, mutation operators can be defined to add, remove or change the attributes of some entity or relation between entities in the real-time system model. For example, such entities are tasks and resources and attributes are execution times and constraints on the inputs (see Appendix D).

The following guidelines are based on what kinds of extensions are directly supported by the proposed notation and the design of the Flextime tool. This provide an indication of how to model a particular hypothesized fault. We assume that most of the guidelines also apply to extending a model-checking tool to perform execution order analysis, but this depends on the design and limitations of the particular model-checker.

Do not add errors that make the real-time model non-deterministic

If the simulation of a mutant with a particular activation pattern becomes non-deterministic (for example, by modelling the execution time of tasks as a random distribution), then it is difficult to determine if the activation pattern leads to a dangerous execution order or not. Hence, the search for an effective activation pattern is disturbed. If a particular fault is transient, model the behavior when the fault occurs, find an activation pattern to reveal it, and run it multiple times on the non-deterministic target system instead. This is a way to divide and conquer the testing problem.

Combine the effect of existing mutation operators

This means that several mutation operators are applied to the same model, but potentially mutating different entities. For example, by applying two existing mutation operators (“ Δ – Interarrival time” and “ Δ – Execution time”) it is possible to test if a system is untimely when the period of task A is extended and the execution time of task B is decreased. This does not require any changes in the notation for the real-time system model. It is notable that one effect of specifying this type of mutation is that one mutant is created for every possible combination.

Iteratively apply existing mutation operators

This means that several mutation operators are applied in series for mutating a specific entity. For example, it is possible to test the consequences if the execution time of task B is longer than expected and the point in time when a resource is unlocked is also delayed. This does not require any changes in the notation for the real-time system model. Also, this kind of mutation becomes a specialization of

an existing mutation operator so the maximum number of mutants created will not increase.

Add or replace some real-time task

If timeliness errors can be modelled as real-time tasks which execute with a certain activation pattern that is predictable or controllable during testing, then a mutation operator could add such tasks to the system without changing the notation for the real-time system model. For example, if a suspected error is that the application periodically is disturbed by some undocumented kernel overhead (or a task created by a COTS software module), but the frequency or what the consequences might be is unknown, then it is possible to create a mutation operator that adds different variants of a periodic task. The mutants are then analyzed to discover the malignant variants and generate test cases that can reveal such behavior.

Inject or modify specialized execution segments

With a small extension to the TAT-notation and the FlexTime add-on it is possible to model timeliness errors by creating new types of execution items (see section 8.2). Mutation operators can then be specified that inject or modify execution items in the execution pattern of tasks. For example, the system specific mutation operator outlined in section 10.1.1 could be implemented with an operator that removes the execution items which perform the locking and unlocking of resources. Another example is adding an execution segment that cannot be preempted or one that blocks the task and waits on a particular data dependency.

Modifying tasks instance execution patterns

As described in section 8.3, all the current mutation operators change a task execution pattern so that the behaviors of all task instances are changed in the same way. The extended modelling style used in the case study described in section 9.4.1 and the associated support in the timeliness testing tool (T^3) actually allow mutation operators to change a subset of task instances. Hence, it is possible to create mutation operators that modify only one of the execution patterns. For example, it is possible to create a mutant operator that change the second instance of task A to execute twice as long as the previous one.

Mutating some execution environment model property

The mutation-based test case generation approach proposed in this thesis typically assumes that the execution environment model is fixed for all mutants and that only properties of the real-time application models are modified by mutation. However, if properties of the execution environment model are parameterized (so that changes do not require recompilation of the kernel simulator), then it is possible to define mutation operators that change aspects of the execution environment. For example, a mutant can be that the priority inheritance protocol has been implemented incorrectly.

10.2 Test Case Generation Issues

The test case generation method proposed in this thesis is primarily associated with generating activation patterns for event-triggered tasks. Since the target platform contains sources of non-determinism (for example caches), several different execution orders are expected for a particular activation pattern of sporadic tasks. Once a particular activation pattern has been generated the test execution techniques presented by Thane and Pettersson (Thane 2000, Pettersson & Thane 2003) can be used to gain coverage of these expected execution orders. Furthermore, the execution order descriptions generated by our approach provide added value since it is possible to determine a subset of these execution orders that are most the relevant to cover for testing of timeliness. The generated execution orders can also be used to support prefix-based testing (see section 10.3).

The execution order analysis performed during test case generation can be seen as a rudimentary form of automated sensitivity analysis for event-triggered real-time systems (c.f. Andersson, Wall & Norström (2006)). However, since we cannot guarantee that our method will find timeliness violations in a model, we do not claim it is rigorous enough to be used this purpose. Testing has the fundamental limitation that it can only show the presence of errors, but not their absence (Dahl et al. 1972). The heuristic-driven execution order analysis shares this property and should therefore be used primarily for testing.

Another possibility is to take the activation patterns generated from mutated models and simulate them on the un-mutated model. This can be seen as testing the timeliness of the model. Obviously, this does not state anything about the properties of the implemented system, but it is likely that such test cases can reveal timeliness violations in models difficult to analyze formally due to reachable state space explosion.

10.2.1 Scalability of Test Case Generation

As opposed to model-checking, the memory consumption does not increase as quickly as the reachable state space of the analyzed model when using a method based on heuristic driven simulation. In addition, the analysis can be halted any time to give the current “best-try” for killing a mutant. Nevertheless, when test cases should be generated from complex system models, the computing time required for on analyzing mutants increases. The following three factors contribute to this effect:

1. The number of mutants created by the selected mutation operators (for fulfilling a given test criteria) increases in a polynomial way with the number of tasks and resources, as outlined in section 6.3.
2. As the search problem expands, more iterations of the genetic algorithm search are required to gain confidence that all the malignant mutants have been killed. The impact of this factor depends on the characteristics of each particular search problem (c.f., Michalewicz & Fogel (1998)), for example, how many execution orders that kill a mutant or if those execution orders correspond to local optima which the heuristic cross overs are likely to exercise. Clearly, it is difficult to estimate the impact of this factor in a general way.
3. The third factor is the time required for each simulation run in the fitness function. The dominating variable in this context is the number of events in the simulation interval and consequently the length of the simulation interval. The length of the simulation interval is determined by the hyper-period of the periodic tasks, that is, the least common multiplier of their periods³. Hence, if there are many periodic tasks with relative prime periods, then the simulation interval may be long. However, this factor can be bounded to a constant by using a global offset as part of the genome and simulating a shorter sub-interval starting from that offset. With this modification, the simulation time can be considered constant with respect to the input domain.

10.2.2 Genetic Algorithm Enhancements

The genetic algorithm suggested for generating test cases in this thesis should be considered a prototype for investigating the feasibility of the test case generation

³Plus the time for the longest offset in the task set

method. The method can be extended in several ways to improve its performance, for example, by evaluating more sophisticated fitness and cross-over functions.

One such extension is preparing the initial population of a genetic algorithm with activation patterns suspected of causing long response times. For example, letting all the activations occur as fast as possible is an intuitive heuristic for stressing a real-time system. Another case that could result in long response times is to activate all the sporadic tasks at the same time. In general, if there exists algorithms (with polynomial time complexity) that can generate the “worst-case” activation pattern under specific conditions, these can be added for preparing the run on the initial population. Hence, if the execution environment model used happens to fulfill these conditions, then it becomes trivial to kill mutants.

Another useful extension is modifying the implementation of the heuristic cross-over functions so that they can be used on a larger class of triggering TA templates.

We believe that the cross-over definitions presented in section 7.2 are useful in the general case. However, if the triggering automata templates are complex and contain additional application specific constraints (except task constraints, such as minimum interarrival times), it is non-trivial to implement the cross-over functions to change the activation pattern in the defined way without violating automata constraints. In particular, a transformation is needed from an activation pattern to a particular valid trace in a specified triggering automata. As long as the triggering automata patterns are small and can be isolated (using parallel decomposition (c.f., Fersman (2003))), we assume that an exact implementation can be made using constraint solving. Another alternative is to implement an artificial neural network that is continuously trained on the “inverse mapping function” for a specific automata. Since the heuristic operators already contain stochastic elements, this may provide a fast and sufficiently precise approximation for generalizing the cross-over implementation.

10.3 Prefix-based Test Case Execution

This section outlines a method that enhances test case execution in event-triggered real-time systems. Further, the approach has the possibility of exploiting the information about critical execution orders that is generated using mutation-based testing of timeliness.

When executing timeliness test cases in an event-triggered system we want to concentrate our efforts on testing the situations in which the critical tasks are likely to miss their deadlines. However, a problem when executing tests in real-

time systems is the controllability issue which is related to non-determinism (see section 3.3). This leads to problems in ensuring that a particular execution order has been sufficiently tested.

As previously mentioned, most test case generation methods for testing timeliness are limited to generating activation patterns, that is, sequences of events with time-stamps. However, the system-wide state in which these inputs occur is seldom specified in test cases. This implies that the same activation patterns must be run multiple times to gain confidence in the system's behavior, hoping that some particular execution order occurs. In some situations, this becomes problematic. As an example, consider the simple case when two tasks are executed on the same processor and 70% of all the executions tasks are interleaved in the same way. To reveal an error associated with the reversed execution order of these tasks, the system must be executed at least 13 times to give 99% probability of observing the reversed order one time. Further, assume that the error in this particular execution order is only revealed using one of 100 input data. If non-deterministic testing is used we must execute and analyze 1300 tests to have a high probability of revealing the error.

As discussed by Schütz (Schütz 1994), the number of possible execution orders grows very quickly with the number of activated tasks instances in an event-triggered system, hence, the probability of observing a specific execution order decreases accordingly, even if a uniform distribution of execution orders is assumed⁴.

The alternative is to have some mechanism that controls the execution so that a desired execution order always occurs, or a mechanism that increases the probability that of an execution order occurring during testing.

However, when performing timeliness testing (and other time dependent testing), intrusive probes cannot be inserted in the execution environment during test execution and then removed, as this would change the temporal behavior of the tested system so that it becomes different from the system used in operation (see section 3.3). For the same reason, a "forced" execution must be exactly the same as a behavior that would occur without the enforcing mechanism. According to Schütz, it is difficult to force a single execution order without modifying the target system in a way that alters its timing (Schütz 1993).

A potentially less intrusive approach to enhance the efficiency of test case executions is to increase the probability that a particular execution order occurs. One way of doing this is to include a state description as a test-prefix in each test case, and hence, enable a more controlled test execution even if no single execution order

⁴It is possible that some execution orders occur much more seldom than others, making it even harder to reach them.

is forced (Hwang et al. 1995).

However, adding state-information prefixes to test cases has some implications for test case generation and execution. First, the test case generation method must support the generation of prefixes. Second, the test harness must be able to force the system to an arbitrary specified system state. This generally requires that the system can be monitored and controlled at instruction level precision – to pre-empt each task at the exact instruction specified by the test case. This is generally not feasible on a non-deterministic hardware platform.

By using constraints on the system architecture as described by Mellin (1998) and Birgisson et al. (1999), the problems associated with realizing prefix based testing are simplified. In particular, including designated preemption points reduces the number of observable intermediate states in the system and supports the test execution scheme described in section 10.3.2.

