TRITA-MMK 2000:16
ISSN 1400-1179
ISRN KTH/MMK/R--00/16--SE
MRTC Report 00/15
V 1.02

Monitoring, Testing and Debugging of Distributed Real-Time Systems

by

Henrik Thane

ARTES

MRTC
MÄLARDALEN REAL-TIME
RESEARCH CENTRE

Stockholm 2000

Doctoral Thesis

Mechatronics Laboratory,
Department of Machine Design
Royal Institute of Technology, KTH
S-100 44 Stockholm,
Sweden

V 1.02:010212

Akademisk avhandling som med tillstånd från Kungliga Tekniska Högskolan i Stockholm, framlägges till offentlig granskning för avläggande av teknologie doktorsexamen fredagen den 26 maj 2000 kl. 10.00 i sal M2, Kungliga Tekniska Högskolan, Brinellvägen 64, Stockholm.

Fakultetsopponent: Jan Jonsson, Chalmers Tekniska Högskola, Göteborg.

© Henrik Thane 2000

Mälardalen Real-Time Research Centre (MRTC),
Department of Computer Engineering
Mälardalen University (MDH)
S-721 23 Västerås

(www.mrtc.mdh.se, henrik.thane@mdh.se)

To
Marianne and Cornelia

Mechatronics Laboratory,
Department of Machine Design
Royal Institute of Technology (KTH)
S-100 44 Stockholm, Sweden.

TRITA-MMK 2000:16
ISSN 1400-1179
ISRN KTH/MMK/R--00/16—SE
MRTC Report 00/15

Document type
Doctoral Thesis

Date
May 1, 2000

Author
Henrik Thane

Supervisors
Jan Wikander, Hans Hansson

Sponsors
ARTES/SSF, Scania AB

Abstract

Testing is an important part of any software development project, and can typically surpass more than half of the development cost. For safety-critical computer based systems, testing is even more important due to stringent reliability and safety requirements. However, most safety-critical computer based systems are real-time systems, and the majority of current testing and debugging techniques have been developed for sequential (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 and debugging 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. The current trend of consumer and industrial applications goes from single micro-controllers to sets of distributed micro-controllers, which are even more challenging than handling real-time per-see, since multiple loci of observation and control additionally must be considered. In this thesis we try to remedy these problems by presenting an integrated approach to monitoring, testing, and debugging of distributed real-time systems.

For monitoring, we present a method for deterministic observations of single tasking, multi-tasking, and distributed real-time systems. This includes a description of what to observe, how to eliminate the disturbances caused by the actual act of observing, how to correlate observations, and how to reproduce them.

For debugging, we present a software-based method, which uses deterministic replay to achieve reproducible debugging of single tasking, multi-tasking, and distributed real-time systems. Program executions are deterministically reproduced off-line, using information concerning interrupts, task-switches, timing, data accesses, etc., recorded at runtime.

For testing, we introduce a method for deterministic testing of multitasking and distributed real-time systems. This method derives, given a set of tasks and a schedule, all execution orderings that can occur at run-time. Each such ordering is regarded as a sequential program, and by identifying which ordering is actually executed during testing, techniques for testing of sequential software can be applied.

For system development, we show the benefits of considering monitoring, debugging, and testing early in the design of real-time system software, and we give examples illustrating how to monitor, test, and debug distributed real-time systems.

Keywords
Monitoring, testing, debugging, testability, distributed real-time systems, deterministic replay, scheduling.

Language
English

Preface

Science and technology have fascinated me as long as I can remember. When I was a kid, comics, TV-shows, popular science magazines, movies, school, and science fiction books set off my imagination.

When I was a teenager, computers started to fascinate me, and I started to play around with games programming, which proved to be a very creative, intellectually stimulating, and fun activity. This experience went on for years and made me realize that computers and computer programming were things I really wanted to work with in the future.

And, so it became. Parallel with studies at the university, I worked part time as a computer programmer for a company that designed real-time system applications, and I kept at it for a year after graduation. After some escapades around the world, I began studying for a Ph.D. at the Royal Institute of Technology (KTH), in April 1995. My research topic was safe and reliable software in embedded real-time system applications. This resulted in a Licentiate degree in Mechatronics, in the fall of 1997. The research topic was a bit broad, but gave me great insight into the problems of designing and verifying safe and reliable computer software. While still being associated with the Mechatronics laboratory at the KTH, I moved in the fall of 1997 to Mälardalen University in Västerås, and started working as part time teacher and part time Ph.D. student. At the same time my research narrowed and I begun focusing on testability of real-time systems, which for natural reasons led me to also consider testing, debugging and monitoring. That work gave fruit – this thesis.

The work presented in this thesis would not have been possible if I had not had such stimulating and creative people around me, including my supervisors Jan Wikander and Hans Hansson. I am also very grateful for the financial support provided by the national Swedish Real-Time Systems research initiative, ARTES, supported by the Swedish Foundation for Strategic Research, as well as for the donation provided by Scania AB during 1995-1997.

However, my greatest thanks go Hans Hansson, Christer Norström, Kristian Sandström, and Jukka-Mäki Turja for all creative work, and intense, and stimulating discussions we have had during the years. Other special thanks go to Anders Wall, Mårten Larsson, Gerhard Fohler, Mohammed El-Shobaki, Markus Lindgren, Björn Alvin, Sasikumar Punnekkat, Iain Bate, Mikael Sjödin, Mikael Gustafsson and Martin Törngren for also providing insightful input. Not only being a friendly bunch of people, they have all also over the years provided input and feedback on drafts of papers – not forgetting, this thesis. Thank you very much.

A very special thank you goes to Harriet Ekwall for handling all the ground service here at MRTC, and for always being such happy and friendly spirit. Thank you.

Finally, the greatest debt I owe to my family, Marianne and Cornelia, who have been extremely supportive, not to mention Cornelia's artwork on my thesis drafts.

Västerås, a beautiful and warm spring day in April 2000.

Publications

1. Thane H and Hansson H. *Testing Distributed Real-Time Systems.* Submitted for journal publication.
2. Thane H. and Hansson H. *Using Deterministic Replay for Debugging of Distributed Real-Time Systems*. In proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS'00), Stockholm, June 2000.
3. Thane H. *Asterix the T-REX among real-time kernels. Timely, reliable, efficient and extraordinary.* Technical report. In preparation. Mälardalen Real-Time Research Centre, Mälardalen University, May 2000.
4. Thane H, and Wall A. *Formal and Probabilistic Arguments for Reuse and testing of Components in Safety-Critical Real-Time Systems.* Technical report. Mälardalen Real-Time Research Centre, Mälardalen University, March 2000.
5. Thane H. and Hansson H. *Handling Interrupts in Testing of Distributed Real-Time Systems*. In proc. Real-Time Computing Systems and Applications conference (RTCSA'99), Hong Kong, December 1999.
6. Thane H. and Hansson H. *Towards Systematic Testing of Distributed Real-Time Systems*. In proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99), Phoenix, Arizona, December 1999.
7. Thane H. *Design for Deterministic Monitoring of Distributed Real-Time Systems.* Technical report, Mälardalen Real-Time Research Centre, Mälardalen University, November 1999.
8. Norström C., Sandström K., Mäki-Turja J., Hansson H., and Thane H. *Robusta Realtidssytem*. Book. MRTC Press, September, 1999.
9. Thane H. and Hansson H. *Towards Deterministic Testing of Distributed Real-Time Systems.* In Swedish National Real-Time Conference SNART'99, August 1999.
10. Thane H. *Safety and Reliability of Software in Embedded Control Systems*. Licentiate thesis. TRITA-MMK 1997:17, ISSN 1400-1179, ISRN KTH/MMK/R--97/17--SE. Mechatronics Laboratory, the Royal Institute of Technology, S-100 44 Stockholm, Sweden, October 1997.
11. Thane H. and Norström C. *The Testability of Safety-Critical Real-Time Systems.* Technical report. Dept. computer engineering, Mälardalen University, September 1997.
12. Thane H. *Safe and Reliable Computer Control Systems - an Overview.* In proceedings of the 16th Int. Conference on Computer Safety, Reliability and Security (SAFECOMP'97), York, UK, September 1997.
13. Thane H. and Larsson M. *Scheduling Using Constraint Programming*. Technical report. Dept. computer engineering, Mälardalen University, June 1997.
14. Thane H. and Larsson M. *The Arbitrary Complexity of Software*. Research Report. Mechatronics Laboratory, the Royal Institute of Technology, S-100 44 Stockholm, Sweden, May 1997.
15. Eriksson C., Thane H. and Gustafsson M. *A Communication Protocol for Hard and Soft Real-Time Systems.* In the proceedings of the 8th Euromicro Real-Time Workshop, L'Aquila Italy, June 1996.
16. Thane, H. *Safe and Reliable Computer Control Systems - Concepts and Methods*. Research Report TRITA-MMK 1996:13, ISSN 1400-1179, ISRN KTH/MMK/R-96/13-SE. Mechatronics Laboratory, the Royal Institute of Technology, S-100 44 Stockholm, Sweden, 1996.
17. Eriksson C., Gustafsson M., Gustafsson J., Mäki-Turja J., Thane H., Sandström K., and Brorson E. *Real-TimeTalk a Framework for Object-Oriented Hard & Soft Real-Time Systems*. In proceedings of Workshop 18: Object-Oriented Real-Time Systems at OOPSLA,Texas,USA,October 1995.

Contents

1 INTRODUCTION 9

2 BACKGROUND 11

2.1 THE SCENE 11
2.2 COMPLEXITY 11
2.3 SAFETY MARGINS 12
2.3.1 Robust designs 12
2.3.2 Redundancy 13
2.4 THE VERIFICATION PROBLEM 14
2.4.1 Testing 14
2.4.2 Removing errors 15
2.4.3 Formal methods 15
2.5 TESTABILITY 16
2.5.1 Ambition and effort 18
2.5.2 Why is high testability a necessary quality? 20
2.6 SUMMARY 21

3 THE SYSTEM MODEL AND TERMINOLOGY 23

3.1 THE SYSTEM MODEL 23
3.2 TERMINOLOGY 23
3.2.1 Failures, failure modes, failure semantics, and hypotheses 23
3.2.2 Determinism and reproducibility 28

4 MONITORING DISTRIBUTED REAL-TIME SYSTEMS 31

4.1 MONITORING 35
4.2 HOW TO COLLECT SUFFICIENT INFORMATION 36
4.3 ELIMINATION OF PERTURBATIONS 38
4.3.1 Hardware monitoring 38
4.3.2 Hybrid monitoring 39
4.3.3 Software monitoring 40
4.4 DEFINING A GLOBAL STATE 43
4.5 REPRODUCTION OF OBSERVATIONS 44
4.5.1 Reproducing inputs 44
4.5.2 Reproduction of complete system behavior 45
4.6 SUMMARY 45

5 DEBUGGING DISTRIBUTED REAL-TIME SYSTEMS 47

5.1 THE SYSTEM MODEL 49
5.2 REAL-TIME SYSTEMS DEBUGGING 50
5.2.1 Debugging single task real-time systems 50
5.2.2 Debugging multitasking real-time systems 50
5.2.3 Debugging distributed real-time systems 52
5.3 A SMALL EXAMPLE 53
5.4 DISCUSSION 54
5.5 RELATED WORK 56
5.6 SUMMARY 57

6 TESTING DISTRIBUTED REAL-TIME SYSTEMS 59

6.1 THE SYSTEM MODEL 62
6.2 EXECUTION ORDER ANALYSIS 63
6.2.1 Execution Orderings 63
6.2.2 Calculating $EX_o(J)$ 65
6.3 THE EOG ALGORITHM 69

6.3.1 GEX_o – the Global EOG ... *71*
6.4 TOWARDS SYSTEMATIC TESTING ... 76
6.4.1 Assumptions ... *76*
6.4.2 Test Strategy ... *77*
6.4.3 Coverage ... *77*
6.4.4 Reproducibility ... *79*
6.5 EXTENDING ANALYSIS WITH INTERRUPTS ... 80
6.6 OTHER ISSUES ... 82
6.6.1 Jitter ... *82*
6.6.2 Start times and completion times ... *83*
6.6.3 Testability ... *83*
6.6.4 Complexity ... *83*
6.7 SUMMARY ... 85

7 CASE STUDY ... 87

7.1 A DISTRIBUTED CONTROL SYSTEM ... 87
7.2 ADDING PROBES FOR OBSERVATION ... 90
7.3 GLOBAL EXECUTION ORDERINGS ... 93
7.4 GLOBAL EXECUTION ORDERING DATA DEPENDENCY TRANSFORMATION ... 94
7.5 TESTING ... 94
7.6 IMPROVING TESTABILITY ... 96
7.7 SUMMARY ... 97

8 THE TESTABILITY OF DISTRIBUTED REAL-TIME SYSTEMS ... 99

8.1 OBSERVABILITY ... 99
8.2 COVERAGE ... 100
8.3 CONTROLLABILITY ... 100
8.4 TESTABILITY ... 101

9 CONCLUSIONS ... 103

10 FUTURE WORK ... 105

10.1 MONITORING, TESTING AND DEBUGGING ... 105
10.2 FORMAL AND PROBABILISTIC ARGUMENTS FOR COMPONENT REUSE AND TESTING IN SAFETY-CRITICAL REAL-TIME SYSTEMS ... 106
10.2.1 Software components in real-time systems? ... *107*
10.2.2 Component contracts ... *108*
10.2.3 Component contract analysis ... *112*
10.2.4 Summary ... *117*

11 REFERENCES ... 119

1 INTRODUCTION

The introduction of computers into safety-critical systems lays a heavy burden on software designers. Public and legislators demand reliable and safe computer systems, equal to or better than the mechanical or electromechanical parts they replace. The designers must have a thorough understanding of the system and more accurate software design and verification techniques than have usually been deemed necessary for software development. However, since computer related problems, relating to safety and reliability, have just recently been of any concern for engineers, there are no holistic engineering principles for construction of safe and reliable computer based systems. There exist only scarce pools of knowledge and no silver bullets[1] that can handle everything. Some people do nonetheless, with an almost religious glee, decree that their method, principle or programming language handles or kills all werewolves (which these days have shrunken to tiny, but sometimes lethal bugs) [9][38][85].

The motivation for writing this thesis is an ambition to increase the depth of knowledge in the pool of distributed real-time systems verification, which previously has been very shallow. We will specifically address testing and debugging, and as most real-time systems are embedded with limited observability, we will also cover monitoring. Testing is an important part of any software development project, and can typically surpass more than half of the development cost. High testability is therefore of significance for cost reduction, but also for the ability to reach the reliability levels required for safety-critical systems. A significant part of this thesis is accordingly dedicated to discussions on testability, and testability increasing measures for distributed real-time systems.

The shallowness of knowledge in the pool of distributed real-time systems verification, is partly due to the fact that the majority of current testing and debugging techniques have been developed for sequential programs. These techniques are as such not directly applicable to real-time systems since they disregard timing and concurrency issues. The implication is that reproducible testing and debugging are not possible using these techniques. 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 (bugs). The current trend of consumer and industrial applications goes from single micro-controllers to sets of distributed micro-controllers, for which current testing and debugging techniques also are insufficient; they cannot handle the multiple loci of observation and control required. In this thesis we try to remedy these problems by presenting novel approaches to monitoring, testing, and debugging of both single CPU and distributed real-time systems.

