Mälardalen University Licentiate Thesis
No.15

Analysis of Execution Behavior for Testing of Multi-Tasking Real-Time Systems

Anders Pettersson

October 2003

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

ISBN 91-88834-13-1
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

An important issue in software testing is the ability to observe the execution of the software; this is especially true for *real-time systems* (RTS). RTS are difficult to observe, and the ability to test RTS is inherently low. Embedded RTS have few interfaces for observation and the execution of multi-tasking RTS is usually non-deterministic. As a consequence, testing of RTS cannot be exercised with existing tools for sequential programs. New tools and methods are necessary that enable observation of the system despite few interfaces while at the same time address the non-determinism issue.

The contribution in this thesis is three-folded: (1) we present a tool suite that allows deterministic testing of multi-tasking RTS, in which synchronization of tasks is resolved *off-line* or *on-line*. (2) We show by building a test bed how to use the tool suite. (3) We present the design and functionality of *Asterix the Real-Time Kernel*.

In (1) we propose an analysis tool that derives all possible system level control-flow paths of multi-tasking RTS in which synchronization between communicating tasks are resolved on-line by using the *Priority Ceiling Emulation Protocol* (PCEP; also know as the Immediate Inheritance Protocol). The analysis tool is an extension of an existing tool in which synchronization were resolved off-line by using release time offsets or priorities to separate the tasks in time.

Based on the number of derived control-flow paths test coverage criteria are defined, and estimations of test effort can be done early in the development of a system. In (2) we show how the defined test coverage criteria relate to the number of traversed control-flow paths during test execution. We also show how the estimation of tasks' execution times affects the analysis. The analysis tool is applied on multi-tasking RTS in which the tasks are synchronized off-line. The real-time applications are then exercised on the test bed using Asterix as the operating system.

In (3) we present a small-sized real-time kernel named Asterix that has support for software based instrumentation of kernel events as well as application usage of system calls. The major problem of software instrumentation is the change in execution behavior that occurs when a RTS is executed without or without the probes. In Asterix we avoid this probe-effect by leaving the probes in the kernel during normal operation.

Also, we present a literature survey covering the state-of-the-art in the field real-time systems testing.

To Andreas and Emelie

Preface

This licentiate thesis presents the results from my first two and a half years of graduate studies. My studies began as an undergraduate student in 1996 at the department of computer science and engineering, Mälardalen University. I received my Master of Science in computer engineering in August 2000, and started my graduate studies in December 2000, also at Mälardalen University.

I have been surrounded by many helpful colleagues here at the university especially those who sits next to my office space and my supervisors that have given me the opportunity to study here at Mälardalen University. My supervisor Henrik Thane that have guided me during these years and have ensured that the right conditions to do the work have been there. My main supervisor Hans Hansson for giving guidance and courses during this period.

Despite the risk of forgetting someone I will mention a few colleagues that have helped me during this study period: Daniel Sundmark for introducing me to the noble art of disc golf and for interesting discussions, Joel Huselius for being a good study companion and for being the master of beer brewing, Thomas Nolte for good arrangement of social activities and being a good traveling companion, Jonas Neander for being a soul mate, Dag Nyström for good laughs on our traveling around the world. Since I probably forgot to mention someone I have to say: I wish to thank everyone of the staff at the department of computer engineering at Mälardalen University for being friendly, nice and helpful.

This thesis is dedicated to my children, Andreas Pettersson and Emelie Petterson, for making the life worth to live and to my parents, Anna-Lisa Pettersson and Jean Pettersson, and my sister, Susanna Pettersson, for all kinds of support during this period.

This work is funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for strategic research.

Contents

List of Publications

The following articles are included in this licentiate[1] thesis:

A *Testing of Computer Software with Temporal Constraints: A State-of-The-Art Report*,
Anders Pettersson,
Mälardalen Real-Time Research Centre Report ISSN 1404-3041 ISRN MDH-MRTC-115/2003-1-SE, Mälardalen University, October 2003.

B *Testing of Multi-Tasking Real-Time Systems With Critical Sections*,
Anders Pettersson and Henrik Thane,
In Proceedings of 9th International Conference on Real-Time and Embedded Computing Systems and Applications RTCSA'03, Tainan City, Taiwan, R.O.C, 18-20 February 2003. Springer-Verlag.

C *The Asterix Real-Time Kernel*,
Henrik Thane, Anders Pettersson and Daniel Sundmark,
In Proceedings of 13th Euromicro International Conference on Real-Time Systems, Industrial Session, Technical University of Delft, Delft, The Netherlands, June 2001. IEEE Computer Society.

D *Experimental Evaluation of a Test Procedure for Deterministic Testing of Real-Time Systems*,
Anders Pettersson and Henrik Thane,
Mälardalen Real-Time Research Centre Report ISSN 1404-3041 ISRN MDH-MRTC-114/2003-1-SE, Mälardalen University, October 2003.

[1] A licentiate degree is a Swedish graduate degree halfway between MSc and PhD.

Besides the above papers, I have co-authored the following papers not included in this thesis:

I *Integration Testing of Fixed Priority Scheduled Real-Time Systems*
Henrik Thane, Anders Pettersson and Hans Hansson
In Proceedings of IEEE Real-Time Embedded System Workshop, London, UK, December 2001. Technical Report, Department of Computer Science, University of York. Editors Ian Bate and Steve Liu.

II *Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies*
Daniel Sundmark, Henrik Thane, Joel Huselius and Anders Pettersson
Mälardalen Real-Time Research Centre Report ISSN 1404-3041 ISRN MDH-MRTC-96/2003-1-SE, Mälardalen University, April 2003.

III *Replay Debugging of Real-Time Systems Using Time Machines*,
Henrik Thane, Daniel Sundmark, Joel Huselis and Anders Pettersson,
In Proceedings of Parallel and Distributed Systems: Testing and Debugging (PADTAD), Nice, France, April 2003, ACM.

Chapter 1

Thesis

This licentiate thesis is the result of research done at Mälardalen University in the TATOO project, which has received funding from the ARTES program. The goal of this project is to develop novel methods and tools for testing of distributed real-time systems.

This thesis has two main contributions (1) an extension of an analysis tool for deriving the system level control flow of multi-tasking real-time system (paper B in chapter 3) and (2) an experimental case-study of previous results in the TATOO project (paper D in chapter 5).

1.1 Background

In computer software projects the deployment is often delayed and the cost is often exceeding the budget. As a result, project tasks that are performed late in the development are almost always either reduced or ignored. Software testing is one of these tasks that often are neglected because of cost and time, and it gets even worse in projects developing more complex software such as software for real-time systems. Therefore, is it essential that developers have tools that make testing less costly and less time consuming.

For sequential programs there exist a number of analysis and testing tools, and the research on tools for testing of concurrent programs are increasing. However, for *real-time systems* there are still few research results regarding testing tools and methods.

1.1.1 Real-time Systems

A *real-time system* is a computer system in which not only the output from the system is important but also the time at which the output is delivered. That is, real-time systems differs from sequential programs and concurrent programs in the sense that the correctness of the produced output is not only dependent on the functional correctness but also on the temporal correctness [2].

Real-time systems are often *multi-tasking*. A multi-tasking program is divided into two ore more tasks that in cooperation with each other fulfill the requirements of the specification, i.e. the tasks in the system are executing concurrently.

There are two types of real-time systems, *hard* and *soft* real-time systems. In hard real-time systems timing constraint violations lead to fatal consequences. Fatal consequences can be large economical losses or human injuries. In soft real-time systems occasional violations of timing constraints can be accepted.

Each type of real-time system can be further grouped into *event triggered* or *time triggered* systems. Event triggered real-time systems are driven by events, usually external interrupts or message passing. An event can occur at an arbitrary point in time. Time triggered real-time systems are driven by a periodically invoked task scheduler that is responsible for deciding which task is next to run.

1.1.2 Testing

The purpose of testing is to reveal faults and to establish confidence in that the program will not fail during its operation. Testing consists of three steps:

- Test planning
- Extract test cases that are likely to reveal most failures
- Apply the test cases to the program in a test execution

Test Planning

The task of testing starts by setting up a test strategy, defining test coverage criteria, modeling the execution behavior, etc. The model can serve as a specification of the execution behavior of the program. Execution behavior can for example be the order in which the program statements are invoked, the

produced output from program calculations or in which order tasks in multi-tasking programs synchronize with each other.

Test Case Generation

Test cases can be generated from either the specification of the program or the implementation of the program.

Specification based test-case generation derives the test cases from the specification; hence the software can be tested early in the development even before the implementation is completed.

Implementation based test-case generation requires knowledge of the implementation details. Since the specification must have been realized in an implementation, testing with implementation-based test cases is done later in the development.

Test Execution

The correctness of a program can only be established according to what have been observed during test executions. Hence, it is essential that we can observe the produced output, intermediate results and the paths traversed by the program.

Observation of execution behaviors can be done by using software probes [3], hardware instrumentation [5], or a combination of software and hardware instrumentation.

1.2 Problem Formulation

The complexity of real-time systems can be expressed in terms of the number of possible valid execution paths that are traversed during the operation of the system. Complexity caused by the indeterminacy in the interaction between the environment and the program makes exhaustive testing impossible. The complexity increases even more when tasks in the system interact with each other, i.e. interprocess communication. Hence, out of all generated test cases only a very small subset of test cases is chosen.

The problem we consider in this thesis is how to analyze the complexity of multi-tasking real-time systems in which synchronization of communicating tasks are resolved dynamically during run-time or by adding offsets to the release time of tasks.

Complexity of real-time systems is also affected by non-determinism in the execution of real-time systems. The non-determinism can be caused by races between concurrently executing tasks competing for shared resources, jitter in the execution time of the tasks, and insertion of software probes.

Programs that during test runs operates correctly with inserted softwares probes may fail to operate during normal operation with the probes removed (the probe effect [1]). This is caused by different execution behavior in the program with and without the probes. Also, changed execution behavior in consecutive runs can result from significant variations in the execution time and race situations.

In many real-time systems only a few execution paths are exercised leaving the majority of the execution paths untested. In this thesis we consider the problem of complexity estimations of real-time systems at design time, i.e. before the specification is implemented.

We propose a tool for analysis of execution behavior of multi-tasking real-time systems with critical sections. The output from the analysis tool is a graph representing possible execution paths from which test cases can be derived. This work is an extension of an existing tool for analysis of the complexity of real-time systems [3]. Also a case study using the original analysis tool [4] is presented.

1.3 Publications and Technical Contributions

There are four publications included in this thesis. Below, each publication is summarized and the contribution is stated for each publication.

Paper A:

Testing of Computer Software with Temporal Constraints: A State-of-The-Art Report, Anders Pettersson.

This state-of-the-art report (SotA) is a literature survey of computer software testing, focusing on testing of real-time systems. The SotA provides the necessary background of software testing for the discussions in this thesis.

Paper B:

Testing of Multi-Tasking Real-Time Systems With Critical Sections, Anders Pettersson and Henrik Thane.

When tasks in real-time systems communicate with each other there is a need for synchronization in order to preserve precedence constraints and avoid conflicts. Since the order of the synchronization is controlled by the implementation, and since it is likely that the programmer mistakenly introduces synchronization errors in the implementation, there is a need for testing methods that test applications for such errors.

The contribution in this paper is a method that allows testing of multi-tasking real-time systems with critical sections. The method is an extension of an existing testing technique for distributed real-time systems in which the synchronization was resolved off-line by using offset to the tasks release times in order to separate the tasks in time [4].

Anders' contribution in this paper is an extension of an existing analysis tool that derives the system level control flow of multi-tasking real-time systems. The extension allows the analysis tool to handle synchronization of tasks during run-time. Both authors have contributed to the solution through discussions.

Paper C:

The Asterix Real-Time Kernel, Henrik Thane, Anders Pettersson and Daniel Sundmark.

One of the issues in testing and debugging of multi-tasking programs is the ability to observe the behavior of the program during run-time.

In this paper we propose a micro-kernel, Asterix, for real-time systems with support for run-time instrumentation of programs. The purpose of Asterix is to have a real-time kernel with small memory footprint, bringing state-of-the-art theories into state-of-the-practice, and to provide a low cost (both economically and computationally) software-based instrumentation for observation of the systems run-time behavior.

Anders' contribution is the design and implementation of Asterix. All authors have contributed in discussions regarding the design, implementation and supported functionality in Asterix.

Notes:

- This paper has been revised and edited so there are slight differences between the published version and the version found in this thesis.
- The Asterix Real Time Kernel, has been used as a foundation for all the empirical validation of our research theories.

Paper D:

Experimental Evaluation of a Test Procedure for Deterministic Testing of Real-Time Systems, Anders Pettersson and Henrik Thane.

In practice, testing of real-time systems is often a non-trivial task because of the non-determinism caused by the run-time environment. The contribution of this paper is to evaluate an analysis tool used in deterministic testing of real-time systems [3]. The analysis tool derives the system level control flow of multi-tasking real-time systems. The validation will be done by running real-time applications on real hardware. The task sets are generated by applying random values on the temporal attributes of the tasks. The results show that the analysis tool is suitable for deriving test cases, although exhaustive testing is difficult to achieve and that the accuracy of execution time estimations is important.

Anders' contributions in this technical report are the following: design and implementation of tools and servers in the test bed, performing the evaluation and enhancement of the analysis tool.

1.4 Summary and Future Work

Testing is often neglected because of time limits and costs. Therefore, it is a necessity that the real-time system can be analyzed early in the development. Today there exists numerous tools for analyzing sequential programs and the numbers of tools for analyzing concurrent programs are increasing. However, there exist very few tools for analysis of real-time systems. There are even fewer tools that consider essential constructs of real-time systems such as synchronization of tasks and invocation and termination of tasks. In this thesis, an analysis tool for multi-tasking real-time systems is presented. The analysis tool can be applied on single node real-time systems where communicating task are synchronized through the priority ceiling emulation protocol, but should be extended to handle synchronizations of tasks in distributed real-time systems.

The execution behavior of real-time systems is non-deterministic. Therefore, in practice testing is sometimes based on trial and error methods because of lack of control over the test execution. In this thesis, an evaluation of a method for deterministic testing of real-time systems is presented. The goal of the evaluation is to show how to predict the execution behavior during testing of real-time systems. The evaluation is done on single node real-time systems in which synchronization of tasks is resolved off-line. The next step would be

to evaluate the method for real-time systems with critical sections, i.e. systems in which synchronization of tasks is resolved during run-time.

Bibliography

[1] J. Gait. A probe effect in concurrent programs. In *Software - Practice and Experience*, volume 16(3), pages 225–233, Mars 1986.

[2] W. Schütz. Fundamentals issues in testing distributed real-time systems. In *Real-Time Systems*, volume 7, pages 129–157, Boston, 1994. Kluwer Academic Publisher.

[3] H. Thane. Monitoring, testing and debugging of distributed real-time systems. In *Doctoral Thesis*, Royal Institute of Technology, KTH, S100 44 Stockholm, Sweden, May 2000. Mechatronic Laboratory, Department of Machine Design.

[4] H. Thane and H. Hansson. Towards systematic testing of distributed real-time systems. In *Proceedings of The 20th IEEE Real-Time Systems Symposium*, pages 360–369, 1999.

[5] J. J. P. Tsai, Y.-D. Bi, and R. Smith. A noninterference monitoring and replay mechanism for real-time systems. In *IEEE TRansaction on Software Engineering*, volume 16, pages 897–916, 1990.

Chapter 2

Paper A:Testing of Computer Software with Temporal Constraints: A State-of-The-Art Report

Anders Pettersson
Department of Computer Science and Engineering
Mälardalen University, Västerås, Sweden
anders.pettersson@mdh.se

Sammanfattning

Förekomsten av datorer i de konsumentprodukter vi använder dagligen ökar hela tiden. Många av dessa datorer styrs av programvara. För att garantera att produkten är användbar måste programvaran testas. Tyvärr är denna testning ofta åsidosatt på grund av att testning är resurskrävande och kostsam. Ett sätt att underlätta testningen är att tillhandahålla verktyg och metoder som reducerar arbetsinsatsen för utvecklare av programvara vid testning. Denna översikt av litteratur inom testning av programvara har till syfte att peka på existerande metoder och verktyg för testing av programvara, speciellt i datorsystem där det finns krav på att tidsbeteendet inte strider mot specifikationen, så kallade *realtidssystem*.

Abstract

Computers in consumers product are increasing. Many of these computers are controlled by software. To ensure that the consumer can use the product as expected the software must be tested. Unfortunately, testing is often neglected because it is costly and resource demanding. One solution to this is to make test tools and test methods available to the developers of software. The purpose of this report is to point out methods and tools for testing of computer software, especially for software that have constraints on their temporal behavior, i.e., real-time systems.

2.1 Introduction

In our daily life we are more and more dependent on computers and their software. When we travel by airplane, use robots at work, or even watch TV at home, we expect them not to malfunction. Therefore, it is important that the software does what the user expects and that it does not fail.

To establish the quality of the software *Validation* and *Verification* are used. Validation is used to establish that the software supplies the service specified in the requirements. Verification is used to establish that the properties of supplied services are correct according to the requirements in their specifications. Verification can be done by statically analyzing the software or analyzing the software dynamically by executing the program, i.e., *testing*.

Based on the execution behavior, computer software can be categorized into three domains:

- ***Sequential programs***, which are programs that runs from invocation to termination without interruptions or interleaving.
- ***Concurrent programs***, which are programs that execute within the same time interval either by interleaved or simultaneous execution.
- ***Real-time systems***, which are programs where the correctness depends on the functional behavior as well as the temporal behavior.

For these domains, the objective of testing is to find deviations between the specified requirements and the observed results during operation of the software.

Testing is a necessity in development of correct software. However, testing is not trivial even if it seems to be. For example, assume a computer program that takes one input from a user and the user is supposed to press only one key but mistakenly press two keys simultaneously. If it is crucial for the functionality that only one key is pressed at a time, all possible two-key presses must be tested in order to establish the correctness.

In the example above, the software should be tested with all combinations (n-key presses) under all circumstances to ensure that the program is free from defects. But the amount of tests then rapidly grows to be enormous, and so are also the costs for the testing. Consequently, exhaustive testing is in most cases not possible.

In this state-of-the-art report we will discuss software testing and how different test methods can be applied on different types of software: sequential programs, concurrent programs and real-time systems. The focus will be on

testing of real-time systems. But we will also discuss testing of non-real-time software to give an introduction to software testing in general.

2.1.1 Outline

The outline of the rest of this report is as follows: In Section 2.2 we discuss the fundamentals of testing. Testing of sequential programs is discussed in Section 2.3. In Section 2.4 we will discuss *regression testing*, i.e., how to test software after the code is modified or needs a retest. In Section 2.5 we will discuss testing of concurrent programs. In Section 2.6 testing of real-time software is discussed, focusing mainly on functional testing of real-time systems.

2.1.2 Terminology

There exist several standards for the terminology used when discussing computer software testing, both international and national, for example, the IEEE standard, the SIS standard and the ISO standard. In this report a terminology that conforms to IEEE STD 610.12-1990 [26] will be used. Below, we give the terminology that is used in this report.

Correctness By correctness of the software it is meant that the behavior of the program execution conforms to the behavior specified in the program specification.

Regression Testing Selective retest of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [26].

Software, Application and Program In this report *software*, *application* and *program* are all an executable computer file that delivers services according to a specified behavior. Although, *software* can also be the documentation and source code of the program, this will not be the interpretation used here.

Task Each individual *task* can bee seen as a small sequential program and is the smallest user defined execution unit. Two or more tasks can, by communicating with each other, form a more complex program and solve more complex problems than an individual task.

Test Test is an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspects of the system or component [26].

Testing Testing is the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspects of the system or component [26].

Threads and Processes In this report we do not distinguish between tasks, threads and processes. However, in general there is a significant difference between them, but the difference do not affect the assumptions in our discussions, and hence we will here use task to denote all three.

Validation Is the process of evaluating a system or a component during or at the end of the development process to determine whether it satisfies specified requirements [26], i.e., validation aims at answering the question *are we building the right system*?

Verification Is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [26], i.e., verification aims at answering the question *are we building the system right*?

2.2 Computer Software Testing

The objective of testing is to reveal failures to eliminate the faults in the software, and thereby increase the confidence in the software. This is done by applying test data to the software. But this raises several issues, such as how to select test data, how to measure the progress of testing and when to stop the testing.

The test data (*test cases*) must be selected to be sufficient to satisfy the requirements, i.e., the test data adequacy. According to Zhu et al. [48], one way to categorize test data adequacy is to base the classification on the source of information for deriving test cases: *white-box testing* (implementation based) and specification-based *black-box testing* (specification-based). Test cases generated using the black-box approach are based on the specification and used for functional testing and interface testing during *integration testing* and *system testing*. It is also used for performance testing, stress testing, and reliability testing. The white-box (or *glass-box*) approach is based on knowledge of the

implementation and is used during unit testing in order to establish to what extent the software is tested.