10.3.1 Prefixed Test Cases for Testing of Timeliness

Test cases for prefix-based timeliness testing contain an activation pattern, an expected outcome and an initial state description (prefix).

In complex systems such as concurrent real-time computers, a state description can be expanded almost infinitely by considering more detail. However, to enhance test execution on non-deterministic platforms it is sufficient to specify the initial state in such a way that the probability of reaching a specific execution order is increased compared to executing the system from an idle or start-up state. To achieve this, the prefix state-description suggested below comprises:

- Pre-emption points of active tasks
- State of system-wide locks
- State of ready queue and/or schedule
- Time relative start of the loading interval

This means that prefixes can be specified at the granularity of pre-emption points in the code and that an execution order trace of the loading interval and critical interval can be used to define a prefix. Hence, if these kinds of constraints are supported, the mutation-based test case generation methods presented in this thesis can be used to generate interesting prefixes. In particular, this information can be

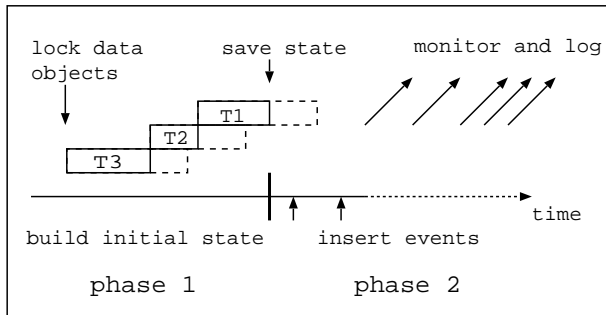


Figure 10.1: Test execution phases

generated by injecting the test-case activation pattern on the mutated real-time system model and then store a snapshot of the simulation state just before the mutated entity may affect the simulation.

In addition to the prefixes, the test case information specified in section 5.2 is still required for prefix-based testing of timeliness.

10.3.2 A Prefix-based Test Case Execution Scheme

The first step in executing a test case with this scheme would be to run the system into the specified prefix state. Under a strict pessimistic concurrency control policy, the active transactions have also acquired locks on all the required resources (Gray & Reuter 1993). Hence, serial execution up to the required transaction states is possible while no time dependencies are reflected in the data objects (in Figure 10.1; transactions T1, T2, and T3 are executed sequentially to specified preemption points). Obviously, the state of the locked objects influences the execution time of the individual transactions; hence, relevant states of shared objects should be treated as task input data. Once each active task is executed to the specified preemption point, they are combined into a concurrent prefix state. Pending transaction tasks which have not yet been allowed to execute, but are part of the prefix specification, are added to the ready queue. If a dynamic scheduler is used, it is assumed to have updated the ready queue as a direct response to incoming events and hence, one must allow it to build an initial schedule before the test execution begins. At this point the specified prefix has been reached.

In the second phase the system is executed from the prefix state and inputs are externally injected according to the activation pattern part of timeliness test cases. The behavior of the system is continuously monitored and logged using the event

monitoring facilities present in the architecture. The event logs are used after the test case execution to verify that a correct execution order was observed and that all the outputs occurred within their time constraints.

10.4 Validation of Mutation-based Testing of Timeliness

This section contains a discussion of the issues related to validating timeliness testing methods in general, and the hypotheses outlined in this thesis in particular.

10.4.1 Evaluating Timeliness Test Cases

A typical way of comparing the effectiveness of testing methods is running test suites to see which ones find the most faults. This requires access to a set of software that contains the type of faults used for comparison. There are three ways of obtaining to such software.

One is to use software systems that already contain faults. These may be faults that have been found, documented and corrected in a new release. It is also possible to use non-tested software that has (at least not tested at this particular level or for a specific type of faults). This approach was used for the experiment described in section 9.4.1. A second way is to automatically seed faults in an assumed correct software by changing the software system directly (as opposed to the mutation operators in section 6.3 that only change the model of the system). This approach results in a number of variants of the system, each containing a fault at a randomly chosen location. This is similar to the approach used in the experiment described in section 9.4.1. A third way is to let a person manually seed faults in copies of the software. If this approach is used, it is important to separate the task of seeding faults from the task of applying a test method, since it may introduce some bias if the tester knows the location of faults.

It is often difficult to manually seed realistic timeliness faults. The reason is that such faults may arise from subtle interactions between software and hardware components, or configuration decisions that are not visible in the application source code. Further, for testing of timeliness there are some constraints on what kinds of faults are meaningful and possible to seed. A dynamic real-time system design executing on a non-deterministic platform would probably contain some slack to tolerate transient disturbances (for example, see section 9.4.1). Hence, seeding very small faults would probably not cause any timeliness failures and would therefore not be detected using any of the compared testing methods. Conversely, faults causing substantial timeliness errors would be detected by running almost any execution

order, and would consequently be detected using almost any testing method. The faults difficult to find, and the ones that best evaluate the relative effectiveness of testing methods, are those that only few of the possible execution orders can reveal. This complicates the manual seeding of faults since it is hard for an independent fault seeder to know what kind of faults that would cause such errors.

Another issue that complicates fault seeding is that some seeded faults might cause logical errors that cannot be detected by the timeliness testing method. For example, an output value might be incorrectly calculated from state variables and certain input values. Such faults are assumed to have been found during unit testing and are not meaningful to seed for evaluating this type of testing methods.

10.4.2 Validation of Thesis Statement

A significant part of the research in this thesis is associated with the development of new techniques and the adaption of existing ones for a new problem domain. This kind of research is guided by experience from small laboratory experiments, intuition and continuous literature studies. Once basic confidence for an idea is established, it is evaluated or demonstrated in a larger or more formal experiment. Furthermore, the research is guided by the objectives presented in section 4.4.2. In particular, the sub-hypotheses from section 4.4.1 and the corresponding objectives and validation experiments are recapitulated and discussed in the following paragraphs.

H1: There is a real-time specification notation which captures the relevant internal behavior of dynamic real-time systems so that meaningful mutation operators can be defined.

Chapter 5 summarizes the activities and requirements for mutation-based testing of timeliness so that both test case generation and test execution can be automated (objective 1). Based on these requirements, chapter 6 adopts a previously defined real-time modelling notation (TAT) that captures the relation between activation patterns and internal execution orders of dynamic real-time systems (objective 2). Subsequently, a set of mutation operators is defined for this modelling notation and evaluated in a model-checking experiment. This experiment corroborates sub-hypothesis H1.

H2: Test cases for testing of timeliness can automatically be generated using mutation-based testing and models of dynamic real-time systems.

In chapter 6 it is recognized that the same model-checking mechanism used for validating the mutation operators can be used for automatically generating timeliness test cases. However, since this method has limitations with respect to system size and extendability, it is not practical for all types of dynamic real-time systems. Instead, chapter 7 presents another approach for automatically generating mutation-based test cases using heuristic driven simulations (objective 3). This method is evaluated in three test case generation case studies that corroborate sub-hypothesis H2.

H3: Test cases generated using mutation-based testing are more effective than random test cases for revealing timeliness errors on a realistic target platform.

To perform experiments and support extendability, a prototype tool was developed. The central features in the design of this tool are presented in chapter 8 (objective 4). In chapter 9, a case-study is described in which the mutation-based testing framework is applied on a small real-time control application (objective 5). The case-study exemplifies how the different activities in the framework can be performed for a specific target system. It also contains an experiment that gives confidence in the effectiveness of mutation-based test cases, given that large deterministic variances in execution patterns are captured during test case generation. Hence, this experiment corroborates sub-hypothesis H3 under the condition that the mutation-based test cases are generated using a suitable model of task instances.

Consequently, we conclude that this thesis provides one possible solution of the problem outlined in section 4.3⁵. A mutation-based testing method has been adapted so that it can be used to automatically generate activation patterns that target execution orders that are likely to reveal timeliness failures. Since the population of dynamic real-time systems is unknown (this holds for most types of software), it is difficult to assess the general applicability of a testing method based on laboratory experiments and even industrial case studies (the problem is to obtain a representative sample). However, the adapted method is defined in the context of an extendable framework and, hopefully, this will encourage practitioners and researchers to apply the method and its associated tool for other platforms and similar testing problems.

⁵Note that it is possible to negate each of the sub-hypotheses and express the experiment results as counter examples. However, we find it more natural to present the sub-hypotheses as statements in the positive form.

Related Work

This chapter presents related work in the joint area of software testing and real-time, reactive systems. The focus is on test case generation but other relevant contributions in the area of testing are also discussed.

11.1 Test Case Generation

This section describes the state-of-the-art methods of generating test cases for real-time systems. In particular, there is a focus on methods that claim to generate test cases for real-time, embedded or reactive systems. Table 11.1 lists the authors of related work and classifies the contributions with respect to three categories. When the same authors have several related publications addressing different aspects of the same method, only one is included in the table. The first category (the column denoted ‘T’) indicates whether the approaches are developed for system level testing of time constraints. Generally, such approaches use some formal or structured model for capturing time constraints.

The column denoted ‘I’ indicates if the related work uses information about concurrent tasks, shared resources and real-time protocols for deciding relevant inputs. In contrast to our approach, few other methods based on formal notations include this in their models (probably to avoid of the associated state space explosion). However, if the internal behavior is not modelled, it is generally impossible to predict the worst case activation pattern for a system that is implemented using conventional real-time operating systems and task models. For example, exactly

#	Authors	T	I	C
1	Braberman, Felder & Marré (1997)	y	y	y
2	Cheung, Chanson & Xu (2001)			n
3	Clarke & Lee (1997)		n	y
4	Petitjean & Fochal (1999)			
5	Hessel & Pettersson (2004)			
6	Krichen & Tripakis (2004)			
7	Mandrioli, Morasca & Morzenti (1995)			
8	Cardell-Oliver & Glover (1998)			
9	En-Nouaary, F. Dssouli & Elqortobi (1998)			
10	Nielsen & Skou (2000)		n	
11	Raymond, Nicollin, Halbwachs & Weber (1998)			
12	Hahn, Philipps, Pretschner & Stauner (2003)			
13	Watkins, Berndt, Aebischer, Fisher & Johnson (2004)			
14	Morasca & Pezze (1990)	n	y	n
15	Pettersson & Thane (2003)			
16	Wegener et al. (1997)			

Table 11.1: Classification of related work

the same input sequence might give completely different behaviors depending on the relative priority of active tasks.

The column denoted ‘C’ lists whether or not the related work proposes testing criteria that are usable together with their method.

The method by Braberman et al. (1997) is the closest related work; they generate test cases from timed Petri-net design models. Similar to our method, a high level notation, SA/SD-RT is used to specify the behavior of concurrent real-time systems. In contrast to our approach, no mutant models are generated, instead their design specification is translated to a timed Petri-net notation from which a reachability tree can be derived and covered. Since the tree grows with the reachable state space of the model, we conjecture that the number of tests becomes high for dynamic real-time systems. Furthermore, no tool is publicly available to model systems and automatically generate test cases using this approach.