The main contributions of this thesis are in the fields of:

- **Monitoring.** We present a method for deterministic observations of single tasking, multi-tasking, and distributed real-time systems. This includes a description of what to observe, how to eliminate the disturbances caused by the actual act of observing, how to correlate observations between nodes, and how to reproduce the observations. We will give a taxonomy of different observation techniques, and discuss where, how and when these techniques should be applied for deterministic observations. We argue that it is essential to consider monitoring early in the design process, in order to achieve efficient and deterministic observations.

[1] Silver bullets, which in folklore are the only means to slay the mythical werewolves.

- **Debugging.** We present a software based technique for achieving reproducible debugging of single tasking, multi-tasking, and distributed real-time systems, by means of deterministic replay. 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. 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.
- **Testing.** We present a method for deterministic testing of multitasking real-time systems, which allows explorative investigations of real-time system behavior. 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. The method includes an analysis technique that given a set of tasks and a schedule derives all execution orderings that can occur during run-time. 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. We also show how this analysis and testing strategy can be extended to encompass interrupt interference, distributed computations, communication latencies and the effects of global clock synchronization. The number of execution orderings is an objective measure of the testability of a system since it indicates how many behaviors the system can exhibit during runtime. In the pursuit of finding errors we must thus cover all these execution orderings. The fewer the orderings the better the testability.

Outline:

The outline of this thesis is such that we begin in chapter 2 with a description of the peculiarities of software in general and why software verification constitutes such a significant part of any software development project. In chapter 3, we define a basic system model and some basic terminology, which we will refine further on in the thesis. In chapter 4, we give an introduction to monitoring of real-time systems (RTS), and discuss and give solutions on how monitoring can be deterministically achieved in RTS, and distributed real-time systems (DRTS). In chapter 5, we discuss and present some solutions to deterministic and reproducible debugging of DRTS. In chapter 6, we present our approach to deterministic testing of DRTS, including results on how to measure the testability of RTS, and DRTS. In chapter 7, we present a larger example illustrating the use of the techniques presented in chapters 3, and 5. In chapter 8, we discuss the testability of DRTS. Finally, in chapter 9 we summarize and draw some conclusions, and in chapter 10 we outline some future work.

2 BACKGROUND

This thesis is about monitoring, testing, and debugging, of distributed real-time systems, but as a background and to set the scene we begin with a description of the peculiarities pertaining to computer software development and why software verification is such a dominant factor in the development process.

2.1 The scene

Experience with software development has shown that software is often delivered late, over budget, and despite all that still functionally incorrect. A question is often asked: ”What is so different about software engineering? Why do not software engineers do it right, like traditional engineers? Or at least once in a while?”

These unfavorable questions are not uncalled for; the traditional engineering disciplines are founded on science and mathematics and are able to model and predict the behavior of their designs. Software engineering is more of a craft, based on trial and error, rather than on calculation and prediction. However, knowing the peculiarities of software, this comparison is not entirely fair; it does not acknowledge that computers and software differ from physical systems on two key accounts:

(1) They have discontinuous behavior and

(2) Software lacks physical attributes like e.g., mass, inertia, size, and lacks structure or function related attributes like e.g., strength, density and form.

The sole physical attribute that can be modeled and measured by software engineers is *time*. Therefore, there exist sound work and theories regarding modeling and verification of systems’ temporal attributes [4][75][117][58]. The theory for real-time systems gives us a platform for more “engineering wise” modeling and verification of computer software, but since the theory is mostly concerned with timing, and ordering of events, we still have to face functional verification.

We will now in the remainder of this chapter discuss the peculiarities of software development compared to classical engineering. We will discuss the impact that the two fundamental differences (1 and 2 above) have on software development as compared to classical engineering of physical systems. We will progress in several steps: beginning with software complexity, and then continuing on with safety margins, testing, modeling, validation, and finally testability.

2.2 Complexity

The two properties (1) and (2) above give rise to both advantages and disadvantages compared to regular physical systems. One good thing is that software is easy to change and mutate hence the name software. The bad thing is that complexity easily arises. Having no physical limitations, complex software designs are possible and no real effort to accomplish this complexity is needed. That is, it is very easy to produce solutions to a problem that are vastly more complex than the intrinsic complexity of the problem.

Complexity is a source of design faults. Design faults are often due to failure to anticipate certain interactions between components in the system. As complexity increases, design faults are more prone to occur since more interactions make it harder to identify all possible behaviors. Since, software does not suffer from gravity, or have any limits to structural strength there is nothing that hinders our imagination in solving problems using software. Although not explicitly expressed, programming languages, programming methodologies and processes in fact introduces virtual physical laws and restrains the imagination of the programmers. At a seminar Nobel Prize winner Richard Feynman once said: "Science is imagination in a straightjacket."

2.3 Safety margins

In the classical engineering disciplines such as civil engineering it is common practice to make use of safety margins. Bridges are, for example, often designed to withstand loads far greater than they would encounter during normal use. For software it is in the classical sense not possible to directly make use of safety margins because software is pure design – like the blue prints for bridges. Safety margins on software would be like substituting the paper in the blue prints for thick sheets of steel. Software is pure design and can thus not be worn-out or broken by physical means. The physical memory where the software resides can of course be corrupted due to physical failures (often transient), but not the actual software. All system failures due to errors in the software are design-flaws, built into the system from the beginning (*Figure 2-1*).

Figure 2-1. Cause-consequence diagram of fault, error and failure. Where the failure signifies a behavior non-compliant with the specification. Where an error signifies a state that can lead to a failure, and where the fault is the hypothesized cause for the error.

2.3.1 Robust designs

A plausible substitute to safety margins in the software context is defensive programming using robust designs (*Figure 2-2*). Every software module has a set of pre-conditions and post-conditions to ensure that nothing *unexpected* happens. The pre-conditions must be valid when entering the software module and the post-conditions must be valid at the end of execution of the module. If these conditions are violated the program should do something *sensible*. The problem is that if the *unexpected* does happen, then the design might be deficient and a *sensible* local action might have a non-predictable effect on the entire system.

Figure 2-2. Robust designs intend to stop infections of the system.

2.3.2 Redundancy

Another method for achieving safety margins that does not directly work in the context of software design, as it does in the physical world, is fault-tolerance by redundancy (*Figure 2-3*).

Figure 2-3. Fault-tolerant designs intend to stop the propagation of errors before they lead to failures, by means of redundancy.

Physical parts can always succumb to manufacturing defects, wear, environmental effects or physical damage. Thus, it is a good idea to have spares handy that can replace defective components, but in order for a redundant system to function properly it must by all means avoid common mode failures. For example,

> *Two parallel data communication cables were cut in Virginia, USA, 1991. The Associated Press (having learned from earlier incidents, had concluded that a spare could be a good idea) requested two separate cables for their primary and backup circuits. However, both cables were cut at the same time because they were adjacent* [84].

Design faults are *the* sources for common mode failures, so fault tolerance against design faults seems futile. An adaptation of the redundancy concept has, nonetheless been applied to software. It is called *N*-version programming and uses *N* versions of *dissimilar* software produced from a common specification. The *N* versions are executed in parallel and their results voted upon, as illustrated in *Figure 2-4*. Empirical studies have unfortunately concluded that the benefit of using *N*-version programming is questionable. A clue is common mode errors in the requirement specifications and the way humans think in general [48][49][99].

Spatial redundancy and time redundancy have proven to be effective against permanent and transient physical faults, and are applied in a multitude of different engineering disciplines. Redundancy against design errors in software has not been proven efficient though.

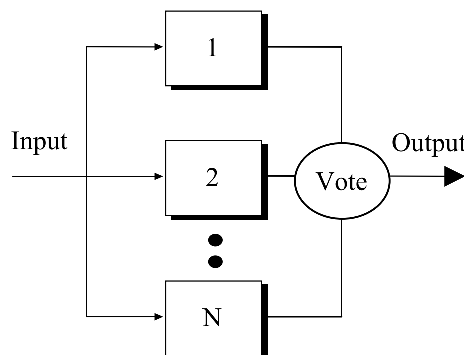


Figure 2-4. N-modular redundancy using N-version programming.

2.4 The verification problem

The application of defensive programming and redundancy may sometimes be of benefit against systematic (design) errors but mostly the system designers are left only with the choice of eliminating all errors, or at least those with possibly serious consequences. It is however not sufficient to only remove errors, the system designer must also produce evidence that the errors indeed have been eliminated successfully – usually through the process of verification.

2.4.1 Testing

The task of considering all system behaviors and all the circumstances a system can encounter during operation is often intractable. Physical systems can be tested and measured. There often exist piece-wise continuous relationships between the input and the output of a system. Only a few tests, for each continuous piece, need to be performed. The behavior of the system intermediate to the samples can be interpolated, and the behavior of the system exterior to the samples can be extrapolated. Thus the number of behaviors to be considered is reduced. However, it is not possible, in general, to assume that the behavior of computers and software is continuous because quantization errors are propagated and boundaries to the representation of numbers can affect the output (*Figure 2-5*). Equally influential is the fact that the execution paths through software change for every decision, depending on whether or not a condition is true. For example, a simple sequential list of 20 if-then-else statements may, in the worst case, yield 2^{20} different behaviors due to 2^{20} possible execution paths. A small change in the input can have a severe effect on which execution path is taken, which in turn may yield an enormous change in output [93]. That is, software behavior is discontinuous and has no inertia like physical systems do.

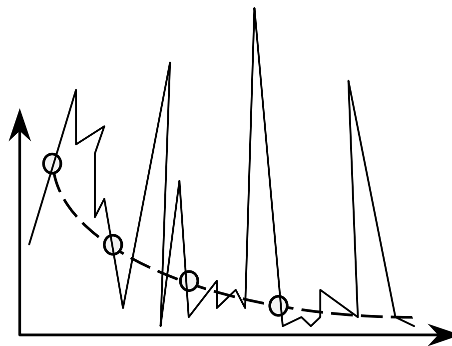


Figure 2-5. Interpolation cannot represent the discontinuos behavior of software.

In order to achieve complete confidence in program correctness we must thus explore all behaviors of the program. But, for a program that takes two 32 bit integers as input, we must cover 2^{64} possible input combinations, and if each test case takes 1 µs to perform we would need 565 000 years to cover all possible input combinations. Not surprisingly E. Dijkstra concluded [19]:

> "Non-exhaustive testing can only show the presence of errors not their absence."

Aggravating the situation further is the fact that typically more than half of the errors in a system are due to ambiguous or incomplete requirement specifications [20][24][67][76]. The intention of testing is to verify that a specific input will yield a specific output *defined* by the specification. Possibly erroneous requirements thus further limits the confidence gained by testing software.

2.4.2 Removing errors

Software does not wear out over time. It is therefore reasonable to assume that as long as errors are discovered, reliability increases for each error that is eliminated – of course given that no new errors are introduced during maintenance. This has led to the development of a wide range of reliability growth models as introduced by Jelinski et al. [44][72][71][69][1][47]. Reliability growth models assume that the failures are distributed exponentially. Initially a system fails frequently but after errors are discovered and removed the frequency of failures decreases. Due to the history of encountered failures and the removal of errors, these models can make predictions about the future occurrence of failures (extrapolation). For ultra reliable systems (10^{-9} failures/hour or less) it has been proven that these types of models cannot be used because a typical system would have to be tested for 115000 years or more [10][70]. In addition, for typical testing environments it is very hard to reproduce the operational profiles for rare events. Taking these factors into account and considering the time required to restart the system for each test run, the failure rates that can be verified empirically are limited to about 10^{-4} failures/hour [10].

2.4.3 Formal methods

Just as traditional engineers can model their designs with different kinds of continuous mathematics, formal methods is an attempt to supply the software engineers with mathematical logic and discrete mathematics as modeling and verification tools.

Formal methods can be put to use in two different ways, (Barroca et al.[57]): (1) They can be used as a syntax to describe the semantics of specifications which are later used as a basis for the development of systems in an ordinary way. (2) Formal specifications can be produced as stated by (1) and then used as a fundament for verification (proof) of the design (program).

If (1) and (2) are employed, it is possible to prove equivalence of program and specification, i.e., to prove that the program does what it is specified to do. This stringency gives software development the same degree of certainty as a mathematical proof [57].

Unfortunately a proof (when possible) cannot guarantee correct functionality or safety. In order to perform a proof, the *correct* behavior of the software must first be specified in a formal, mathematical language. The task of specifying the correct behavior can be as difficult and error-prone as writing the software to

begin with [67][68]. In essence the difficulty comes from the fact that we cannot know if we have accurately modeled the ”real system”, so we can never be certain that the specification is complete. This distinction between model and reality attends all applications of mathematics in engineering. For example, the ”correctness” of a control loop calculation for a robot depends on the fidelity of the control loop’s dynamics to the real behavior of the robot, on the accuracy of the stated requirements, and on the extent to which the calculations are performed without error.

These limitations are however, minimized in engineering by empirical validation. Aeronautical engineers believe that fluid dynamics accurately models the air flowing over the wing of an airplane, and civil engineers believe that structural calculus accurately models the structural integrity of constructions. Aeronautical engineers and civil engineers have great confidence in their respective models since they have been validated in practice many times. Validation is an empirical pursuit to test that a model accurately describes the real world.

Worth noting is that the validation of the formal models has to be done by testing, and as formal models are based on discrete mathematics they are discontinuous. The implication is thus that we cannot use interpolation or extrapolation, leaving us in the same position – *again,* as when using simple testing. We have nonetheless gained something compared to just applying testing solely; we can assume that the specifications are non-ambiguous and complete within themselves. Although, they may still be incomplete or erroneous with respect to the real target system they attempt to model.

Further limitations to current formal methods are their lack of scalability (due to exponential complexity growth), and their inability to handle timing and resource inadequacies, like violation of deadlines and overload situations (although some tools do handle time [60]). That is, any behavior (like overload) not described in the formal model cannot be explored. Testing can however do just that, because testing explores the *real system*, not an abstraction.

2.5 Testability

So, even if we make use of formal methods and fault-tolerance we can never eliminate testing completely. But, in order to achieve complete confidence in the correctness of software we must explore all possible behaviors, and from the above discussion we can conclude that this is seemingly an impossible task.

A constructive approach to this situation has led to research into something called testability. It is known that some software is easier to verify than other software, and therefore potentially more reliable.

Definition. *Testability*. The probability for failures to be observed during testing when errors are present.

Consequently, a system with a high level of testability will require few test cases in order to reveal all errors, and vice versa a system with very low testability will require intractable quantities of test cases in order to reveal all errors. We could thus potentially produce more reliable software, if we knew how to design for high testability.

In order to define the testability of a program we need to know when an error (bug) causes a failure. This occurs if and only if:

(1) the location of the error is executed in the program, (**E**xecution)

(2) the execution of the error leads to an erroneous state, (**I**nfection) and

(3) the erroneous state is propagated to output (**P**ropagation).