Testing approaches can be divided into: *coverage-based testing* (structural testing), *fault-based testing* and *error-based testing*. Coverage-based testing methods can further be divided into: control-flow based and data-flow based testing.

Both control-flow and data-flow structural testing are often based on a flow-graph model of the structure of the program. The model is derived by statically analyzing the software either by the compiler or an analysis tool.

In fault-based testing it may not be sufficient to select test data to meet some coverage criterion, but also chose test data based on to what extent the test is expexted to reveal a failure. Based on the approach to reveal failures, testing methods can be divided into: *fault seeding*, *mutation-based* and *fault injecting*.

Fault seeding testing is to intentionally add faults that are known to reveal a failure. If m faults are seeded and n faults are found, then based on n and m an estimation of the remaining non-seeded faults can be made.

Mutation testing is to create a set of mutated programs based on an original program. Each of the mutated programs is expected to reveal a single failure. If the failure is revealed the test is then later used to test the original program. When all mutated programs are tested an optimal set of test inputs can be determined.

Fault injection evaluates the impact of changing the code or the state of the software. This is done by using perturbation to change the code and observe the result by instrumentation. Fault injection is mostly used to test the reliability of the software.

Testing can also be used (1) to establish the level of confidence that the program will not fail during its operation and (2) to establish that specified properties are satisfied. In contrast to fault-revealing tests this is done by applying test cases to demonstrate the absence of faults. That is, a successful test case does not reveal failures.

To be successful in testing there must be guidance for when to test, how to test, what to test and what tools to use for testing, i.e., there is a need for a test plan.

2.2.1 Planning for Testing

Test Plan

A test plan is the documentation of the conditions and requirements that must be set for testing. The documentation can be formal or informal but it is important that there are no ambiguous requirements. One way to achieve unambiguous documentation is to use mathematics [28].

A well-defined test plan should include, at least, well documented requirements in a specification, strategies for initial testing, integration testing and system testing. According to Leung et al. [22] the test plan must also include:

- A strategy for regression testing.
- A guideline for the test procedure, including a test design strategy, coverage criteria and information on how to handle test cases that do not need to be re-executed.
- Information for identification of test classes, test case execution order, and changes made to the software.

However this covers the general case. For real-time systems, especially safety-critical systems, it is often the case that all test cases are exercised in a retest. Then we do not have to have strategies for selecting which test-cases to execute.

In Bertolino et al. [4] the authors present an approach for deriving test plans for integration testing from a formal description based on software architectures. The purpose of the derived test plan is to describe the components of the software and the connections between these components.

Rational Unified Process (RUP) is a software development tool that enforces creation of test plans divided into well-defined phases during the life cycle of the software. RUP also encourages developers to start testing the software as early as possible by performing inspections on documents such as design specification and functional requirements. It has been shown that early inspections of source code and documentation can reveal 80% of specification and programmer faults [8]. By using RUP and inspections in the early phases of the development, test efforts are reduced in the later phases.

Fault Hypothesis

The fault hypothesis is the definition of what a failing behavior is according to the specification of the software [3]. What a failing behavior is depends on the current failure mode of the system. In Clarke et al. [7] a classification of

different failure modes of sequential programs are defined: *control failures*, *value failures*, *addressing failures*, *termination failures* and *input failures*.

For concurrent programs, in addition to the failure modes above the following failures must be considered: *ordering failures*, *synchronization errors* and *interleaving failures*.

In [19] the propagation of a programmers mistake or an erroneous output leading to a failure is defined as $Fault \rightarrow Error \rightarrow Failure$. However, according to the IEEE STD [26] the words fault and error are used interchangeably. The above definition of fault and error is used in the fault tolerance discipline. Consequently, with the same meaning as above the definition $Error \rightarrow Fault \rightarrow Failure$ can also be found in the literature.

Test Cases

There must also be specified input sequences for the test execution. These input sequences are called *test cases*; a *test suite* is a collection of test cases.

For sequential programs a test case is often the input parameter to the program and the expected output from the program. Whereas, in concurrent programs the test cases are an input parameter, output parameter and some specified behavior of the system. For example, if the testing strategy is to find errors with respect to in which order the task are synchronizing, then the input would consists of the input parameter to the program and a valid synchronization sequence [6].

Initial Test Case Selection

In the initial testing, the first test cases can be created based on a specification (black-box). Later on when the specification is implemented, the set of test cases can be extended with structural-based test cases (white-box).

Rothermel et al. [30] define a test case as $< identifier, input, output >$ in order to achieve maintainability and storage of test cases in a database. As complement to a test case definition, a test history must often be maintained together with scripts for test case execution. A test history is helpful when revalidating the test cases in a re-test of modified programs. Scripts for test case execution are helpful in larger software projects, if the number of test cases are too many to handle manually or when tests are exercised during non-working time [27].

Test Case Selection for Re-testing

Leung et al. [22] propose how to categorize test cases into different classes. These classes are: reusable test cases, re-testable test cases and obsolete test cases.

- Reusable test cases are testing the unmodified parts of the specification and the program constructs. Re-execution of these test cases is hopefully not necessary since they produce the same output as the previous tests.
- Re-testable test cases are testing the program constructs that are modified, although the specification is not modified.
- Obsolete test cases are test cases that are no longer relevant because of that input/output relations no longer are valid, the program has been modified so the test case no longer test the program construct, or the test case no longer contribute to the structural coverage.

There is also a need to distinguish re-testable test case from obsolete test cases. This introduces two new classes of test cases.

- New-structural test cases
- New-specification test cases

Wong et al. [44] propose a method that produces a *complete* but not *precise* set of test cases. A complete set of test cases contains only the test cases that should be used for re-validating the inherited functionality from the previous version of the program. Precise sets of test cases do not include test cases where the previous version and the new version produce the same output. They discuss the cost of to being too ambitious in the effort to get a complete and precise subset of regression tests. Their proposed method concentrate on the *flexibility* of the test selection.

2.2.2 Analysis of Computer Software Execution Behavior

Analysis of programs are based on the test data adequacy criteria; specification-based and implementation-based criteria. For sequential programs, the test coverage can be determined by analyzing the code at unit-level. By the use of the coverage-based approach the following type of code coverage can be achieved:

- in control-flow based testing
  - *all-node*, *all-branch* and *all-paths*
- in data-flow based testing
  - *all defs coverage*, *all p-uses coverage*, *all c-uses coverage*, *all c-uses/some p-uses coverage*, *all p-uses/some c-uses coverage*, *all uses coverage*, *all du-paths coverage*

The test coverage criteria can be used to (1) determine when we have tested the software enough and (2) when to stop testing.

Because of the interleaved execution in concurrent programs and real-time systems, the analysis is more complex than analysis of sequential programs, and implies the use a programming constructs to synchronize the task in order to avoid conflicts.

A common approach to derive serializations is to statically analyze the structure of the implementation [39]. Serializations can also be derived dynamically by instrumentation of the exercising program [35]. However, in both approaches the competition for shared resources and synchronizations of tasks must be considered. Using synchronization constructs, such as, semaphore protocols or rendezvous in ADA, can do this. There are issues that should be considered when using such approach, for example, uniqueness of input, possible exposure of errors during test execution, and infeasible concurrent program serializations [41].

For real-time systems a common cause for interleaved execution is that the tick scheduler schedules a higher prioritized task, or using programming constructs that put the program on hold and lets other programs run.

If synchronization constructs are used, then some of the serializations become infeasible. For example, if two tasks, task *A* and task *B* in a concurrent program have precedence constraints such that a read operation in task *A* must be exercised before a write operation in task *B*. Then, all serializations in which the write operation of task *B* is exercised before the read operation in task *A* are invalid serializations.

Also for multi-tasking programs two consecutive executions with the same input may have different execution behavior and even produce different output [5]. Hence, making it impossible to test the program.

```
Program A( void )        Program B( void )
{                        {
  read  x;                 read  z;
  write x;                 write z;
}                        }
```

Program order	Possible serializations
A - B	read x; write x; read z; write z;
B - A	read z; write z; read x; write x;

Figure 2.1: Example of possible serializations of the execution behavior of two sequential programs.

2.2.3 Execution Behavior

What is the execution behavior of a program? It can for example be output values, signals, or the statements traversed in the executions [46]. The behavior of a program can be based on synchronization sequences, rendezvous sequences, and execution paths [46, 37].

Execution paths define in which order the statements in a program are traversed. For sequential programs the execution path of the statements is exercised in the order of the implementation and in which order the programs are invoked, see figure 2.1.

For concurrent programs and multi-tasking real-time systems the complexity of deriving the serializations is increased. In Figure 2.2 it is shown how the interleaving can affect the traversed execution paths.

There are two types of execution characteristics of real-time systems *multi-tasking* and *single-task* real-time systems. Multi-tasking systems can further be of two types *pre-emptive* and *non pre-emptive*. Single task and non pre-emptive real-time systems have similarities with execution characteristics of sequential programs since the task in such systems are exercised in sequence without interruption. Multi-tasking and pre-emptive real-time systems have the same fundamental execution characteristic as concurrent programs.

Synchronization

Multi-tasking programs may have requirements that restrict the order of interleaving between programs; such requirements may be due to data dependencies

```
Task A( void )          Task B( void )
{                          {
  read  x;                   read  z;
  write x;                   write z;
}                          }
```

Program order	Possible serializations
A - B	read x; write x; read z; write z;
B - A	read z; write z; read x; write x;
A - B - A	read x; read z; write z; write x;
B - A - B	read z; read x; write x; write z;
A - B - A - B	read x; read z; write x; write z;
B - A - B - A	read z; read x; write z; write x;

Figure 2.2: Example of possible serializations of the execution behavior of two tasks in a concurrent program.

between programs. Without these constraints *race situations* can occur.

A race situation is when two or more tasks are competing for limited resources and it is not possible to á priori determine which of the tasks that is going to win the competition. Example of limited resources can be CPU, I/O ports, and shared variables. In figure 2.3 a race situation is visualized by exemplifying access to a shared resource X by two tasks, task A and task B. Initially X is assigned the value 1. The two possible orderings and results of the computations are that task A starts to executes because of earlier release time, folloed either by task B preempting task A before A have completed its operation on X (see figure 2.3 (a)) computing the result of $X = 32$, or in the case that A perform its operation on X before task B preempts A (see figure 2.3 (b)) the produced output is then $X = 17$.

Observability

Observability is the ability to observe the state before and after an operation. Consequently, it must be possible to observe the input, output and the internal state.

Observing the input and output in sequential programs is straightforward, that is if the program does not include any non-deterministic statements [33].

The inputs are observed to determine the behavior of the program's envi-

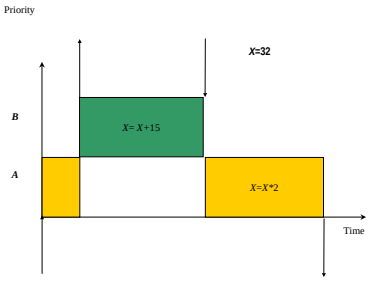

Figure 2.3 (a): Example 1

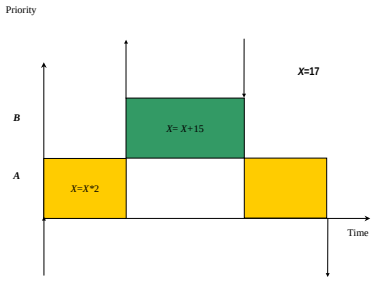

Figure 2.3 (b): Example 2

Figure 2.3: Example of two possible execution orderings from repeated executions of two tasks A and B accessing the shared resource X initialized to 1. In figure 2.3 (a) task B precedes task A and in figure 2.3 (b) task A precedes task B

ronment. By observing the internal state, the exact cause of the failure can be located, and internal state changes that have no effect on the output can be detected. The internal states of sequential programs are observed using interactive debuggers or printouts in the code. One of the problems of using interactive debuggers and auxiliary output to observe real-time systems is that the temporal behavior are changed during the observation [36], i.e., even if we can stop the program and observe the state, the time cannot be stopped in the environment.

Observations can be done in different ways; visually by looking at the screen or printouts, or by using instrumentation. We assume here that all observations are achieved by instrumentation.

There are three approaches for monitoring the state of software: hardware based, software based and a combination of hardware and software. Observations achieved by inserting monitoring probes into the code and then removing the probes during normal operation could affect the behavior of the execution. This phenomenon is called the “probe effect” [9].

In [33] three techniques to handle the probe effect are discussed, the probe effect can be ignored, minimized or avoided. When observing concurrent programs and real-time systems we must avoid the probe effect, that is the software used for monitoring must remain in the application or non-intrusive hardware must be used. Another problem that occur when using monitors in real-time systems is that temporal delays are introduced leading to longer response times.

Yann-Hang et al. [20] propose a tool suite for testing real-time ADA ap-

plications. The tool suite includes an instrumentation tool implemented as an ADA run-time library. Output generated from the analysis and the instrumentation are flow graphs and trace files that are used to determine the code coverage criteria of the ADA-program. The analysis tool can handle different kinds of coverage criteria, e.g., basic blocks coverage, c-use coverage, and p-use coverage. However, in their paper the proposed test analysis do not consider the temporal behavior of the application.

Determinism

Executions of sequential programs are repeatable and deterministic. That is, for an input we get the same output regardless of how many times we run the program with that input. This is true if the program does not include any statements that depend on the temporal behavior and/or random behavior. Examples of such statements are random statements or dependencies of a clock readings in sequential programs [32].

In concurrent programs, each task is executed independently and therefore it is often impossible to determine which execution path the program follows each time we run the program. That is, for a unique input we can get different output for several consecutive runs.

Sang et al. [6] achieve deterministic testing by controlling in which order the programs synchronize for accesses to shared resources. In this case, in addition to the input to the program a synchronization sequence that is derived from the specification must be added. Between the forced synchronization points the programs run nondeterministically, and the nondeterministic execution is then used to check for nondeterminacy conformance between the specification and the implementation. Running the program nondeterministically tests behavioral conformance, and during each execution of the program the synchronization sequences are logged. The logged synchronization sequences are then analyzed to see if the behavioral conformance is satisfied. Sang et al. emphasize that in nondeterministic execution not only valid synchronization sequences are executed but also invalid synchronization sequences.

Controllability

Controllability is the ability to force the program into a desired state. For sequential programs it is sufficient to give the input to the program and set a break-point at the desired program statement to achieve controllability. For multi-tasking programs controllability is achieved at a coarser scale than for

sequential programs. Here synchronization sequences are derived by statically analyzing the concurrent program and then forcing the program to traverse the same trajectory as the derived synchronization sequence.

Reproducibility

Reproducibility – test repeatability – is the ability to reproduce a previous execution of a program. In other words, for a given input the system always computes the same output in repeated runs of the system [26].

After errors have been corrected the tester wants to assure that the error have been removed and that no new errors have been introduced. Therefore it is necessary to test the system repeatedly. During repeated test runs with the same test cases, the same outputs must be observed in order to determine if the software is correct [26]. If test executions are not reproducible re-testing cannot determine that corrections have removed the errors. For concurrent programs and real-time systems, in which *races* have impact on the execution path, the program is not usually reproducible.

To reproduce the exact execution behavior of a sequential program it is sufficient to run the program repeatedly with the same input. In order to reproduce the execution behavior of multi-tasking programs it is not sufficient to repeatedly feed the same input to guarantee the same output. This is because of the race situations that can occur when programs concurrently accesses shared resources. For real-time systems it is not sufficient to consider only the ordering of the accesses, in addition the time at which the access occurred must also be considered [33]. Other causes for making RTS non-reproducible can be non-determinism in hardware, communication protocols, network traffics, etc. [25], and reading of real-time and random numbers [33].

There are two approaches to reproducibility (or test repeatability): the *language based* and the *implementation based* approaches.

The language based approach transforms a program into a new program that includes program constructs that constrains the execution in order to force the control of the execution. Carver et al. [5] propose a tool for transformation of concurrent programs. Based on the language used and what synchronization constructs that are available in the language, e.g., ADA-rendezvous or monitors, the tool creates a new program that forces the execution to follow a derived synchronization sequence of the concurrent program.

The implementation based approach requires an event history. The behavior of a program is logged during run-time in a history log. The information in the history log is then used to reproduce the behavior of the execution. In

Thane [36] an implementation based method for creating a history log and reproducing the behavior of real-time systems by deterministic replay is introduced.

Testability

The IEEE standard [26] defines testability as the ability to create test cases that satisfies the test criteria. An extended definition that not only includes the metrics of creating test cases but also consider the probability of revealing a failure during testing is proposed by Voas et al. [40]. They also propose approaches for analyzing the software to measure the testability. One important issue is to determine the parts of the code that are most likely to hide faults. This analysis is based on *information loss* in the data; *explicit* and *implicit* information loss.

Explicit information loss is when computations of data are not observed during test execution. Hence, explicit information loss can only be found by static analysis of the implementation. This makes analysis to be performed possible only late in the development, since the implementation must have been completed. The most frequent cause of explicit information loss is the hiding of internal information. However, information hiding is often used in well structured programming approaches to prevent unintended tampering with internal data of software modules. Explicit information loss is a design issue and can be solved by designing the software not to hide internal information.

Implicit information loss is when different data are fed as input but when the same data is presented as result. There exists a correlation between the cardinality of the input and cardinality of the output, called domain/range ratio (DRR). If software has high DRR it is considered to have low testability. Solutions to reduce implicit information loss include isolating implicit information loss using specification decomposition, minimizing variable reuse and increasing the number of out parameters. The benefit of analysis for implicit information loss is that it can be performed early in the development. Based on the above assumption Voas et al. propose an analysis method that measures the probability of software failure [40].

Testability analysis and testing complements each other in that the testability analysis can give guidance on where in the code testing efforts should be spent.

2.3 Testing of Sequential Programs

In most software projects the testing phase often stands for up to 50% of the development cost. Mainly because the testing process often involves manual tasks, and that expensive test equipment often is needed and that these resources are limited and shared between testers. Other causes that increase the cost of testing are: the difficulty to create test cases, a huge number of test cases, the need for re-tests, the time to execute each test case, etc.

Testing of computer software can be divided into four phases *modeling the software's environment*, *selecting test scenarios*, *running and evaluating test scenarios* and *measuring test progress* [43]. The test execution can further be divided into three sub-phases *unit testing*, *integration testing* and *system testing*.

2.3.1 Unit Testing

A unit can be a function, a collection of functions, a task, a collection of tasks, etc. Rarely a unit is a whole program unless the program is very small.

Unit testing is often performed by the programmer. The programmer compiles the unit on the development platform and feeds the input manually or by a test program. However, this technique cannot reveal failures that may occur during execution in the program's real envronment.

There have been several structural testing methods proposed such as statement coverage, branch coverage and path coverage. To determine paths and coverage, often a control-flow graph that represents the structure is used [48].

Functional testing techniques aims to test that the output from the function correlates to the given input and is correct with respect to the requirements. The functional test also aims to assure that the interface of functions is correct and is properly used. There are several approaches for generation of inputs for unit tests, for example *boundary value tests*, *random tests* or *statistical tests*.

2.3.2 Integration Testing

Integration testing is the phase when the units are integrated with each other and tested. Approaches for integration testing are *incremental*, *top down*, *bottom up* or the *big-bang* approach. Incremental integration testing is to stepwise integrate the program unit for unit. Top down integration testing is to integrate the program by starting with the main unit and then integrate the units as they are called from the units above in the hierarchy. Bottom up approach is the

opposite to top down approach, the ingration is started from the units that is in the lowest level of the call hierarchy. In both the top down and the bottom up approach it can be neccessary to use stubs, dummy units, for those units, which are not yet subject for testing. The big bang apprach for integration testing means that all units functionality are implemented and then all units are intgrated at the same time.

2.3.3 System Testing

When integration testing have been performed, system testing is performed in the programs real environment, with realistic scenarios of inputs, outputs and the load of the system.

Despite that there exists several phases the different types of testing are not isolated activities; testing is an iterative process. For example, system testing can be done several times in a project because we have subsystems that will be put together into the final system, and during maintenance faults are corrected and new functionality are added or removed.

2.4 Regression Testing

Regression testing strategies can be of two types. Either software can be re-tested with all test cases (*re-test all*) or with a subset of the test cases (*selective regression test*). Selective regression testing can be to select enough test cases to reveal all failures, minimal number of test cases or select test cases that only traverse the modified paths of a program. Retesting a software with a subset of test cases can reduce the cost of testing the software, and is therefore the most common approach in academic papers. Onoma et al. [27] discuss approaches for regression test selection. In their paper a framework is presented, the multilevel regression testing framework, that developers can use for regression testing during development and maintenance. They emphasize the difference between the academic and industrial view of what is important issues in regression testing:

> "*While researchers are mostly concerned with reducing the number of test cases for re-testing, there are other important issues in using regression testing in an industrial environment.*" [27]

One issue is that although the re-test all strategy is costly and time consuming, it is not always desirable to find a subset of test cases. Especially for

those companies that must use retest-all method because of certain constraints such as safety-critical programs, etc [27]. Examples of other issues can be the use of tools for automation when regression testing are used extensively and frequently. A drawback of regression testing is that the suite of test cases increases when the software is maintained and this makes testing even more time consuming.