Cheung et al. (2001) present a framework for testing multimedia software, including temporal relations between tasks with “fuzzy” deadlines. In contrast to our approach, the test cases generated are targeted for testing multi-media applications and their specific properties. Similar to our approach, information about tasks and precedence constraints are considered during test case generation.

There are several methods for testing timeliness based on different kinds of formal models. As mentioned above, these methods focus on covering the structure of the model and typically do not model the behavior of real-time tasks and protocols on the tested system. Further, none of these methods use mutation-based testing techniques. These differences from our approach hold for all model-based methods in this category, hence, they are not explicitly compared to our approach (see table 11.1, rows 3 - 13).

For example, Clarke & Lee (1997) propose a framework for testing time constraints on the request patterns of real-time systems. Time constraints are specified in a constraint graph, and the system under test is specified using process algebra. In contrast to our approach, only constraints on the inputs are considered and the authors mention that it would be very difficult to test constraints on the output since it depends on internal factors.

Petitjean & Fochal (1999) present a method where time constraints are expressed using a clock region graph. A timed automation specification of the system is then “flattened” to a conventional input output automation that is used to derive conformance tests for the implementation in each clock region. This method does not describe the execution environment of the system such as scheduling protocols and shared resources. However, a discussion of how clocks in the target system can be handled when conducting model-based conformance testing is presented.

Hessel & Pettersson (2004) presents a method for automatically generating conformance tests for real-time systems that are modelled using timed automata. One of the main contributions of this work is the ability to use reachability analysis tools to generate a “minimal” set of activation patterns that fulfill classical structural coverage criteria for graphs (for example, as presented by Fujiwara, Bochmann, Khendek, Amalou & Ghedamsi (1991)).

Krichen & Tripakis (2004) address limitations in the applicability of previous black-box approaches and suggest a method for conformance testing using non-deterministic and partially observable models. The testing criteria presented are inspired by Hessel, Larsen, Nielsen & Skou (2003) but extended for test case specifications that allow several possible interactions with the implementation.

Mandrioli et al. (1995) suggest a method for testing real-time systems based on specifications of the system behavior in temporal logic. The elements of test cases are timed input-output pairs. These pairs can be combined and shifted in time to create a large number of partial test cases; the number of such pairs grows quickly with the size and constraints on the software. In a more recent paper, SanPietro, Morzenti & Morasca (2000) expand the previous results to incorporate high-level, structured specifications to deal with larger scale, modular software.

Cardell-Oliver & Glover (1998) propose a method for generating tests from timed automata models to verify the conformance to sequences of timed action transitions. This method attempts to cover all reachable states and transitions in the model, and hence, the number of test cases can become substantial. In a more recent paper (Cardell-Oliver 2000), test views are introduced to limit the number of test cases. However, no method for generating relevant test views are presented. Another automata based approach is described by En-Nouary et al. (1998). Their approach exploits a sampling algorithm using grid-automata and non-deterministic finite-state machines as an intermediate representation to reduce the test effort. Similarly, Nielsen & Skou (2000) use a subclass of timed automata to specify real-time applications. The main contribution of their method is a coarse equivalence partitioning of temporal behaviors over the time constraints in the specification. In a more recent paper (Nielsen & Skou 2003), the authors suggest strategies for choosing delays within partitions, one of which is similar to random testing while another tries to stress the system by choosing as short delays as possible. We assume that the latter strategy produces test cases similar to the manually created tests used in the experiment described in section 9.4.1. Raymond et al. (1998) present a method of generating test cases for reactive systems. Instead of explicitly generating activation patterns, this approach focuses on constraints imposed by the environment and on generating external observers.

Hahn et al. (2003) describe a method for generating test cases based on models of reactive systems that interact with physical processes in real-time. The main contribution of their work is the concept of using a separate continuous model to generate expected outcomes in the value domain. No testing criteria are used, instead feedback from the test execution and the continuous process model is used in a “closed-loop” fashion to generate new tests. By using the control system modelling approach, described in section 7.3 of this thesis, similar expected outcomes could also be generated using mutation-based testing.

In contrast to our approach and the other methods in this category, Watkins et al. (2004) do not use a formal model as basis for test case generation. Instead, genetic algorithms are used directly to test complex systems that contain time constraints. Data are gathered during the execution of the real system and visualized for post analysis. The fitness of a test case is calculated based on its uniqueness and what exceptions are generated by the systems and test-harness during test execution. Similar to this method, our genetic algorithm extensions can be used directly on a target system instead of on a model. However, the search problem would be more complex and no testing criteria could be used for measuring the progress of such a test method.

There is some related methods, not developed for system-level testing of timeliness, that nevertheless are relevant or complementary to our approach (see table 11.1, rows 14 - 16).

Morasca & Pezze (1990) propose a method for testing concurrent and real-time systems that uses high-level Petri-nets for specification and implementation. This method does not explicitly handle timeliness, nor does it provide testing criteria, but it is one of the first to model the internal concurrency of the tested real-time system using Petri-nets.

Thane (2000) proposes a method to derive execution orders of a real-time system before it is put into operation. It was suggested that each execution order can be treated as a sequential program where conventional test methods can be applied. During test execution, the tests are sorted according to the pre-analyzed execution orders. In a more recent paper, Pettersson and Thane (Pettersson & Thane 2003) extend the method by supporting shared resources. In contrast to our method, this method is developed for real-time systems where all task activation times are known at design time.

Wegener et al. use genetic algorithms to test temporal properties of real-time tasks (Wegener et al. 1997). However, the main focus of their work is determining suitable inputs for producing worst and best-case execution times. This approach is a valuable complement to our method, since we assume that relevant classes of input data exist for each real-time task before system-level testing of timeliness starts.

11.2 Testing of Real-Time Systems

Thane (2000) presents results for testing and debugging distributed real-time systems. As previously mentioned, Thane focuses on statically scheduled systems where it is tractable to analyze the possible execution orders before the system goes into operation. The system is then executed until there is enough test coverage of each of the anticipated execution orders. Thane does not present any method for generating timeliness test cases or relevant execution orders; instead he suggests that conventional methods can be used in combination with his execution order analysis. Thane also proposes a method for deterministic replay and debugging facilities for real-time systems. Both the non-deterministic test execution approach and the replay and debugging facilities are useful complements to the testing framework presented in this thesis.

Although the focus of Schütz's (Schütz 1993) work is on testing time-triggered

systems, he discusses test execution in event-triggered systems. The factors that he identifies as important when instrumenting test objects are synchronization of processes, access to time and synchronous interrupts. Further, Schütz discusses the test effort of the two design approaches and concludes that the theoretical number of execution orders in an event-triggered system increases exponentially.

Mellin (1998) and Birgisson et al. (1999) also elaborate on the test effort associated with the different paradigms and suggest some constraints that would potentially decrease the test effort of event-triggered systems while maintaining their event-triggered semantics.

For example, a task that executes in an event-triggered real-time system can be preempted multiple times by different higher prioritized tasks. Each combination of preemptions potentially results in a different state of the overall system. The enormous number of states complicates the testing of event-triggered systems, since to gain confidence in the timely delivery of a specific result, the introduction of related events may need to be tested at each possible system state (Birgisson 1998).

Recently, Andersson et al. (2006) presented the ART framework for analyzing the temporal behavior of complex embedded systems. The purpose of this framework is mainly to increase analyzability of the current system behavior and to be able to analyze potential changes to the real-time design. This means that a model is constructed based on the execution behavior of an system, it is then changed and simulated to reveal potential problems. This is quite similar to the execution analysis part of mutation-based testing, as presented in this thesis, but the emphasis is in analyzing the simulations of the changed system instead of deriving test cases for the new version. Another difference is that the models of real-time applications extracted by this approach are stochastic, meaning that the analysis becomes different and does not support the divide and conquer separation between test case generation and test execution, advocated in this thesis (see section 10.1.2). It might be possible to integrate support for mutation-based testing in the ART framework and thereby increase the usefulness of both approaches. For example, such integration may provide the possibility of generating tests and formulating testing criteria using the ART models.

Conclusions

This chapter summarizes this thesis and elaborates on the impact of the results and contributions. Future directions for research on this topic are also suggested.

12.1 Summary

Timeliness is a property that is unique for real-time systems and deserves special consideration during both design and testing. A problem when testing timeliness of dynamic real-time systems is that response times depend on the execution order of concurrent tasks. Other existing testing methods ignore task interleaving and timing and, thus, do not help determine what kind of test cases are meaningful for testing timeliness. This thesis presents several contributions for automated testing of timeliness for dynamic real-time systems. In particular, a model-based testing method, founded in mutation testing theory, is proposed and evaluated for revealing failures arising from timeliness faults. One contribution in this context is that mutation-based testing and models developed for generic schedulability analysis, can be used to express testing criteria for timeliness and for automatic generation of mutation-based test cases. Seven basic mutation operators are formally defined and validated to represent error types that can lead to timeliness failures. These operators can subsequently be used to set up relevant testing criteria.

Two approaches for automatically generating test cases for timeliness are defined. One approach reliably generates test cases that can distinguish a correct system from a system with a hypothesized timeliness error. The second approach is

designed to be extendible for many target platforms and has the potential to generate test cases that target errors in the interaction between real-time control systems and physical processes, modelled in MATLAB/Simulink. In addition, this thesis outlines a scheme for prefix-based test case execution and describes how such a method can exploit the information from mutation-based test case generation to focus testing on the most relevant scenarios.

The contributions outlined above are put into context by a framework for automated testing of timeliness. This framework specifies activities, techniques and important issues for conducting mutation-based timeliness testing – from system modelling to automated test case execution.

The validation of the proposed testing approaches is done iteratively through case studies and proof-of-concept implementations. In particular, the mutation-based testing criteria are validated through a model-checking case study (see section 6.6). This case study reveals that the generated mutants actually lead to timeliness violations in a simple real-time system model. The simulation based test case generation method is evaluated in a set of test case generation experiments with different target system characteristics (see section 7.4). These experiments indicate that the approach is applicable for generating test cases for non-trivial dynamic real-time systems and real-time control systems with mixed task loads. This was not possible using previously existing methods due to problems with the size of the reachable state space and limitations in tool support. Finally, the proposed framework for testing of timeliness is demonstrated on a small robot control application running on Linux/RTAI. This case study indicates that the mutation-based test cases, that are generated using assumptions of the internal structure of the real-time system, can be more effective than both naively constructed stress tests and a suite of randomly generated test cases that are approximately ten times larger.

Consequently, our case studies and experiments corroborate the hypothesis presented in section 4.4.1. Mutation-based testing can be used for generating effective test cases by exploiting models that take internal behaviors into consideration.

12.2 Contributions

This section briefly summarizes the contributions of this thesis. The contributions are the result of pursuing the objectives outlined in section 4.4.2.

- A mutation-based framework for testing of timeliness. The framework is constructed to address the problems summarized in chapter 4 and is demon-

strated to be effective in finding timeliness errors in dynamic real-time systems.

- An extendible set of mutation operators defined in terms of a modelling notation which captures varying execution orders. This kind of mutation operators allows coverage to be expressed when testing timeliness.
- A method for automatically generating mutation-based test cases for timeliness using model checking.
- A method for automatically generating mutation-based test cases for timeliness using heuristic driven simulations.
- A tool prototype that allows automated test case generation according to the proposed framework and supports modelling of the environment using MATLAB/Simulink.