Based on these three steps 1-2-3 Voas et al. [116][115][114] formally define the testability, q, of a program P, due to an error l, with input range I and input data distribution D as:

$$
\begin{aligned}
q_l = \; & \text{Probability \{executing error\}} \\
& \times \text{Probability \{infection | execution\}} \\
& \times \text{Probability \{propagation | infection\}}
\end{aligned}
$$

Where the value of q_l is the probability of failure caused by the error l, and where q_l is viewed as the *size* of the error l. The testability, q, for the entire program is given by the smallest of all errors in P, that is, $q = \min\{\forall l \mid q_l\}$. This smallest error, q, will consequently be the most elusive during testing, and require the greatest number of test cases before leading to a detectable failure. This model has been named the PIE model, by Voas [116].

Example 2-1

Figure 2-6 depicts an erroneous program which testability we will explore. Assume that the program has an input space of all positive integers with a rectangular distribution, and that there is an erroneous statement in line 8, which is executed 25% of the time (E=0.25). Based on this we can calculate the probability of failure for this program due to the error in line 8. The probability of failure according to the PIE model is then $q = 0.25 \times 1.0 \times 2/30000 = 0.00001667$. The error in line 8 will always infect the variable x (I=1.0), and the probability for propagation is defined by the value difference between the correct statement and the error, and the integer division (P=2/30000).

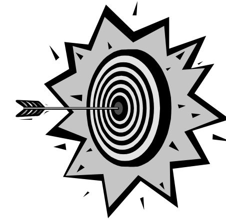
```
1:  int f(positive integer a)
2:  {
3:    positive integer x;
4:    if(...)
5:    {
6:      if(...)
7:      {
8:        x = a - 1;/* Correct is: x = a + 1;*/
9:        x = x div 30000; /* integer division*/
10:     }
11:   }
12:   return x;
13: }
```

Figure 2-6. An erroneous function.

If a certain program P has a testability of 1.0, a single test case is needed to verify whether the program is correct or not. Vice versa, if a program P' has a testability measure of 0.0 it does not matter how many test-cases we devise and run – we will never be able to detect an error. These two extremes are both desirable attributes of a program. The benefit of the first attribute, 1.0, is obvious because

we can easily explore program correctness. The use of the second attribute, 0.0, might not be as obvious, but if we can never detect an error, it will never cause a failure either. A testability score of 1.0 is utopia for a tester and a testability score of 0.0 is utopia for a designer of fault-tolerance. A tester wants by all means to detect a failure, and a designer of fault-tolerance wants by all means to stop the execution, infection and propagation of an error before it leads to a failure. There is thus a fundamental trade-off between testability and fault-tolerance.

2.5.1 Ambition and effort



Testing of software is done with a certain ambition and effort. We might for example, have the ambition to assess that a program has a reliability of 10^{-9} failures per hour. By analogy this could be viewed as hitting bull's eye on a dartboard blindfolded. Depending on how many darts we throw we get a certain confidence in how likely it is that we have hit bulls eye. The testability of a system can be regarded as the size of the bull's eye. Since the testability q gives the probability of finding the smallest error, we can after hitting testability bull's eye assume that the system is correct, since there should be no error smaller than q. That is, in order to find the smallest error with a certain confidence we have to run a specific minimum number of test cases. This minimum number is larger than the number needed to find any other error larger than the smallest error.

The confidence can be regarded as a measure of how thick the darts are. That is, as confidence increases, the size of the dart increases – meaning that it will eventually fill the entire dartboard and that we thus have hit the bull's eye.

It has been argued that reliability is not the software attribute we want to determine, but rather what we really want is confidence in the correctness of software. The probable correctness model by Hamlet [37] provides this ability. The model works like this: Assume that we have some estimated probability f, for program failure according to some input data distribution D. Then the probability that the software will not fail during the next input data is

$$(1 - f).$$

The probability that it will not fail during the next N inputs is

$$(1 - f)^N.$$

The probability of at least one failure during the next N inputs is

$$1 - (1 - f)^N.$$

If we test the program with N inputs without failure we get the confidence

$$C = 1 - (1 - f)^N$$

that the real probability of failure is less than or equal to f [114]. Using this relation between confidence, probability of failure, and the number of test cases we can derive an expression for how many test cases we need to execute without failure in order to verify a certain failure frequency with a certain confidence:

$$N = \left\lceil \frac{\ln(1-C)}{\ln(1-f)} \right\rceil \qquad (2\text{-}1)$$

Table 2-1 illustrates the relation between the ambition of achieving a certain failure frequency and the number of test cases required (effort) for achieving this with a certain confidence. *Table 2-2* illustrates how the number of test cases required corresponds to the time needed to perform all these tests.

From *Table 2-2* we can conclude that safety-critical software ($f<10^{-9}$ failures/hour) require a ridiculous number of hours of testing. However, if we know the testability, q (the size of the bull's eye) we can reduce the number of test cases required, because when we have found the smallest error, we can assume that the software is correct – there should be no error smaller. Based on this, we can substitute the f in equation (2-1) with the testability q, and calculate how many test cases, N, are required before we can assume that the program is correct with a certain confidence, C.

Table 2-3 illustrates the number of test cases needed in relation to the testability, q, of the program, before we can assume that the program is correct (compare with *Table 2-1*).

Table 2-1. The relation between targeted reliability f, the number of test cases and the confidence, C, in f according to formula (2-1).

***f* Failures/test case**	***C=.9***	***C=.99***	***C=.999***
10^{-1}	22	44	66
10^{-2}	230	460	690
10^{-3}	2,300	4,600	6,900
10^{-4}	23,000	46,000	69,000
10^{-5}	230,000	460,000	690,000
10^{-6}	2,300,000	4,600,000	6,900,000
10^{-7}	23M	46M	69M
10^{-8}	230M	460M	690M
10^{-9}	2,300M	4,600M	6,900M
10^{-10}	23,000M	46,000M	69,000M
10^{-11}	230,000M	460,000M	690,000M

Table 2-2. The relation between reliability, the number of test cases and the time required. Confidence C=0.99.

***f* Failures/test case**	***Number of test cases***	***1 test per second***	***1 test per hour***
10^{-1}	44	44 seconds	1.8 days
10^{-2}	459	7.6 minutes	19.1 days
10^{-3}	4600	1.25 hours	191 days
10^{-4}	46,000	13 hours	5.25 years
10^{-5}	460,000	5.5 days	52.5 years
10^{-6}	4,600,000	1.75 months	525 years
10^{-7}	46M	1.5 years	5250 years
10^{-8}	460M	15 years	52 500 years
10^{-9}	4,600M	150 years	525 000 years
10^{-10}	46,000M	1,500 years	5,25 10^{6} years
10^{-11}	460,000M	15,000 years	5,25 10^{7}years

Table 2-3. The relation between the testability, *q*, the confidence, *C*, and the number of test cases *N,* according to formula (2-1).

***q* Failures/test case**	***C***	***N***
0,0000001	0,00	1 000
0,0000001	0,7	10 000 000
0,00001	0,01	1 000
0,00001	0,63	100 000
0,00001	0,999955	1 000 000
0,001	0,63	1000
0,001	0,999955	10 000
0,001	0,999999998	20 000
0,1	0,999973	100
0,1	0,99999999929	200

2.5.2 Why is high testability a necessary quality?

There are many arguments for designing systems with high testability:

- The cost of verification and maintenance of software encompass typically more than half of the cost for a software project [6][16][30][40]. If we have a limited amount of resources we can achieve higher levels of reliability if we design for high testability rather than if we do not. *"Get more bang for the buck."*
- More than 50% of all system failures can be traced back to the requirement specifications [20][24][67][76]. This has led to the application of iterative development processes where the requirements are refined for each iteration and prototype produced – which are then validated against the customer. Following this scheme it is possible to decrease the portion of failures coming from the requirements due to wanting or erroneous requirements.

  However, for each iteration and prototype we must re-test the system. High testability is thus of utmost importance in order to decrease the testing effort for each iteration.
- For safety critical software (10^{-9} failures/hour) [23] it is obligatory to design for high testability since reliability growth models have proven that testing of software to below 10^{-4} failures/hour is infeasible [10]. Testability scores (error sizes) of software must thus be larger than 10^{-4} failures/hour. If we have achieved a reliability corresponding to the testability of the system, we can assume that the system is correct, there should be no error smaller than indicated by the testability of the program [116][115][114].

2.6 Summary

In this chapter we have given an introduction to the peculiarities of computer software. From that we conclude that the problems of designing and verifying software are fundamental in character: Software has discontinuous behavior, no inertia, and has no physical restrictions what so ever, except for time. Silver bullets, i.e., techniques or methods that solely will eliminate all bugs have shown to be myths [9][38][85]. Any Programming language, any formal method, any theory of scheduling, any fault tolerance technique, and any testing technique will always be flawed or incomplete. To design reliable software we must thus make use of all these concepts in union. Consequently, we will never be able to eliminate testing completely. However, with respect to testing, debugging and monitoring of real-time systems software there has been very little work done. We will therefore in the remainder of this thesis discuss and give solutions to the peculiarities of monitoring, debugging and testing of single program real-time systems, multitasking real-time systems, and distributed real-time systems.

3 THE SYSTEM MODEL AND TERMINOLOGY

In this chapter we define a basic system model which we will refine and extend in subsequent chapters. We will also introduce basic terminology and vocabulary.

3.1 The system Model

We assume a distributed system consisting of a set of nodes. Each node is a self sufficient computing element with CPU, memory, network access, a local clock and I/O units for sampling and actuation of an external process. We further assume the existence of a global synchronized time base [27][51] with a known precision δ, meaning that no two nodes in the system have local clocks differing by more than δ.

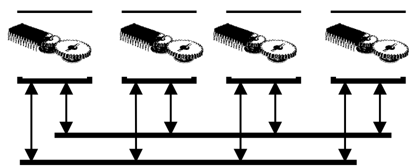


Figure 3-1. A distributed system.

Figure 3-2.An observer.

The software that runs on the distributed system consists of a set of concurrent tasks and interrupt routines, communicating by message passing or via shared memory, all governed by a real-time kernel. Tasks and interrupts may have functional and temporal side effects due to preemption, message passing and shared memory.

We assume that there exists a set of observers, which can observe/monitor the system behavior. These observers can be situated on different levels, ranging from dedicated nodes, which eavesdrop on the network, to programming language statements inside tasks that outputs significant information. These observers are fundamental for monitoring, testing and debugging of real-time systems (RTS) and distributed real-time systems (DRTS).

We will in subsequent chapters make additions and refinements to this system model.

3.2 Terminology

In this section we will define some of the basic vocabulary that is used in the remainder of the thesis. We begin with failures, and conclude with determinism and reproducibility. We will refer to software modules, tasks, and groups of tasks as components.

3.2.1 Failures, failure modes, failure semantics, and hypotheses

What constitutes a failure, what is a failure mode, what is a fault hypothesis?

3.2.1.1 Fault, error, and failure

Definition. A *failure* is the nonperformance or inability of the system or component to perform its intended function for a specified time under specified environmental conditions [67]. That is, an input, X, to the component, O, yields an output, $O(X)$, non-compliant with the specification.

Definition. An *error* is a design flaw, or a deviation from a desired or intended state [67]. That is, if we view the program as a state machine, an error (bug) is an unwanted state. We can also view an error as a corrupted data state, caused by the execution of an error (bug) but also due to e.g., physical electromagnetic radiation.

Definition. A *fault* is the adjudged (hypothesized) cause for an error [59]. Generally a failure is a fault, but not vice versa, since a fault does not necessarily lead to a failure.

The relation between the definitions of fault, error, and failure, is depicted in *Figure 3-3*.

Figure 3-3. Cause consequence diagram of fault, error and failure.

Systematic and physical failures

Failures are usually divided into two categories:

- *Systematic failures* which are caused by specification or design flaws, i.e., behaviors that do not comply with the goals of the intended, designed and constructed system. Examples of contributing causes, are erroneous, ambiguous, or incomplete specifications, as well as incorrect assumptions about the target environment. Other examples are failures caused by design and implementation faults. Wear, degradation, corrosion, etc. do not cause these types of failures, all errors are built in from the beginning, and no new errors will be added after deployment.

  Definition. A systematic *failure* occurs if and only if:

  1) the location of an error is executed in the program,
  2) the execution of the error leads to an erroneous state, and
  3) the erroneous state is propagated to the output.

  This means, that if an error is not executed it will not cause a failure. If the effect of the execution of the error (infection) is indistinguishable from a correct system state it will not cause a failure. If the system's state is infected but not propagated to output there will be no failure.

- *Physical failures* which are the result of a violation upon the original design. Environmental disturbances, wear or degradation over time may cause such failures. Examples, are electromagnetic interference, alpha and beta radiation, etc.

  Definition. A physical *failure* occurs if and only if:

  1) the system state is corrupted or infected, and
  2) the erroneous state is propagated to the output.

  Fault-tolerance mechanisms usually try to prevent (1) by applying robust designs, and (2) by applying redundancy, etc.

3.2.1.2 Failure modes

Depending on the architecture of the system we can assume different degrees, and classes, of failure behavior. That is, certain types of failures are extremely improbable (impossible) in some systems, while in other systems it is very likely that they occur. For example, consider multitasking systems where we have to resolve access to shared resources by means of mutual exclusion. One approach is to make use of semaphores, and another to make use of separation in time. In the latter case deadlock situations are impossible, while in the previous case deadlocks certainly are possible. Using synchronization in time we thus eliminate an entire class of failures, and can therefore during testing eliminate the search for them.

Components can fail in different ways and the manner in which they fail can be categorized into failure modes. The failure modes are defined through the effects, as perceived by the component user. We are going to present categories, i.e., failure modes, (1 to 6) ranging from failure behavior that sequential programs, or single tasks in solitude, can experience, to failure behavior that is only significant in multitasking, distributed systems and real-time systems, where more than one task is competing for the same resources, e.g., processing power, memory, computer network, etc.

Failure modes:

1. ***Sequential failure behavior*** (Clarke et. al. [17]):
   - *Control failures*, e.g., selecting the wrong branch in an if-then-else statement.
   - *Value failures*, e.g., assigning an incorrect value to a correct (intended) variable.
   - *Addressing failures*, e.g., assigning a correct (intended) value to an incorrect variable.
   - *Termination failures*, e.g., a loop statement failing to complete because the termination condition is never satisfied.
   - *Input failures, e.g.*, receiving an (undetected) erroneous value from a sensor.

Multitasking and real-time failure behavior

2. ***Ordering failures***, e.g., violations of precedence relations or mutual exclusion relations.
3. ***Synchronization failures***, i.e., ordering failures but also deadlocks.
4. ***Interleaving failures***, e.g., unwanted side effects caused by non-reentrant code, and shared data, in preemptively scheduled systems.
5. ***Timing failures***. This failure mode yields a correct result (value), although the procurement of the result is time-wise incorrect. For example, deadline violations, too early start of task, incorrect period time, too much jitter, too many interrupts (too short inter-arrival time between consecutive interrupt occurrences), etc.
6. ***Byzantine and arbitrary failures***. This failure mode is characterized by a non-assumption, meaning that there is no restriction what so ever with respect to which effects the component user may perceive. Therefore, the failure mode has been called malicious or fail-uncontrolled. This failure mode includes

two-faced behavior, i.e. a component can output “X is true” to one component user, and “X is false” to another component user.