Leung et al. [22] have identified two types of regression testing *corrective* and *progressive*. Progressive regression testing is caused by modification of both code and specification, whereas corrective regression testing only comprise code modification.

When using regression testing selection techniques the basic concept is to test only the modified parts of the program, but this can lead to undisclosed failures since not all test cases that possibly reveals failures are re-executed. There have been extensive research on regression testing techniques and most of them address the regression selection problem [1, 12, 13, 29, 31, 42, 44]. Many of the algorithms aim to select test cases were the new and the old version of the program differs in output. Others are concentrated to achieve certain degree of coverage. Wong et al. [44] propose a technique that use both of these approaches. The proposed approach is based on two techniques: *minimization* and *test case prioritization*.

A definition of regression testing problems is found in Rothermel et al. [30]. They define four problems and describe how to proceed when exercise regression testing: *Let P be a procedure or program, let P' be a modified version of P and let T be a test suit for P. A typical regression test proceeds as follows:*

1. *Select $T' \subseteq T$, a set of test cases to execute on P.*
2. *Test P' with T', establishing P''s correctness with respect to T'.*
3. *If necessary, create T'', a set of new functional or structural test cases for P'.*
4. *Test P' with T'', establishing P''s correctness with respect to T''.*
5. *Create T''', a new test suit and test history for P', from T', T'', and T'''.*

The first step is the regression test selection problem. Execution of test cases are addressed in steps 2 and 4. In step 3 the problem of how to select test case to get enough coverage is defined. While step 5 address the problem of maintenance of a test suit.

Leung et al. [22] divides the regression testing problem into two subproblems: the test selection problem (1) and the test plan update problem (2). In

selection of test cases to test a modified program, select test cases from an existing test plan, and create new test cases based on made modifications. For (2), see section 2.2.1, where the update problem is included.

Most of the existing regression test techniques are using structural based test cases (white box testing). This because most of them compare the structure of the old program with the new program, and then the tester tries to get the same coverage percentage as the previous test.

According to Kim et al. [15] most regression testing techniques concentrate on the test selection problem, ignoring other important issues of regression testing. Equal important to the selection problem is the issue of what triggers the regression testing event. Should tests be executed periodically or at some pre-determined instance, for example, after all changes, after modification of critical components, or during final testing. The hypothesis for their work is based on the fact that the amount of changes between regression testing sessions affect the cost effectiveness, since the test suit grows for each modification leading to that more test cases are selected in every session. The impact of this is that the effort to select failure revealing test cases also increase and makes selection algorithms less cost effective.

2.4.1 General Regression Test Assumptions

Many regression testing strategies assume that the program is sequential; regression testing is performed at unit level, and there exists a well designed specification, well designed test plan, and control- and/or data-flow graph with a single entry and exit point. Leung et al. [22] discuss assumptions regarding characteristic of programs such as:

- the program is single entry and single exit
- the program is not too simple
- the program is not too complex

Functions can easily be adapted to the first assumption by inserting a start and end node in a control-flow graph. White et al. [42] propose a method where the control-flow is modeled as a call graph, and here the single entry and exit assumption is not an issue since a call graph is represented as a tree and the call chain starts and ends in the root node of the tree.

The second and third assumptions are more difficult to overcome. Here are cost/benefit trade-offs important and should be considered. If the program is

too simple then the set of test cases may be small and the cost for selecting test cases is more than re-execute all the tests.

Leung [21] have examined the fundamentals of selective regression testing when divided into two strategies: *selective regression unit testing* and *selective regression function testing*. Where function testing addresses integration and system test. Unit testing refers to strategies where structural coverage criteria must be met. For unit testing Leung discuss *proper scope assumptions*. The approach is to choose a scope such as that all faults in a modified program are revealed. This can however lead to unnecessary testing if the scope is large. They claim the problem is to design a selective regression unit testing strategy to minimize the scope and code to re-test.

Cost Models

Onoma et al. [27] talk about costs in regression testing, but they only consider the cost of the testers and not the cost of machine time. The cost is approximated by the time spent on *developing test cases*, *re-validating*, *execution of the test suit*, *comparing the results*, and *fault identification*. They emphasize that if the cost for analyzing which test cases to select for a subset of the previous set of test cases exceeds the cost of running the unselected test cases, then the retest-all method is more cost-effective than a selection technique.

Leung et al. [23] discuss a cost model that compares regression testing strategies, and in particular selective regression test strategies. The cost can be of two types: *direct* and *indirect*. Direct costs are the cost for resources used during test, such as test analyst´s time and equipment for executing the test. Indirect costs are the cost for development of tools, management and database storage. The model takes into account *system analysis cost*, *test selection cost*, *test execution cost* and *result analysis cost*.

2.4.2 Regression Test Techniques

There exists numerous regression test selection techniques [15, 30], for example:

- *retest-all* techniques
- *random/ad-hook* techniques
- *minimization* techniques
- *safe* techniques

- *data-flow/coverage* techniques
- *prioritization* techniques

In the retest-all method, as the name indicates, all the previous test cases are used in the regression test phase. This can however be acceptable if the program is small; the number of re-tests is small. For larger software projects and where regression testing is used frequently rerun of all test cases is not acceptable. If we have to consider the cost of running test cases or the amount of test cases is too large we can use a regression test selection techniques that select a subset of the previous set of test cases.

By randomly selecting a subset of test cases there are no guarantees that those test cases that need to be rerun are in the subset. Testers with á priori knowledge of the system can see to that if some test cases are missing these test cases can be added to the test suit.

In minimization techniques the goal is to select a minimal subset of test cases, where each test case corresponds to the impact of the modification in the program. The method selects at least one test case that execute every modified or added statement.

Safe techniques selects test cases necessary to reveal all faults in the modified program. This can lead to that in some cases the retest-all and safe techniques selects the same set of test cases. According to Kim et al., on average the safe method selects 68% of the test cases [15]. If the safe method selects all test cases, then it is less effective than the retest-all technique since in safe method an analysis is done to determine what modifications have been made. By using regression testing techniques that select *potentially revealing test* [29] no á priori knowledge is needed as in techniques that concentrate less on coverage criteria. Rothermel et al. [29] assumes that four criteria must hold for their algorithm to select a safe subset of test cases: *safety*, *precision*, *efficiency* and *generality*.

Coverage techniques aim to select only those test cases that traverse paths were the program have been changed. The coverage can lead to that some test cases are not selected, although they should have because they traverse possibly affected parts of the program.

With a prioritization technique, test cases can be exercised in such an order that those test cases that reveal failures early in the testing process or get confidence and coverage criteria increased at a faster rate [31] are re-executed first. It depends on the application what should be concerned when choosing prioritization criteria. Wong et al. [44] propose that test cases are prioritized by cost per additional coverage to reveal failures early.

Algorithms for Regression Test Selection

- Slicing Algorithm
- Incremental Algorithm
- Firewall Algorithm (Adapted firewall Algorithm/Firewall at Integration Algorithm)

Slicing Algorithm

Gupta et al. [13] propose a selective regression testing technique using a data flow based slicing algorithm that satisfies the *all-uses* test criteria. By using a slicing algorithm the authors aims to remove the need of maintenance of a test suit. The test suit can be omitted since all *def-use pairs* are explicitly detected. A def-use pair is a pair of a definition and the use of the variable. Since the def-use pairs must be computed there is a need for data flow information. This information can be represented by nodes in a control flow graph. A def-use pair can be either *computation uses* (C-uses) or *predicate uses* (P-uses). C-uses that occurs in computation statements and P-uses in conditional statements.

Def-use pairs affected by program changes can be divided into two categories:

- *Directly affected*
- *Indirectly affected*

Directly affected def-use pairs are program changes due to insertion or deletion of variable uses and definitions. For example, if a statement is changed from $Var = 1$ to $Var = 1 + VarZ$ then new def-use pairs must be created and the use of $VarZ$ must be tested.

Indirectly affected def-use pairs can be of two types:

(I) A program is edited, for example $Var = 1$ is changed to $Var = -1$, and no new def-use pairs are created but the change affects the value for computation in a def-use pair. The def-use pairs that use this new value must be re-tested.

(II) A def-use pair that test a condition path and is affected by a program change, for example if $Var < 0$ is changed to $Var > 0$, then the def-use pairs must be re-tested.

To identify all def-use pairs that are affected by program changes the authors use a slicing algorithm. The program must be modeled as a control flow graph (CFG) where each node represent a program statement and each edge represent a path between two statements. A *backward* and *forward* walk algorithm are applied on the CFG, these algorithms identifies all def-use pairs. Backward walk algorithm identifies definitions of variables by traversing the CFG in a backward direction from the use of the variables until it have found all definitions and their corresponding paths.

Incremental Regression Testing

As most regression testing algorithms, the incremental regression testing algorithm proposed by Agrawal et al. [1] assumes that the program can be represented as a control-flow graph (CFG) extended with information of the data-flow. The algorithms is built on a simple model of changes:

(1) to fix faults.

(2) the specification is changed.

In (1) all test cases that produce an incorrect output in the previous test must be rerun to verify that the output after the changes is correct. In (2) all test-cases that are considered to be incorrect according to the changed specification must be rerun even if they produced a correct output in the previous test.

The authors propose three methods for incremental regression testing:

- The execution slice technique.
- The dynamic slice technique.
- The relevant slice technique.

The methods are based on four observations, and these observations have been verified through experiments that reveal how many statements that are executed under each test case.

The methods can be applied on changes that hold for the following assumptions:

- No changes are made to the CFG.
- No changes are made to the left-hand-side of an assignment.

The execution slice technique strategy is to off-line determine the set of statements executed under each test case. Then, when retesting the program, only those test cases with sets of statements containing a modified statement need to be rerun. The execution slice technique can also be used to test a program at unit and function level. This makes it suitable even for larger software projects or when the test strategy is black-box testing. This is done by not considering which statement that are executed, instead the method can determine which module that are executed under a test case.

In some cases a modification to a statement leads to that all test cases are selected. This happens when for example the predicate of a conditional statement is changed but the conditional block does not affect the output. The problem of selecting all test cases in some cases are solved by using a dynamic program slice technique. The method determines in which test cases the modified statement affects the output. However the method can not determine what type of modification that is made. The problem with this is that if the changes introduces new faults then the faulty change is not detected since relevant test cases are not rerun.

The proposed solution identifies potential dependencies of variables in an execution history. The relevant slice technique determines potential dependencies of a variable if in a path of the execution history no definitions of the variable can be found between a predicate and the use of the variable, and there exist a definition of the variable in another path. Both paths start in the predicate and end in the computation that use the potentially dependent variable.

The authors give an example when the relevant slicing technique have deficiencies. This can happen when a use of a variable is control dependent of a previous predicate. This can lead to unnecessary rerun of test cases. They solve this by excluding the statements that have control dependencies to a predicate that may affect the output from the set of statements.

Firewall concept for Regression Testing

As the previous methods the *firewall* concept proposed by White et al. [42] requires a model of control-flow which is modeled as a call graph (CG). The CG shows the control-flow at module level. There are three basic assumption for the firewall approach. All module dependencies must be modeled in the CG, there are no other errors than those caused by the modified modules, and the unit and integration test must be reliable.

The firewall is a boundary such as that the firewall comprise the functions that need to be modified. The main idea is to aim for that after a modification

the number of modules inside the firewall is not increased.

Besides the CG there must also exist a module/test matrix that dynamically obtain which modules that each test case tests. The matrix also includes which modules that calls a module. These calls are mapped to the call graph. A firewall is applied to the call graph and contains the set of modified modules that need to be re-tested. Then a subset is selected from the set of test cases bounded by the fire-wall and determined by the module/test matrix.

Even though the paper aims at integration regression testing the authors discuss the importance of using regression testing in all phases of a program's life-cycle, and that the sooner a regression error is found the less are the testing costs. Since the dependencies are computed from a call graph there is no information about the internal structure of the modules. In other words, the method is a black-box regression testing strategy.

2.5 Testing of Concurrent Programs

Race situations and nondeterministic execution behavior increase the complexity of concurrent programs. This leads to two major problems when testing concurrent or multi-tasking programs: the ability to observe and control the execution of concurrent programs.

Methods

In Carver et al. [5] the authors propose a method that by synchronization constructs force an execution to follow a derived synchronization sequence. What synchronization constructs to choose is determined by the programming language. Examples of such synchronization construct can be semaphores, monitors, or rendezvous in the programming language ADA. The method is called *deterministic execution testing*. During an execution, information of the synchronization sequence is logged and when the implementation is re-tested not only the input is feed to the program but also the previous logged synchronization sequence. The program is forced to exercise the synchronization sequence. This is done by constructing a new version of the original program. The new program is constructed by a tool that add synchronization constructs, supported by the language that is used, that guarantees that the program follows the derived synchronization sequence.

During the program execution of the guaranteed synchronization sequence debugging information can be logged at re-execution of the program. The de-

bugging information can then be used to assure that the error have been corrected. To make sure that the correction have not introduced new bugs not only the test that revealed the error must be re-executed, also previous test must be re-executed.

Sang Chung et al. [6] propose a method that use synchronization sequences derived from message sequence charts. However, in both Carver et al. and Sang Chung et al. they focus on concurrent program testing and do not consider the temporal behavior of the program. Sang Chung et al. [6] use logic clocks [18] to determine the order of the messages. Logic clocks can be used as long as we do not have to consider at which time an event have happen, which is the case of software in real-time systems.

To achieve testing of real-time systems we also need to consider at which time events occur. Then logical clocks cannot be used since it cannot be determined when an event occur only in which order the occur.

In Yang et al. [46] the authors propose a test method for testing of concurrent programs. In the paper a model of the execution behavior consist of the input value (α), the produced execution path (δ or C-path), and a rendevouze path (σ or C-route). The basic idea in their paper is to find unique pairs of input and rendevouze path (α, σ) and determine which execution path the unique pair can be correlated to. By adding the C-route to the input the uniqueness of each produced C-path is guaranteed. In other words for each repeated test run with the same input that traverse the same C-path must also traverse the same C-route. Not all C-route are feasible. There can be dependencies between tasks that makes some C-paths infeasible, for example, rendevouz paths that lead to deadlocks.

The test method proposed in the paper is performed in six different steps that involves static analysis of each individual task on order to find C-paths and C-routes. When the analysis and selection of test cases, (α, σ) is done the test execution is performed in two stages first a nondeterministic test execution with only α from the test case. Second stage is a controlled test execution in which the execution is forced. The second stage, forced execution of σ, can be used to determine feasible C-routes since the forced execution of an infeasible C-route will lead to a failure of the execution for example, deadlocks.

2.6 Testing of Real-Time Systems

Real-time systems are software and hardware that in cooperation with their environment, and based on inputs from the environment, produce and deliver

results within specified time intervals. The time intervals are determined by the temporal constraints derived from the temporal properties of the environment. Because of the temporal constraints on the interaction between the real-time system and its environment the date of the data (inputs and outputs) is important [33]. Below is a definition of a real-time system:

> "*A real-time system is a system whose correctness depends not only on the logical result(s) of a computation, but also on the time at which the result(s) are produced*" [33].

There is a widespread range of programs that apply to the definition of real-time systems, ranging from video and audio streaming over Ethernet to pacemakers. To make distinction between different types of real-time systems they are categorized into two major types based on their criticality *hard real-time systems* and *soft real-time systems*:

Hard Real-Time Systems are systems in which a failure or violation of temporal constraints often leads to unacceptable consequences such as huge financial losses or human injuries.

Soft Real-Time Systems are systems in which it can be acceptable to allow occasional violations of temporal constraints. However, there may be constraints on how many violations that are allowed and the frequency of the violations.

Each type of system can further be grouped into the following subgroups based on the underlying execution model in the system and the invocations of the tasks; *event triggered* and *time triggered* real-time systems:

Event Triggered Real-Time Systems are systems that are driven by external and/or internal events. Examples of events are signals, message passing, internal interrupts (i.e. software interrupts), external interrupts. In other words, the run-time environment allow instances of tasks to be invoked at arbitrary points in time so the granularity of the release times is in the domain of continuous time.

Time Triggered Real-Time Systems are systems that are driven by a timer that periodically starts a scheduler that invokes instances of a task. That is, the run-time environment only allows tasks to be invoked at predetermined points in time. Each instance of a task is therefore released at discrete points in time.

2.6.1 Distributed Real-Time System

Distributed real-time systems are systems where computations are performed at self-contained computers (nodes) that are interconnected by a network. Communication between the nodes is achieved by messages passing and the processes can use synchronization to maintain a precedence relation or mutual exclusion between processes on different nodes. Processes on the same node also use the communication service. A designer of real-time system often chose distributed solutions because of increasing complexity and safety requirements. A distributed solution makes it possible to achieve greater reliability through redundancy. Also the inherited distribution of the system, for example, control systems on a factory floor can be a cause to chose a distributed solution. [33].

2.6.2 Testing of Real-Time Systems

Hassan Gomaa [11] propose a software development approach for real-time systems that incorporates tools for automated testing of real-time systems. The development approach and the tools have been evaluated in a case-study in the development of a robot controller. The development approach is based on the software design method DARTS (Design Approach for Real-Time Systems) [10]. The design of a real-time system is to decompose the system into task and defining the tasks' interfaces according to the requirements in the specification. Thus, it is important to formally review the design specifications and to verify that the task decomposition conforms to the specification. After a detailed design specification has been accomplished the functionality of the tasks are implemented.

The functional requirements are described by *data-flow diagrams* for each task, this can be done since each task alone is a sequential program. The synchronization of tasks is assumed to be solved by using events. The receiver of the event blocks itself in order to wait for a wake-up signal. The control flow of the events is described in an *event sequence diagram*, based on a task structure chart, which makes it possible to define the flow in more detail and finer grained than the data-flow diagram. Since the input and output events are described in the event sequence diagram it is possible to derive test cases for the integration test phase. But before starting integration testing each task is functionally tested on the host computer.

The unit testing and the initial functional integration testing are exercised on the host computer. The reason for this is that there are often more and better tools on the host computer than for the target system. Also, it is much more

efficient to test on the host computer than on the target platform because testing can be more easily automated on the host.

For system testing and the initial temporal integration testing, the synchronization of tasks are tested by creating a skeleton of the main module of the task. This skeleton consists of the synchronization constructs. Testing can then be performed by using a stub that sends signals to wake up the task that is waiting for the signal. After the synchronization parts are tested more functionality can be added to the skeleton and be tested. Automated testing of real-time system on the host computer can only test for logical correctness. It cannot test for temporal behavior.

After integration testing on the host the real-time system is tested on the target platform. Preferably this is done in a incremental bottom up approach. This is because of the more low level functionality implemented the less drivers and environment simulators must be implemented. The automation of the system testing assumes that there exists a secondary storage for storing of test results. Preferably, the target is tested with an environment simulator that are feeding inputs and receiving and time-stamping outputs from the system. The testing can be controlled by test scripts from an external computer that is running the environment simulator.

In order for the host/target testing approach to work the development tools used (e.g. compilers) must support both the host and the target platform.

Koehnemann et al. [16] observed that testing (and debugging) are limited by the constraints of the software in real-time systems. Example of such constraints are concurrent designs, real-time constraints and embedded target environment. They also discuss increased complexity of concurrent and real-time software that leads to increased complexity of the testing.

Test execution of real-time systems (that often also are embedded systems) can be divided into four phases:

1. Unit Testing
2. Integration Testing
3. System Testing
4. Hardware/Software Integration Testing

in which the three first phases are similar to the phases of test execution of sequential programs. The fourth step is the testing of correctness of the control of devices attached to the system, i.e. the environment which the real-time

system are controlling. In practice, the test execution in each phase is often performed in two steps [33]. The first step consists of execution of the application while recording the behavior. Then in the second step, the recorded behavior is analyzed.

Testing for Functional Correctness

Thane et. al. [37] is addressing the problem of testing distributed real-time systems in a deterministic way. The difference in testing sequential programs and concurrent programs is that for the same sequence of inputs different output can be produced by the concurrent program. Therefore, sequential testing techniques cannot be used to test concurrent programs and real-time systems. The authors propose a approach for testing of distributed real-time systems using sequential test tools.

The test approach is divided into three iterative steps:

1. identify the set of possible execution orderings (serializations),
2. test the system using any test technique of choice,
3. map each test case and output onto the correct execution ordering, based on observation and
4. repeat 1-3 until required coverage is achieved.

In the first step a static off-line analysis of the software is performed. This is done by using a analysis tool that derives all possible execution orderings and creates a *Execution Order Graph*(EOG). The EOG is a output from a simulation of the behavior of a preemptive scheduling policy [2, 24, 45]. More exactly the graph is showing the non-deterministically behavior in the execution of the real-time software. The analysis tool assumes that execution time, priority and release time are known. Release times and the priorities of the tasks are determined at design time. However, execution times of tasks cannot be easily determined neither at design or when specification is realized in a implementation.