A part from these contributions, this thesis contains an example of how the proposed framework can be applied for testing a simple robot arm control system. Also, a scheme for pre-fix based test execution, that increases controllability and is compatible with the mutation-based framework, is outlined.

12.3 Future Work

This section contains some spin-off ideas of how the framework for automatic testing of timeliness can be extended beyond the scope of this work.

Field-study on Framework Applicability

The framework presented in this thesis has been shown effective using a combination of simulation, proof-of-concept and a limited feasibility case-study. To further validate the applicability and flexibility of the presented framework it is desirable to use it to test a wide range of commercial real-time applications. This kind of study would reveal how well the proposed framework can be adapted for platforms and applications used in commercial real-time system development. Furthermore, it would be interesting to compare the effectiveness of ad-hoc methods used in industry with our framework.

Testing of Timeliness during Mode-changes

The testing experiments presented in this thesis have focused on testing the timeliness of a real-time system from startup. This assumes that the system and its environment continues to operate in a similar way during the rest of its execution. However, for some systems the mode of operation may change at certain points in time. This means that the system task set are altered or that different types of services is requested or become more critical. A typical example of this is a control system on an aircraft, where there is one mode of operation when the aircraft is taxiing on the ground, and another when the plane is airborne. In these kinds of systems it may be important that some services remain timely during the transition from one mode of operation to another. The testing framework presented in this thesis should be evaluated for generating timeliness test cases during the transition from one mode to the other. We believe that such extension is possible and can increase the usefulness of the timeliness testing framework even more.

Testing of Timeliness for Multi-Processor Systems

A emerging trend in general computing is to increase performance by utilizing parallel processors and specialized computing elements (for example, chips containing field programmable gate arrays). Eventually, this trend will force developers to use parallel platforms for dependable real-time applications.

Unfortunately, the problems motivating a structured and generalizable approach for testing of timeliness are even more prevalent in architectures with several active processing units. In particular, the relation between activation patterns and execution orders is more complex and the problems associated with finding critical scenarios are elevated.

The framework presented in this thesis has currently only been evaluated for single processor architectures. However, assuming that the scheduling protocol in a multiprocessor system behave deterministically given a specified task set and activation pattern, it may be possible to use the proposed methods and tools to test timeliness of a multi-processor system. Consequently, the divide and conquer approach provided by the combination of mutation testing, execution order analysis and non-deterministic test execution should be evaluated for this domain. An initial future work along this line is to perform a series of test case generation experiments using a simulated multiprocessor platform.

Evaluation of the Iterative Delta Reduction Mode

During initial experimentation with the different test case generation modes (described in section 8.3), it was observed that, for some system models, the iterative delta reduction mode killed mutants with small deltas more reliably than using the fixed delta mode directly. However, it is still inconclusive whether the iterative tightening of deltas actually improves the capabilities of the heuristic search or if this effect only is a consequence of running more simulations. A series of controlled experiments would be needed to investigate this. Furthermore, we leave as a future work to determine if the hybrid test suites generated using the iterative delta reduction mode can be as effective as the test suites generated using fixed deltas.

Validation of the Prefix-based Test Execution Scheme

The prefix-based test execution approach outlined in section 10.3.2 generally requires support from the test harness and from the operating system where a real-time application is run. Despite indications that designated pre-emption points increase the controllability of dynamic real-time systems, it is still unclear what penalties are associated with including them in real-time systems. An interesting project would be to integrate the prefix-based test execution scheme with a real-time operating system and compare the effort to observe a particular execution order.

A partial implementation of designated pre-emption points has been made on the RTAI platform using a globally shared semaphore (execution token) that is temporarily released and re-taken when certain system calls are issued or when specific points in the task code are reached. During the state enforcement phase a high priority “enforcer task” is used for reading the prefix-description and waking up application tasks in a specified order. A structured experiment is needed to determine if this implementation of the prefix-based test execution scheme can be used without a probe effect.

A Testable Platform for Dynamic Real-time Systems

Even if the testing framework presented in this thesis is developed to be general and adaptable for many platforms, there are potential benefits in standardizing the target system platform. For example, by providing a suite of tools for application modelling, mutation-based test generation, predictable event monitoring and prefix based test execution for a specific platform, real-time application developers can try both the methods and the platform directly.

One candidate platform for event-triggered real time applications is the Distributed active real-time Database System – DeeDS. The DeeDS prototype already contains predictable event monitoring, transaction support and real-time scheduling protocols. However, it is important that the real-time properties of such a platform are evaluated experimentally and captured in an accurate execution environment model.

Evidence-based online monitoring

The test case experiments presented in this thesis indicate that the tasks' specified worst case execution behavior often can be violated without jeopardizing system level timeliness.

Malignant mutants contain cases where such violations can lead to problems. Using execution order analysis of malignant mutants, it is possible to characterize situations where timeliness failures may occur. From this it might be possible to automatically create composite event descriptions that can trigger on sequences of events leading up to a situation where a system level time constraint might be violated. This could be used to predict dangerous situations before they occur and prevent errors leading to timeliness failures.

References

- Alur, R. & Dill, D. (1994), 'A theory of timed automata', *Theoretical Computer Science* **126**, 183–235.
- Ammann, P. & Black, P. (1999), A specification-based coverage metric to evaluate test sets, *in* 'HASE '99: Proceedings of the 4th IEEE International Symposium on High-Assurance Systems', Washington, DC, pp. 239–248.
- Ammann, P., Black, P. & Majurski., W. (1998), Using model checking to generate tests from specifications, *in* 'Proceedings of the Second IEEE International Conference on Formal Engineering Methods', IEEE Computer Society, pp. 46–54.
- Ammann, P., Ding, W. & Xu, D. (2001), Using a model checker to test safety properties, *in* 'Proceedings of the Seventh IEEE International Conference on Engineering of Complex Computer Systems', IEEE Computer Society, Skövde, Sweden, pp. 212–221.
- Ammann, P. & Offutt, J. (1994), Using formal methods to derive test frames in category-partition testing, *in* 'Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS 94)', pp. 69–80.
- Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P. & Yi, W. (2002), Times - A tool for modelling and implementation of embedded systems, *in* 'Proceedings of TACAS'02', number 2280, Springer-Verlag, pp. 460–464.
- Andersson, J., Wall, A. & Norström, C. (2006), A framework for analysis of timing and resource utilization targeting complex embedded systems, Uppsala University, pp. 297–329.

- Andrews, J. H., Briand, L. C. & Labiche, Y. (2005), Is mutation an appropriate tool for testing experiments?, in 'Proceedings of the 27th IEEE International Conference on Software Engineering (ICSE 2005)', St. Louis, Missouri, pp. 402–411.
- Åström, K. J. & Wittenmark, B. (1997), *Computer-Controlled Systems*, Prentice Hall.
- Baker, T. P. (1991), 'Stack-based scheduling of real-time processes', *The Journal of Real-Time Systems* (3), 67–99.
- Beizer, B. (1990), *Software Testing Techniques*, Von Nostrand Reinhold.
- Berkenkötter, K. & Kirner, R. (2005), Real-time and hybrid systems testing, in M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker & A. Pretschner, eds, 'Model-based testing of Reactive Systems', Vol. LNCS, Springer-Verlag, pp. 355–387.
- Bianchi, E., Dozio, L., Ghiringhelli, G. & Mantegazza, P. (1999), Complex control systems, applications of diapm-rtai at diapm, in 'Realtime Linux Workshop', Vienna.
- Birgisson, R. (1998), Improving testability of applications in active real-time database environments, Master's thesis, University of Skövde. HS-IDA-MD-98-001.
- Birgisson, R., Mellin, J. & Andler, S. F. (1999), Bounds on test effort for event-triggered real-time systems, in 'Proceedings of the 6th International Conference on Real-Time Computing, Systems and Applications (RTCSA'99)', IEEE Computer Society Press, Hong-Kong, pp. 212–215.
- Bowen, J. P. & Hinchey, M. G. (1995), 'Seven more myths of formal methods', *IEEE Software* 12(4), 34–41.
- Braberman, V., Felder, M. & Marré, M. (1997), Testing timing behavior of real-time software, in 'International Software Quality Week'.
- Burns & Wellings (2001), *Real-Time Systems and Programming Languages*, Addison Wesley.
- Cardell-Oliver, R. (2000), 'Conformance tests for real-time systems with timed automata specifications', *Formal Aspects of Computing* 12(5), 350–371.

- Cardell-Oliver, R. & Glover, T. (1998), 'A practical and complete algorithm for testing real-time systems', *Lecture Notes in Computer Science* **1486**, 251–261.
- Cheung, S. C., Chanson, S. T. & Xu, Z. (2001), Toward generic timing tests for distributed multimedia software systems, in 'ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)', IEEE Computer Society, Washington, DC, USA, p. 210.
- Chevalley, P. & Thévenod-Fosse, P. (2002), 'A mutation analysis tool for java programs', *Software Tools and for Technology Transfer* pp. 1–14.
- Clarke, D. & Lee, I. (1997), Automatic generation of tests for timing constraints from requirements, in 'Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems', Newport Beach, California.
- Dahl, O., Dijkstra, E. W. & Hoare, C. (1972), *Structured Programming*.
- DeMillo, R. A., Lipton, R. J. & Sayward, F. G. (1978), 'Hints on test data selection: Help for the practicing programmer', *IEEE Computer* **11**(4), 34–41.
- DeMillo, R. A. & Offutt, A. J. (1991), 'Constraint-based automatic test data generation', *Transactions on Software Engineering* **SE-17**(9), 900–910.
- DeMillo, R. A. & Offutt, A. J. (1993), 'Experimental results from an automatic test case generator', *ACM Transactions on Software Engineering and Methodology*.
- En-Nouaary, R., F. Dssouli, K. & Elqortobi, A. (1998), Timed test case generation based on a state characterization technique, in 'Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98)', Madrid, Spain.
- Fersman, E. (2003), A Generic Approach to Schedulability Analysis of Real-Time Systems, PhD thesis, University of Uppsala, Faculty of Science and Technology.
- Fersman, E., Petterson, P. & Yi, W. (2002), Timed automata with asynchronous processes: Schedulability and decidability, in J.-P. Katoen & P. Stevens, eds, 'Proc. of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems', number 2280 in 'Lecture Notes in Computer Science', Springer-Verlag, pp. 67–82.