The above listed failure modes build up a hierarchy where byzantine failures are based on the weakest assumption (a non-assumption) on the behavior of the components and the infrastructure, and sequential failures are based on the strongest assumptions. Hence byzantine failures are the most severe and sequential failures the least severe failure mode. The byzantine failure mode covers all failures classified as timing failures, which in turn covers synchronization failures, and so on (see *Figure 4-4*).

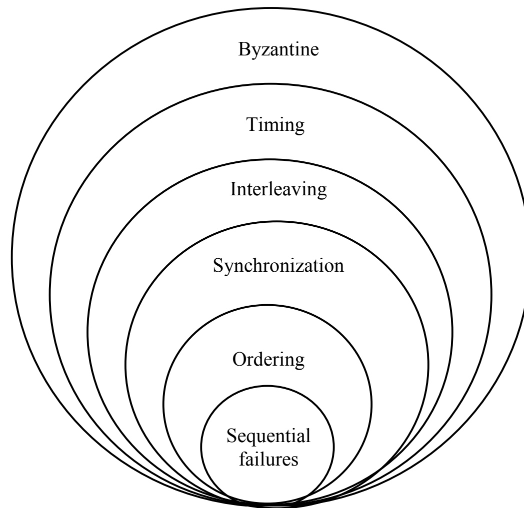


Figure 4-4. The relation between the failure modes.

The component user can also characterize the failure modes according to the viewpoints domain. A distinction can be made between primary failures, secondary failures and command failures (Leveson [67]):

- ***Primary failures***

  A primary failure is caused by an error in the software of the component so that its output does not meet the specification. This class includes sequential and byzantine failure modes, excluding sequential input failures.

- ***Secondary failures***

  A secondary failure occurs when the input to a component does not comply with the specification. This can happen when the component is used in an environment for which it is not designed, or when the output of a preceding task does not comply with the specifications of a succeeding task’s input. This class includes interleaving failures, sequential input failure modes, as well as changed failure mode assumptions.

- ***Command failures***

  Command failures occur when a component delivers the correct result but at the wrong time or in the wrong order. This class covers timing failures, synchronization failures, ordering failures, as well as sequential termination failures.

The persistence of failures

The persistence of failures can be categorized into three groups:

- ***Transient failures***. Transient failures occur completely aperiodic, meaning that we cannot bound their inter-arrival time. They can appear once, and then never appear again. Typically, these types of failures are induced by electromagnetic interference, or radiation, which may lead to corruption of memory, or CPU registers – *bit-flips*. Transient failures are mostly physical failures.

- ***Intermittent failures***. The inter-arrival time of intermittent failures can be bounded with a minimum and/or maximum inter-arrival time. These types of failures typically take place when a component is on the verge of breaking down, for example, due to a glitch in a switch. Examples from the software world could be failures due to sporadic interrupts.

- ***Permanent failures***. A permanent failure that occurs, stays until removed (repaired). A permanent failure can be a damaged sensor, or typically for software, a systematic failure – caused by an error in a program, which stays there until removed.

3.2.1.3 Failure semantics

The above classification of failure modes is not restricted to individual instances of failures, but can be used to classify the failure behavior of components, which is called a component's failure semantics (Poledna [87]):

- ***Failure semantics***

  A component exhibits a given failure semantic if the probability of failure modes, which are not covered by the failure semantic, is sufficiently low.

If a given component is assumed to have synchronization failure semantics, then all individual failures of the component should be synchronization-, ordering-, or sequential failures. The possibility of more severe failures, like timing failures, should be sufficiently low. The failure semantic is a probabilistic specification of the failure modes a component may exhibit. The semantic has to be chosen in relation to the application requirements. In other words, the failure semantics defines the most severe failure mode a component should experience. Fault-tolerant systems are designed with the assumption that any component that fails will do so according to a given failure semantic. When we *test* a system we do so also with a certain failure semantic in mind. That is, we look for failures of a certain kind. For plain sequential programs we usually do not look for interleaving failures, or timing failures. However, if the component will be used in a multitasking or real-time system we certainly have to look for these types of failures.

3.2.1.4 Fault hypothesis

When a system is designed for fault-tolerance or when testing is performed it is always based on a *fault hypothesis*, which is simply the assumption that the system will behave according to a certain failure semantic.

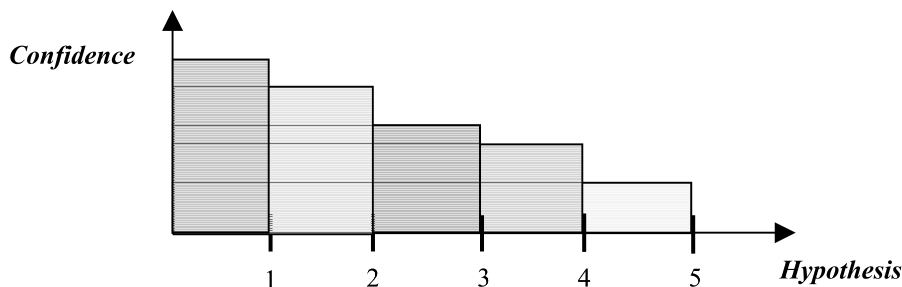


Figure 3-5. The achieved confidence (of the reliability) for different fault hypothesis.

This means that if a system is tested with a specific fault hypothesis, and a certain confidence in its reliability is achieved (*Figure 3-5*), then if we later assume a more severe fault hypothesis, the confidence in the achieved reliability decreases (*Figure 3-6*). For example, if we have tested a system, which has memory protection, and then remove the memory protection we cannot say anything about the achieved reliability with respect to that fault hypothesis. Changes of this type typically give rise to secondary failures.

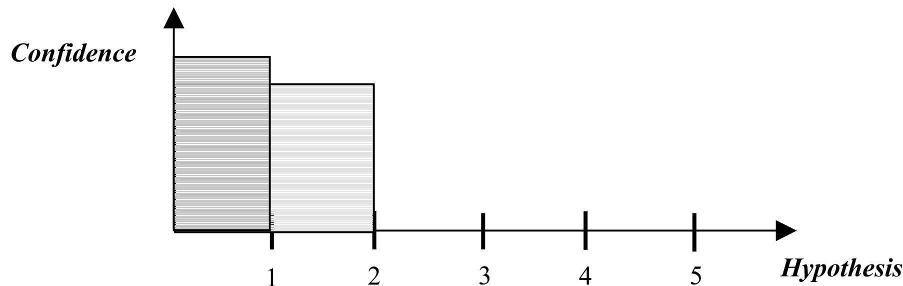


Figure 3-6. The confidence in the reliability for more severe fault hypothesis collapses when basic assumptions do not hold due to e.g., the removal of memory protection.

3.2.2 Determinism and reproducibility

Sequential programs are usually regarded as having deterministic behavior, that is, given the same initial state and inputs, the sequential program will consistently produce the same output on repeated executions, even in the presence of systematic errors. For example,

```
int SUM(int a, int b, int c)
{
        int s;
        s = a+b;
        printf("c=%d\n", c);
        return(s);
}
```

Given that the parameters *a* and *b* were equal on repeated calls to *SUM*(*a,b,c*) then the function would deterministically reproduce the sum of *a* and *b* – regardless of the value of *c*.

The determinism of a system with respect to an observed behavior can be defined as:

Definition. *Determinism*. A system is defined as deterministic if an observed behavior, *P*, is uniquely defined by an observed set of necessary and sufficient parameters/conditions, *O*.

Definition. *Partial Determinism*. A system is defined as partially deterministic if an observed behavior, *P*, is uniquely defined by a *known* set of necessary and sufficient parameters/conditions, *O,* but the observations are limited to a subset of *O*.

The implication of the definition of determinism is that if we have a function *f(a*, *b*, *c)* and the observed behavior, *P,* of this function is deterministically determined by the necessary and sufficient conditions (or parameters) of *a* and *b*, then we can execute the function *f(a, b, c)* an infinite number of times and deterministically

observe this behavior by observing the output of *f* and by observing *a* and *b*. The value of *c* is of no significance because it is not necessary for *P's* determinism. If we can also control, not only observe, the values of *a* and *b* we can also reproduce the observation of behavior *P*.

Definition. *Reproducibility*. A system is reproducible if it is deterministic with respect to a behavior *P*, and if it is possible to *control* the entire set of necessary and sufficient conditions, *O*.

Definition. *Partial reproducibility*. A system is partially reproducible if it is deterministic with respect to a behavior *P*, and if it is possible to *control* a subset of the necessary and sufficient conditions, *O*.

Hence, the relation is such that the property reproducibility is stronger than the property determinism, i.e., if some observations are reproducible they are deterministic, but not necessarily vice versa, thus:

Determinism ⊂ Partial reproducibility ⊂ Reproducibility

This is an important distinction to make, since the desired behavior, the fault hypothesis and the infrastructure dictates how many conditions/variables/factors we need to observe in order to guarantee determinism of observations, as well as how many conditions we must control for reproducibility of observations.

4 MONITORING DISTRIBUTED REAL-TIME SYSTEMS

We will in this chapter discuss how to observe the behavior of embedded real-time systems (RTS), and how to observe and correlate observations between nodes in distributed real-time systems (DRTS). There are two significant differences between debugging and testing of software for desktop computers and embedded real-time systems:

- It is more difficult to observe embedded computer systems, simply because they are embedded, and that they thus have very few interfaces to the outside world, and
- the actual act of observing RTS and DRTS can change their behavior.

In order to dynamically verify a system, i.e., to test or debug it, we must *observe* its run-time behavior and deem how well these observations comply with the system requirements. Fundamental in all physical sciences, as well as in testing of software, is the non-ambiguity, or determinism, of observations, and the ability to reproduce observations. Of equal importance is that the actual act of observation does not disturb, or intrude on, system behavior. If nonetheless the observations are intrusive then it is imperative that their effect can be calculated and compensated for. If we cannot, there are no guarantees that the observations are accurate or reproducible.

Race conditions with respect to order of access to shared resources occur naturally in multi-tasking real-time systems. Different inputs to the racing tasks may lead to different execution paths. These paths will in turn lead to different execution times for the tasks, which depending on the design may lead to different *orders* of access to the shared resources. As a consequence there may be different system behaviors if the outcome of the operations on the shared resources depend on the ordering of accesses.

Example 4-1.

Consider figures *4-1* and *4-2*. Assume that two tasks *A* and *B*, use a shared resource *X,* which they both do operations on, and that the resource *X* is protected by a semaphore *S*. Task *B* has higher priority than task *A*. Depending on the inputs, the execution time of task *A* will vary, which will result in different accesses to the shared resource:

(1) *Figure 4-1* illustrates a scenario, in which task *B* locks the semaphore, and enters the critical region before task *A*. Task *B* then preempts *A* and performs an operation on *X*. The new value of *X* is *B*(*X*). The entire transaction will yield a value of *X* corresponding to *A*(*B*(*X*)).

(2) *Figure 4-2* illustrates a different scenario, in which task *A* terminates before task *B* is released, and thus performs an operation on *X* before *B*. The new value of *X* is *A*(*X*). The entire transaction will yield a value of *X* corresponding to *B*(*A*(*X*)).

If we now add a probe to task *A*, in order to test its behavior, we may extend its execution time so that only scenario (1) is run. Consequently scenario (2) will never be executed during run-time, and if *B*(*A*(*X*)) is erroneous due to e.g., an error in task *A,* this will not be revealed during testing. If we later, after satisfactory testing, remove the probe in task *A*, scenario (2) may occur again and the erroneous calculation *B*(*A*(*X*)) may be executed, leading to a failure. This non-

deterministic effect of intrusively observing a system is called the *probe-effect* [31][78] or the *Heisenberg uncertainty in software* [64][77].

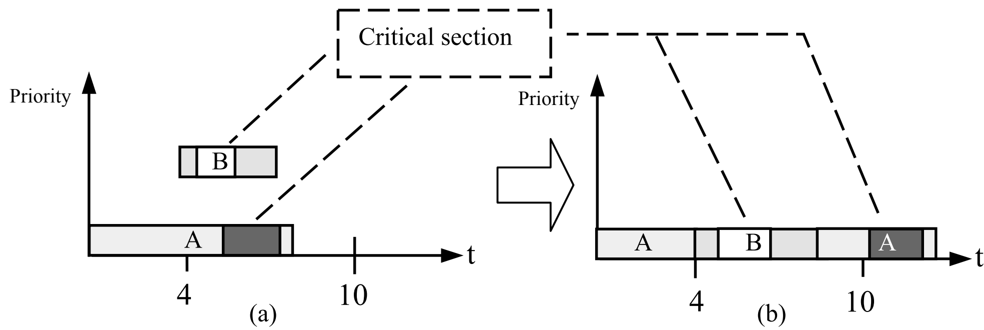


Figure 4-1. The releases of tasks A and B - Figure (a). Where task B has higher priority than task A. Task B enters the critical section before A, when A has its worst case execution time. Figure (b) depicts the resulting execution, where A is preempted by B.

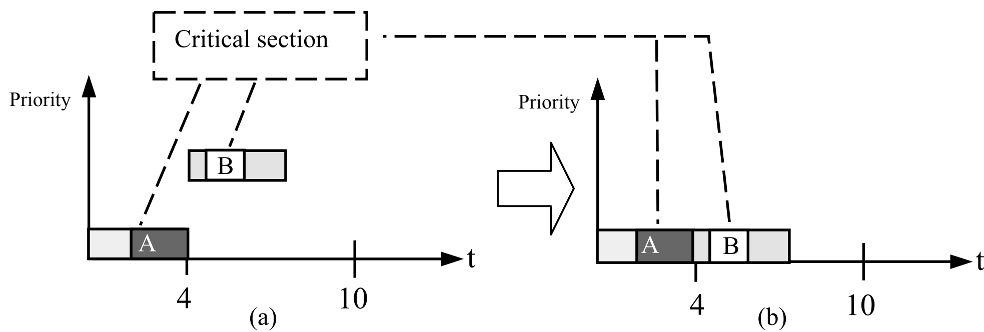
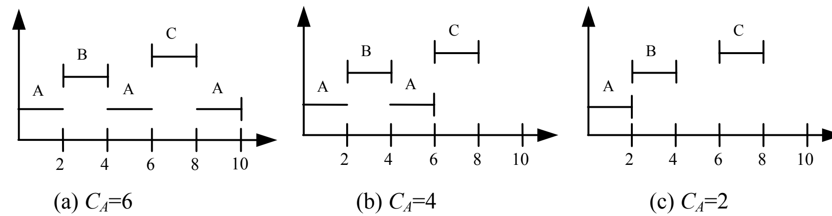


Figure 4-2. The releases of tasks A and B - Figure (a). Where B has higher priority than task A. Due to shorter execution time task A starts and terminates before task B is released. Figure (b) depicts the resulting execution, where A precedes B.