The second step are the exercise of test case on the target by using appropriate testing techniques. During the run of test cases the execution behavior, i.e. the control flow of a particular test run, are monitored and saved in a log. Since the test approach do not consider the the no-deterministic behavior until later steps testing tools for test of sequential programs can be used in this step.

In the next step the analyzed and observed execution behavior are compared. If a test case and corresponding execution behavior can be mapped onto a branch in the EOG the mapping are noted and the steps are repeated until coverage criteria are fulfilled. The coverage criteria are of two types the first is how many times each branch have been observed during the test runs and the second how many of the unique branches have been observed.

The deterministic approach in testing of distributed real-time systems is achieved in step 3. The definition of determinism are; for each test case during repeated test runs the same output is observed. By in addition to the test case also observe the execution behavior as output determinism is achieved in step 3 when mapping the output onto the EOG.

In distributed systems during the exercise of the test cases on each node the control flow are saved in a log. The difference between testing of a single node system and a distributed system is that on each node the local clock must be synchronized with other local clocks on other nodes and the increase of complexity when analysis in step 1 is performed.

Testing for Temporal Correctness

Tsai et al. [38] provides methods for dynamic analysis of correctness of temporal constraints of real-time software. The approach is based on a non-intrusive monitoring technique that record run-time information. The run-time information is then used to analyze the software for violations of temporal constraints. From the run-time information graphs are constructed for analysis of temporal constraints. The graphs created are *Timed Process Interaction Graph* and *Dedicated Timed Process Interaction Graph*.

In Khoumsi [14] the author propose a method to test the temporal constraints of the output from distributed real-time systems. The method consists of three phases how to specify a distributed real-time system, a distributed test architecture and a procedure for distributing test sequences.

The method assumes that the distributed real-time system is modeled as a n-port Timed Automata. Based on this model the temporal constraints are derived and transformed into global test sequences that are distributed to *testers*. Testers are independent nodes that feed the system with inputs at the appropriate instance of time and receive output for analysis of the temporal correctness.

To verify the order and timing of the inputs and outputs each tester have an assigned local clock that can be asked for the time and the local clock can be used as an alarm for the timing of the input.

This method test the timing and order of the output from the distributed

real-time system. This is an important aspect of a real-time system since the correctness of such system depends on at which time the result is produced. However, the author do not discuss the problem of having clocks on different sites in a distributed system. The drift of clocks is a problem for the global view of what the time it is. It is not mentioned how the clock drift effect the analysis of the timing of the outputs.

Test Strategies

Test strategies are descriptions on how to set-up the system, perform the test execution and analyze the result of the test execution of a test case.

Schütz [32, 34] have proposed a test strategy for testing of distributed real-time systems, designed for the MARS architecture. The test strategy consists of five different test phases

- *Task Test*,
- *Cluster Test*,
- *Interface Test*,
- *System Test* and
- *Field Test*.

Task Test are functional testing and preliminary interface testing, performed on the individual tasks. Task test are performed entirely on the host system. This demands that the task programmer are supplied with appropriate programming tool set.

Cluster Test are performed on the target system. The author propose two types of Cluster Tests; open-loop Cluster Test and closed-loop Cluster Test. Open-loop Cluster Test tests the functional correctness of a cluster and the temporal correctness of the interaction of task. Open-loop Cluster Test is also used when testing for loss of messages in communication between clusters. In closed-loop Cluster Testing more realistic inputs can be fed and robustness test can be performed since the output are dynamically analyzed and re-calculated and can be fed back as input to the cluster and thereby close the loop. The main difference of open-loop and closed-loop Cluster Testing is that in closed-loop Cluster Testing the application is run without modification with a environment simulator and can therefore include test of temporal correctness. However, in both approaches a special test system has to be build to behave as the surrounding system from the clusters point of view.

Interface Test are tests that peripheral devices attached to the systems Interface Buses behaves in an expected manner.

System Test tests the interaction between clusters and that the system as whole behaves according to the specification.

Field Test tests the system with the real environment and real peripheral devices. In this test phase the system is in its operational environment and can therefore be used as customer acceptance test.

This test strategy test distributed real-time systems. However, the application must be designed to follow the assumptions for the MARS system. Several drawbacks are discussed in the paper and one of most important for debugging and testing on the target is the coupling of the monitored to the high level language used when programming. For the aspect of real-time scheduling the off-line scheduling assumption, as in any other real-time system, reduce the flexibility of the system but simplifies the analysis of the number of test case needed for code coverage. Since, off-line scheduled real-time systems can be seen as a sequential program where the execution behavior is known a-priori.

Test Bed Architectures

Kopetz et al. [17] propose a architecture for running distributed fault-tolerant real-time systems. The architecture is called Maintainable Real-Time Systems (MARS) architecture and supports statically scheduled hard real-time systems. MARS consist of clusters that can be interconnected by an arbitrary network topology. Tasks that have functionality relation are allocated to the same cluster. There are no tools for automating the allocation of tasks to a cluster so the designer itself is responsible for the appropriateness of task allocations on clusters.

Each cluster consists of a set of components that are interconnected by a MARS-bus. A component is a self-contained computer that have identical copies of the MARS-OS and tasks. The tasks are communicating through the MARS-bus by using MARS standardized messages. In the cluster there is also an Interface Component that is connected to a Interface Bus that makes it possible to communicate with the environment (another MARS cluster or the physical process).

In Thane et al. [36] the authors presents a test architecture that is suitable for testing of embedded systems. The test-rig consists of the system itself, with one ore more nodes, and a test node on which the result of the computations in the system are analyzed. On the test node it is determined if the computations produced the expected results or not.

Environment Simulators

As discussed in previous sections a real-time system is a system that interacts with its environment. In testing of such systems there may be the case that the environment does not exist yet because of parallel development of hardware and software or when the cost or safety inhibits the use of the real hardware. In these cases the environment must be simulated in order to enable testing of the real-time software. A simulation is the execution of a computer program that represents a model of a real hardware. From the simulation the behavior can be used as stimuli to the system that is to be tested.

2.6.3 Regression Testing of Real-Time Systems

Zhu et al. [47] have proposed a framework for how to automate regression testing of real-time software in distributed environment. They discuss testing of safety-critical real-time systems such as pacemakers and defibrillators. Testing of software in pacemakers cannot be performed in its natural environment since a failure of the pacemaker can lead to human injuries and therefore requires expensive specialized hardware for testing. Thus, automating the testing procedure is of importance for reducing the cost, using the test equipment in an efficient way and to remove the error-prone manual handling. The framework is developed based on Onomas [27] regression testing process.

The distributed regression testing framework is built upon three components *test server, test stations* and *test clients*. In this context components can be general purpose computers or specially designed systems. All instances of the three components are connected to a local area network for efficiency and high utilization of the test stations. The test server serves as an oracle and have access to the test database. When a test is to be exercised the test client first creates test cases based on the information from test databases and the test engineer. After test case creation the test clients are responsible for submitting the test and control and monitor the exercise of the test case. The test station are the component on which the actual test case execution is performed.

The framework is designed with an object-oriented approach. This makes it easier, for example, for composing of complex test cases that are composition of several test cases and using different test case selection strategies.

The framework consists of four different layers *network layer, support layer, task layer* and *interface layer*. In the network layer existing communication mechanisms provided by the operating system are used. The support layer have three responsibilities: connection for access of test database, trans-

portation of files between the three components and remote control of method invocations. The task layer is a set of programs that performs tasks such as test case submission, test case selection and test case execution. For easy use of the framework for test engineers the interface layer provides visual interfaces.

Other important issues for automation and flexibility of the framework are the test case allocation, test load balancing, test interruption and recovery, composite test cases and dynamic test station configuration.

To able to perform regression testing and to be able to tell if the faults are removed the real-time software must have deterministic execution behavior. The framework proposed by Zhu et al. seems to be aimed to real-time software that is single-tasking or non-premptive programs that run sequentially and therefore have deterministic execution behavior. Unless a test method that can handle the non-determinism in the execution behavior is used the framework cannot be used for achieving regression testing of multi-tasking real-time systems.

2.7 Summary

There are many types of software and each of these software types may require specialized tools and methods for testing. For example, testing of sequential programs can be performed by feeding inputs to the program and then observing the output in order to tell if the behavior of the execution is correct according to the requirements. This is because of that sequential programs have a deterministic execution behavior. To locate the defect a debugger can be used.

Testing techniques that test the behavior of sequential computer programs is a well established and explored area both for the industrial users and researchers. However, testing of sequential programs is not a trivial task and can only in rare cases be done with small efforts. This is because when testing computer programs a large amount of test cases must be exercised (usually manually.

To succeed in testing we need not only be concerned about the execution of the software to reveal failures, we must also design the software so that it can be tested with little effort. It is also important that testing is integrated with the development of the software. This has the benefit that testing is considered at early stages of the design of the software and that it can decrease the cost of finding faults.

When failures are revealed the source code is corrected and the program is re-tested. This retest is time consuming and costly because of:

- an analysis is performed in order to chose a subset of test cases that must be exercised,
- for each iterative step in the regression testing new test cases are added that increases the number of test cases to run, and
- by not running test cases there is a potential risk of faults being present in the software.

Academia is interested in reducing the test efforts by reducing the number of test cases while industry is interested in more effective tools and automated testing, leading to the situation where there are numerous research results on test case selection tools, but few on automation of retests.

Testing of concurrent programs is more complex than testing of sequential programs. The complexity is caused by the interleaved execution leading to indeterminacy of the execution behavior. That is, because of the nondeterministic execution behavior it is impossible to establish the correctness of the program since each input can produce different outputs.

The common approach to test concurrent programs is to derive test cases based on the execution behavior (synchronization sequences) when tasks are communicating with each other. By the use of the synchronization sequences the execution can be controlled at the synchronzation events, and hence deterministic testing can be achieved.

A real-time system must be tested for both functional correctness and temporal correctness. There are very few tools for testing real-time systems and existing tools often requires special hardware or software architectures.

Regression testing of multi-tasking real-time systems is hard since it requires not only control of the inputs and the state in the program but also control over the time at which events occur.

Bibliography

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *Proceedings of Conference on Software Maintenance*, pages 348–357, 1993.

[2] N. C. Audsley, A. Burns, R. I. Davis, and K. W. Tindell. Fixed priority pre-emptive scheduling: A historical perspective. In *Real-Time Systems journal*, volume 8(2/3). Kluwer A.P., March/May 1995.

[3] C. Bernardeschi, L. Simoncini, and A. Fantechi. Validating the design of dependable systems. In *Proceedings First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 98)*, pages 364–372, Apr 1998.

[4] A. Bertolino, F. Corradini, P. Inverardi, and H. Muccini. Deriving test plans from architectural descriptions. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 220–229, 2000.

[5] R. H. Carver and K-C. Tai. Replay and testing for concurrent programs. In *IEEE Software*, volume 8(2), pages 66–74, 1991.

[6] Sang Chung, Hyeon Soo Kim, Hyun Seop Bae, and Don Gil Lee Yong Rae Kwon. Testing of concurrent programs after specification changes. In *Proceedings IEEE International Conference on Software Maintanence (ICSM '99)*, pages 199–208, 1999.

[7] S. J. Clarke and J. A. McDermid. Software fault trees and weakest preconditions: A comparison and analysis, 1993.

[8] M. E. Fagan. Design and code inspections to reduce errors in program development. In *IBM Systems Journal*, volume 15(3), pages 182–211, 1976.

[9] J. Gait. A probe effect in concurrent programs. In *Software - Practice and Experience*, volume 16(3), pages 225–233, Mars 1986.

[10] H. Gomaa. A software design method for real-time systems. *Communications of the ACM*, 27(9):938–949, 1984.

[11] H. Gomaa. Software development of real-time systems. *Communications of the ACM*, 29(7):657–668, 1986.

[12] I. Granja and M. Jino. Techniques for regression testing: Selecting test case sets taylored to possibly modified functionalities. In *Proceedings of the Third European Conference., Software Maintenance and Reengineering*, pages 2–11, 1999.

[13] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings., Conference on Software Maintenance*, pages 299–308, 1992.

[14] A. Khoumsi. Testing distributed real-time systems using a distributed architecture. In *Proceedings of the 2000 International Conference on Software engineering*, pages 126–135, 2000.

[15] J. M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *Proceedings of the 2000 International Conference on Software engineering*, pages 126–135, 2000.

[16] Harry Koehnemann and Timothy Lindquist. Towards target-level testing and debugging tools for embedded software. In *Proceedings of the conference on TRI-Ada '93*, pages 288–298. ACM Press, 1993.

[17] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The mars approach. In *IEEE Micro*, volume 9(1), pages 25–40, 1989.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, volume 21(7), pages 558–565, July 1978.

[19] J.C. Laprie. Dependability: Basic concepts and associated terminology. In *Dependable Computing and Fault-Tolerant System*, volume 5. Springer Verlag, 1992.

[20] Yann-Hang Lee, YoungJoon Byun, Ji Xiao, O. Goh, W. E. Wong, and A. Lee. A toolsuite for testing analysis of real-time ada applications. In *Proceedings of 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, pages 65–69, 2000.

[21] H. K. N. Leung. Selective regression testing assumptions and fault detecting ability. In *Information and Software Technology*, volume 37(10), pages 531–537, 1995.

[22] H. K. N. Leung and L. White. Insights into regression testing. In *Proceedings., Conference on Software Maintenance*, pages 60–69, 1989.

[23] H. K. N. Leung and L. White. A cost model to compare regression test strategies. In *Proceedings., Conference on Software Maintenance*, pages 201–208, 1991.

[24] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. In *Journal of the ACM*, volume 20(1), 1973.

[25] C. E. McDowell and D. P. Helmbold. Debugging concurrent program. In *ACM Computing Surveys*, volume 21(4), pages 593–622, December 1989.

[26] IEEE Standard Glossary of Software :Engineering Terminology. Ieee standards collection, ieee std 610.12-1990. September 1990.

[27] A. K. Onoma, W. T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. In *Proceedings. IEEE Transactions on Software Engineering*, volume 22(8), pages 529–551, 1996.

[28] D. L. Parnas. Tabular representation of relations. In *Technical Report, Telecommunications Reasearch Institute of Ontario, Communicaton Research Laboratory. Department of Electrical and Computer Engineering, McMaster University, Hamilton, Ontario Canada L8S 4K1, CRL Report*, number 260, 1992.

[29] G. Rothermel and M. J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings. IEEE International Conference on Software Maintenance (CMS '93)*, pages 358–367, 1993.

[30] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. In *Proceedings. Communications of the ACM*, volume 41(5), pages 81–86, 1998.

[31] G. Rothermel, R. H. Untech, and M. J. Harrold. Test case prioritization: an empirical study. In *Proceedings. IEEE International Conference on Software Maintenance (ICMS '99)*, pages 179–188, 1999.

[32] W. Schütz. A test strategy for the distributed real-time system mars. In *Proceedngs of the 1990 IEEE International Conference on Computer Systems and Software Engineering*, pages 20–27, 1990.

[33] W. Schütz. Fundamentals issues in testing distributed real-time systems. In *Real-Time Systems*, volume 7, pages 129–157, Boston, 1994. Kluwer Academic Publisher.

[34] Werner Schütz. Testing a distributed real-time system – the mars approach. Research Report 11/1989, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1989.

[35] K.-C. Tai, R.H. Carver, and E.E. Obaid. Debugging concurrent ada programs by deterministic execution. In *IEEE Transactions on Software Engineering*, volume 17(1), pages 45–63, January 1991.

[36] H. Thane. Monitoring, testing and debugging of distributed real-time systems. In *Doctoral Thesis*, Royal Institute of Technology, KTH, S100 44 Stockholm, Sweden, May 2000. Mechatronic Laboratory, Department of Machine Design.

[37] H. Thane and H. Hansson. Towards systematic testing of distributed real-time systems. In *Proceedings of The 20th IEEE Real-Time Systems Symposium*, pages 360–369, 1999.

[38] J. J. P. Tsai, K.-Y. Fang, and Y.-D. Bi. On real-time software testing and debugging. In *Proceedings of Fourteenth Annual International Computer Software and Application Conference*, pages 512–518, Oct 1990.

[39] Naoshi Uchihira, Shinichi Honiden, and Toshibumi Seki. Hypersequential programming: A new way to develop concurrent programs. 5(3):44–54, July/September 1997.

[40] J. M. Voas and K. W. Miller. Software testability:the new verification. In *IEEE Software*, volume 12(3), pages 17–28, May 1995.

[41] S. N. Weiss. A formal framework for the study of concurrent program testing. In *Proceedings of the Second Workshop on Software Testing, Verificaion and Analysis*, pages 106–113, July 1988.

[42] L. J. White and H. K. N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings., Conference on Software Maintenance, 1992*, pages 262–271, 1992.

[43] J. A. Whittaker. What is software testing and why is it so hard. In *IEEE Software*, January/February 2000.

[44] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings. The Eight International Symposium on Software Reliability Engineering*, pages 264–274, 1997.

[45] J. Xu and D. Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. In *IEEE Transaction on Software Engineering*, volume 16(3), pages 360–369, 1990.

[46] R-D. Yang and C-G. Chung. Path analysis testing of concurrent program. In *Information and Software Technology*, volume 34(1), pages 43–56, Jan 1992.

[47] F. Zhu, S. Rayadurgam, and W.-T. Tsai. Automating regression testing for real-time software in a distributed environment. In *Proceedings of First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 98)*, pages 373–382, 20-22 April 1998.

[48] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.

Chapter 3

Paper B: Testing of Multi-Tasking Real-Time Systems with Critical Sections

Anders Pettersson and Henrik Thane
Mälardalen University, Mälardalen Real-Time Research Centre,
P.O. Box 883 SE-721 23 Västerås, Sweden
{anders.pettersson, henrik.thane}@mdh.se

In Proceedings of 9th International Conference on Real-Time and Embedded Computing Systems and Applications, February 2003

Abstract

In this paper we address the problem of testing real-time software in the functional domain. In order to achieve reproducible and deterministic test results of an entire multitasking real-time system it is essential not to only consider inputs and outputs, but also the order in which tasks communicate and synchronize with each other. We present a deterministic white-box system-level control-flow testing method for deterministic integration testing of real-time system software. We specifically address fixed priority scheduled real-time systems where synchronization is resolved using the Priority Ceiling Emulation Protocol or offsets in time. The method includes a testing strategy where the coverage criterion is defined by the number of paths in the system control flow. The method also includes a reachability algorithm for deriving all possible paths in terms of orderings of task starts, preemption and completions of tasks executing in a real-time system. The deterministic testing strategy allows test methods for sequential programs to be applied, since each identified ordering can be regarded as a sequential program.

3.1 Introduction

Testing software is challenging. A typical solitary program has a large state space and a discontinuous behavior. The latter due to containers with limited resolution, e.g., 32 bit integers, quantization errors, and program flow selections. The implication is that it is highly unreliable to make use of interpolation when testing programs. Consequently, a large part of the state space must be explored in order to verify that inputs produce correct outputs according to the specification. It is not surprising that a large part of software development budgets is spent on maintenance. Elevating to the level of real-time software testing, the challenge is even greater. Real-time software is usually built on an aggregate of multiple concurrently executing programs, i.e., it is multi-tasking. To begin with, this entails testing of multiple programs. What is worse however, is the state space explosion that occurs due to the interactions between the tasks when they execute concurrently. These interactions are not limited to the functional domain but are also a function of the timing and the ordering of the tasks' execution in the system. The majority of current testing and debugging techniques have been developed for solitary (non real-time) programs. These techniques are not directly applicable to real-time systems, since they disregard issues of timing and concurrency. This means that existing techniques for reproducible testing cannot be used. Reproducibility is essential for regression testing and cyclic debugging, where the same test cases are run repeatedly with the intention of verifying modified program code or to track down errors. It is common that real-time software has a non-reproducible behavior. This is due to the fact that giving the same input and same internal state to a program is not sufficient. There are hidden variables that are ignored: Race conditions and ordering. An aspect of this is intrusive observations caused e.g., by temporary additions of program code, which incur a temporal probe-effect [6] by changing the race conditions in the system.

In theory it is possible to reproduce the behavior of a real-time system if we can reproduce the exact trajectories of the inputs to the system with an exact timing. For guaranteed determinism we would in addition need to control the frequency of the temperature dependent real-time clock that generates the periodic timer tick, which is the basis for all time driven scheduling decisions. The inputs, and state, of the tasks dictates their individual control flow paths taken, which in turn dictates the execution time of the tasks, which in the end dictates the preemption pattern for strictly periodic systems. Trying to perform exhaustive black-box testing of individual programs is in the general case infeasible, due to the large number of possible inputs. For example,

two 32 bit inputs yields 2^{64} possible input combinations, not considering state, which for a test case every 10^{-12} seconds would take about half a year to execute. For a typical multitasking real-time system the number of possible input combinations is similarly bordering on the incomprehensible due to all possible temporal and functional interactions between the tasks. However, just as individual program's control flow structure can be derived and used for white-box testing (where the number of paths is usually significantly lower than the number of inputs), we can make use of the system level control flow for deterministic white-box testing of the multitasking real-time system software. We will elaborate on this issue in this paper.