- Fujiwara, S., Bochmann, G. V., Khendek, F., Amalou, M. & Ghedamsi, A. (1991), 'Test selection based on finite state models', *IEEE Transactions on software engineering* **17**(6), 591–603.
- Gait, J. (1986), 'A probe effect in concurrent programs', *Software - Practice and Experience* **16**(3), 225–233.
- Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software.*, Addison-Wesley.
- Gray, J. & Reuter, A. (1993), *Transaction Processing, Concepts and Techniques*, Morgan Kaufmann.
- Hahn, G., Philipps, J., Pretschner, A. & Stauner, T. (2003), 'Prototype-based tests for hybrid reactive systems', *14th IEEE International Workshop on Rapid System Prototyping* **00**, 78.
- Hansson, J., Son, S., Stankovic, J. & Andler, S. (1998), Dynamic transaction scheduling and reallocation in overloaded real-time database systems, in 'Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications'.
- Henriksson, D. (2006), Resource-Constrained Embedded Control and Computing Systems, PhD thesis, Department of Automatic Control, Lund University.
- Henriksson, D., Cervin, A. & Årzén, K.-E. (2003), TrueTime: Real-time control system simulation with MATLAB/Simulink, in 'Proceedings of the Nordic MATLAB Conference', Copenhagen, Denmark.
- Hessel, A., Larsen, K., Nielsen, B. & Skou, A. (2003), Time optimal real-time test case generation using UPPAAL, in 'Proceedings of Workshop on Formal Approaches to Testing of Software (FATES)', Montreal.
- Hessel, A. & Pettersson, P. (2004), A test case generation algorithm for real-time systems, in 'Proceedings of the Fourth International Conference on Quality Software', pp. 268–273.
- Houck, C., Joines, J. & Kay, M. (1995), A genetic algorithm for function optimization: A Matlab implementation, Technical Report NCSU-IE TR 95-09, Department of Computer Science, North Carolina State University.

- Hwang, G., Tai, K. & Hunag, T. (1995), 'Reachability testing : An approach to testing concurrent software', *International Journal of Software Engineering and Knowledge Engineering* **5**(4).
- Kim, S., Clark, J. & McDermid, J. (2000), Class mutation: Mutation testing for object-oriented programs, in 'Net.ObjectDays Conference on Object-Oriented Software Systems'.
- Kopetz, H., Zainlinger, R., Fohler, G., Kantz, H., Puschner, P. & Schütz, W. (1991), An engineering approach to hard real-time system design, in 'Proceedings of the Third European Software Engineering Conference', Milano, Italy, pp. 166–188.
- Krichen, M. & Tripakis, S. (2004), Black-box conformance testing for real-time systems, in 'Proceecings of SPIN'04 Workshop on Model-Checking Software'.
- Laprie, J., ed. (1994), *Dependability: Basic Concepts and Terminology*, Springer-Verlag for IFIP WG 10.4.
- Larsen, K. G., Pettersson, P. & Yi, W. (1995), Compositional and symbolic model-checking of real-time systems, in 'IEEE Real-time Systems Symposium', pp. 76–89.
- Liu, C. L. & Layland, J. W. (1973), 'Scheduling algorithms for multiprogramming in a hard-real-time environment', *Journal of the ACM* **20**(1), 46–61.
- Ma, Y. S., Offutt, A. J. & Kwon, Y. R. (2005), 'Mujava: An automated class mutation system', *Software Testing, Verification and Reliabilty* **2**(15), 97–133.
- Mandrioli, D., Morasca, S. & Morzenti, A. (1995), 'Generating test cases for real-time systems from logic specifications', *ACM Transactions on Computer Systems* **4**(13), 365–398.
- Mellin, J. (1998), Supporting system level testing of applications by active real-time databases, in 'Proceedings 2nd International Workshop on Active, Real-Time, and Temporal Databases, ARTDB-97, number 1553 in LNCS. Springer-Verlag'.
- Michalewicz, Z. & Fogel, D. B. (1998), *How to solve it : Modern Heuristics*, Springer.

- Morasca, S. & Pezze, M. (1990), 'Using high level Petri-nets for testing concurrent and real-time systems', *Real-Time Systems: Theory and Applications* pp. 119–131. Amsterdam North-Holland.
- Nielsen, B. & Skou, A. (2000), Automated test generation from timed automata, in 'Proceedings of the 21st IEEE Real-Time Systems Symposium', IEEE, Walt Disney World, Orlando, Florida.
- Nielsen, B. & Skou, A. (2003), 'Automated test generation from timed automata', *International Journal on Software Tools for Technology Transfer (STTT)* 5(1), 59 – 77.
- Nilsson, K. (2005), Personal communication.
- Nilsson, R. (2000), Automated selective test case generation methods for real-time systems, Master's thesis, University of Skövde.
- Nilsson, R. (2003), Thesis proposal : Automated timeliness testing of dynamic real-time systems, Technical Report HS-IDA-TR-03-006, School of Humanities and Informatics, Univeristiy of Skövde.
- Nilsson, R., Andler, S. & Mellin, J. (2002), Towards a Framework for Automated Testing of Transaction-Based Real-Time Systems, in 'Proceedings of Eighth International Conference on Real-Time Computing Systems and Applications (RTCSA2002)', Tokyo, Japan, pp. 109–113.
- Nilsson, R. & Henriksson, D. (2005), Test case generation for flexible real-time control systems, in 'Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation', Catania, Italy, pp. 723–731.
- Nilsson, R., Offutt, J. & Andler, S. F. (2004), Mutation-based testing criteria for timeliness, in 'Proceedings of the 28th Annual Computer Software and Applications Conference (COMPSAC)', IEEE Computer Society, Hong Kong, pp. 306–312.
- Nilsson, R., Offutt, J. & Mellin, J. (2006), Test case generation for mutation-based testing of timeliness, in 'Proceedings of the 2nd International Workshop on Model Based Testing', pp. 102–121.
- Norström, C., A.Wall & Yi, W. (1999), Timed automata as task models for event-driven systems, in 'Proceedings of RTCSA'99', Hong Kong.

- Offutt, A. J., Xiong, Y. & Liu, S. (1999), Criteria for generating specification-based tests, in 'Proceedings of the fifth International Conference on Engineering of Complex Computer Systems', Las Vegas.
- Offutt, J., Alexander, R., Y.Wu, Xiao, Q. & Hutchinson, C. (2001), A fault model for sub-type inheritance and polymorphism, in 'Proceedings of the 12th International Symposium on Software Reliability Engineering', IEEE Computer Society Press, Hong Kong China, pp. 84–93.
- Offutt, J., Jin, Z. & Pan, J. (1999), 'The dynamic domain reduction approach to test data generation', *Software–Practice and Experience* **29**(2), 167–193.
- Offutt, J., Ma, Y.-S. & Kwon, Y.-R. (2006), The class-level mutants of mujava, in 'Workshop on Automation of Software Test (AST 2006)', Shanghai, China, pp. 78–84.
- Petitjean, E. & Fochal, H. (1999), A realistic architecture for timed testing, in 'Proc. of Fifth IEEE International Conference on Engineering of Complex Computer Systems', USA, Las Vegas.
- Petters, S. M. & Färber, G. (1999), Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible, in 'Proc. 6th Int'l Conference on Real-Time Computing, Systems and Applications (RTCSA'99)', Hong Kong.
- Pettersson, A. & Thane, H. (2003), Testing of multi-tasking real-time systems with critical sections, in 'Proceedings of Ninth International Conference on Real-Time Computing Systems and Applications (RTCSA'03)', Tainan city, Taiwan.
- Puschner, P. P. & Schedl, A. V. (1991), 'Computing maximum task execution times - a graph based approach'. Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.
- Ramamritham, K. (1995), The origin of time constraints, in M. Berndtsson & J. Hansson, eds, 'Proceedings of the First International Workshop on Active and Real-Time Database Systems (ARTDB 1995)', Springer, Skövde, Sweden, pp. 50–62.
- Raymond, P., Nicollin, X., Halbwachs, N. & Weber, D. (1998), Automatic testing of reactive systems, in 'Proceeding of the 19th IEEE Real-Time Systems Symposium (RTSS98)'.

- Rothermel, G. & Harrold, M. J. (1997), 'A safe, efficient regression test selection technique'.
- SanPietro, P., Morzenti, A. & Morasca, S. (2000), 'Generation of execution sequences for modular time critical systems', *IEEE Transactions on Software Engineering* **26**(2), 128–149.
- 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.
- Sha, L., Rajkumar, R. & Lehczy, J. P. (1990), 'Priority inheritance protocols: An approach to real-time synchronization', *IEEE Transactions on Computers* **9**(39), 1175–1185.
- Stankovic, J. A., Spuri, M., Ramamritham, K. & Buttazzo, G. C. (1998), *Deadline scheduling for real-time systems*, Kluwer academic publishers.
- Stankovic, J., Spuri, M., Di Natale, M. & Buttazzo, G. (1995), 'Implications of classical scheduling results for real-time systems', *IEEE Computer* **28**(6), 16–25.
- Thane, H. (2000), *Monitoring, Testing and Debugging of Distributed Real-Time Systems*, PhD thesis, Royal Institute of Technology. KTH, Stockholm, Sweden.
- Watkins, A., Berndt, D., Aebischer, K., Fisher, J. & Johnson, L. (2004), Breeding software test cases for complex systems, in 'HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9', IEEE Computer Society, Washington, DC, USA, p. 90303.3.
- Wegener, J., StHammer, H. H., Jones, B. F. & Eyres, D. E. (1997), 'Testing real-time systems using genetic algorithms', *Software Quality Journal* **6**(2), 127–135.
- Young, S. J. (1982), *Real-Time Languages: Design and Development*, Chichester: Ellis Horwood.

Demonstrator Task Code

This appendix describes the implementation of the different tasks that realize the robot application. The purpose of this is primarily to show examples of the application code to increase the understanding of the tested application and its properties. The current implementation of the robot application is small, consisting of approximately 1000 lines of c code. This allows short response-times and high sampling rates. The following subsections describe the implementation of each of the application tasks.

TrajectoryMgr Task

As outlined in section 9.2 the trajectory manager (TrajectoryMgr) is responsible for responding to requests from users or higher level control applications on other nodes. For example, another node can send a sequence of viapoints for moving the robot hand from position A to position B without smashing into an object or breaking some rigid object attached to the robot hand (such as the beam in the Flexcon application). The commands that can be issued to the trajectory manager task is;

- Add a trajectory point
- Adjust current trajectory
- Adjust a trajectory point

```
typedef
struct {
    SetPoint J1;    // Contains the required setpoint for joint 1
    SetPoint J2;
    SetPoint J3;
    double x;      // distance from robot base to hand in mm
    double y;      // height of the hand over ground in mm
    double angle;  // desired angle of the robot hand in radians
    int live;      // via-point is part of the current trajectory
    double psens;  // precision
    double asens;
} ViaPoint;
```

Figure A.1: Definition of the ViaPoint data structure

The trajectory points are defined by the data-structure listed in figure A.1. As seen in that figure, a “viapoint” store of both the coordinates in the plane of the robot hand, and the corresponding setpoints for the joint motor controllers. When a new via-point should be added to the trajectory the joint setpoints (angles) are calculated from the coordinates by the trajectory manager (using the inverse kinematics function in figure A.3). The system uses a circular buffer to keep track of the via points in a trajectory, the global data structure is protected using a priority inheritance enabled semaphore (handled by the LockResource and UnlockResource functions). The source code for the trajectory manager task is listed in figure A.2.