Example 4-2.

Consider *Figure 4-3* which depicts the execution orderings of tasks during the Least Common Multiple (LCM) of the period times of the involved tasks *A*, *B* and *C*, based on a schedule generated by a static off-line scheduler and where later release time gives higher priority. Due to a varying execution time of task *A,* with a best case execution time (*BCET*) of 2 and a worst case execution time (*WCET*) of 6*,* we get three different scenarios, depicted in figures *4-3(a–c).* As exemplified below the execution of these different execution orderings will give different results.



(a) C_A=6 (b) C_A=4 (c) C_A=2

Figure 4-3 Three different execution order scenarios.

Assume in addition that all tasks call a common subroutine *f*(), that is by error non-reentrant, and that the tasks *A*, *B*, and *C* execute the program code in *Figure 4-4*. Task *A* is also in error by assigning 7 to *b,* when it should be 10. A critical point for the values computed by task *A* is indicated in the code for task *A*, by the preemption point.

int f(int a)
{
   int sum;
 	
   sum =a+b;
   return(sum);
}
	Task A:
	int b; /* global*/
	int ans;
	
	/*1st assignenment in prg. */
	b=7; /* error*/
	…
	…
	/* Preemption point */
	…
	…
	/*last assignment in prg.*/
	ans = f(3);
	...
		Task B:
		...
		int ans;
		...
		b=10;
		ans = f(2);
		...
			Task C:
			...
			int ans;
			...
			b=10;
			ans = f(5);
			...

Figure 4-4. The tasks A, B and C and the called function f()

The values calculated for task *A*, depending on which scenario is run, would thus be scenario (a) ans = 13 (correct), scenario (b) ans = 13 (correct), and scenario (c) ans = 10 (erroneous).

Hypothesize now, that we add a probe to task *A*, in order to test its behavior, and thus extend its execution time to always exceed its BCET. As a consequence scenario (c) will never be executed during run-time, and the error in task *A* will not be revealed during testing. If we later, after satisfactory testing, remove the probe in task *A*, scenario (c) can occur again and task *A* will fail. Thus giving rise to the probe effect.

Besides race-conditions, and the occurrence of the probe-effect in DRTS, there is also a difference between DRTS and sequential software with respect to control. Achieving deterministic observations of regular sequential programs is easy because in order to guarantee reproducibility we need only control the sequence of inputs and the start conditions [78]. That is, given the same initial state and inputs, the sequential program will deterministically produce the same output on repeated executions, even in the presence of systematic faults [94], or in the presence of intrusive probes. Reproducibility is essential when performing regression testing or cyclic debugging [92][96], where the same test cases are run repeatedly with the intent to validate that either an error correction had the desired effect, or simply to make it possible to find the error when a failure has been observed [59], or to show that no new errors have been introduced when correcting another error. However, trying to directly apply test techniques for sequential programs on distributed real-time systems is bound to lead to non-determinism and non-reproducibility, because control is only forced on the inputs, disregarding the significance of order and timing of the executing and communicating tasks.

Consequently, in order to facilitate reproducible monitoring of DRTS we must:

1. Reproduce the inputs with respect to contents, order, and timing
2. Reproduce the order and timing of the executing and communicating tasks.
3. Eliminate the probe-effect.

However, if deterministic monitoring is sufficient it is enough to only observe all entities with respect to contents, order and timing. A system can be defined as deterministic with respect to a certain set of behaviors if we can observe all necessary and sufficient conditions for the set of behaviors to occur. As for sequential software, it would be necessary to observe inputs and outputs in order to deterministically deem if the outputs comply with the requirements. If the control flow of the sequential program also depends on random number generators, we would have to observe these also for determinism. For a multitasking real-time systems, with ordering failure semantics or stricter semantics assumed, it would be necessary to observe contents, order and timing of all inputs, outputs, and executions of the involved tasks in order to deterministically deem if the system fulfills its requirements. For reproducibility however, it would also be necessary to control all necessary and sufficient conditions for a set of behaviors to deterministically occur. A system is partially reproducible if we can deterministically observe it but only control some of the necessary and sufficient conditions.

Reproducibility is a necessity when debugging, when regression testing [92], or when sufficient coverage during testing is sought (we will in section 4.5, chapter 5 and section 6.4.4 elaborate on this).

Contributions

- In this chapter we present a software-based method for deterministic observations of single tasking, multi-tasking, and distributed real-time systems.
- This includes a description of what to observe, how to eliminate the disturbances caused by the actual act of observing, how to correlate observations between nodes, and how to reproduce the observations.
- We will give a taxonomy of different observation techniques, and discuss where, how and when these techniques should be applied for deterministic observations.

- We argue that it is essential to consider monitoring early in the design process, in order to achieve efficient and deterministic observations.

4.1 Monitoring

Research on testing of shared memory multiprocessor systems and distributed systems have been penetrated in some detail over the years. The focus has mainly been on monitoring, i.e. gathering of run-time information for debugging [86][111][112] and performance measurements [11].

The research issues have been:

- The intrusiveness (perturbation) of observations in software [31][64][78] and how to eliminate the perturbations using special hardware [113].
- How to deterministically reproduce the observations using mechanisms in software [22][63] [104] or mechanisms in hardware [113]
- The problem of defining a global state in distributed systems [29] using logical clocks [12][58] or synchronized physical clocks [51][53][87].

However, the number of references on research regarding monitoring for testing and debugging of single node real-time systems, and multiple node (distributed) real-time systems, that consistently handle time, distribution, interrupts, clock synchronization, and scheduling, dwindle fast (to zip).

The observational requirements for testing and debugging differ, in the amount and type of information required. The quintessential difference comes from the fact that testing is used for finding failures (or showing their absence), while debugging is used for finding the errors that cause the failures. Another difference is that testing can easily be automated, while debugging is essentially a manual task. For the verification of safety-critical software (failure rates of 10^{-4} to 10^{-9} failures/hour) it is necessary that the test process can be automated since the number of test cases required is enormous [10][70] (see chapter 2).

For testing of sequential programs it is usually sufficient to monitor inputs, and outputs via predefined interfaces of the programs, and based on that information deem, according to the specification, if a test run was successful or not. For (distributed) real-time systems we need also observe the timing and order of the executing and communicating tasks, since the outputs depend on these variables, and thus also the determinism of the observations.

To detect errors using debugging it is also necessary to monitor the internal behavior of the programs with respect to intermediate computed values, internal variables, and the control flow. For interactive debugging, in the classical sense of sequential programs, it is required that the control flow can be altered via manual or conditional breakpoints, or via traces, all in order to be able to increase the observability of the program. Consequently, for debugging of (distributed) real-time systems, we need to control the timing and order of the executing and communicating tasks, otherwise we cannot achieve deterministic debugging. However, the problem of defining a global state in distributed real-time systems, and break-pointing tasks on different nodes at exactly the same time, is a serious obstacle when debugging. Either we need to send *stop* or *continue* messages from one node to a set of other nodes with the problem of nonzero communication latencies, or we have a priori agreed upon times when the execution on the processors should be halted or resumed. In the latter case there is the problem of non-perfectly synchronized clocks, so the tasks may not halt or resume

their execution at the same time. Most real-time systems are also driven by the environment, so just because we breakpoint one task on one node, does not stop the external process.

When monitoring a DRTS there are some fundamental questions that must be answered:

- How to extract enough information from the system?
- How to eliminate the perturbations that the observations cause?
- How to correlate the observations, i.e., how to define a consistent global state?
- How to reproduce the observations?

We are now going to address each of these questions in turn.

4.2 How to collect sufficient information

The amount of monitoring needed in order to collect sufficient information is dependent on two basic factors:

- *What is the fault hypothesis*? That is, the more severe failure semantics, the more information we need to store and process in order to achieve deterministic observations. For sequential software it is sufficient to observe inputs and outputs. For multi-tasking systems we need also observe task execution orderings and their access to shared resources. For real-time systems we further need to observe the timing of the tasks. However, if we want to test a multitasking real-time system and assume sequential failure semantics we need only test tasks in solitude, since we can regard each task as a sequential program. The probability that the multitasking real-time system will only exhibit sequential failure behavior in practice is not very high though. It is therefore very important to chose a realistic fault-hypothesis and observe the system based on that fault hypothesis.
- *What is the a priori knowledge* of the system with respect to its behavior and attributes? The validity of the fault-hypothesis is based on the support that the environment and the infrastructure (real-time kernel, hardware, etc.) give the fault hypothesis.

  Does the system have memory protection? How does the system synchronize access to shared resources (time or semaphores)? Is the execution strategy time-triggered or event-triggered?

  For example, if the system is time-triggered and scheduled using e.g., strict periodic fixed priority scheduling or static scheduling, we know that the system will repeat its execution every LCM. For event-triggered systems we have no such general limit and might have to observe and store copious amounts of information.

The actual information to be observed can be categorized into three groups:

- Data flow (internal and external)
- Control flow (execution and timing)
- Resources (memory and execution resources)

Data flow

- *Inputs* – determine for which input the task will execute, this is important if the actual input is not provided by a test oracle, but rather by an external process or an environment simulator.
- *Outputs* – what are the produced outputs via the predefined interfaces of the task? This includes messages that are passed between the system nodes, in a DRTS.
- *Auxiliary outputs* – output intermediately computed values, or program state, which are not visible via the predefined interfaces. For sequential software these are commonly implemented using e.g., assertions, or even using printf statements in C. For distributed real-time systems the situation is more complex and we need to define the type of data we *additionally* need. Typically these are related to memory mapped I/O interfaces, for example received messages over the network, readings of A/D converters, readings of the local clock, etc. Because any additional outputs will require more memory, communication bandwidth, and execution time, we need to take these auxiliary outputs into account when designing and scheduling, in order to avoid the probe-effect. These auxiliary outputs could also be parameterized, i.e., we can during run-time switch between different auxiliary outputs, given of course that this parameterization is designed in such a way that the timing behavior is constant.

Control flow

- *Inputs, outputs, auxiliary outputs, and inter-node messages*. At what time and in what order were the inputs received? At what time and in what order were the outputs produced?
- *Task switches* – which, when, and in what order are tasks starting, preempting, and finishing? We can make use of this information for deriving the synchronization and execution orderings between tasks. We can also make use of the timing information in order to deem if tasks start too early, too late, finish too late or finish too early. We can further measure the periodicity of the tasks and thus deem if jitter requirements are met. Making use of this timing information we can also measure the execution times of the tasks.
- *Interrupts* – which, when, how long, and in what order are interrupts interfering with tasks. Using this information we can judge how the interrupts interfere with the execution of the tasks. We can thus measure if basic assumptions of interrupt overhead are true.
- *Real-time kernel overhead*. What is the execution time of the real-time kernel? What are the latencies due to interrupt disable, that is, when the kernel needs to perform atomic operations it usually disables all interference by interrupts, for how long time can the kernel block all interrupts?

- *Tick rate*. The tick rate is the frequency at which the real-time kernel is invoked, and at which new scheduling decisions are taken. However, the tick rate can vary due to global clock synchronization. That is, the inter arrival time between ticks might increase or decrease if the local clock is to slow or to too fast compared to the global time base.

Resources

- *Memory use* – stack use, etc. How much of the memory is used by the tasks, interrupt service routines, or the kernel?
- *CPU utilization*. How much of the CPU's calculating power is used?
- *Network utilization*. How much of the networks bandwidth is used?
- *The state of the real-time kernel*: Which tasks are waiting for their turn to execute (waiting queue, list, or table)? Which task is running? Which tasks are blocked?

It is very important to deem which information is necessary for monitoring of the system, because if you extract too much there will be a heavy performance and cost penalty. If you extract too little, the precision of the observations will be too coarse or simply non-deterministic for judging how and why the system behaved as it did. For determinism there is a least necessary level of observability required, i.e., we need to observe the necessary and sufficient parameters as defined in section 3.2.2. Any additional (useful) information surpassing the level of necessity for determinism will increase the precision of the observations. Think of inputs, and outputs for sequential software as the least necessary level for determinism, while debugging provides a higher level of observability (precision) since we can inspect the internal control flow and the contents of the variables.

4.3 Elimination of perturbations

After a decision on what entities to observe we need to decide on how to eliminate the probe-effect. There are basically three approaches (1) non-intrusive/passive hardware, (2) intrusive software instrumentation, and a hybrid (3) where the software instrumentation is minimized.

4.3.1 Hardware monitoring

A transparent non-intrusive approach towards monitoring, is the application of special hardware, e.g. hardware that allows buss sniffing, or non-intrusive access to memory via dual-port memories, etc., but also through the use of hardware CPU emulators, (Lauterbach et al. [61]). Hardware monitoring has been applied for performance measurements [8], execution monitoring of multiprocessor systems [73], and real-time systems [74][86][113]. Since the monitoring hardware is interfaced to the target system's hardware via the CPU socket (emulator) or via the data and address busses, it can observe the target system without interfering with its execution, and thus not introduce any probe-effects. The drawbacks are that the monitoring mechanisms must be very target specific and therefore very expensive, but also that the observations will be on a very low level of detail, since only the external interfaces of the microprocessors and shared resources such as dual-port memories, can be monitored. The ever-increasing integration of functionality in current general-purpose micro-controllers makes it correspondingly harder to observe the internal behavior of the micro controllers/CPUs, due to cache memory, on-chip memory, etc. Hardware

monitoring must also be considered early in the design of the system since monitoring mechanisms will be difficult to integrate when the rest of the hardware configuration is set. Non-intrusive monitoring of distributed real-time systems also requires that we have dedicated monitoring hardware on each node, and that the nodes are interconnected via a dedicated monitoring network for data transfer and synchronization, in order to avoid the probe-effect. We need also establish a globally synchronized time base, relative which all observed events on the nodes can be correlated otherwise there can be no guarantees of the consistency between observations.

It can be argued that the cost for the monitoring hardware will only impact the development budget, not the production cost, since the monitoring hardware can be removed from the target system. Experience of software development has however shown that maintainability is a necessity also after deployment. The non-portability, the lack of scalability and the observations low level of detail severely limit the viability of the hardware approach. The current trend of making application specific hardware using FPGAs and VHDL [11] gives, however, an opportunity to conveniently integrate non-intrusive monitoring mechanisms in the hardware for single node systems.

4.3.2 Hybrid monitoring

In order to increase the level of abstraction and decrease the amount of information recorded, hybrid approaches to monitoring have been suggested. “Triggers” are implemented in software, which using a minimum number of instructions assists the hardware in recording significant events. Software triggers do for example, write to specific addresses that are memory mapped with the monitoring hardware, or use special co-processor instructions. When the monitoring software writes to these addresses the hardware records the data passed on the data bus of the processor. Using this approach the limitations of hardware monitoring can be alleviated, although the cost and non-portability issues still remain. The monitoring instructions in the software must also be resource adequate, and remain in the target system in order to avoid the probe-effect. Hybrid performance monitoring of distributed systems have been covered by Haban et al. [36], and performance monitoring of multiprocessor systems by Mink et al. [81][91][34].