Testing real-time systems controlling only the inputs have been attempted previously - mostly in the formal methods community were formal specifications models have been used for generating inputs to the system to test either the temporal [5][10] [13] or the functional [9] behavior. In comparison to other sub-fields within the real-time systems research community the list of references dealing with testing of real-time software is quite meager, rather famished in fact. One reference that has inspired us is the work by Yang and Chung [18]. They define a system level control-flow testing method for testing of concurrent Ada programs (not real-time but concurrent). The system control flow is defined by all synchronization sequences (rendezvous) in the system. When testing a concurrent Ada program the executed synchronization sequence is defined as being part of the output. If a test case is applied twice, and the same synchronization sequence is observed, then the same behavior has been exercised - thus deterministic testing is achieved. However, it is not certain that the tests are reproducible, since there exist no explicit control over the synchronization sequences. The number of paths executed divided by the number of paths derived is used to define coverage. Similar work can be found in Hwang et al. [7] where they also attempt deterministic replay [15] in order to achieve reproducibility. Since Yang et al. and Hwang et al. only concentrate on the rendezvous sequences they do not handle more intricate real-time operating system issues like preemption, interrupts or critical sections.

In this paper we extend the method for achieving deterministic testing of distributed real-time systems by Thane and Hansson [16][14]. They addressed task sets with recurring release patterns, executing in a distributed system, where the scheduling on each node was handled by a fixed priority driven preemptive scheduler supporting offsets. The method transforms the non-deterministic distributed real-time systems testing problem into a set of deterministic sequential program testing problems. Similarly to Yang's work, but with the inclusion of preemption, interrupts and communication delays,

Thane and Hansson define the executed orderings between tasks (derived from task-switch monitoring) to be part of the system's output. Thus, achieving determinism is an issue of correlating inputs, with outputs and execution orderings (the system control-flow). Coverage is defined by the number of unique system control-flow paths tested, and by the number of test cases run per each path. The former criterion is derived from a system control-flow analysis and the latter criterion is defined by the testing technique applied, e.g., statistical confidence in black-box testing.

In their system control-flow analysis method they assumed that all synchronization was resolved offline, e.g., by an off-line scheduler, which assigns offsets and priorities to all tasks in the distributed system. That is, on-line synchronization mechanisms like semaphores are not allowed. All tasks in the system are also assumed to receive all input immediately at their start, and to produce all output at their termination. These limitations were quite severe, although the analysis proved that even off-line scheduled systems could yield enormous numbers of different scenarios, when subjected to preemption and jitter (execution time-, communication time-, and interrupt induced jitter), especially when the tested systems were of multi-rate character.

In this paper we elaborate on the approach presented by Thane and Hansson in [16][14] and expand the task model to also include critical sections, governed by the Priority Ceiling Emulation Protocol (PCEP) [3], a.k.a. the immediate inheritance protocol and immediate priority ceiling protocol. Since tasks may synchronize/communicate via critical sections, we will also relax Thane's and Hansson's input output assumption. Our extension is however only valid for the individual nodes in the distributed real-time system, unless we assume a global PCEP, which is quite complex to achieve [12]. The subsequent analysis in this paper is hence focused on a single node. The results by Thane and Hansson [16][14] on how to derive the global system control-flow can however successfully be applied if global scheduling is relying on offsets between tasks on different nodes, but this is outside the scope of this paper.

The basic intuition behind deterministic testing can be illustrated as follows. Consider Figure 3.1, which depicts two execution scenarios of two tasks A, B, who share a common resource x, which they do operations on. The resource, x, is protected by a semaphore governed by the priority ceiling emulation protocol which raises the priority of the task that is granted the resource to the priority ceiling of the tasks using the resource. In Figure 3.1 scenario (a), task A enters the critical section before B and thus accesses x before B - with end result of scenario (a), $x = 25$. In Figure 3.1 scenario (b), task B enters the critical section before A and thus accesses x before A - with the end result of

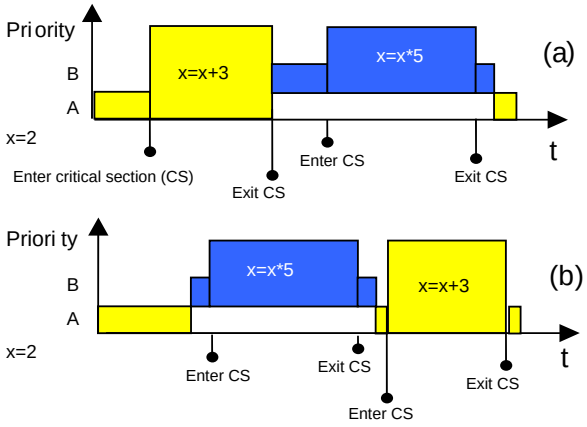

Figure 3.1: Two different execution orderings with different results, caused by race conditions in accesses of a shared resource x.

scenario (b), $x = 13$.

As we can see in Figure 3.1, even though the same input is provided, $x = 2$, the end result of the execution is dependent of the task execution ordering, i.e., the system level control-flow path taken. However, if we run the same scenario with the same input, the result will always be the same on repeated executions. That is, the multitasking real-time system is deterministic if we consider both inputs and execution orderings.

3.1.1 Contribution

The contribution of this paper is a deterministic white-box system level integration testing method that includes:

- A testing strategy for achieving a required level of coverage, with respect to the number of paths in the system control-flow. The testing strategy also allows test methods for sequential programs to be applied, since each identified ordering can be regarded as a sequential program.

- A reachability technique for deriving the system level control-flow. The system control-flow is defined by all possible orderings of task starts,

preemptions and completions for tasks executing in a system where synchronization is resolved using offsets or using PCEP.

The result in this paper substantially extends the applicability of the results by Thane and Hansson [16][14], since we now can handle systems with on-line synchronization, for which it is actually more likely that errors are caused by implementation and synchronization problems. Also, PCEP has been adopted in industry standards, like *POSIX*, *ADA95*, and *OSEK*, for its implementation simplicity [8][1].

The organization of the paper is as follows: Section 3.2 presents our deterministic integration testing strategy. Section 3 introduces a method for deriving the system control-flow when synchronization is resolved by the PCEP protocol or offsets. Finally, in Section 4, we conclude.

3.2 The Deterministic Test Strategy

In the scope of our test strategy, we define a single executed *system level control-flow path* (SLCFP) to be part of the system's output.

By correlating inputs with outputs and executed SLCFPs deterministic test results are achieved. Coverage is defined by the number of unique SLCFPs tested, and by the number of test-cases run per each path. The former criterion is based on a system control-flow analysis, which we present in section 3.3. The latter criterion is defined by the testing technique applied, e.g., statistical confidence in black-box testing [4].

For the testing strategy to work we need in addition to the inputs and outputs, means to extract the system control flow, usually in the form of task-switches and access to semaphores: activation of task, entering critical section, leaving critical section, preemption, and task completion. We thus expand on the work by Thane and Hansson [19][20] to also include races to critical sections. This SLCFPs extraction can be facilitated in a number of ways, ranging from intrusive software instrumentation, and hooks into the real-time kernel, to special non-intrusive hardware like In Circuit Emulators, with OS awareness. If the instrumentation is implemented in software, it is necessary to eliminate the probe effect, usually by leaving the instrumentation code in the deployed system. In our experience the execution time overhead for software instrumentation of the SLCFPs is minimal, typically below 0,1 $^{0}/_{00}$ of processor utilization.

Definition. The deterministic test procedure (as illustrated in Figure 3.2) with no knowledge of the number of possible SLCFPs is defined as:

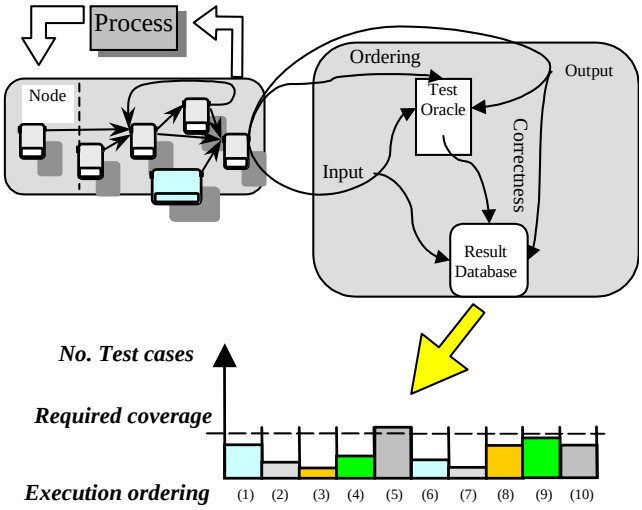

Figure 3.2: A test rig with a set of system level control-flow buckets, and where the coverage for each bucket is illustrated.

1. Test the system using any sequential technique of choice, and monitor the 3-tuple (input, output, SLCFP) for each test case. A test case includes all inputs to the participating tasks that are part of the SLCFP during the interval $[0, T^{MAX}]$, where T^{MAX} typically is equal to the *Least Common Multiple* (LCM) of the tasks period times.

2. Map the 3-tuple for the interval $[0, T^{MAX}]$ into a "bucket" for each unique SLCFP.

3. Repeat 1-2 until required coverage for the sequential testing technique applied is reached for every bucket.

With the above defined testing procedure we can achieve deterministic testing, with respect to failures that pertain to ordering, and its effect on the inputs and outputs via the systems legal interfaces. That is, the method is not deterministic with respect to failures like transient bit-flips, or arbitrary memory corruption from e.g., non-reentrant code, unless we regarded every assembly write operation as a critical section - which is unreasonable.

The above defined testing strategy is however not complete, since we do not know when to stop testing. We do not know how many SLCFPs there exist. In the next section we will present a technique for deriving all possible SLCFPs from which we can calculate the maximum number of SLCFPs and thus derive a stopping criterion. The stopping criterion can either be based on the system control flow for all tasks in the system or for just a sub set of the tasks. If we during testing after a while notice that certain paths have attained a low level of coverage (e.g., 0) then this can either be attributed to a pessimism in the system control flow analysis (e.g., two tasks may not execute their worst case execution time in the same execution scenario) such that too many paths are derived, or that certain paths are simply rare during execution. In any case deterministic replay technology [15] can be used for enforcing certain paths such that the required coverage for these paths is attained. The application of deterministic replay is however out of scope for this paper, and is something we will present in a later publication.

3.3 System Control-Flow Analysis

In order to derive a stop criterion for the deterministic testing strategy we now define the system level control-flow in terms of a System Level Control-Flow Graph (*SLCFG*) and present an algorithm that generates SLCFGs, from which we can derive all possible system level control-flow paths (*SLCFP*). We begin however, with a definition of the system task model.

3.3.1 Task Model

The real-time system software consists of a set of concurrent tasks. Tasks communicate by non-blocking message passing or shared memory. All synchronization, precedence or mutual exclusion, is resolved either offline by assigning different release-times/offsets and priorities, or during runtime by the use of semaphores which have PCEP semantics. Further, we assume a task model that includes both preemptive scheduling of off-line generated schedules [17] and fixed priority scheduling of strictly periodic tasks [2][11].

- The system contains a set of jobs J, i.e. invocations of tasks, which are released in a time interval $[t, t+T^{MAX}]$, where T^{MAX} is typically equal to the LCM of the involved tasks period times, and t is an idle point within the time interval $[0, T^{MAX}]$ where no job is executing. The existence of such an idle point, t, simplifies the model such that it prevents

temporal interference between successive T^{MAX} intervals. To simplify the presentation we will henceforth assume an idle point at 0.

- Each job $j \in J$ have the following attributes: a release time r_j, worst case execution time ($WCET_j$), best case execution time ($BCET_j$), a deadline D_j, and a unique base priority bp_j. J represents one instance of a recurring pattern of job executions with period T^{MAX}, i.e., job j will be released at time r_j, $r_{j+T^{MAX}}$, $r_{j+2T^{MAX}}$, etc. Jobs may have identical release times.

3.3.2 Synchronization using PCEP

For PCEP we assume that:

- Each job $j \in J$ has a current priority p_i that may be different from the statically allocated base priority, bp_j, if the job is subject to priority promotion when granted a resource.

- Each resource R, used by a set of jobs S_R, has a statically computed priority ceiling defined by the highest base priority in S_R increased by one, i.e., $p_R = MAX(bp_i | I \in S_R) + 1$. We assume that all jobs have unique priorities so we need to increase p_R by one to achieve a unique priority for the priority ceiling; jobs that have higher priorities than p_R are also adjusted to have unique priorities.

- Each job, j, that enters a critical section protecting a resource R is immediately promoted to the statically allocated priority ceiling of the resource, if $p_R > p_j$ then $p_j = p_R$.

- Each job, j, that is executing and releases a resource R is demoted immediately to the maximum of the base priority bp_j, and the ceilings of the remaining resources held by the job.

- Each critical section, k, has a worst case execution time ($WCET_k$) and a best case execution time ($BCET_k$) and a release time interval $[er_k, lr_k)$ ranging from the earliest release time to the latest release time.

- All resources are claimed in the same order for all paths through the program in a job.

3.3.3 The System Level Control-Flow Graph

In essence, to derive the system level control-flow graph, we perform a reachability analysis by simulating the behavior of a real-time kernel conforming to our task model during one $[0, T^{MAX}]$ period for the job set J. The System Level Control-Flow Graph ($SLCFG$) is a finite tree for which the set of possible paths from the root contains all possible execution scenarios.

We define a SLCFG as a pair $< N, A >$, where

- N is a set of *nodes*, each node being labeled with a job, the job's current priority, and a continuous time interval, i.e., for a job set J : $N \subseteq J \cup$ "_" $\times P \times I(T^{MAX})$, where "_" is used to denote a node where no job is executing. P is the set of priorities, and $I(T^{MAX})$ is the set of continuous intervals in $[0, T^{MAX}]$.
- A is the set of edges (directed arcs; transitions) from one node to another node, labeled with a continuous time interval, i.e., for a set of jobs J : $A \subseteq N \times I(T^{MAX}) \times N$.

Basic transitions

Intuitively, an edge corresponds to the transition (the task-switch) from one job to another, or when a job enters or leaves a critical section. The edge is annotated with a continuous interval of when the transition can take place, as illustrated in Figure 3.3, showing SLCFGs for simple jobs without critical sections.

$$\xrightarrow{[a,b)} \overset{[\alpha,\beta)}{A : p_A} \xrightarrow{[a',b')} B : p_B$$

Figure 3.3: Two transitions, one to job A and one from job A to job B.

The interval of possible start times $[a', b')$ for job B, in Figure 3.3, is defined by:

$$\begin{aligned} a' &= max(a, r_A) + BCET_A \\ b' &= max(b, r_A) + WCET_A \end{aligned} \tag{3.1}$$

The max() functions are necessary because the calculated start times a and b can be earlier than the scheduled release of the job A. In the SLCFG a node

represents a job annotated with a continuous interval of its possible execution time, $[\alpha, \beta)$, as depicted in Figure 3.4.

$$\xrightarrow{[a,b)} \overset{[\alpha,\beta)}{A : p_A}$$

Figure 3.4: A job annotated with its possible execution, start time and current priority.

We define the interval of execution, $[\alpha, \beta)$ as the interval in which job A can be preempted:

$$\begin{aligned} \alpha &= max(a, r_A) \\ \beta &= max(b, r_A) + BCET_A \end{aligned} \tag{3.2}$$

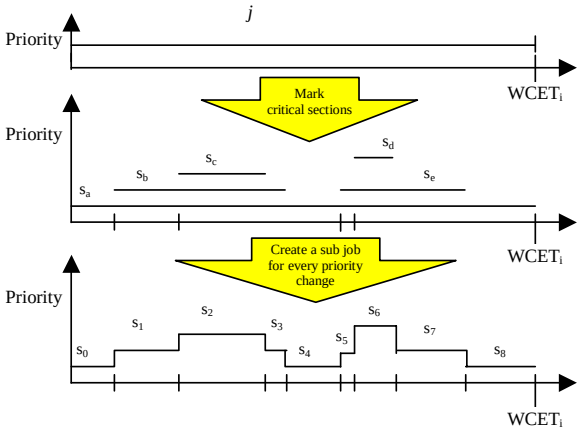

Figure 3.5: A job split into a set of sub jobs, in order of changes in effective priority. The sub jobs s_0, s_4, and s_8 represent the base priority job.

Critical section transitions

Critical sections will be introduced by transforming the job set, such that a job with critical sections is split into a set of jobs corresponding to the different

critical sections and executions in between. We assume that each job $i \in J$, which has a set of critical sections CS_i, is split into an ordered list of sub jobs, SJ_i, such that every time there is a change in the job's effective priority a new sub job is added (as illustrated in Figure 3.5). Each sub job $s_i \in SJ_i$ of original job i have a release time interval $[er_s, lr_s)$ ranging from its earliest release time to its latest release time. The release time interval for a sub job s_i is given in terms of execution time run by the immediately preceding sub job, q_i, before it enters the critical section represented by sub job s_i, rather than in terms of the system clock tick. This means that all $BCET$s and $WCET$s for all sub jobs are calculated such that they represent execution time before entering the immediately succeeding critical section except the last sub job, which runs until termination.

$$\xrightarrow{[a,b)} \underset{q_i : p_q}{[\alpha,\beta)} \quad \xrightarrow{[a',b')} \underset{s_i : p_s}{[\alpha',\beta')} \quad \xrightarrow{[a'',b'']} \underset{z_i : p_z}{[\alpha'',\beta'')}$$

Figure 3.6: Three transitions, one to sub job q_i, one demoting transition from sub job q_i to sub job s_i, and one promoting transition from sub job s_i to sub job z_i.

The interval of possible start times $[a', b')$ for the sub job s_i, as illustrated in Figure 3.6, is defined relative to its predecessor, q_i, by:

$$\begin{aligned} a' &= max(a, r_q) + BCET_q \\ b' &= max(b, r_q) + WCET_q \end{aligned} \tag{3.3}$$

The $max()$ function in Equation 3.3 is needed since the sub job cannot be released earlier than scheduled release of the original job i. The transition interval can represent a promoted priority, denoted $[a, b]$, or demoted priority, denoted $[a, b)$. A node represents a sub job in the same manner as a node represents a job, i.e., the node is annotated with a continuous interval of its possible execution and a priority, in this case the priority ceiling of the critical section. We define the execution interval, $[\alpha', \beta')$ for the sub job s_i:

$$\begin{aligned} \alpha' &= max(a, r_q) \\ \beta' &= max(b, r_q) + WCET_q \end{aligned} \tag{3.4}$$

That is, the interval, $[\alpha', \beta')$, specifies the interval in which sub job s_i with priority p_s can be preempted by a higher priority job.

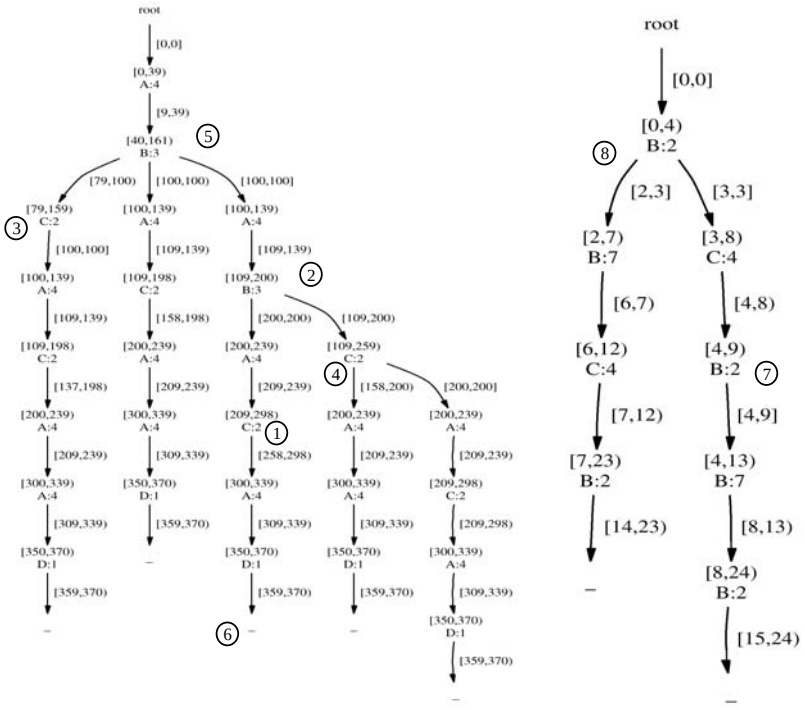

Figure 3.7: The resulting execution order graphs for the job set in Table 3.1 and Table 3.2.

Transition rules

Below are rules for transitions to create a SLCFG, as exemplified and annotated in Figure 3.7. The first six rules correspond to the basic transitions, and the remaining rules are rules for critical sections.