RefAdjuster Task

When the robot hand reaches within specified bounds of its current setpoint or after a new trajectory is calculated by the trajectory manager, this task becomes activated. This task consumes the available via points in the trajectory buffer and updates the setpoints for the controller tasks so that the robot hand is directed towards the next point along the trajectory. If the next via-point is farther away than a certain max-distance, an intermediate setpoint is calculated. This task also checks that the next coordinate in the buffer is correct and within the operational range of the robot arm. Figure A.4 contain a listing of the code performed by this task at each activation.

```

void SPT0_work(SPT0_data* TD,int MyID){
    LockResource(NETWORK,MyID); // Lock network for reading data
    SPT0_inputs IN;
    ReadNetworkBuffer(&IN);
    UnlockResource(NETWORK,MyID);
    LockResource(TRAJBUF,MyID); // lock trajectory-buffer
    if (IN.operation==ADD){ // Add coordinate
        ViaPoint tmp;
        tmp=InverseKinematics(IN.x,IN.y,IN.angle);
        // Calculate angles corresponding to coordinate
        tmp=CorrectCoordinate(tmp); // Check and move..
        Trajectory[TCurrentP]=tmp;
        Trajectory[TCurrentP].live=1;
        int next=TCurrentP+1;
        if (next>=CBUFFERSIZE) next=0;
        if (Trajectory[next].live==0){
            TCurrentP=next;
        } else {
            // commandbuffer overflow !! Perform error handling
        }
    } else if (IN.operation==ADJUSTALL){ // Move current Trajectory
        int i;
        for(i=0; i<CBUFFERSIZE;i++){
            ViaPoint TVP;
            TVP=Trajectory[i];
            if (TVP.live==1){
                ViaPoint tmp;
                tmp=InverseKinematics(TVP.x+IN.x,TVP.y+IN.y,TVP.angle+IN.angle);
                tmp=CorrectCoordinate(tmp);
                Trajectory[i]=tmp;
            }
        }
    } else if (IN.operation==ADJUSTONE) { //Move single Via-point;
        int Cindex=TCurrentP-IN.index;
        if (Cindex<0) Cindex=Cindex+CBUFFERSIZE;
        if (Cindex>=0 && Cindex<CBUFFERSIZE){
            if (Trajectory[Cindex].live==1){
                ViaPoint TVP=Trajectory[Cindex];
                ViaPoint tmp;
                tmp=InverseKinematics(TVP.x+IN.x,TVP.y+IN.y,TVP.angle+IN.angle);
                tmp=CorrectCoordinate(tmp);
                Trajectory[Cindex]=tmp;
            }
        }
    }
    UnlockResource(TRAJBUF,MyID);
}

```

Figure A.2: Source code for trajectory manager task

```

ViaPoint InverseKinematics(double x, double y, double angle){

    // Temp variables
    double B;
    double q1;
    double q2;
    ViaPoint T;
    double Angle1;
    double Angle2;
    double Angle3;

    B=(x*x) + (y*y);
    // uses pythagoras to calculate distance from robot
    // base to desired endpoint along a imaginary line b
    // Hence, B contains the length of line b squared
    q1 = atan2(y,x);
    // calculates angle between ground and line b
    // (atan2 is arc-tan + knowledge of quadrants)
    q2 = acos(((seglen1*seglen1) + B - (seglen2*seglen2))/(2*seglen1*sqrt(B)));
    // uses the cosine theorem to calculate angle between
    // line b and robot arm segment 1
    Angle1=q1+q2; // The angle between the ground and robot arm segment 1
    Angle2= acos(((seglen1*seglen1)+(seglen2*seglen2)-B)/(2*seglen1*seglen2));
    // uses the cosine theorem to calculate angle between Arm segment1
    // and Arm segment2
    Angle3=angle-Angle1-Angle2;
    // determine the angle of the last joint so that the
    // robot "hand" get the desired angle.
    T.J1.angle=Angle1;
    T.J2.angle=Angle2;
    T.J3.angle=Angle3;
    T.x=x;
    T.y=y;
    T.angle=angle;
    return T;
}

```

Figure A.3: Source code for simplified inverse kinematics

```

void SPT1_work(SPT1_data* TD,int MyID){
    ViaPoint TVP;
    TVP=Trajectory[TCurrentC];
    LockResource(TRAJBUF,MyID); // Lock Trajectory buffer

    if (isSame(&CurrSetPoint,&TD->CurrViaPoint)){
        // try to progress main counter
        int next=TCurrentC+1;
        if (next>=CBUFFERSIZE){
            next=0;
        }
        if (Trajectory[next].live==1){
            TCurrentC=next;
            TD->CurrViaPoint = Trajectory[TCurrentC];
        } else {
            // Ultimate target reached
        }
    }
    UnlockResource(TRAJBUF,MyID);
    LockResource(SETPOINT,MyID); // Lock Current SetPoint
    // check distance from here to next via-point
    // if more than max, interpolate help-point
    // else set curr-setpoints --> currViapoint
    double Deltax=CurrSetPoint.x-(TD->CurrViaPoint.x);
    double Deltay=CurrSetPoint.y-(TD->CurrViaPoint.y);
    double dist = sqrt((Deltax*Deltax)+(Deltay*Deltay));
    if (dist>MAXDIST){ // makes sure we don't get div/0 error
        // uses uniformity between triangles sides to split up
        // the maxdistance on x and y
        double newx=CurrSetPoint.x + ((-1* Deltax)/(dist/MAXDIST));
        double newy=CurrSetPoint.y + ((-1* Deltay)/(dist/MAXDIST));
        CurrSetPoint=InverseKinematics(newx,newy,CurrSetPoint.angle);
        CurrSetPoint=CorrectCoordinate(CurrSetPoint);
    } else {
        CurrSetPoint=TD->CurrViaPoint;
    }
    UnlockResource(SETPOINT,MyID); // unlocks Current SetPoint data-structure
}

```

Figure A.4: Source code for the RefAdjuster Task

ForceHandler Task

The ForceHandler task is assumed to be triggered when a change in the force applied to the tip of the robot hand is detected. The force values read by the force sensor are assumed to differ at each reading, due to imprecision of the hardware, hence, a sliding average is used to determine if the registered change is large enough to be reported to subscribers on the network. Figure A.5 lists the code that is exe-

```

void SPT3_work(SPT3_data* TD, int MyID){

    LockResource(HARDWARE, MyID); // lock hardware interface

    SPT3_inputs IN;
    ReadSPT3Data(&IN);

    if (IN.force>MAXF) IN.force=MAXF;
    if (IN.force<MINF) IN.force=MINF;
    int i;
    for(i=0;i<5;i++){
        TD->oldforce[i]=TD->oldforce[i+1];
    }
    oldforce[5]=IN.force;
    double omean=(TD->oldforce[0]+TD->oldforce[1]+TD->oldforce[2])/3;
    double nmean=(TD->oldforce[3]+TD->oldforce[4]+TD->oldforce[5])/3;

    double fchange=abs(nmean-omean);
    if (fchange>DELTA){
        LockResource(NETWORK, MyID);
        SendForceUpdate(fchange, omean, nmean);
        // Reports large changes in force read
        UnlockResource(NETWORK, MyID);
    }

    UnlockResource(HARDWARE, MyID);
}

```

Figure A.5: Source code for Force sensor handler

cuted for the force sensor handler.

The Controller Tasks

The controller tasks used in this prototype are of PID type, the source code of one such controller is listed in figure A.7. The controllers read the current angles of the joints and then adjust the motor torques so that the joint angle changes and the robot arm moves smoothly towards the desired position. The current setpoint is shared with the refadjuster task and the other controllers, and is protected by a semaphore. Furthermore, the hardware interface (access to I/O ports or memory mapped registers) is protected by another semaphore. The controller parameters need to be adjusted for the properties of the particular process. Figure A.6 lists the

```
void PT2_init(PT2_data* TD,int MyID){
    // Example controller parameters

    const double K = 50;        // Decide P part
    const double Ti = 0.25;     // Decide I part
    const double Td = 0.2;     // Decide D Part
    const double h = 0.1;      // Controllers assumed sampling period
    const double N = 10;       // Maximum derivative Gain

    const double Tr = 1;       // Tracking constant - for anti-windup
    const double beta = 1;     // Setpoint weighting

    // Pre-calculated coefficients
    TD->K = K;
    TD->bi = K*h/Ti;
    TD->ad = Td/(Td + N*h);
    TD->bd = K*Td*N/(Td + N*h);
    TD->tk = h/Tr;
    TD->beta = beta;

    // State variables
    TD->yold = Get_init_pos(3);
    TD->ui = 0;
    TD->ud = 0;
}
```

Figure A.6: A. Initialization of constants

initialization part of the controllers where controller parameters are set and constant expressions are precalculated.

Watchdog task

The watchdog task runs periodically to check the integrity of the control system. This is done by examining a set of time-stamps indicating when the controller tasks last read data from the hardware interface. If the controllers have omitted reading the sensor values within a certain time the watch-dog task performs a dummy read-operation to avoid an incorrect state of the hardware. In addition, the periodic controller tasks update individual time-stamps so that the watch-dog task can detect they are alive and operating. The watch-dog task also sends status messages to a monitoring node on the network so that cooperating nodes can detect system failures. Figure A.8 contains source code for this task.

```

void PT2_work(PT2_data* TD,int MyID){

    LockResource(HARDWARE,MyID); // Locks hardware
    PT2_inputs IN;
    ReadPT2Data(&IN);
    LastRead=PT2alive=GetTimeD(1); // Tell Watch-dog that we are alive

    LockResource(SETPOINT,MyID);
    // Locks Current setpoint structure
    double ysp;
    if (IN.ref==NOVAL){
        ysp = CurrSetPoint.J1.angle;
        // Gets the setpoint as assigned by the Ref-adjuster
    } else {
        ysp=IN.ref; // Gets the setpoint from ''manual mode''
    }
    if (ysp>YMAXJ3){
        ysp=YMAXJ3;
    } else if (ysp<YMINJ3) ysp=YMINJ3;

    double y = CurrPos.J3.angle = IN.y_actual;
    UpdatePosition(&CurrPos,&CurrSetPoint); // Performs Kinematics-update
    UnlockResource(SETPOINT,MyID);

    // Calculate control signal (u)
    double up = TD->K*(TD->beta * ysp - y);
    TD->ud = TD->ad*TD->ud - TD->bd*(y - TD->yold);
    // U = P + I + D
    double u = up + TD->ui + TD->ud;

    // Makes sure output signal is within specified bounds
    double usat;
    if (u>UMAXJ3){
        usat=UMAXJ3;
    } else if (u<UMINJ3){
        usat=UMINJ3;
    } else usat=u;

    WriteOutput(usat,0,0,MyID);
    UnlockResource(HARDWARE,MyID);
    TD->ui = TD->ui + TD->bi*(ysp - y) + (TD->tk* (ysp - y));
    // last term is added for (anti wind-up)tracking...
    TD->yold = y;
}

```

Figure A.7: Source code for PID-controller task

```
void PT3_work(PT3_data* TD,int MyID){

    LockResource(HARDWARE,MyID); //Locks hardware interface
    double Now = GetTimeD(1);
    double SinceR=(Now-LastRead);
    double SincePT0=(Now-PT0alive);
    double SincePT1=(Now-PT1alive);
    double SincePT2=(Now-PT2alive);
    int status=0;
    if ((LastRead!=0.0) && (SinceR>MAXPOLLTIME)){
        DummyReadHWinterface();
        LastRead=GetTime(1);
    }
    UnlockResouce(HARDWARE,MyID);

    if ((PT0alive!=0.0)&&(SincePT0>(2*PT0_PERIOD))){
        // Perform some local error-handling
        status+=ErrorHandling();
    }
    if ((PT1alive!=0.0)&&(SincePT1>(2*PT1_PERIOD))){
        // Perform some local error-handling
        status+=ErrorHandling();
    }
    if ((PT2alive!=0.0)&&(SincePT2>(2*PT2_PERIOD))){
        // Perform some local error-handling
        status+=ErrorHandling();
    }
    LockResource(NETWORK,MyID);
    // Send i'm alive message
    SendIMAlive(status);
    UnlockResource(NETWORK,MyID);
}
```

Figure A.8: Source code for Watchdog Task

Demonstrator Tasks Testing

This appendix contains the detailed description of how equivalence partitioning was done on each task to derive classes of input data. As previously mentioned, there are several methods for deriving this kind of input data. The method used for this experiment is a refinement of equivalence partitioning and pair-wise combination coverage (for example, compare with Ammann and Offutt (Ammann & Offutt 1994)), adapted for temporal unit testing.