4.3.3 Software monitoring

Historically, the contributing motivations for using hardware, and hybrid, monitoring approaches have been the problem of predicting the perturbations caused by instrumenting software [28]. That is, any instrumentation of the software will require memory and execution time resources, while hardware can passively monitor the system with no interference. For software instrumentation there will be a probe-effect, if the probes are removed after satisfactory monitoring, or if the probes are added to a system that has already been shown, e.g., using scheduling theory, to always meet its deadlines. If the probes are not removed there will be a financial penalty due to the dedicated resources (memory, processing, bandwidth, etc.) or they might hamper the performance of the system.

In order to test and debug a system to satisfactory levels of reliability we fundamentally need to observe the system, and by including instrumentation code in the software (application and kernel), we can observe significantly more than possible with hardware approaches. Software monitoring of real-time systems have been covered by Chodrow et al. [14], distributed systems by Joyce et al. [46][80], and distributed real-time systems by Tokuda et al. [111][90]. In general it is necessary to leave the software probes in the target system in order to eliminate the probe-effect. If the target system is a real-time system, which can be scheduled e.g., using fixed priority or static scheduling it is straightforward to analyze the effects that the probes have on the system. Just make the probes part of the design, i.e., allocate execution time and memory, and then make use of execution time analysis [89] and scheduling theory [4][75][117]. For monitoring of distributed real-time systems we need also to allocate communication bus bandwidth and account for the probes when making the global schedule. We need also establish a global time-base in order to correlate observations on different nodes.

For software monitoring of distributed real-time systems we have identified the following four different types of probes, depending on where they are implemented:

- *Kernel-probes* – System and kernel level probes, monitor task switches, interrupt interference, etc. (*Figure 4-5*). These types of probes are typically not programmable by the application designer, but rather given as an infrastructure by the real-time kernel. In order to avoid the probe effect, these types of probes should be left permanently in the kernel, their contributing overhead must also be predictable, and minimized.

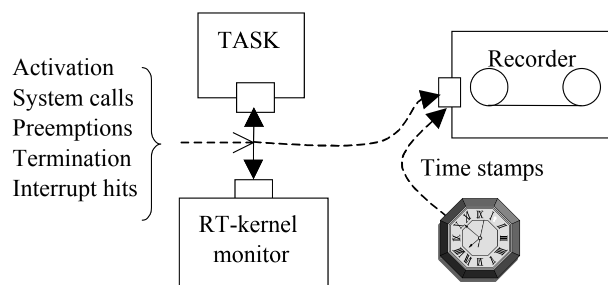


Figure 4-5. Kernel-probe mechanism.

- *Inline-probes* – Are task level probes that add auxiliary outputs to the task they instrument (*Figure 4-6*). These types of outputs are outputs that are regarded necessary from a monitoring, testing or debugging perspective, rather than from a functional application requirement perspective. As they are part of the application code they will also be covered in the estimations, or measurements of the execution times. This also means that we usually need to let them remain in the target system in order to eliminate the probe effect.

```
      ….
      ….
printf("red alart");
      …
```

Figure 4-6. Inline- probe.

- *Probe-tasks* – Are tasks dedicated to collecting data from kernel-probes, inline-probes and other probe-tasks. As depicted in the *Figure 4-7*, a dedicated probe-task receives data from a set of tasks. All probe-tasks must be taken into account when designing and scheduling the system. These types of probes need also remain in the target system in order to avoid the probe effect. We will however soon elaborate on this, and show that there are certain circumstances that allow for these probes to be removed from the target system.

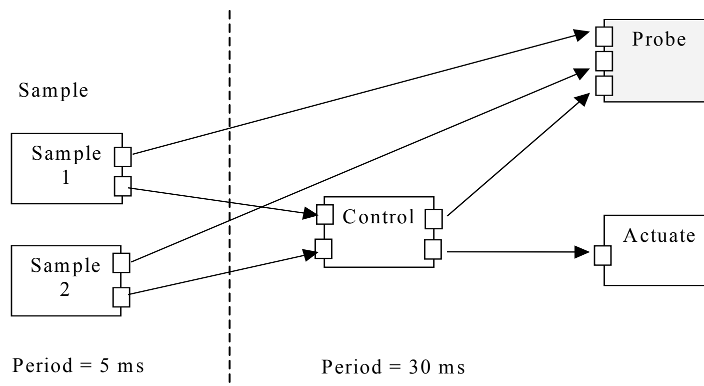


Figure 4-7. Probe-task receives data from other tasks.

- *Probe-nodes* – Are dedicated nodes that collect data from probe-tasks and are able to monitor communication busses (*Figure 4-8*). The probe-node can also analyze the collected data for performance estimations or for testing and debugging. These probe-nodes must also be taken into account when allocating resources for the system since they will require communication bus bandwidth, unless they passively eavesdrop on the network. These types of probes are however easier to remove from the target system since they usually are self-contained computing elements, and can thus be regarded as passively observing hardware monitoring elements, and consequently they can be removed with minimal interference.

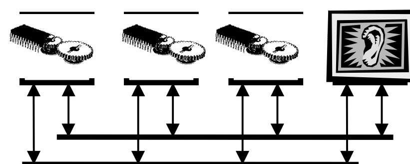


Figure 4-8. Probe-node.

Resource allocation

The prerequisites for avoiding the probe-effect when using the above-defined probes are the allocation of sufficient resources e.g., execution time, network bandwidth and memory. It is not very plausible that these resources will “pop” up in the right place during the test phase if they have not been taken into account during the design phase.

Eliminating the probes?

In most embedded systems the execution time (CPU speed) and memory are limited resources, mostly due to large production volumes, where per unit cost reduction is of significance. From an end-quality, and verification point of view, it is not hard to motivate the extra cost for dedicated probes. That is, you fundamentally need the probes for testing. If you do not have the probes you cannot assess the reliability, or find the errors. Accidents have however, shown that having non-functional code in the target system can be hazardous [67]. Some testing measures are sometimes also deemed so hazardous that they must by all means be eliminated from the target system. Examples include test procedures for train signaling systems, where the test procedures actually change the state of the signals, and consequently could an inadvertent execution of these procedures during runtime cause severe accidents.

So, is there any possibility for us to remove probes after satisfactory testing without introducing the probe-effect? For some execution strategies, e.g., statically scheduled real-time systems, probes can be removed without temporal side-effects if they are situated within *temporal firewalls* [96] (*Figure 4-9*). That is, as long as we do not change the start and completion times of tasks, and change their times of output (communication or access to shared resources), we can remove the probes. The probes can also be eliminated in fixed priority scheduled systems, if we make use of offsets and thus erect temporal firewalls, or if the probes have lower priority than the rest of the tasks in the system (*Figure 4-10*). In the latter case we must also guarantee that the monitoring probes cannot ever block a higher priority task.

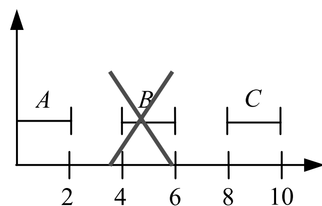


Figure 4-9. Probe task B can be removed due to fixed release times of A and C.

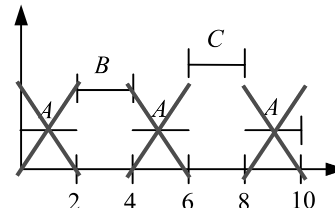


Figure 4-10. Low priority probe task A can be removed without side effects.

If it now would be possible to eliminate the probes, to what end could we use the spare resources (memory and time)? If we remove the probes and thus decrease the processor utilization, we could possibly use the spare resources for non-critical activities (soft real-time tasks) [26]. However, how do we guarantee that the non-critical software does not introduce new errors? In most micro-controllers we do not have memory protection schemes, and consequently can a soft real-time task wreak havoc in the memory space where the hard real-time tasks reside and operate. Could we use a cheaper and slower processor? It is not very likely that we could use a processor with different timing characteristics than the one tested, because all execution times and scheduling are based on the timing specifics of the target

processor. That is, if we change the processor we need to reschedule the entire system, and consequently retest the entire system again; thus gaining nothing.

Memory reduction

One possible benefit however, could be the reduction of memory use. If it can be shown that the removal of probes will not change the functional behavior of the system with respect to memory access, and memory side effects, and all the memory used by the probes have been allocated in a specific address space, we could remove this memory and thus save money.

4.4 Defining a global state

In order to correlate observations in the system we need to know their orderings, i.e., determine which observations are concurrent, and which precede and succeed a particular event. In single node systems or tightly coupled multiprocessor systems with a common clock this is not a problem, but for distributed systems without a common clock this is a significant problem. An ordering on each node can be established using the local clocks, but how can observations between nodes be correlated?

One approach is to establish a causal ordering between observed events, using for example logical clocks [58] derived from the messages passed between the nodes. However, this is not a viable solution if tasks on different nodes work on a common external process without exchanging messages, or when the duration between observed events is of significance. In such cases we need to establish a total ordering of the observed events in the system. This can be achieved by forming a synchronized global time base [27][51]. That is, we keep all local clocks synchronized to a specified precision δ, meaning that no two nodes in the system have local clocks differing by more than δ.

Figure 4-11 illustrates the local ticks in a distributed system with three nodes, all with tick rate Π, and synchronized to the precision δ. There is no point in having $\Pi \leq \delta$, because the precision δ dictates the margin of error of clock readings, and thus a $\Pi \leq \delta$ would result in overlaps of the δ intervals during which the synchronized local ticks may occur [56].

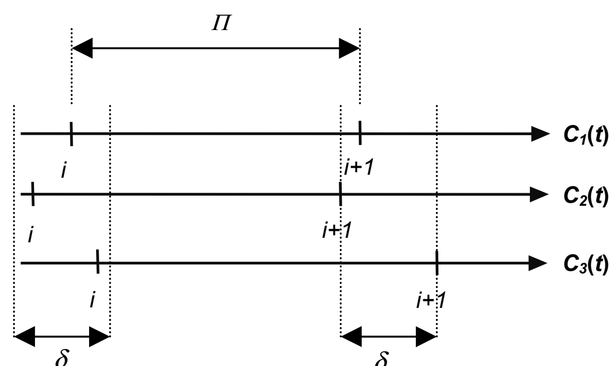


Figure 4-11. The occurrence of local ticks on three nodes

Consider *Figure 4-12*, illustrating two external events that all three nodes can observe, and which they all timestamp. Due to the sparse time base [53] and the precision δ, we end up with timestamps of the same event that differ by 1 time unit (i.e., Π) while still complying with the precision of the global time base. This means that some nodes will consider events to be concurrent (i.e., having identical time stamps), while other nodes will assign distinct time stamps to the same events. This is illustrated in *Figure 4-12*, where node 2 will give the events *e1* and *e2* identical time stamps, while they will have difference 2 and 1 on nodes 1 and 3, respectively. That is, only events separated by more than 2Π can be globally ordered with consensus among the nodes. Due to the precision of the global clock synchronization there is thus a smallest possible granule of time defined by 2δ for deterministic ordering events in the system, since tick overlaps are not acceptable, i.e., $2\Pi > 2\delta$. Consequently the ultimate precision of the global state, i.e., the observed state, will be defined by the precision of the global clock synchronization.

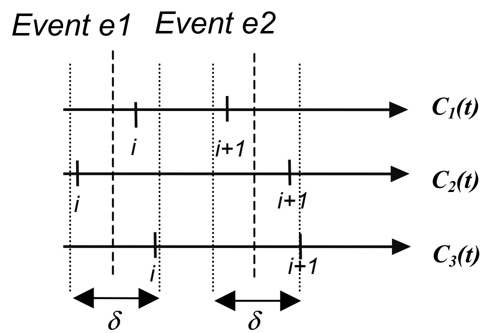


Figure 4-12. Effects of a sparse time base.

4.5 Reproduction of observations

In order to reproduce observations we must bring about the exact same circumstances as when the original observations were made. That is, for distributed real-time systems we need to reproduce the inputs, and as the behavior of a distributed real-time system depends on the orderings, and timing of the executing and communicating tasks, we need also reproduce that behavior in order to reproduce their outputs.

4.5.1 Reproducing inputs

For the reproduction of inputs it is common to use environment simulators [33][65]. The simulators are models of the surrounding environment, e.g., models of the hardware, or the user and user interface, that can simulate the environment's inputs and interactions with the target system, with respect to contents, order and timing.

Classically, the environment simulators have not focused on reproducing inputs to the system, but rather been necessities when the target hardware has not been available, due to concurrent development, or when the application domain has been safety-critical. For the verification of safety-critical systems it is necessary to produce very rare scenarios (10^{-9} occurrences/hour) that would be extremely difficult (or even dangerous) to produce even if the target system and target environment were available to the testers [85][97]. Examples are space applications, weapons systems, and medical treatment devices.

4.5.2 Reproduction of complete system behavior

When it comes to the reproduction of the state and outputs of single tasking RTS, multitasking RTS and DRTS, with respect to the orderings, and timing of the executing and communicating tasks, there are two approaches:

(1) *Off-line replays, i.e., record* the runtime behavior and examine it while replaying it off-line.

(2) *On-line reproduction*, i.e., rerun the system while controlling all necessary conditions.

We will in subsequent chapters discuss and give solutions to (1) and (2). That is, elaborate on how reproduction of observations can be made with respect to debugging and testing of distributed real-time systems.

4.6 Summary

We have in this chapter presented a framework for monitoring single tasking, multi-tasking, and distributed real-time systems. This includes a description of what to observe, how to eliminate the disturbances caused by the actual act of observing (the probe effect), how to correlate observations (how to define a global state), and how to reproduce them. We have given a taxonomy of different observation techniques, and where, how and when these techniques should be applied to obtain deterministic observations. We have argued that it is essential to consider monitoring early in the design process in order to achieve efficient and deterministic observations. Software monitoring is also the preferable approach since it *scales* better than the hardware approaches. Software monitoring can compared to hardware monitoring also observe the system on many levels of abstraction while hardware monitoring is limited to observation of low level details.

5 DEBUGGING DISTRIBUTED REAL-TIME SYSTEMS

We will in this chapter present a method for reproducible debugging of real-time systems (RTS) and distributed real-time systems (DRTS). Classical debugging techniques used for cyclic debugging of sequential programs like breakpoints, watches and single stepping (tracing) cannot immediately be applied to real-time systems software without sacrificing determinism and reproducibility. We will in this chapter discuss and describe a software based technique for achieving reproducible debugging, based on on-line recording and off-line replay, that covers interleaving failure semantics.

Testing is the process of revealing failures by exploring the runtime behavior of the system for violations of the specifications. Debugging on the other hand is concerned with revealing the errors that cause the failures. The execution of an error infects the state of the system, e.g., by infecting variables, memory, etc, and finally the infected state propagates to output. The process of debugging is thus to follow the trace of the failure back to the error. In order to reveal the error it is imperative that we can reproduce the failure repeatedly. This requires knowledge of the start conditions and deterministic executions. For sequential software with no real-time requirements it is sufficient to apply the same input and the same internal state in order to reproduce a failure. For distributed real-time software the situation gets more complicated due to timing and ordering issues.