1. **If** the current job j_i completes without preemption, and there are no higher priority jobs that immediately succeeds j_i, **then** add a transition, $j_i \xrightarrow{[a',b')}$, where $[a', b')$ is the interval of possible finishing times of j_i.

2. **If** the current job j_i completes without preemption and a higher prioritized job j_k immediately succeeds j_i, **then** add a transition, $j_i \xrightarrow{[r_k,r_k)} j_k$

Task	*r*	*p*	*WCET*	*BCET*
A	0	4	39	9
B	40	3	121	39
C	40	2	59	49
A	100	4	39	9
A	200	4	39	9
A	300	4	39	9
D	350	1	2	9

Table 3.1: A job set for a schedule with a LCM of 400 ms.

Task	*r*	*p*	*WCET*	*BCET*
B	0	2	4	2
	-	7	4	4
	-	2	9	7
C	3	4	5	1

Table 3.2: A job set for a schedule where job B accesses a shared resource, and when entering the critical section boost its priority to 7. B is split into 3 sub jobs.

, where r_k is the release time of j_k and represents the preemption. In addition, **if** there is a lower prioritized job j_l ready, or made ready during the execution interval of j_i, **then** add a transition, $j_i \xrightarrow{[a'_l, b'_l)} j_l$, where $[a'_l, b'_l)$ is the interval of possible finishing times of j_i.

3. **If** the current job j_i has a $BCET$ such that it definitely is preempted by another job j_k **then** add a transition, $j_i \xrightarrow{[r_k, r_k]} j_k$, where r_k is the release time of j_k and represents the preemption.

4. **If** the current job j_i has a $BCET$ and $WCET$ such that it may either complete or be preempted before any preempting job j_k is released **then** add a transition, $j_i \xrightarrow{[r_k, r_k]} j_k$, where r_k is the release time of j_k and represents the preemption. In addition, **if** the set of ready jobs is empty **then** add a transition, $j_i \xrightarrow{[a_i, r_k)} j_k$, where $[a_i, r_k)$ is the interval of completion times of j_i.

5. **If** the current job j_i has a $BCET$ and $WCET$ such that it may either complete or be preempted before any preempting job j_k is released **then** add a transition, $j_i \xrightarrow{[r_k, r_k]} j_k$, where r_k is the release time of j_k and represents the preemption. In addition, **if** there are lower prioritized jobs j_l ready and $\beta_i > \alpha_i$ holds **then** add a transition, $j_i \xrightarrow{[a'_l, b'_l)} j_l$, where $[a'_l, b'_l)$ is the interval of start times of j_l and a transition, $j_i \xrightarrow{[r_k, r_k)} j_k$, where r_k is the release time of j_k and represents the completion of j_i immediately before j_k.

6. **If** the current job j_i is the last job scheduled in this branch of the tree **then** add a transition, $j_i \xrightarrow{[a'_i,b'_i)} \_$, where $[a'_i, b'_i)$ is the interval of finishing times of j_i.

7. **If** the current sub job s_i succeeded by a higher priority sub job s_j before the release of any higher priority job j_k. That is if $b'_i < r_k$, and $p_j > p_k > p_i$ **then** add a transition, $s_i \xrightarrow{[a'_i,b'_i)} s_j$, where $[a'_i, b'_i)$ is the interval of start times of s_j.

8. **If** the current sub job s_i succeeded by a higher priority sub job s_j before the release of any higher priority job j_k or is preempted by j_k. That is, $a'_i < r_k < b'i$, and $p_j > p_k > p_i$ **then** add a transition, $s_i \xrightarrow{[a'_i,r_k]} s_j$, where $[a'_i, r_k]$ is the possible start interval of s_j. And a transition, $s_i \xrightarrow{[r_k,r_k]} s_j$, where r_k is the release time of j_k and $[r_k, r_k]$ represents the preemption.

9. **If** the current sub job s_i is succeeded by a lower priority sub job s_j before the release of any higher priority job j_k, that is $a'_k < r_k$, **then** s_i is entered into the set of ready jobs and then governed by rule 4 or rule 5, above.

3.4 The algorithm

We will now define an algorithm for generating a System Level Control-Flow Graph ($SLCFG$). Essentially, the algorithm simulates the behavior of a task scheduler that supports scheduling of strictly periodic tasks and exercised with a fixed priority preemptive real-time kernel, complying with the previously defined task model and SLCFG transition rules. The SLCFG for a set of jobs is generated by a call to the algorithm SLCFG (NODE, RDYSET,...) given in appendix A (Listing 1), where NODE is a node that represents the root node of the SLCFG. RDYSET represents the set of tasks that is ready to run and is initially the empty set. The interval is the release interval and is initially , and the considered simulation interval, initialized to $[0, T^{MAX}]$. The algorithm is a recursive function to which the initial arguments are given, as defined above.

In the remainder of this section we will go through the details of the algorithm, the references to line numbers corresponds to the line numbers in Listing 1, Listing 2 and Listing 3 in appendix A

In the algorithm, line 1: we look ahead one job at a time, this is achieved by extracting the release time of the next job. To acquire the next release time that succeeds the currently running job the simulation interval is searched until the next job is found.

In lines 2-6 it is determined if the simulation has come to an end of a control-flow path. This is done by determining the state of the set of jobs ready to execute, if the ready queue is empty and there are no jobs in the simulation interval to put into the ready queue then we have reached the end of a path. Line 6: Draws the end node of the path that corresponds to rule 6. If the simulation is in a state such that that it has not reached the end of a path, line 7-46, we consider if the current job may be preempted, line 13-29, or is definitely not preempted, line 30-46. Rule 1-2 will continue in the non-preemption case while rule 3, rule 4 and rule 5 will continue in the preemption case.

In the preemption case, for rule 4 and rule 5 it must be determined if the current job terminates before the release of a higher priority job, line 14. In those cases that the current job terminates before the release of any higher priority job, it must also be determined if there exists any succeeding lower priority job, line 20, or if any higher priority job immediately succeeds the current job, line 23. Line 27-29 will be visited for rule 3, rule 4 and rule 5 and represents the branch of the preemption of the current job.

Lines 33-34 corresponds to the case when a critical section is entered and the priority is promoted, rule 7. For rule 8, when the current job may enter the critical section before it is preempted there is two outgoing transitions from the current job and are govern by lines 16-17 for the sub job that is entering the critical section and lines 27-29 for the preemption before entering the critical section.

3.4.1 The stop criterion

By enumerating the possible and unique paths in the system control flow we get a measure of the number of system level control flow paths we need to test using the deterministic testing strategy for full coverage. The stopping criterion can be scaled such that it encompass a single task, multiple transactions or all tasks in the system. The above analysis is however pessimistic in the sense that it does not take into account the correlation between actual input and the execution time of a task, this introduces a pessimism such that in practice two tasks may never exhibit their worst case (or best case) execution time during the same system level control flow path. We thus run into the possibility of deriving too many paths that may never be executed in practice.

3.4.2 Conclusion

In this paper we have present a method for deterministic integration testing of strictly periodic fixed priority scheduled real-time systems where synchronization is either resolved using on-line synchronization, complying with the Priority Ceiling Emulation Protocol (PCEP) [3] (a.k.a., the immediate inheritance protocol), or offsets. The paper extends the results by Thane and Hansson [16][14] with handling of online synchronization. This substantially increases the applicability of the method, since it is more likely that errors are caused by synchronization and implementation problems.

Essentially the method is a structural white box testing method applied on the system level rather than on the individual tasks. The method includes a testing strategy where the coverage criterion is defined by the number of paths in the system control flow. The method also includes a reachability algorithm for deriving all possible paths in terms of orderings of task starts, preemptions and completions of tasks executing in a real-time system. The deterministic testing strategy allows test methods for sequential programs to be applied, since each identified ordering can be regarded as a sequential program.

In the presented analysis and testing strategy, we consider task sets with recurring release patterns, and accounted for the effects of variations in start and execution times of the involved tasks, as well as the variations of the arrival and duration of the critical sections.

For future work we plan to introduce deterministic replay technology [15] to testing in order to enforce certain system level control flow paths.

Bibliography

[1] Technical committee on operating systems and application environments of the ieee. portable operating system interface (posix) - part 1: System application program interface (api), 1996. ansi/ieee std 1003.1, 1995 edition, including 1003.1c:amedment 2: Threads extension c language.

[2] N. C. Audsley, A. Burns, R. I. Davis, and K. W. Tindell. Fixed priority pre-emptive scheduling: A historical perspective. In *Real-Time Systems journal*, volume 8(2/3). Kluwer A.P., March/May 1995.

[3] T. Baker. Stack-based scheduling of real-time processes. In *Real-Time Systems Journal*, volume 3(1), pages 67–99, 1991.

[4] B. Beizer. Software testing techniques. In *Van Nostrand Reinhold*, 1990.

[5] R. Cardell-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. In *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, pages 251–261, 1998.

[6] J. Gait. A probe effect in concurrent programs. In *Software - Practice and Experience*, volume 16(3), pages 225–233, Mars 1986.

[7] G. H. Hwang, K. C. Tai, and T. L. Huang. Reachability testing: An approach to testing concurrent software. In *International Journal of Software Engineering and Knowledge Engineering*, volume 5(4), pages 493–510, 1995.

[8] ISO/IEC. Iso/iec 8652l 1995 (e). In *Information Technology - Programming Languages - Ada*, February 1995.

[9] T. K. Iversen, K. J. Kristoffersen, G. K. Larsen, M. Laursen, R. G. Madsen, S. K. Mortensen, P. Pettersson, and C. B. Thomasen. Model-checking of real-time control programs. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 147–255, June 2000.

[10] A. Khoumsi. A new method for testing real time systems. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 441–450, December 2000.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. In *Journal of the ACM*, volume 20(1), 1973.

[12] F. Mueller. Priority inheritance and ceilings for distributed mutual exclusion. In *Proceedings 20th IEEE Real-Time Systems Symposium*, pages 340–349, December 1999.

[13] B. Nielsen and A. Skou. Test generation for time critical systems: Tools and case study. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 155–162, June 2001.

[14] H. Thane and H. Hansson. Towards systematic testing of distributed real-time systems. In *Proceedings of The 20th IEEE Real-Time Systems Symposium*, pages 360–369, 1999.

[15] H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, June 2000.

[16] H. Thane and H. Hansson. Testing distributed real-time systems. In *Journal of Microprocessors and Microsystems*, pages 463–478. Elsevier, 2001.

[17] J. Xu and D. Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. In *IEEE Transaction on Software Engineering*, volume 16(3), pages 360–369, 1990.

[18] R-D. Yang and C-G. Chung. Path analysis testing of concurrent program. In *Information and Software Technology*, volume 34(1), pages 43–56, Jan 1992.

Appendix A

Listing of the System Control Flow algorithm.

```
Algorithm SL_CFG (NODE, RDYSET, [ a, b ), [ Sl, Su ) )
{
1       From the simulation interval [Sl, Su), get the next release time t
2       If the set of jobs ready to execute is empty do
3               Add the job at time t to the queue of jobs ready to execute
4               if the set of jobs ready to execute not is empty do
5                       SL-CFG ( NODE, RDYSET, [ a, b ), [ Sl, Su ) );
                else
6                       Draw the node that represents the end of a trajectory
        else
7               Extract the highest prioritized job j from the ready queue
8               Calculate the execution window for job j, [ a, ß )
9               Calculate the start interval of the next job at time t, [ a', b' )
10              Draw the transition and the node of the current job, NODE->j
11              Add all jobs that have lower priority than the jobs at time t to the ready queue
12              if the job j is preempted by a job at time t then do
13-29           SL_CFG_preemption ( j, RDYSET, [ a, b ), [ a, ß ), [ a', b' ) , [ Sl, Su), t )
                else
30-46           SL_CFG_nopreemption ( j, RDYSET, [ a, b ), [ a, ß ), [ a', b' ) , [ Sl, Su), t )
}
```

List. 1. The listing of the main loop of the System control Flow algorithm.

```
Algorithm SL_CFG_preemption (j, RDYSET, [ a, b ), [ a, ß ), [ a', b' ) , [ Sl, Su), t )
{
13      Extract the next critical section sj from job j
14      If job j completes before the release of a job at time t then do
15              if job j enter a critical section and the priority is promoted then do
16                      Add sub job sj to the queue of jobs ready to execute
17                      SL_CFG ( j , RDYSET, [ a', t ] , [ t, Su ] )
                else
18                      Add sub job sj to the queue of jobs ready to execute
19                      SL_CFG ( j , RDYSET, [ a', t ), [ t, Su ] )
20                      if the set of jobs ready to execute not is empty do
21                              Add the job at time t to the queue of jobs ready to execute
22                              SL_CFG ( j, RDYSET, [t, t), ( t, Su])
23                      else if there are jobs that immediately succeeds job j then do
24                              Add the job at time t to the queue of jobs ready to execute
25                              Add sub job sj to the queue of jobs ready to execute
26                              SL_CFG ( j, RDYSET, [ t, t), (t, Su] )
27      Add the job at time t to the queue of jobs ready to execute
28      Recalculate the execution time for job j
29      SL_CFG ( j, RDYSET, [ t, t], (t, Su])
}
```

List. 2. The listing of the System Control Flow algorithm, the part in which the job may or may not be preempted.

```
Algorithm SL_CFG_nopreemption (j, RDYSET, [a, b), [ a, ß ), [a', b'), [Sl, Su), t)
{
30      Extract the next critical section sj from job j
31      if this is the possible end of the simulation then do
32              Add sub job sj to the queue of jobs ready to execute
33              if job j enter a critical section and the priority is promoted then do
34                      SL_CFG ( j, RDYSET, [ a', b' ], [ b', Su ] )
                else
35                      SL_CFG ( j, RDYSET, [ a', b' ), [ INF, INF ] )
        else
36              if job j enter a critical section and the priority is promoted then do
37                      Add sub job sj to the queue of jobs ready to execute
38                      SL_CFG ( j, RDYSET, [a', b' ], [ b' , Su ] )
39              else if there exists a job that immediately succeeds job j then do
40                      Add the job at time t to the queue of jobs ready to execute
41                      Add sub job sj to the queue of jobs ready to execute
42                      SL_CFG ( j, RDYSET, [t, t), ( t, Su] )
43                      if the job at time t immediately succeeds job j the do
44                              SL_CFG ( j, RDYSET, [ a', b' ), [ t, sr] );
                else
45                      Add sub job sj to the queue of jobs ready to execute
46                      SL_CFG ( j, RDYSET, [ a', b' ), [ t, sr ] );
}
```

List. 3. The listing of the System Control Flow algorithm, the part in which the job completes before the release of a higher prioritized job.

Chapter 4

Paper C: The Asterix Real-Time Kernel

Henrik Thane[1,2], Anders Pettersson[1] and Daniel Sundmark[1,2]
Mälardalen Real-Time Research Centre[1], Sweden,
Zealcore Embedded Solutions AB[2]
{henrik.thane, anders.pettersson, daniel.sundmark}@mdh.se

In Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS'01), Industrial Session.

4.1 Introduction

This paper describes a real-time kernel, Asterix, that in a practical manner makes use of many of the recent advances made in the real-time systems research community. The basic ambition behind the development of the Asterix real-time kernel was to pack state-of-the-art research results into a package such that it can be easily used and understood by people in the embedded systems industry. From an academic point of view the Asterix real-time kernel fulfills all the basic requirements necessary for facilitating different types of timing analyses. For a software designer this signifies that the Asterix real-time kernel has the means to satisfy engineering of real-time software in the same fashion as civil engineers make use of structural calculus when designing bridges or houses. The Asterix real-time kernel is in combination with its support environment in a unique position to provide the embedded systems industry with a development kit that can increase the reliability, safety, and testability of their applications with several magnitudes compared to existing development systems.

From the outset of the development project we decided that the kernel would be distributed as an open source program. For a customer this has several benefits: nothing can be cheaper than free of charge, and risks taken by relying on a small company for providing a real-time kernel can be minimized by having access to the source code. In summary, the kernel packs state-of-the-art features into a package that is all free of charge and open.

Although the Asterix real-time kernel defines the state-of-the-art with respect to other real-time kernels its greatest strengths lies in its open platform and its support by extremely powerful development, and verification tools.

Key features of the Asterix kernel are:

- **The execution strategies**. The Asterix real-time kernel handles execution strategies ranging from strictly statically scheduled systems via fixed priority scheduled systems to event-triggered systems, or any combination of them.

- **Memory consumption**. From an industrial point of view we have during the design of the kernel considered memory consumption and minimized the kernel and the application memory footprints.

- **Monitoring support**. Built in monitoring support makes it possible to use of state-of-the-art testing, and debugging tools like deterministic testing [15] and deterministic replay debugging [16] and visualization. The

kernel also provides facilities for measuring execution times of tasks with a high resolution.

- **Wait and Lock-free communication**. The kernel supports a communication type, using buffers, that decreases the need for explicit synchronization using e.g., semaphores. As a consequence blocking times can be reduced as well as increasing the analyzability of the entire real-time system [4][8][17].

- **Execution time jitter reduction**. The kernel provides a mechanism for minimizing the execution time jitter of individual tasks as well as the jitter originating from the kernel it self. This has been shown to increase the testability of the application enormously, since all executions of the target system will be reproducible [15]. It is also very important for control applications in general to minimize the jitter.

- **Compiling kernel**. The Kernel is compiling which means that the kernel is very resource efficient in terms of allocating memory only necessary for a set of tasks. This means that if the target system only contains 5 tasks, data is only allocated for 5 tasks. If the system contains 127 tasks data is only allocated for 127 tasks. The obvious benefits are that we minimize the overhead, both in memory and in execution time jitter. This overhead is otherwise inherent to all kernels that handle an arbitrary number of application tasks.

- **Exception handling**. The kernel has a well defined exception handling architecture, with different layers of abstraction, separating temporal and functional error handling, as well as system level and user level exceptions.

- **Formally verified**. The kernel design is also in the process of being formally verified. This means that we will be able to provide customers with verified versions of the Asterix real-time kernel.

- **Portable**. The kernel design is such that it will be easy to port to any processor. We will provide test suites for validation of new implementations.

- **Analyzable**. Applications which employs the services provided by the Asterix real-time kernel will be possible to analyze with respect to temporal behavior, synchronization correctness, and proper use of communication mechanisms. This analyzability is not only of importance when

Task	*Period*	*Offset*	*Priority*	*Deadline*
A	400	0	4	100
B	400	40	3	400
C	400	40	2	400
A	400	100	4	200
A	400	200	4	300
A	400	300	4	400
D	400	350	1	400

Table 4.1: A static schedule for a period/LCM of 400 ms.

Task	*Period*	*Offset*	*Priority*	*Deadline*
A	100	0	4	100
B	400	40	3	400
C	400	40	2	400
D	400	350	1	400

Table 4.2: A fixed priority schedule. Same system as in table 4.1.

developing safety critical application it is also a desirable property which can be used to shorten the time to market. The reason is that the Asterix kernel enables analysis of designs in an early phase of a project and thereby avoids costly and time-consuming re-designs in a late phase of the project because of lack of computational resources.

Document outline. We will in the remainder of this document describe the key features of the Asterix real-time kernel in more detail. We begin with the execution strategy, and continue with monitoring mechanisms, and jitter reduction.

4.2 The Asterix Execution Strategy

The execution strategy of the Asterix real-time kernel is based on fixed priority scheduling with support for preemption. This means that the kernel is multitasking, i.e., multiple tasks can share the same computing resource (CPU), where the task with the highest priority executes. Preemption, or task interleaving, means that an executing task can be preempted during its execution by

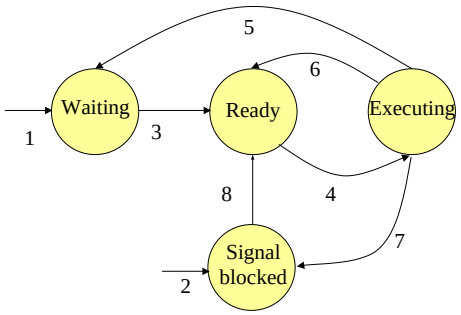

Figure 4.1: The states and transitions possible in the Asterix real-time kernel.

another task, and then allowed to resume after the completion of the preempting task. The kernel supports both periodic time triggered tasks and aperiodic event triggered tasks [7]. All tasks are of terminating character, which means that when a task has completed its execution, it terminates and waits until a new period or a new event occurs. This has the benefit of decreasing the coupling in the system by abstracting the reactivation of the tasks away from the source code. The responsibility for reactivation is left to the kernel and the schedule where analyses are easier to apply. A task in Asterix is defined by its:

- **P** - *Priority* (which is unique),
- **T** - *Period-time* (for aperiodic tasks the period-time is not defined),
- **O** - *Offset* (a periodic task can delay its reactivation with an offset relative its period-time),
- **BCET** and **WCET** (the *best case execution time* and *worst case execution time*), and
- **DL** - *Deadline*.