TrajectoryMgr testing

The behavior of this task when invoked is dependent on the inputs read from the network and the state of the circular trajectory buffer at the beginning of the test. Three properties of these input data were suspected of influencing the temporal behavior of this task. The parameters and their associated classes of inputs are listed in table B.1. One of these parameters is the size of the coordinate buffer when a new coordinate is received. This parameter is denoted “Size of Buffer” in the table. Another parameter is the actual type of operation sent to the trajectory manager. This parameter is called “Operation” in the table and we distinguish between an adjust command for a coordinate at position 10 in the buffer and coordinate 99 in the buffer. The last parameter is termed “Coordinate” and refers to the structure of the actual coordinate that should be added to the trajectory. The variables N1 and N2 used in table B.1 are random positive real values between 200 and 1000. The values with dagger symbols denote values outside the assumed operational profile.

Size of buffer	Operation	Coordinate
0	Insert	(N1,N2,1.57)
15	Move All	(-N1,N2,0.0)
99	Adjust (10)	(N1, -N2, 3.15) †
100	Adjust (99)	(0,0,0.0)†

Table B.1: Equivalence partitioning of TrajectoryMgr

Test case	Size of buffer	Operation	Coordinate
1	0	Insert	(N,N,1.57)
2	0	Move All	(N, -N, 3.15)
3	0	Adjust(10)	(0,0,0.0)
4	0	Adjust (99)	(-N,N,0.0)
5	15	Insert	(0,0,0.0)
6	15	Move All	(-N,N,0.0)
7	15	Adjust(10)	(N,N,1.57)
8	15	Adjust(99)	(N, -N, 3.15)
9	99	Insert	(N, -N, 3.15)
10	99	Move All	(N,N,1.57)
11	99	Adjust(10)	(-N,N,0.0)
12	99	Adjust(99)	(0,0,0.0)
13	100	Insert	(-N,N,0.0)
14	100	Move All	(0,0,0.0)
15	100	Adjust(10)	(N, -N, 3.15)
16	100	Adjust(99)	(N,N,1.57)

Table B.2: Pair-wise combination of test parameters

Test Case	Execution pattern								
1	10,8	+T	4,9	-T	4,4	+S	13,7	-S	3,4
2	6,6	+T	4,6	-T	3	+S	50,1	-S	3,6
3	6,7	+T	4,7	-T	3,2	+S	1,9	-S	3,4
4	6,5	+T	4,7	-T	3,6	+S	1,9	-S	3,3
5	5,5	+T	4,3	-T	3	+S	11,8	-S	3,2
6	5,7	+T	4,7	-T	3	+S	10,3	-S	3,1
7	5,6	+T	4,4	-T	3	+S	5,1	-S	3
8	5,4	+T	4,2	-T	3	+S	1,9	-S	3
9	5,6	+T	4,2	-T	2,9	+S	9	-S	3
10	10,5	+T	5,5	-T	4,5	+S	399,9	-S	3,5
11	5,3	+T	4,3	-T	2,9	+S	5,1	-S	3
12	5,7	+T	5,3	-T	2,8	+S	1,9	-S	3,2
13	5,3	+T	4,3	-T	3,1	+S	9,1	-S	3,1
14	5,9	+T	4,3	-T	2,9	+S	346,7	-S	3,2
15	5,2	+T	4,4	-T	3	+S	5,2	-S	3
16	5,5	+T	4,3	-T	2,9	+S	1,9	-S	3

Table B.3: Max execution times measured for TrajectoryMgr

Each value for a parameter in table B.1 should be pair-wise combined with each value of the other parameters. Table B.2 lists an example test suite that provides pair-wise coverage of these parameter values. Table B.3 table list the worst case execution times recorded for each segment when executing the task with input data satisfying the test case descriptions in table B.2. In the execution pattern table, +X means that the task locks resource X, -X means that the task unlocks resource X.

Refadjuster testing

The behavior of this task depends on the contents of the trajectory buffer. The parameters in table B.4 have been identified to influence the execution pattern of this task. Here, Q1 and Q2 denote any coordinate in the first and second quadrant of the operational plane. The constant MAXD is the parameter deciding the maximum allowed distance between two Viapoints in the trajectory.

Size of buffer	Distance	From quadrant	To quadrant
15	$< MAXD$	Q1	Q1
90	$> MAXD$	Q2	Q2

Table B.4: Equivalence partitioning of RefAdjuster Task

Test case	Buffer size	Distance	From quadrant	To quadrant
1	15	$< MAXD$	Q2	Q2
2	15	$> MAXD$	Q1	Q1
3	90	$< MAXD$	Q1	Q1
4	15	$> MAXD$	Q1	Q2
5	15	$< MAXD$	Q2	Q1
6	90	$> MAXD$	Q2	Q2

Table B.5: Pair-wise combination of test parameters for RefAdjuster

Test Case	Execution pattern								
1	5,8	+S	3,8	-S	3,4	+R	2	-R	3,3
2	14,7	+S	6,7	-S	4,8	+R	1,9	-R	4
3	6,3	+S	3,5	-S	3,6	+R	1,9	-R	3,4
4	5,8	+S	3,5	-S	3,2	+R	1,9	-R	3,1
5	5,6	+S	3,4	-S	3,7	+R	1,9	-R	3,1
6	6	+S	3,6	-S	3,2	+R	1,9	-R	3,2

Table B.6: Max execution times measured for RefAdjuster

Burst of 6 sensor reads	Latest sensor read
1 change $> DELTAF$	inside valid range
3 cons. changes $> DELTAF$	outside valid range
no changes $> DELTAF$	

Table B.7: Equivalence partitioning of ForceHandler Task

Burst of 6 sensor reads	Latest sensor read
no changes $> DELTAF$	inside valid range
1 change $> DELTAF$	inside valid range
3 cons. changes $> DELTAF$	inside valid range
no changes $> DELTAF$	outside valid range
1 change $> DELTAF$	outside valid range
3 cons. changes $> DELTAF$	outside valid range

Table B.8: Pairwise test-data used for ForceHandler Task

ForceHandler testing

This task locks different resources depending on the relation between input values. This needs some special consideration during testing of timeliness, since the worst case behavior for each of the possible task execution patterns needs to be measured. From a unit testing perspective, the behavior of this task depends on the last sensor values read. Also, the sizes of the sensor values read may affect the execution pattern of the task. Table B.7 contains the equivalence partitioning used for testing this task. Table B.8 lists the test data that fulfill pair-wise coverage of these classes.

Since the execution patterns for this task are different depending on the input data, two different worst-case execution behavior tables can be extracted. These are listed in tables B.9 and B.10.

Controller tasks testing

The controller tasks have relatively simple control flows, but the robot arm behavior depend on the correctness and timeliness of the calculations performed by these tasks. The behavior of a controller task depends on the active setpoints and the current angle of the robot arm joints. The behavior also depends on the state variables calculated by the controller in the previous activation step. Since the Flexcon application can be executed in closed loop with a simulated robot arm in

Test Case	Execution pattern								
1	7	+Q	5,4	+T	4,1	-T	2,7	-Q	3,1
2	6,4	+Q	5,6	+T	4,3	-T	2,6	-Q	3
3	6,4	+Q	5,5	+T	4,2	-T	2,6	-Q	3
4	6,5	+Q	5,6	+T	4,2	-T	2,5	-Q	3
5	6,6	+Q	5,4	+T	4,3	-T	2,7	-Q	2,9
6	6,6	+Q	5,5	+T	4,3	-T	2,7	-Q	2,9

Table B.9: Max execution times measured for ForceHandler, execution pattern A

Test Case	Execution pattern				
1	2,9	+Q	2,9	-Q	3,4
2	-	-	-	-	-
3	-	-	-	-	-
4	3	+Q	2,9	-Q	3,3
5	-	-	-	-	-
6	2,8	+Q	2,8	-Q	3,4

Table B.10: Max execution times measured for ForceHandler, execution pattern B

Matlab/simulink this was used for deriving sequences of low-level input data for the higher level commands. Table B.11 shows the parameters varied to produce such sequences of task input data. The combinations needed to obtain pair-wise coverage across these categories are shown in table B.12 and the results from measurements are revealed in tables B.13, B.14 and B.15.

Watchdog testing

In the Watchdog task, there are dependencies from the execution order of the system back to the behavior of the task. This kind of dependencies must be specially considered during testing of timeliness.

Setpoint size	setpoint position	difference
valid size	same quadrant	large
invalid size	other quadrant	small

Table B.11: Equivalence partitioning for Controller tasks

Setpoint size	setpoint position	difference
valid size	same quadrant	large
invalid size	same quadrant	small
invalid size	other quadrant	large
valid size	other quadrant	small

Table B.12: Pairwise test-data used for controller tasks

Test Case	Execution pattern								
1	11,5	+Q	8	+R	12,3	-R	6,7	-Q	3,5
2	8,5	+Q	6,2	+R	10,2	-R	6,4	-Q	3
3	8,5	+Q	5,7	+R	10,6	-R	6,6	-Q	3,2
4	8,6	+Q	5,5	+R	10,5	-R	6,3	-Q	3

Table B.13: Max execution times measured for Controller 1

Test Case	Execution pattern								
1	12,2	+Q	12,4	+R	13,1	-R	14,6	-Q	11,9
2	11,4	+Q	12,4	+R	13,9	-R	13,8	-Q	3,1
3	11,2	+Q	12,6	+R	14,4	-R	13,7	-Q	3
4	11,5	+Q	6,5	+R	13,8	-R	13	-Q	3

Table B.14: Max execution times measured for Controller 2

Test Case	Execution pattern								
1	9,5	+Q	7,8	+R	10,6	-R	7,6	-Q	3,3
2	6,8	+Q	5,3	+R	8,1	-R	5,2	-Q	2,9
3	9,6	+Q	7,7	+R	10,2	-R	7,7	-Q	3,4
4	7,1	+Q	5,7	+R	7,9	-R	5,8	-Q	3

Table B.15: Max execution times measured for Controller 3

Task	Execution pattern								
WatchDog	7,5	+Q	4,5	-Q	3,9	+T	4,1	-T	3,1

Table B.16: Max execution times measured for Watchdog

In particular, the task's execution behavior (control flow) depends on the relation between a set of shared time stamps and the current time. That means that the execution time depend on the actual order and timing of other tasks. These variables cannot be controlled when all tasks are run concurrently without introducing probe effects. Optimally, this kind of relation between execution order and execution time should be included in the model used for test case generation. In the mutation-based testing framework, this addition can be implemented by a specialized execution segment as described in section 10.1.2. Since the impact is small and the dependencies are limited to a single task for this application, we do not model these dependencies in this case study.