There are several problems to be solved in moving from debugging of sequential programs (as handled by standard commercial debuggers) to debugging of distributed real-time programs. We will briefly discuss the main issues by making the transition in three steps:

Debugging sequential real-time programs

In moving from debugging sequential non real-time programs to debugging sequential real-time programs temporal constraints on interactions with the external process have to be met. This means that classical debugging with breakpoints and single stepping (tracing) cannot be directly applied, since it would make timely reproduction of outputs to the external process impossible. Likewise, using a debugger we cannot directly reproduce inputs to the system that depend on the time when the program is executed, e.g., readings of sensors and accesses to the local real-time clock. A mechanism, which during debugging faithfully and deterministically reproduces these interactions, is required.

Debugging multi-tasking real-time programs

In moving from debugging sequential real-time programs to debugging multitasking real-time programs executing on a single processor we must in addition have mechanisms for reproducing task interleavings. For example, we need to keep track of preemptions, interrupts, and accesses to critical regions. That is, we must have mechanisms for reproducing the interactions and synchronizations between the executing tasks.

Reproducing rendezvous between tasks has been covered by Tai et al. [104], as have reproduction of interrupts and task-switches using special hardware, by Tsai et al. [113]. Reproducing interrupts and task switches using both special hardware and software have been covered by Dodd et al. [22]. However, since both the two latter approaches are relying on special hardware and profiling tools they are not very

useful in practice. They all also lack support for debugging of distributed real-time systems, even though Dodd et al. claim they have such support. We will in section 5.5 elaborate on their respective work.

Debugging of distributed real-time systems

The transition from debugging single node real-time systems to debugging distributed real-time programs introduces the additional problems of correlating observations on different nodes and break-pointing tasks on different nodes at exactly the same time.

To implement distributed break-pointing we either need to send *stop* or *continue* messages from one node to a set of other nodes with the problem of nonzero communication latencies, or we need *á priori* agreed upon times when the executions should be halted or resumed. The latter is complicated by the lack of perfectly synchronized clocks (se section 4.4), meaning that we cannot ensure that tasks halt or resume their execution at exactly the same time. Consequently a different approach is needed.

Debugging by deterministic replay

We will in this chapter present a software based debugging technique based on deterministic replay [63][78], which is a technique that records significant events at run-time and then uses the recorded events off-line to reproduce and examine the system behavior. The examinations can be of finer detail than the events recorded. For example, by recording the actual inputs to the tasks we can off-line re-execute the tasks using a debugger and examine the internal behavior to a finer degree of detail than recorded.

Deterministic replay is useful for tracking down errors that have caused a detected failure, but is not appropriate for speculative explorations of program behaviors, since only recorded executions can be replayed.

We have adopted deterministic replay to single tasking, multi-tasking, and distributed real-time systems. By recording all synchronization, scheduling and communication events, including interactions with the external process, we can off-line examine the actual real-time behavior without having to run the system in real-time, and without using intrusive observations, which potentially could lead to the probe-effect [31]. Probe-effects occur when the relative timing in the system is perturbed by observations, e.g., by breakpoints put there solely for facilitating observations. We can thus deterministically replay the task executions, the task switches, interrupt interference and the system behavior repeatedly. This also scales to distributed real-time systems with globally synchronized time bases. If we record all interactions between the nodes we can locally on each node deterministically reproduce them and globally correlate them with corresponding events recorded on other nodes.

Contribution

In this chapter we present a method for debugging of real-time systems, which to our knowledge is

- The first entirely software based method for deterministic debugging of single tasking and multi-tasking real-time systems.
- The first method for deterministic debugging of distributed real-time systems.

5.1 The system model

In addition to the original system model in chapter 3 we assume the use and existence of a run-time real-time kernel that supports preemptive scheduling, and require that the kernel has a recording mechanism such that significant system events like task starts, preemptions, resumptions, terminations and access to the real-time clock can be recorded [108], as illustrated in *Figure 5-1*. The detail of the monitoring should penetrate to such a level that the exact occurrence of preemptions and interrupt interference can be determined, i.e., it should record program counter values where the events occurred. All events should also be time-stamped according to the local real-time clock.

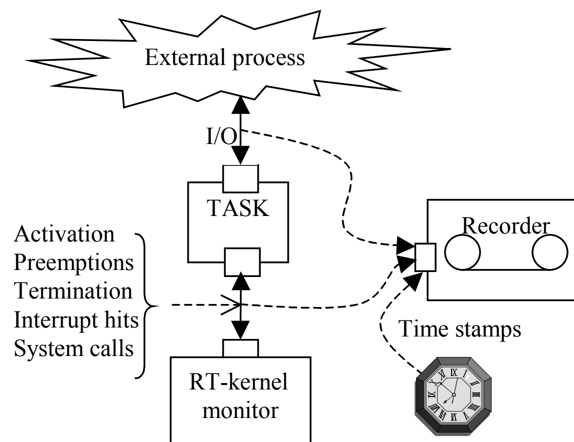


Figure 5-1. Kernel with monitoring and recording.

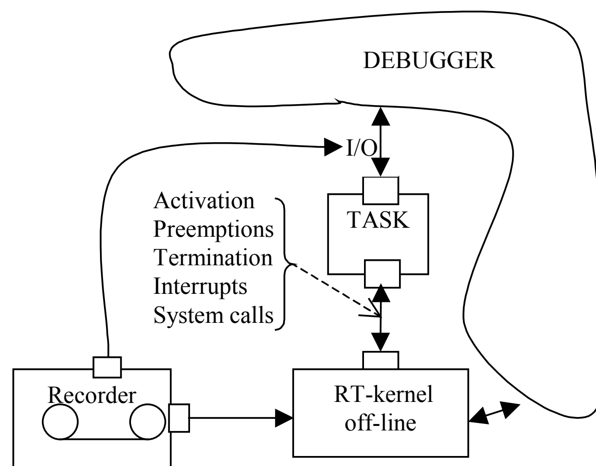


Figure 5-2. Offline kernel with debugger

We further require that the recording mechanism governed by the run-time kernel supports programmer defined recordings. That is, there should be system calls for recording I/O operations, local state, access to the real-time clock, received messages, and access to shared data.

All these monitoring mechanisms, whether they reside in the real-time kernel or inside the tasks, will give rise to probe-effects [31][78] if they are removed from the

target system. That is, removing the monitoring probes will affect the execution, thereby potentially leading to new and untested behaviors.

We further assume that we have an off-line version of the real-time kernel (shown in *Figure 5-2*), where the real-time clock and the scheduling have been disabled. The off-line kernel with support by a regular debugger is capable of replaying all significant system events recorded. This includes starting tasks in the order recorded, and making task-switches and repeating interrupt interference at the recorded program counter values. The replay scheme also reproduces accesses to the local clock, writing and reading of I/O, communications and accesses to shared data by providing recorded values.

5.2 Real-time systems debugging

We will now in further detail discuss and describe our method for achieving deterministic replay. We follow the structure in the introduction and start by giving our solution to debugging sequential software with real-time constraints, and then continue with multitasking real-time systems, and finally distributed multitasking real-time systems.

5.2.1 Debugging single task real-time systems

Debugging of sequential software with real-time constraints requires that debugging is performed in such a manner that the temporal requirements imposed by the environment are still fulfilled. This means, as pointed out in the introduction, that classical debugging with breakpoints and single-stepping cannot be directly applied, since it would invalidate timely reproduction of inputs and outputs.

However, if we identify and record significant events in the execution of the sequential program, such as reading values from an external process, accesses to the local clock, and outputs to external processes, we can order them. By ordering all events according to the local clock, and recording the contents of the events (e.g., the values read) together with the time when they occurred we can off-line reproduce them in a timely fashion. That is, during debugging we "short-circuit" all events corresponding to the ones recorded by substituting readings of actual values with recorded values.

An alternative to our approach is to use a simulator of the external process, and synchronize the time of the simulator with the debugged system. However, simulation is not required if we already have identified the outputs that caused the failure.

5.2.2 Debugging multitasking real-time systems

To debug multitasking real-time systems we need, in addition to what is recorded for single task real-time systems, to record task interleavings. That is, we have to record the transfers of control. To identify the time and location of the transfers we must for each transferring event assign a time stamp and record the program counter (PC).

To reproduce the run-time behavior during debugging we replace all inputs and outputs with recorded values, and instrument all tasks by inserting trap calls at the PC values where control transfers have been recorded. These trap calls then execute the off-line kernel, which has all the functionality of a real-time kernel, but all transfer of control, all accesses to critical regions, all releases of higher priority tasks, and all preemptions by interrupts are dictated by the recording. Inter-process communication

is however handled as in the run-time kernel, since it can be deterministically reproduced by re-executing the program code.

Figure 5-3 depicts an execution with the three tasks *A*, *B*, and *C*. We can see that task *A* is being preempted by task *B* and *C*. During debugging this scenario will be reproducible by instrumenting task *A* with calls to the kernel at *A*'s PC=*x* and PC=*y*.

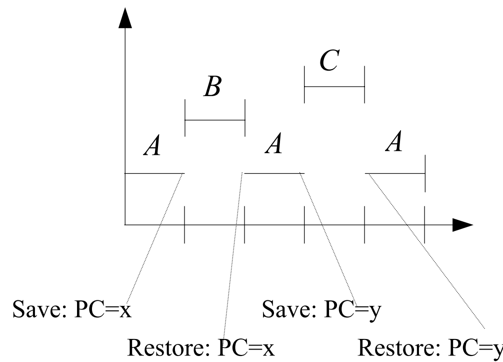


Figure 5-3. Task A is preempted twice by task B and C.

The above reasoning is a bit simplistic when we have program control structures like loops and recursive calls, since in such structures the same PC value will be executed several times, and hence the PC value does not define a unique program state. This can be circumvented if the target processor supports instruction or cycle counters. The PC will together with any of these counters define a unique state. However, since these hardware features are not very common in commercial embedded micro-controllers, we suggest the following alternative approach:

Instead of just saving the PC, we will save all information stored by the kernel in context-switches, including CPU registers (address and data), as well as stack and program counter registers pertaining to the task that is preempted. The entire saved context can be used as a unique marker for the preempted program. The program counter and the contents of the stack register would for example be sufficient for differentiating between recursive calls.

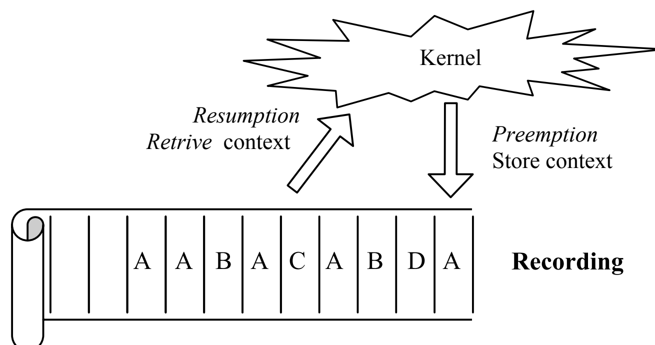


Figure 5-4. A kernel that stores and retrieves contexts from the recording when making context switches.

For loops, this approach is not guaranteed to uniquely identify states, since (at least theoretically) a loop may leave the registers unchanged. However, for most realistic programs the context together with the PC will define a unique state. Anyhow, in the unlikely situation, during replay, of having the wrong iteration of a loop preempted due to indistinguishable contexts, the functional behavior of the replay will be

different from the one recorded – and therefore detectable; or if the behaviors are identical, then it is of no consequence.

Any multitasking kernel must save the contexts of suspended tasks in order to resume them, and in the process of making the recording for replay we must store contexts an additional time. To reduce this overhead we can make the kernel store and retrieve all contexts for suspended tasks from the recording instead, i.e., we need only store the contexts once (see *Figure 5-4*).

Our approach eliminates the need for special hardware instruction counters since it requires no extra support other than a recording mechanism in the real-time kernel. If we nonetheless have a target processor with an instruction counter, or equivalent, we can easily include these counter values into the recorded contexts, and thus guarantee unique states.

To enable replay of the recorded event history we insert trap calls to the off-line kernel at all recorded PC values. During replay we consequently get plenty of calls to the kernel for recorded PC values that are within loops, but the kernel will not take any action for contexts that are different from the recorded one.

An alternative approach to keep track of loop executions is to make use of software instruction counters [79] that count backward branches and subroutine calls in the assembly code. However, this technique requires special target specific tools that scan through the assembly code and instrument all backward branches. The approach also affects the performance, since it usually dedicates one or more CPU registers to the instruction counter, and therefore reduces the possibility of compiler optimizations.

5.2.3 Debugging distributed real-time systems

We will now show how local recordings can be used in achieving deterministic distributed replay. The basic idea is to correlate the local time stamps up to the precision of the clock synchronization. This will allow us to correlate the recordings on the different nodes. As we by design can record significant events like I/O sampling and inter-process communication, we can on each node record the contents and arrival time of messages from other nodes. The recording of the messages therefore makes it possible to locally replay, one node at a time, the exchange with other nodes in the system without having to replay the entire system concurrently. Time stamps of all events make it possible to debug the entire distributed real-time system, and enables visualizations of all recorded and recalculated events in the system. Alternatively, to reduce the amount of information recorded we can off-line re-execute the communication between the nodes. However, this requires that we order-wise synchronize all communication between the nodes, meaning that a fast node waits up until the slow node(s) catch up. This can be done truly concurrently using several nodes, or emulated on a single node for a set of homogenous nodes.

Global states

As defined in section 4.4 we can only globally order events separated by more than 2Π, where Π is the inter-arrival time of operating system clock-ticks. Due to the precision of the global clock synchronization there is thus a smallest possible granule of time defined by 2δ for deterministic ordering events in the system, since tick overlaps are not acceptable, i.e., $\Pi > \delta$. Consequently the precision of the global state, i.e., the observed state, will be defined by the precision of the global clock synchronization.

5.3 A small example

We are now going to give an example of how the entire recording and replay procedure can be performed. The considered system has four tasks *A*, *B*, *C*, and *D* (*Figure 5-5*). The tasks *A*, *B*, and *D* are functionally related and exchange information. Task *A* samples an external process via an analog to digital converter (A/D), task *B* performs some calculation based on previous messages from task *D*, and task *D* receives both the processed A/D value and a message from *B*; subsequently *D* sends a new message to *B*.

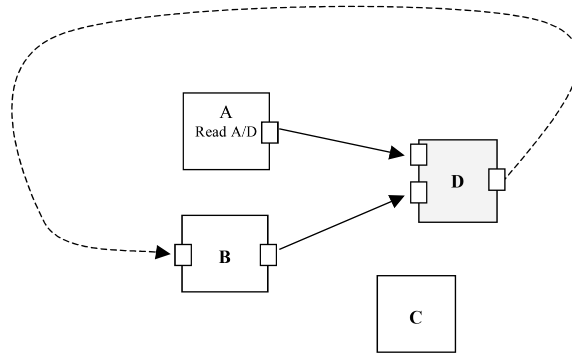


Figure 5-5 The data-flow between the tasks.