Depending on how we define these task attributes the Asterix real-time kernel can be configured to run a static schedule, i.e., a predefined timetable [18, 9] (Table 4.1), or periodic tasks according to Fixed Priority Scheduling [1, 2] (Table 4.2). A system can also be defined as event triggered by not giving period

Transition	*Cause*
1	Periodic task starts
2	Aperiodic task starts
3	A task is ready to execute due to period start
4	The scheduler decides to start the highest priority task available in ready and Executing.
5	When a periodically executing task has terminated.
6	When the executing task is preempted.
7	When an executing task is suspended until a signal occurred.
8	When an aperiodic task is triggered by a signal.

Table 4.3: The transitions in the Asterix real-time kernel.

times of the tasks and by defining activator signals. Due to this general execution strategy, we can mix any of the different execution paradigms. We can for example have statically scheduled systems where some tasks are event triggered [12] or fixed priority scheduled. The approach of assigning priorities to tasks even for statically scheduled timetables gives robustness since lower priority tasks cannot disrupt the execution of higher priority tasks if they fail and run berserk.

Every tasks in the Asterix kernel can be in four states: Waiting, Signal blocked, Ready, and Executing. The states and valid transitions between them are illustrated in Figure 4.1 and exemplified in Table 4.3.

4.2.1 Synchronization

In the Asterix real-time kernel we can synchronize tasks in two distinct ways: on-line, in the source code, using semaphores and signals, and off-line, in the schedule, using offsets. The decision to allow both types of synchronization mechanisms was that they have mutually exclusive benefits depending on the type of problem to be solved. That is for certain problems the solution or analysis would be more complex if we used e.g., offsets than semaphores and vice versa. In the Asterix real-time kernel we allow both types of synchronization to be used at the same time. For multitasking systems the use of semaphores is notorious due to the possibility of deadlocks and starvation caused by priority inversion. Semaphores have proven to cause many intricate problems and elu-

sive bugs. However, depending on the actual algorithm used to implement the semaphore synchronization mechanism we can eliminate deadlock and starvation situations. This can be achieved by using priority ceiling algorithms. In the Asterix real-time kernel we have implemented a very simple, memory conservative and predictable algorithm, the Immediate Inheritance Protocol. Another benefit of this protocol in conjunction with the terminating character of all tasks is that we can make use of a single stack, and thus decrease memory use.

4.2.2 Communication

In the Asterix real-time kernel we provide a mechanism called wait and lock-free communication (WLFC). This type of shared memory communication allows tasks to communicate with each other without any blocking, that is, the need for explicit synchronization using e.g., semaphores is eliminated. The shared memory communication is of simplex type and based on a set of buffers that can be read by a set of tasks and written to by *one* task. The reading tasks are guaranteed that the value they read is non-volatile during their execution, i.e., atomicity is guaranteed. The writing task is also guaranteed a free buffer for writing. The wait-free in WLFC means that a reading task does not have to wait for the latest produced value; it will always be available for the receiver. When a task A, starts to execute it is given a reference to the latest written value by a producer, P. This value is guaranteed to be unmodified during the execution of task A, even if it is preempted by the writer, P, and P produces a new value. However if now a second reading task, B, preempts A after P has written a new value, task B will at its start receive a reference to this new latest value.

The cost for this type of communication is memory. The number of buffers need for each wait and lock-free communication channel is $no\_buffers = no\_readers + 2$. If a software designer feels that the memory needed for WLFC is too costly then the designer can resort to shared memory and use semaphores for synchronization. Another option is if the communicating tasks run with the same periodicity then we can make use of offsets for synchronization and just use one memory buffer.

Wait and lock-free communication is superior for systems where communication between asynchronous/multi-rate tasks occur [4][8]. In addition WLFC gives decreased blocking times and reduced scheduling complexity. All data transfer is also performed by the tasks themselves, not by the kernel, which decreases kernel overhead and jitter. The data transfer overhead is debited to

the tasks involved in the transaction, and therefore subject to execution time estimation.

4.2.3 Hard and soft tasks

The tasks in the Asterix real-time kernel are divided into two categories: Hard tasks and soft tasks. The difference between the tasks are that the hard tasks are required to have passed a schedulability analysis (see Section 4.2.5), while the soft tasks have no such requirement. Since we can mix both types of tasks in a system we must guarantee that the soft tasks cannot disrupt the execution of the hard tasks. This guarantee is fulfilled by statically assigning priorities to soft tasks that are all lower than the priorities of the hard tasks. In order to guarantee that no soft task can block a hard tasks the semaphore mechanism is devised such that the set of semaphores used by the soft tasks are disjunct with the set of semaphores used by the hard tasks. These sets are defined off-line, and are actually required in order to set the correct priority ceilings in the immediate priority inheritance protocol. If a soft task under suspicious circumstances still would access a hard semaphore an exception handler would be invoked and the situation would be detected.

4.2.4 Pre-runtime configuration

As the Asterix real-time kernel is compiling we need a means to specify the constitution of the system and to initialize all data structures describing the tasks and their attributes. This is done in a configuration file, which upon execution outputs the necessary data structures that can be complied together with the application code and the kernel. Figure 4.2 illustrates a configuration file.

4.2.5 Timing analysis

Timing analysis is performed at two levels, at the task level and at the system level [6][11].

At the task level the worst case execution time for each task is analyzed or estimated. This analysis is comparably simple on Asterix compared to do on tasks running on traditional RTOS such as WxWorks and QNX since Asterix requires terminating tasks. Further, since a task cannot be blocked after it has entered the state execution (only pre-empted), the worst case execution time can be calculated or measured for the code of task in isolation.

```
SYSTEMMODE = NORMAL;
RAM = 512000;
  MODE mode_1{
    RESOLUTION = 1000;
    HARD_TASK ht_1{
         ACTIVATOR     = 100; //period time
         OFFSET        = 0;
         DEADLINE      = 50;
         PRIORITY      = 10;
         STACK         = 50;
         ROUTINE       = ht_1_routine;
         ARGUMENTS     = "1, 2, 3";
         ERR_ROUTINE   = ht_1_error_routine;};
    HARD_TASK ht_2{
         ACTIVATOR     = 50; //period time
         OFFSET        = 0;
         DEADLINE      = 20;
         PRIORITY      = 20;
         STACK         = 50;
         ROUTINE       = ht_2_routine;};
    SOFT_TASK st_1{
         ACTIVATOR     = sig_1; //trigger signal
         OFFSET        = 0;
         DEADLINE      = 10;
         PRIORITY      = 10;
         STACK         = 50;
         ROUTINE       = st_1_routine;};
    WAITFREE w_1{
         WRITER        = ht_1;
         READER        = ht_2;
         NUM_BUF       = 3;
         TYPE          = "my_type";};
    SIGNAL sig_1{
         USER          = ht_1;
         USER          = st_1;};
    SEMAPHORE sem_1{
         USER          = ht_1;
         USER          = ht_2;};
  };
```

Figure 4.2: The configuration of a system.

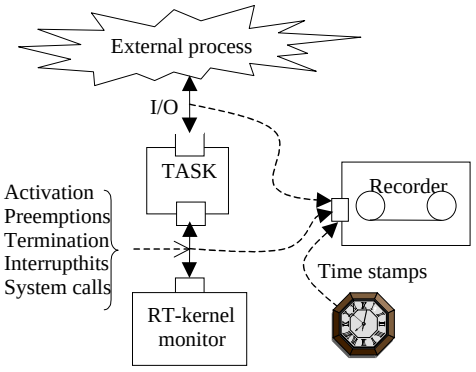

Figure 4.3: Kernel monitoring and recording.

At system level we analyze if the composed system fulfil its timing requirements by using either fixed priority analysis or a pre-runtime scheduler. Both kind of analysis is mature and proven to be useful in industrial applications [10][3].

When designing a system we can assign time budgets [5] to the tasks that are not implemented by intelligent guesses based on experience. By doing this we gain two positive effects. First, the system level timing analysis can be done before implementation and hence we have a tool for estimating the performance of the system. Second, the time budgets can be used as an implementation requirement.

By applying this approach we make the design process [5] less adhoc with respect to real-time performance. That is, the first time one can find timing problems in traditional system design is when the complete system or subsystem has been implemented. If a timing problem is found adhoc optimization starts which most surely will make the system difficult to maintain.

4.3 Monitoring

In the Asterix real-time kernel we have unique support for observations [13] of the target application, i.e. an event recorder, Figure 4.3. These observations can be used for execution time measurements of the application tasks, but most significantly these observations can be used for visualization, deterministic re-

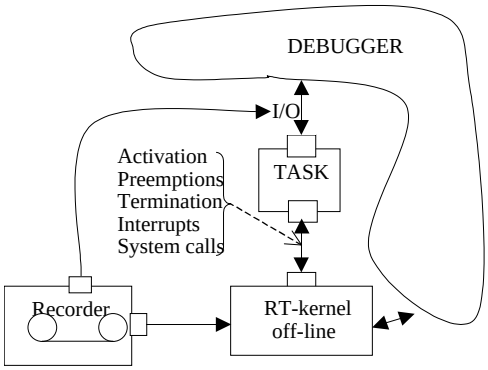

Figure 4.4: Offline kernel with debugger.

play debugging and deterministic testing of the target system.

4.3.1 Deterministic replay

Deterministic replay is a software based technique proprietary to the Asterix development environment that allows reproducible debugging of single tasking, multi-tasking, and distributed real-time systems [16]. During runtime, information is recorded with respect to interrupts, task-switches, timing, and data. The system behavior can then be deterministically reproduced off-line using the recorded information, Figure 4.4. A standard debugger can be used without the risk of introducing temporal side effects, and we can reproduce interrupts, and task-switches with a timing precision corresponding to the exact machine instruction at which they occurred. The technique also scales to distributed real-time systems, so that reproducible debugging, ranging from one node at a time, to multiple nodes concurrently, can be performed.

4.3.2 Deterministic testing

For testing of sequential software it is usually sufficient to provide the same input (and state) in order to reproduce the output. However, for real-time systems it is not sufficient to provide the same inputs for reproducibility - we need also to control, or observe, the timing and order of the inputs and the concurrency of the executing tasks. Based on the monitoring mechanisms built into the

Asterix real-time kernel and a proprietary testing method found in the Asterix development environment deterministic testing of the target application can be easily performed. The testing method includes an analysis technique that given a set of tasks and a schedule derives all execution orderings that can occur during run-time [13]. The method also includes a testing strategy that using the derived execution orderings can achieve deterministic, and even reproducible, testing of real-time systems. Each execution ordering can be regarded as a sequential program and thus techniques used for testing of sequential software can be applied to real-time system software. The analysis and testing strategy can also be extended to encompass interrupt interference, distributed computations, communication latencies and the effects of global clock synchronization.

4.4 Jitter reduction

The kernel provides a mechanism for minimizing the execution time jitter of individual tasks as well as the jitter originating from the kernel it self. This has been shown to increase the testability of the application enormously, since all executions of the target system will be reproducible [15][14]. It is also very important for control applications in general to minimize the jitter.

Figure 4.5 illustrates the possible execution orderings for a schedule running on the Asterix real-time kernel without jitter reduction. That is all the possible task starts, task preemption and task terminations yielded by the varying execution times of tasks and their varying start times due to delay caused by higher priority tasks. Figure 4.6 illustrates the same system with jitter reduction turned on.

From a verification perspective the fewer the scenarios the system exhibit the better the possibilities are for testing and debugging the system since the number of behaviors to consider is reduced. In fact there is an exponential relation between the jitter in the system and the number of execution scenarios. In other words the testability of a system is enormously influenced by the jitter. The potential testability, and therefore indirectly the reliability, of the system benefits greatly by the jitter reduction techniques used in the Asterix real-time kernel. Typical testability gains are in the range of billions for an industrial application with modest complexity and even more for more complex systems.

This feature makes the Asterix real-time kernel suitable for use in mission critical, and safety critical applications, or simply in applications where the funds are limited but the desire is to get more-bang-for-the-buck (reliability).

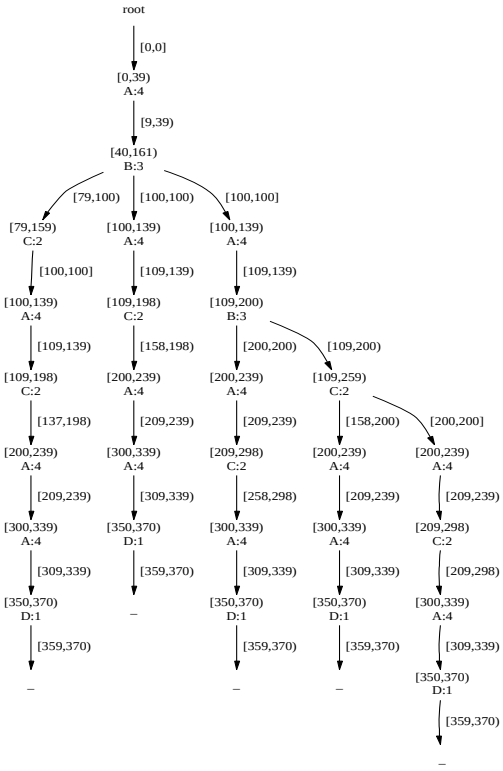

Figure 4.5: The possible execution ordering scenarios for a system with jitter.

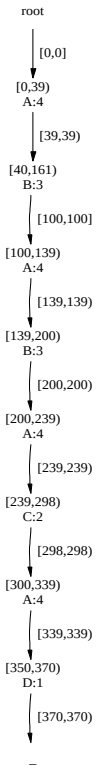

Figure 4.6: No jitter.

4.5 Conclusions

In this paper have we presented a novel real-time kernel Asterix which support development of hard real-time systems. We have presented the supported execution strategy, the monitoring and testing facilities, and a mechanism for jitter reduction. We are currently in the process of adopting Asterix to different micro controllers.

Bibliography

[1] N. C. Audsley, A. Burns, R. I. Davis, and K. W. Tindell. Fixed priority pre-emptive scheduling: A historical perspective. In *Real-Time Systems journal*, volume 8(2/3). Kluwer A.P., March/May 1995.

[2] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings of 8th IEEE Workshop on Real-Time Operating Systems and Software Real-Time Systems journal*, pages 127–132, Atlanta, Georgia, May 1991.

[3] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano a revolution in on-board communications. Technical report, 1998.

[4] J. Chen and A. Burns. Asynchronous data sharing in multiprocessor real-time systems using process consensus. In *Proceedings of 10th Euromicro Workshop on Real-Time Systems*, June 1998.

[5] C. Eriksson, J. Mäki-Turja, K. Post, M. Gustafsson, J. Gustafsson, K. Sandström, and E. Brorsson. An overview of rtt: A design framework for real-time systems. In *Journal of Parallel and Distributed Computing*, volume 36, pages 66–80, October 1996.

[6] M. Joseph and P. Pandya. Finding response times in a real-time system. In *The Computer Journal - British Computer Society*, volume 29(5), pages 390–395, October 1986.

[7] H. Kopetz. Event-triggered versus time-triggered real-time systems. In *Lecture Notes in Computer Science*, volume 563, Berlin, 1991. Springer Verlag.

[8] H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: A solution to a real- time synchronization problem. In *Proceedings of he 14th Real-Time Systems Symposium*, pages 131–137, 1993.

[9] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. In *Journal of the ACM*, volume 20(1), 1973.

[10] C. Norström, K. Sandström, M. Gustafsson, J. Mäki-Turja, and N.-E. Bånkestad. Experiences from introducing state-of-the-art real-time techniques in the automotive industry. In *Proceedings of 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS01)*, Washington, US, April 2001. IEEE Computer Society.

[11] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. In *Journal of Real-time systems*, volume 1(2), pages 159–176. Kluwer A.P., September 1989.

[12] K. Sandström, C. Eriksson, and G. Fohler. Handling interrupts with static scheduling in an automotive vehicle control system. In *Proceedings of the 5th Internationell Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, Japan, October 1998.

[13] H. Thane. Design for deterministic monitoring of distributed real-time systems. In *Technical report*. Mälardalen Real-Time Research Centre, Dept. Computer Engineering, Mälardalen University, 1999.

[14] H. Thane and H. Hansson. Handling interrupts in testing of distributed real-time systems. In *Proceedings of the Real-Time Computing Systems and Applications Conference (RTCSA'99)*, Hong Kong, December 1999.

[15] H. Thane and H. Hansson. Towards systematic testing of distributed real-time systems. In *Proceedings of The 20th IEEE Real-Time Systems Symposium*, pages 360–369, 1999.

[16] H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, June 2000.

[17] K. W. Tindell, A. Burns, and A. J. Wellings. Analysis of hard real-time communications. In *Journal of Real-Time Systems*, volume 9(2), pages 147–171, September 1995.

[18] J. Xu and D. Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. In *IEEE Transaction on Software Engineering*, volume 16(3), pages 360–369, 1990.

Chapter 5

Paper D: Experimental Evaluation of a Test Procedure for Deterministic Testing of Real-Time Systems: Technical Report

Anders Pettersson and Henrik Thane
Department of Computer Science and Engineering
Mälardalen University, Västerås, Sweden
{anders.pettersson, henrik.thane}@mdh.se

Abstract

In this paper we present an experimental evaluation of the deterministic test method for real-time system software as presented by Thane et al. In order to evaluate the method we have built a test bed and on that test bed we have tested a number of real-time applications. Specifically we have evaluated the relation between the theoretically derived coverage criterion and the actually observed coverage during test. The evaluation shows that coverage criteria can be derived but it is still hard to achieve complete coverage. It also shows that the derived coverage criteria depend on the accuracy of estimated execution times.

5.1 Introduction

In this paper we present an experimental evaluation of the deterministic test method for real-time system software as presented by Thane et. al. [9]. Specifically we have evaluated the relation between the number of theoretically derived system level control-flow paths (SLCFP) and the number of actually observed system level control-flow paths. We define the system level-control flow paths in accordance with Thane et. al. [9] as every unique sequence of task-switches during a specified duration of time.

In order to facilitate this evaluation we have developed a number of programs:

- An analysis tool that derives all possible SLCFP, which defines the theoretical coverage criterion.
- A test case generation tool that uses the derived SLCFP for achieving deterministic testing.
- A test bed complying with the method
- A set of programs (real-time applications) to test.

We have applied the analysis tool on a set of randomly generated multitasking real-time applications and showed the relation between the derived SLCFP and the observed SLCFP for each generated application. All real-time applications are schedulable according to the rate monotonic scheduling policy. The tasks in the real-time applications are assumed to not communicate with each other.

The analysis tool derives all possible SLCFP, which are stored at the test server. During the test runs, test cases applied on the real-time applications running on the test bed, the SLCFP are extracted and sent to the test server for comparison to the derived SLCFP. The test server is keeping track of the number of each unique SLCFP and the number of observations of each unique SLCFP in order to establish the fulfillment of the test coverage criteria.

The remainder of this paper is organized as follows: in Section 5.2 we present a procedure for deterministic testing of real-time systems. Furthermore, in Section 5.3 an analysis tool for deriving the system level control-flow is discussed. Followed by a description of the test bed in Section 5.4. The result from the experimental evaluation is discussed in Section 5.5, and the paper concludes in Section 5.6 and Section 5.7 with a summary and a brief discussion of the continuation of this work.

5.2 The Test Procedure

The evaluated test method, presented by by Thane et al. [9], consists of a number of steps:

1. identify the set of possible system level control-flow paths (serializations),
2. test the system using any test technique of choice,
3. map each test case and output onto the correct control flow path based on run-time observations, and
4. repeat 1-3 until required coverage is achieved.

In step 1 the test coverage criterion is derived. During step 2 test runs are applied on the application in order to log input, output and the control-flow. In step 3 the logged information is compared to the output from step 1.

5.3 Analysis of Real-Time Systems

In this section we discuss the analysis tool we have used in this evaluation. The analysis tool derives all possible system level control-flow paths (SLCFP), i.e. the tool is an off-line analysis tool. With off-line analysis we mean two things, (1) pre-analysis of the *system level control-flow* (SLCF) and (2) post-analysis of the SLCF extracted during test runs.

5.3.1 Pre-Analysis Tool

The pre-analysis tool simulates the execution behavior in a real-time system. Basically, the tool simulates the run time behavior of a fixed priority scheduler based on the temporal attributes of the tasks. In our task model a task is defined as $< T, O, DL, P, BCET, WCET >$, where:

- T is the period time of the task,
- O is the release time offset,
- DL is the deadline of the task,
- P is the priority of the task,

- $BCET$ is the best case execution time,
- $WCET$ is the worst case execution time.

As a consequence, the bounds on the tasks' execution times must be known á priori.

Period time, offset, deadline and priority is determined at design time while the execution time is more difficult to estimate. We have used an approach to estimate the execution time of tasks similar to the one used for Real-Time Talk [3], in which the design method allows a coarse estimation of execution times based on time budgets that are stipulated in the early design.