For the measurements, the shared time stamps were set so that the program path with the intuitively longest execution time would occur when the task was run in singularity (see table B.16).

Case-Study Test Execution Results

This appendix contains figures that visualize the results from the experiment described in section 9.4.2. In particular, figure C.1 shows the average number of test cases that resulted in timeliness failures when five different test suites fulfilling the ($\Delta = 20$) execution time testing criteria were run on 7 software variants with seeded timeliness errors. Each test suite contained 7 test cases, therefore this is the maximum number of effective test cases (max of the z-axis). In figure C.2 and figure C.3 the most and least effective test suite for each software variant are plotted in the same way.

For comparison, five test suites with 10 randomly generated test cases were run on each of the seven software variants. In the same way, the average number of effective test cases are plotted in figure C.4. The result from the most effective random test suites for each version is plotted in figure C.6 and the least effective random test suites are plotted in figure C.5.

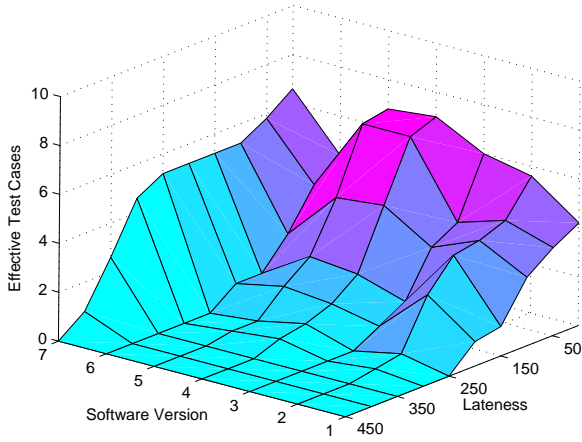


Figure C.1: Average number of failures detected in seeded variants using mutation testing

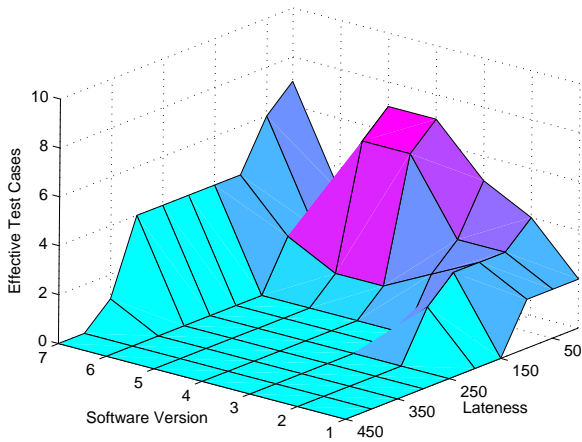


Figure C.2: Least number of failures detected in seeded variants using mutation testing

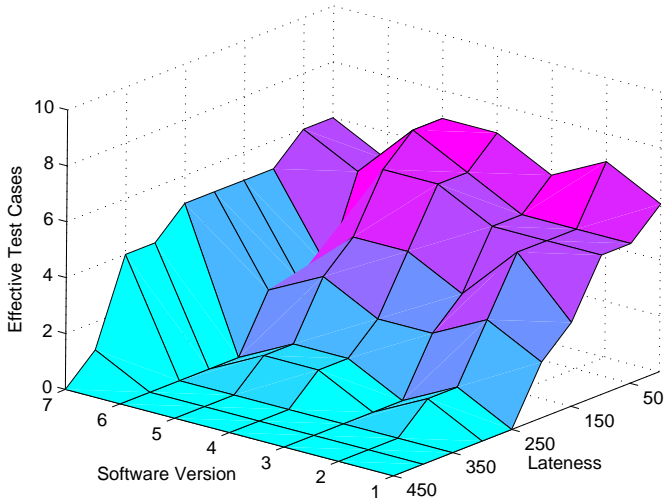


Figure C.3: Most number of failures detected in seeded variants using mutation testing

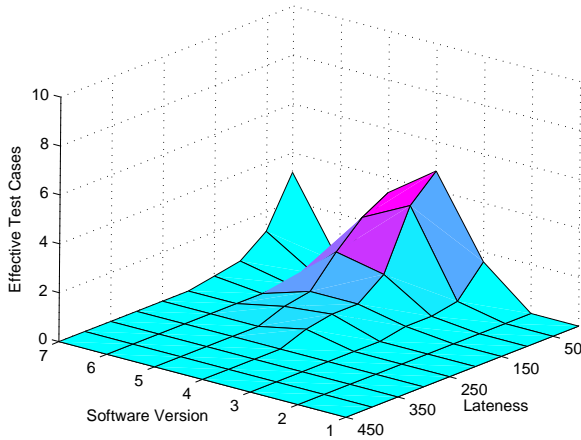


Figure C.4: Average number of failures detected in seeded variants using random testing

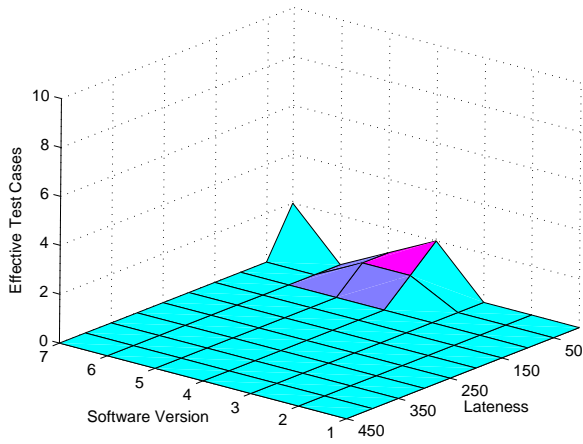


Figure C.5: Least number of failures detected in seeded variants using random testing

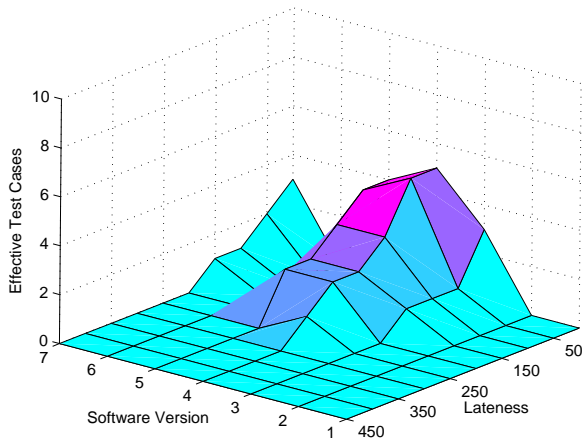


Figure C.6: Most number of failures detected in seeded variants using random testing

Mutation Operators Overview

This appendix contains a classification (see figure D.1) of the mutation operators defined in chapter 6 as well as an overview of how related mutation operators can be implemented using the guidelines in section 10.1.2.

Entity/Relation			Temporal attribute			Mutation type			Mutation operators	
Task x_i	Task x_j	Resource r_m	Exec. time	Critical section	Input constraints	Add	Remove	Manipulate	Defined	Extension
X						X				Add task
X							X			Remove task
X			X					start		Add exec. segment
X			X					end	Δ +execution time, Δ +execution time	
X					X	X				System specific
X					X		X			System specific
X					X			Offset	Δ +pattern offset, Δ - pattern offset	
X					X			Interarrival- time	Δ +interarrival- time, Δ -interarrival- time	
X		X		X		X				Add exec. segment
X		X		X			X			Lock omission + Unlock omission
X		X		X				start	Δ +hold time shift, Δ -hold time shift, Δ +lock time, Δ - lock time, Lock omission	
X		X		X				end	Δ +hold time shift, Δ -hold time shift, Δ +unlock time, Δ -unlock time, Unlock omission	
X	X					X			Precedence constraint +	
X	X						X		Precedence constraint -	
X	X							Relative priority		Trivial
		X				X				Add exec. segments
		X					X			Lock omission + Unlock
		X						Ceiling value		Mutate exec. env. property

Figure D.1: Classification of defined mutation operators and related extensions

Dissertations

Linköping Studies in Science and Technology

- No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.
- No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.
- No 18 **Mats Cedwall:** Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.
- No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.
- No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.
- No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.
- No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.
- No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.
- No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.
- No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.
- No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.
- No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.
- No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.
- No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.
- No 109 **Peter Fritzon:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.
- No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.
- No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.
- No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.
- No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.
- No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.
- No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.
- No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.
- No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.
- No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.
- No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.
- No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.
- No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.
- No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.
- No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.
- No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.
- No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.
- No 270 **Ralph Rönquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.
- No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.
- No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.
- No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.
- No 281 **Christer Bäckström:** Computational Complexity

- 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 Lambrich:** 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 Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.
- No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmönster 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 Bjärelund:** 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-supported 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 förändring i samband med införande av informa-

- tionsystem, 2003, ISBN 91-7373-618-X.
- No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.
- No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.
- No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.
- No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.
- No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Empirical Study in Software Engineering, 2003, ISBN 91-7373-779-8.
- No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.
- No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.
- No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.
- No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.
- No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.
- No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.
- No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.
- No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5
- No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004, ISBN 91-7373-966-9.
- No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.
- No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modelling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.
- No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.
- No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.
- No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.
- No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.
- No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.
- No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.
- No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.
- No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.
- No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.
- No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.
- No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.
- No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.
- No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005, ISBN 91-85457-07-8.
- No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.
- No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.
- No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.
- No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.
- No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.
- No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.
- No 1005 **Aleksandra Tesanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.
- No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.
- No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.
- No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.
- No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.
- No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8.

- 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 1023 **Sonia Sangari:** Some Visual Correlates to Focal Accent in Swedish, 2006, ISBN 91-85523-67-4.
- No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

polisarbete, 2005, ISBN 91-85299-43-X.

Linköping Studies in Information Science

- No 1 **Karin Axelsson:** Metodisk systemstrukturerings- att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.
- No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.
- No 3 **Anders Avdic:** Användare och utvecklare - om utveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.
- No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000. ISBN 91-7219-811-7.
- No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X
- No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.
- No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.
- No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.
- No 9 **Karin Hedström:** Spår av datoriseringens värden - Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.
- No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.
- No 11 **Fredrik Karlsson:** Method Configuration - method and computerized tool support, 2005, ISBN 91-85297-48-8.
- No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.
- No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för