The System Level Control-Flow Analysis Tool

The SLCF-tool relies on the use of the Fixed Priority Scheduling [1] policy of strictly periodic tasks or static scheduling [11, 5]. Based on stipulated time budgets and the task model, the pre-analysis tool can be used to estimate the number of control-flow paths early in the design phase. To not obstruct the scheduling and the SLCF analysis the functionality of the tasks must be implemented such that the time budgets are met.

The output from the analysis tool is a *System Level Control-Flow Graph* (SLCF-graph). In which each branch corresponds to a single system level control-flow path of the real-time system. Also each branch represents a finite duration of time (in most cases the least common multiple of the tasks' period times). Each node in the graph represents the execution of a task and each edge represents the transition from one task to another, i.e. a task-switch. The start of a tasks is affecting the control-flow if the start of a task leads to a task-switch to another task (thus producing more than one outgoing transition from a node). The termination of a task is affecting the control-flow if a task start immediately succeeds the termination.

The number of control-flow paths can be roughly estimated by 3^{p*n}, where 3 is the maximum number of outgoing transitions from a node, p is the number of preemption points and n is the number of task instances.

One of the purposes of the analysis tool is to derive a coverage criterion. To establish the coverage criterion it must be possible to derive all possible control-flow paths. However, this may be impossible because of the state space explosion during the analysis. The reasons for this state space explosion are contributed by long intervals of time (increased n) or that the start times of tasks are such that tasks frequently preempt each other (increased p).

Furthermore, synchronization of communicating tasks is resolved off-line by separating the task in time using offsets and/or priority [9]. The input data to receiving tasks are available at the start of the receiving task and data sent by tasks are available at the termination of the sending task.

A detailed description of the SLCF-graph and the rules for constructing the SLCF-graph can be found in Thane et al. [9].

5.3.2 Post-Analysis Tool

During run-time, the traversed control-flow paths are continuously logged. The log file is parsed by the post-analysis tool in order to derive the control-flow paths that occurred during the test runs.

Each control-flow path that is extracted from the log file is compared to the control-flow paths derived in the pre-analysis. If a match is found between the compared control-flow paths, then this control-flow path is marked as unique, or if the control-flow path has been previously matched, a counter for the number of occurrences of this control-flow path is increased. By increasing the counter for every unique control-flow path discovered and increasing the counter for how many times each control-flow path have been traversed the test coverage is tracked.

5.4 Test Bed

Our test bed consists of three main components *off-line analysis host*, *test server* and *target system* (see figure 5.1). Where the off-line analysis host and the test server are running on the same physical computer. In addition to the off-line analysis host, we have used another host on a separate physical computer to generate task sets and to perform schedulability analysis of the task sets.

The off-line analysis host is the computer on which the system level control-flow analysis of the real-time software is performed. The test server communicates with the off-line analysis host by sending the control-flow paths that occurred during the test runs and receiving the control-flow paths from the target system. The test results are handled according to the test method, see section 5.2.

The test server distributes test cases and receives the set of control flow paths that occurs during the test runs. Also, on the test server the test oracle is running. The test procedure does not rely on any particular communication

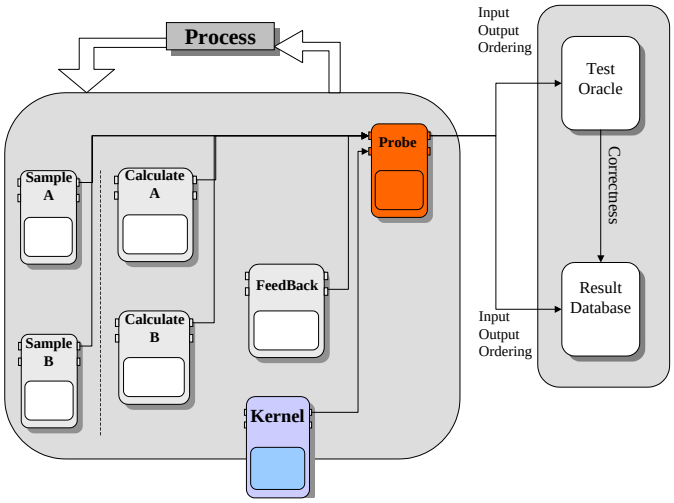

Figure 5.1: The figure shows the test bed used for the experimental evaluation. The test server and the real-time system nodes are connected via a Local Area Network.

protocol or architecture for sending the run-time information to the test server. It must however, fulfill a basic requirement: when a message is sent, it must be guaranteed that the test server receives the message. This is achieved by letting the test server keep track of the received messages based on their sequence numbers.

The system level control-flow paths are extracted during the test runs and continuously sent to the test server node by a probe task. When the test coverage criteria are met or when a large enough number of test cases have been exercised the target is rebooted in order automate the testing. Rebooting the target allows the test server to allocate the target for test runs of other applications or the same application with different configurations.

5.4.1 The System Under Test

The system under test is a multi-rated multi-tasking real-time software conforming to the assumptions in section 5.3 running on the target node. The test procedure relies on the existence of non-intrusive instrumentation techniques on the target. For this purpose we have used the *Asterix real-time kernel* [10]

```
void kernel_TASK taskswitch ( void )
  {
      probe_control_flow ( old_task );
      switch_out ( old_task );
      schedule ( task_set );
      switch_in ( new_task );
      probe_control_flow ( new_task );
  }
```

Figure 5.2: Example of a software probe inserted in order to monitor the task-switch.

that supports software based instrumentation.

5.4.2 Instrumentation

Figure 5.2 shows an example of software probes inserted in order to instrument the system level control-flow. The control-flow paths are temporarily stored in a circular buffer. Since there is a space limitation there is a need to send the extracted information to the test server before any of the information is overwritten. In our experiment the information is sent periodically to the test server by a low prioritized probe task. The probe task has a periodicity such that there is always room for new entries and such that higher prioritized tasks do not block the probe task so that inconsistency occurs in the buffer.

The probe effect [4] is solved by leaving the instrumentation probes in the code during normal operation of the system. This will increase the resource utilization of the system, but the execution behavior will remain unchanged.

5.4.3 Information Extraction

For extraction of the information from the target we are using UDP/IP over an Ethernet link. UDP is a connectionless communication protocol without any transmission guarantees. This is not in compliance with our basic requirement on communication, but since we in our evaluation only have two nodes, we have moved the responsibility of lost messages to the test server and hence we fulfil the requirement.

5.4.4 Hardware

The hardware used for the target platform is a PC-104 board equipped with Intel 80486 CPU. The PC-104 board is a PC compliant board often used in embedded systems in which the physical size is of no matter. The board consists of, besides the CPU, built-in graphic devices, an Ethernet controller and I/O controllers.

5.5 Experimental Results

5.5.1 Task Set Generation

The real-time application used in this evaluation consists of randomly generated task sets, which all are schedulable according to the rate monotonic scheduling policy [5]. We have 200 task sets consisting of 5 to 8 tasks. The attributes of the tasks generated are: period time, worst case execution time and best case execution time. The period times of the task is randomly selected from the values 40, 50, 60, 70 and 80 ms. This choice was made for two main reasons:

- to get a low value as possible of the least common multiple of the tasks' period times, but yet achieve a task set that is prone to trigger preemptions, and
- to be able to chose execution times of tasks so that it is possible to introduce execution time jitter.

The execution time of the task is based on the best execution time and the worst execution time. In Figure 5.3 an application task is shown, in which the best execution time is set to a constant and a varied execution time that is randomly chosen between the best execution time and the worst execution time, minimum 2 ms and maximum 10 ms execution time. Different tasks are forced to take different control-flow paths by controlling the input, $BCET$ and $WCET$, to the tasks. However, the software cannot be forced to traverse a unique control-flow path.

We have divided the set of tasks into two groups of 100 task sets. One group represents the sets with significant idle time, and the other group the sets with little idle time. Idle time is the slack in the system where the idle task is executing.

```
void sim_sample(void *ignore)
{

  /* Here the task consumes execution time without
     introduced jitter */
  poll_timer_hwticks ( BCET );

  /* Here the task consumes varied execution time
     that introduce jitter. */
  poll_timer_hwticks ( 0 + (unsigned long)(WCET-BCET
                       * genrand() )   );

}
```

Figure 5.3: Example of a application task with introduced randomly generated execution time jitter.

In our task sets we assume that the tasks are not communicating with each other, because in the evaluated test procedure [8] it is not explicitly stated how to handle invalid cotrol-flow paths caused by application design or execution time estimations. Since we randomly generate the task set and the tasks' execution times we cannot guarantee these properties.

5.5.2 Results

During the test runs of the real-time applications we have exercised 100 and 1000 test cases on each task set and compared the number of found control-flow paths traversed during the test runs with the number of derived control-flow paths for each task set. Graph 5.4 and graph 5.5 illustrate the number of derived and exercised control-flow paths during 1000 test runs for all task sets.

During the test runs we have observed derived control-flow paths, which are never exercised. This phenomenon can be seen in graph 5.6 and graph 5.7, task sets 60 to 80. Despite the number (up to 1000) of derived control-flow paths in some task sets there are very few (1 to 200) control-flow paths that are exercised.

Properties that have impact on the number of exercised control-flow paths are the amount of execution time jitter of a task, the preemption by higher prioritized task and the number of instances of a task.

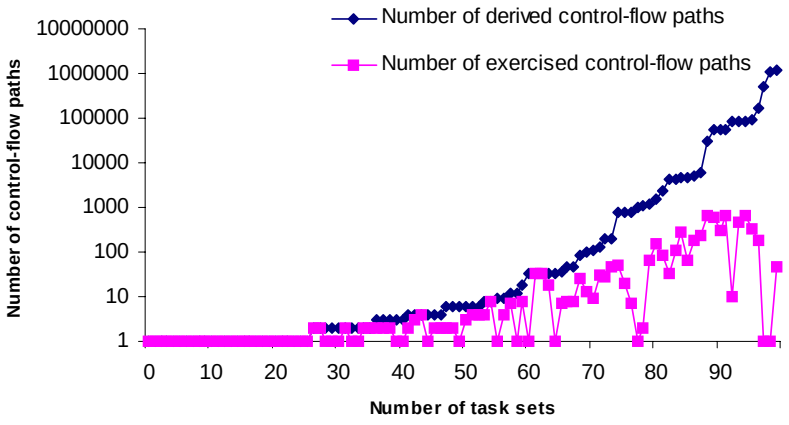

Figure 5.4: The graph shows the number of derived and exercised control-flow paths during 1000 test runs of each task set. It can be seen how many of the exercised control-flow paths that are unique. In this graph the task set is considered to have a significant execution time slack.

The number of instances of a task will not affect the results in the comparison between the derived and exercised control-flow paths since the same number of task instances is run both during analysis and during runtime. The number of preemption points will significantly affect the number of exercised control-flow paths in the following ways:

1. The tasks execution time is such that preemption do not occur, and
2. The execution time jitter during the test runs is less than in the stipulated execution time budgets.

For situation 1 our task sets are created to give a high possibility of preemption. But, there are some scenarios that are unlikely to occur during the test runs. In the analysis, different control-flow paths are created if a higher prioritized task is immediately succeeding the currently executing task. However, during run-time of the application this scenario is triggered by a preemption at the last instruction in the execution of the previous tasks (i.e., the last instruction is the last instruction of the task-switch). In our experiment we have not had such control over the execution so that we could trigger such scenario

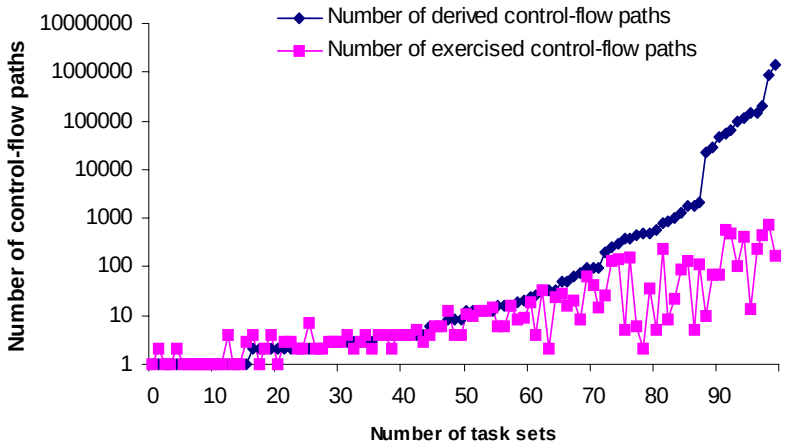

Figure 5.5: The graph shows the number of derived and exercised control-flow paths during 1000 test runs of each task set. It can bee seen how many of the exercised control-flow path that are unique. In this graph the task set is considered to have a low execution time slack.

and it is not desirable to have that control. Even if this scenario is unlikely to occur in applications it is however possible. This observation implies that the coverage could be increased by forcing the program to traverse those scenarios.

In 2, the tasks in the task sets have execution times ranging the whole span between $BCET$ to $WCET$. We have used a linear distribution of the randomly generated execution time jitter. Although, a different result may have been achieved by using a distribution in which we have had more control over the execution times.

In the graphs 5.4 and 5.5 it can be seen that in some task sets the number of traversed control-flow paths are greater than the number of derived control-flow paths, that is, the percentage of found control-flow paths during the test runs exceeds 100%. This has two main explanations: (1) The stipulated execution time budgets set in the early stage of the development must be optimistic. Setting to narrow budgets leads to that the analysis tool reports to few possible control-flows. (2) It is difficult to estimate the execution time of tasks because of the introduced delay by the hardware architecture such that the time budget is exceeded. In the experiment, task sets that do not exceed 100% in the comparison have execution times within their time budgets. This emphasize the

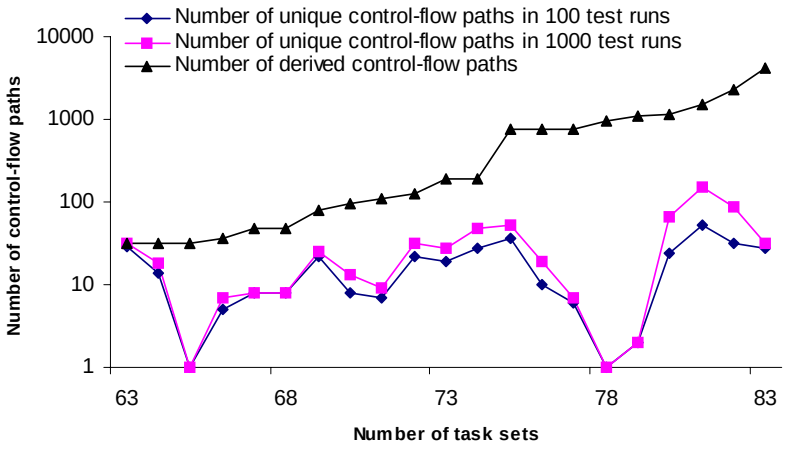

Figure 5.6: This graph shows the number of found system level control-flow paths in 1000 test runs and 100 test runs compared to the number of derived paths in task sets that are considered to have significant execution time slack.

importance of method for accurate estimations of execution times.

During the test runs we found that the original analysis algorithm did not catch all control-flow paths, so in this case study a revised version of the analysis algorithm have been used (see tech note [6]). The control flow paths that was not identified was the case when we have a normal transition (transition rule 1 in Thane et al. [8]) and if a low prioritized job is ready to run before the release of a higher priority job. I.e., the original algorithm looked too far ahead, to the next higher prioritized job. Now we look only until the point in time where a job that affects the control-flow is released, despite the level of priority.

5.6 Conclusions

In this paper we have presented:

- A test bed for automated testing of real-time systems.
- An evaluation of a analysis method used in a test procedure for deterministic testing of multi-tasking real-time systems.

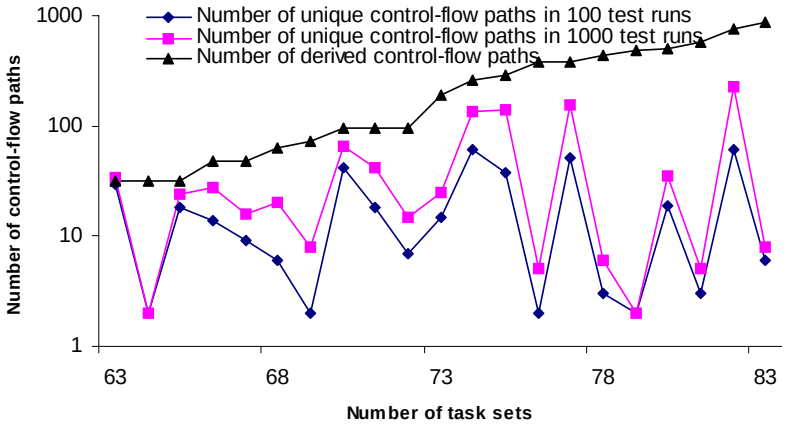

Figure 5.7: This graph shows the number of found system level control-flow paths in 1000 test runs and 100 test runs compared to the number of derived paths in task sets that are considered to have low execution time slack.

The experimental evaluation shows that even if the number of test runs for each test case is increased it is still hard to obtain complete test coverage. We have also shown the usefulness of the ability to analyze the complexity of the software and how to use the information in a testing procedure in order to achieve deterministic testing of real-time systems. Through experiments we have shown the relation between execution time jitter and the complexity of testing of fixed priority scheduled real-time systems. Hence, pointing out the importance of reducing execution time jitter when designing software for high testability.

5.7 Future Work

The original idea of the case study was to use COTS-tools (Matlab, Simulink and Real-Time Embedded Coder) for designing and analysis of real-time systems. We auto generated source code for a control application that controlled an autonomous six-legged robot to be used in rocky terrain. However, we found the following properties of the automated generated C-code that did not add more information than we achieved by the generated C-code with randomly

generated execution time:

- The generated C-code use libraries where the execution time properties are not documented.
- Customizing the generated code is as time consuming as coding from scratch.

Also, the applications resulted into very few tasks 1-2. Instead we decided to choose a solution where we had more control over the generated code. Hence, there is no result from analysis of application modeled with COTS-tools for real-time systems. In future work we will investigate control applications modelled with an extension of MATLAB/SIMULINK by Cervin et al. [2].

We believe it would be of interest to seek a threshold of the number of test runs at which further test runs do not increase the probability to find new unique control-flow paths. This allows design of multi-tasking software to have high testability.

The result in this evaluation is highly dependent on how the randomly generated task sets are created. In future work we will also look at how different distributions of the random numbers will affect the result. In other words having more control of the span of the execution time jitter.

The system level control-flow analysis tool that is evaluated in this paper have been extended [7] to handle synchronization during run time. One obvious extension of this experimental evaluation would be to include synchronization of communication task.

Furthermore, the original system level control-flow analysis tool could handle distributed systems, so an evaluation that includes a system with more than one target node should be performed. However, to analyze distributed systems we think that it is first necessary to reduce the complexity by applying data communication constraints on the system level control-flow graphs in order to reduce the number of valid control-flow paths.

Bibliography

[1] N. C. Audsley, A. Burns, R. I. Davis, and K. W. Tindell. Fixed priority pre-emptive scheduling: A historical perspective. In *Real-Time Systems journal*, volume 8(2/3). Kluwer A.P., March/May 1995.

[2] A. Cervin, D. Henriksson, B. Lincoln, and K.-E. Årzé n. Jitterbug and truetime: Analysis tools for real-time control systems. In *Proceedings of the 2nd Workshop on Real-Time Tools*, Copenhagen, Danmark, 2002.

[3] C. Eriksson, J. Mäki-Turja, K. Post, M. Gustafsson, J. Gustafsson, K. Sandström, and E. Brorsson. An overview of rtt: A design framework for real-time systems. In *Journal of Parallel and Distributed Computing*, volume 36, pages 66–80, October 1996.

[4] J. Gait. A probe effect in concurrent programs. In *Software - Practice and Experience*, volume 16(3), pages 225–233, Mars 1986.

[5] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. In *Journal of the ACM*, volume 20(1), 1973.

[6] A. Pettersson. The revised EOG-algorithm. In *Mälardalen University Technical Note, Department of Computer Science and Engineering, P.O. Box 883, SE-721 23 Västerås, Sweden*, number 0596, October 2003.

[7] A. Pettersson and H. Thane. Testing of multi-tasking real-time systems with critical sections. In *Proceedings of Ninth International Conference on Real-Time and Embedded Computing Systems amd Applications*, Tainan City, Taiwan, R.O.C, 18-20 February 2003.

[8] H. Thane and H. Hansson. Towards systematic testing of distributed real-time systems. In *Proceedings of The 20th IEEE Real-Time Systems Symposium*, pages 360–369, 1999.

[9] H. Thane and H. Hansson. Testing distributed real-time systems. In *Journal of Microprocessors and Microsystems*, pages 463–478. Elsevier, 2001.

[10] H. Thane, A. Pettersson, and D. Sundmark. The asterix real-time kernel. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS'01), Industrial Session*, Delft, June 2001.

[11] J. Xu and D. Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. In *IEEE Transaction on Software Engineering*, volume 16(3), pages 360–369, 1990.