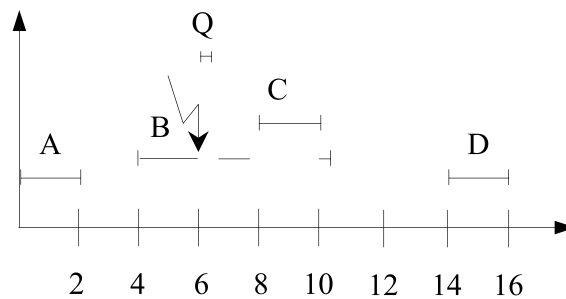


Figure 5-6. The recorded execution order scenario

Task *C* has no functional relation to the other tasks, but preempts *B* at certain rare occasions, e.g., when *B* is subject to interrupt interference, as depicted in *Figure 5-6*. However, task *C* and *B* both use a function that by a programming mistake is made non re-entrant. This function causes a failure in *B*, which subsequently sends an erroneous message to *D*, which in turn actuates an erroneous command to an external process, which fails. The interrupt *Q* hits *B*, and postpones *B*'s completion time. *Q* causes in this case *B* to be preempted by *C,* and *B* therefore becomes infected by the erroneous non-reentrant function. This rare scenario causes the failure. Now, assume that we have detected this failure and want to track down the error.

We have the following control transfer recording for time 0 -16:

1. Task *A* starts at time 0
2. Task *A* stops at time 2
3. Task *B* starts at time 4
4. Interrupt *Q* starts at time 6, and preempts task *B* at PC=*x*
5. Interrupt *Q* stops at time 6,5
6. Task *B* resumes at time 6,5, at PC=*x*
7. Task *C* starts at time 8, and preempts task *B* at PC=*y*
8. Task *C* stops at time 10
9. Task *B* resumes at time 10, at PC=*y*
10. Task *B* stops at time 10,3
11. Task *D* starts at time 14
12. Task *D* stops at time 16

Together with the following data recording:

1. Task *A* at time 1, read_ad() = 234
2. Task *B* at time 4, message from *D* = 78

During debugging all tasks are instrumented with calls to the off-line kernel at their termination, and the preempted tasks *B* and *C* are instrumented with calls to the off-line kernel at their recorded PC values. Task *A*'s access to the read_ad() function is short circuited and *A* is fed with the recorded value instead. Task *B* receives at its start a message from *D*, which is recorded before time 0.

The message transfers from *A* and *B* to *C* is performed by the off-line kernel in the same way as the on-line kernel.

The programmer/analyst can breakpoint, single step and inspect the control and data flow of the tasks as he or she see fit in pursuit of finding the error. Since the replay mechanism reproduces all significant events pertaining to the real-time behavior of the system the debugging will not cause any probe-effects.

As can be gathered from the example it is fairly straightforward to replay a recorded execution. The error can be tracked down because we can reproduce the exact interleavings of the tasks and interrupts repeatedly.

5.4 Discussion

In a survey on the testability of distributed real-time systems Schütz [96] has identified three issues related to deterministic replay in general, which we briefly comment below:

Issue 1: *One can only replay what has previously been observed, and no guarantees that every significant system behavior will be observed accurately can be provided.*

Since replay takes place at the machine code level the amount of information required is usually large. All inputs and intermediate events, e.g. messages, must be kept.

The amount and the necessary information required is of course a design issue, but it is not true that all inputs and intermediate messages must be recorded. The replay can as we have shown actually re-execute the tasks in the recorded event history. Only those inputs and messages which are not re-calculated, or re-sent, during the replay must be kept. This is specifically the case for RTS with periodic tasks, where we can make use of the knowledge of the schedule (precedence relations) and the duration before the schedule repeats it self (the LCM – the Least Common Multiple of the task period times.) In systems where deterministic replay has previously been employed, e.g., distributed systems [83] and concurrent programming (ADA) [104] this has not been the case. The restrictions, and predictability, inherent to scheduled RTS do therefore give us the great advantage of only recording the data that is not recalculated during replay.

Issue 2: *If a program has been modified (e.g., corrected) there are no guarantees that the old event history is still valid.*

If a program has been modified, the relative timing between racing tasks can change and thus the recorded history will not be valid. The timing differences can stem from a changed data flow, or that the actual execution time of the modified task has changed. In such cases it is likely that a new recording must be made. However, the probability of actually recording the sequence of events that pertain to the modification may be very low. This is an issue for regression testing [105][106] which we will discuss in chapter 6, section 6.4.4.

Issue 3: *The recording can only be replayed on the same hardware as the recording was made on.*

The event history can only be replayed on the target hardware. This is true to some extent, but should not be a problem if remote debugging is used. The replay could also be performed on the host computer if we have a hardware simulator, which could run the native instruction set of the target CPU. Another possibility would be to identify the actual high-level language statements where task switches or interrupts occurred, rather than trying to replay the exact machine code instructions, which of course are machine dependent. In the latter case we of course run into the problem of defining a unique state when differentiating between e.g., iterations in loops.

5.5 Related work

There are a few descriptions of deterministic replay mechanisms (related to real-time systems) in the literature:

- A deterministic replay method for concurrent Ada programs is presented by Tai et al. [104]. They log the synchronization sequence (rendezvous) for a concurrent program *P* with input *X*. The source code is then modified to facilitate replay; forcing certain rendezvous so that *P* follows the same synchronization sequence for *X*. This approach can reproduce the synchronization orderings for concurrent Ada programs, but not the duration between significant events, because the enforcement (changing the code) of specific synchronization sequences introduces gross temporal probe-effects. The replay scheme is thus not suited for real-time systems. Further, issues like unwanted side effects caused by preempting tasks are not considered. The granularity of the enforced rendezvous does not allow preemptions, or interrupts for that matter, to be replayed. It is unclear how the method can be extended to handle interrupts, and how it can be used in a distributed environment.
- Tsai et al. present a hardware monitoring and replay mechanism for real-time uniprocessors [113]. Their approach can replay significant events with respect to order, access to time, and asynchronous interrupts. The motivation for the hardware monitoring mechanism is to minimize the probe-effect, and thus make it suitable for real-time systems. Although it does minimizes the probe-effect, its overhead is not predictable, because their dual monitoring processing unit causes unpredictable interference on the target system by generating an interrupt for every event monitored [22]. They also record excessive details of the target processors execution, e.g., a 6 byte immediate AND instruction on a Motorola 68000 processor generates 265 bytes of recorded data. Their approach can reproduce asynchronous interrupts only if the target CPU has a dedicated hardware instruction counter. The used hardware approach is inherently target specific, and hard to adapt to other systems. The system is designed for single processor systems and has no support for distributed real-time systems.
- The software-based approach *HMON* [22] is designed for the HARTS distributed (real-time) system multiprocessor architecture [100]. A general-purpose processor is dedicated to monitoring on each multiprocessor. The monitor can observe the target processors via shared memory. The target systems software is instrumented with monitoring routines, by means of modifying system service calls, interrupt service routines, and making use of a feature in the pSOS real-time kernel for monitoring task-switches. Shared variable references can also be monitored, as can programmer defined application specific events. The recorded events can then be replayed off-line in a debugger. In contrast to the hardware supported instruction counter as used by Tsai et al., they make use of a software based instructions counter, as introduced by Mellor-Crummey et. al. [79]. In conjunction with the program counter, the software instruction counter can be used to reproduce interrupt interferences on the tasks. The paper does not elaborate on this issue. Using the recorded event history, off-line debugging can be performed while still having interrupts and task switches occurring at the same machine code instruction as during run-time. Interrupt occurrences are guaranteed off-line by inserting trap instructions at the recorded program counter value. The paper lacks information on how they achieve a consistent global state, i.e., how the recorded events on different nodes can consistently be related to each other. As they claim

that their approach is suitable for distributed real-time systems, the lack of a discussion concerning global time, clock synchronization, and the ordering of events, diminish an otherwise interesting approach. Their basic assumption about having a distributed system consisting of multiprocessor nodes makes their *software* approach less general. In fact, it makes it a hardware approach, because their target architecture is a shared memory multiprocessor, and their basic assumptions of non-interference are based on this shared memory and thus not applicable to distributed uniprocessors.

5.6 Summary

We have presented a method for deterministic debugging of distributed real-time systems that handles interleaving failure semantics. The method relies on an instrumented kernel to on-line record the timing and occurrences of major system events. The recording can then, using a special debug kernel, be replayed off-line to faithfully reproduce the functional and temporal behavior of the recorded execution, while allowing standard debugging using break points etc. to be applied.

The cost for this dramatically increased debugging capability is the overhead induced by the kernel instrumentation and by instrumentation of the application code. To eliminate probe-effects, these instrumentations should remain in the deployed system. We are however convinced that this is a justifiable penalty for many applications, especially safety-critical such.

6 TESTING DISTRIBUTED REAL-TIME SYSTEMS

In this chapter we will present a novel method for integration testing of multitasking real-time systems (RTS) and distributed real-time systems (DRTS). This method achieves deterministic testing of RTS by accounting for the effects of scheduling, jitter in RTS, and the inherent parallelism of DRTS applications.

A real-time system is by definition correct if it performs the correct *function* at the correct *time*. Using real-time scheduling theory we can provide guarantees that each task in the system will meet its timing requirements [4][75][117], given that the basic assumptions, e.g., task execution times and periodicity, are not violated at run-time. However, scheduling theory does not give any guarantees for the functional behavior of the system, i.e., that the computed values are correct. To assess the functional correctness other types of analysis are required. One possibility, although still immature, is to use formal methods to verify certain functional and temporal properties of a model of the system. The formally verified properties are then guaranteed to hold in the real system, as long as the model assumptions are not violated. When it comes to validating the underlying assumptions (e.g., execution times, synchronization order and the correspondence between specification and implemented code) we must use dynamic verification techniques which explore and investigate the run-time behavior of the real system. Testing [94] is the most commonly used such technique. Testing which is the state-of-practice can be used as a complement to, or a replacement for formal methods, in the functional verification.

Reproducible and deterministic testing of sequential programs can be achieved by controlling the sequence of inputs and the start conditions [78]. That is, given the same initial state and inputs, the sequential program will deterministically produce the same output on repeated executions, even in the presence of systematic faults [93]. Reproducibility is essential when performing regression testing or cyclic debugging, where the same test cases are run repeatedly with the intent to validate that either an error correction had the desired effect, or simply to make it possible to find the error when a failure has been observed [59]. However, trying to directly apply test techniques for sequential programs on distributed real-time systems is bound to lead to non-determinism and non-reproducibility, because control is only forced on the inputs, disregarding the significance of order and timing of the executing and communicating tasks. Any intrusive observation of a distributed real-time system will, in addition, incur a temporal probe-effect [31] that subsequently will affect the temporal and functional behavior of the system (see chapter 4).

The main contribution of this chapter is a method for achieving deterministic testing of distributed real-time systems. We will specifically address task sets with recurring release patterns, executing in a distributed system where the scheduling on each node is handled by a priority driven preemptive scheduler. This includes statically scheduled systems that are subject to preemption [117][95], as well as strictly periodic fixed priority systems [4][5][75]. The method aims at transforming the non-deterministic distributed real-time systems testing problem into a set of deterministic sequential program testing problems. This is achieved by deriving all the possible execution orderings of the distributed real-time system and regarding each of them as a sequential program. A formal definition of what actually constitutes an execution order scenario will be defined later in the chapter. The following small example presents the underlying intuition:

Consider *Figure 6-1a,* which depicts the execution of the of the tasks *A*, *B* and *C* during the Least Common Multiple (*LCM*) of their period times, dictated by a schedule generated by a static off-line scheduler. The tasks have fixed execution times, i.e. the worst and best-case execution times coincide ($WCET_i = BCET_i$, for $i \in \{A,B,C\}$). A task with later release time is assigned higher priority. These non-varying execution times have the effect of only yielding one possible execution scenario during the LCM, as depicted in *Figure 6-1a*. However, if for example, task *A* had a minimum execution time of 2 ($BCET_A=2; WCET_A=6$) we would get three possible execution scenarios, depicted in *figures 6-1a to 6-1c.* In addition to the execution order scenario in *Figure 6-1a*, there are now possibilities for *A* to complete before *C* is released (*Figure 6-1b*), and for *A* to complete before *B* is released (*Figure 6-1c*).

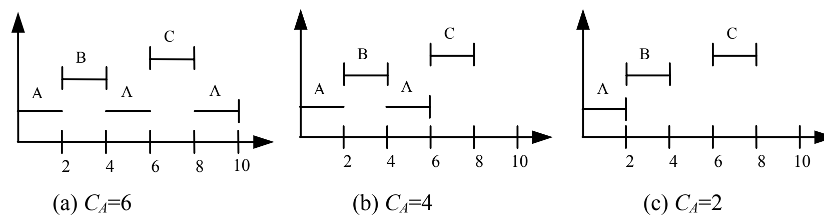


Figure 6-1 Three different execution order scenarios.

Given that these different scenarios yield different system behaviors for the same input, due to the order or timing of the produced outputs, or due to unwanted side effects via unspecified interfaces (caused by bugs), we would by using regular testing techniques for sequential programs get non-deterministic results. For example, assume that all tasks use a common resource *X* which they do operations on. Assume further that they receive input immediately when starting, and deliver output at their termination. We would then for the different scenarios depicted in figures 6-1a to 6-1c get different results:

- The scenario in *Figure 6-1a* would give *A*(*X*)*,* *B*(*X*) and *C*(*B*(*X*)).
- The scenario in *Figure 6-1b* would give *A*(*X*)*,* *B*(*X*) and *C*(*A*(*X*)).
- The scenario in *Figure 6-1c* would give *A*(*X*)*,* *B*(*A*(*X*)) and *C*(*B*(*A*(*X*))).

Making use of the information that the real-time system depends on the execution orderings of the involved tasks, we can achieve deterministic testing, since for the same input to the tasks and the same execution ordering, the system will deliver the same output on repeated executions.

We would thus be able to detect failures belonging to ordering failure semantics as defined in section 3.2. For ordering failure semantics it is sufficient to only observe orderings of identical task starts and completions, it is not necessary to observe exactly where in its execution a task is preempted, as is required for interleaving failure semantics (which we assumed for the deterministic replay in chapter 5).

In order to address the scenario dependent behavior we suggest a testing strategy (as illustrated in *Figure 6-2*) consisting of the following:

1. Identify all possible execution order scenarios for each scheduled node in the system during a single instance of the release pattern of tasks with duration *T*; typically equal to the LCM of the period times of the involved tasks (*Figure 6-2* illustrates eight scenarios.)
2. Test the system using any regular testing technique of choice, and monitor for each test case which execution order scenario is run during *[0,T],* i.e., which, when and in what order jobs are started, preempted and completed. By jobs we mean single instances of the recurring tasks during *T*.
3. Map test case and output onto the correct execution ordering, based on observation. This is handled by the module *execution ordering* in *Figure 6-2*.
4. Repeat 2-3 until the sought coverage is achieved.

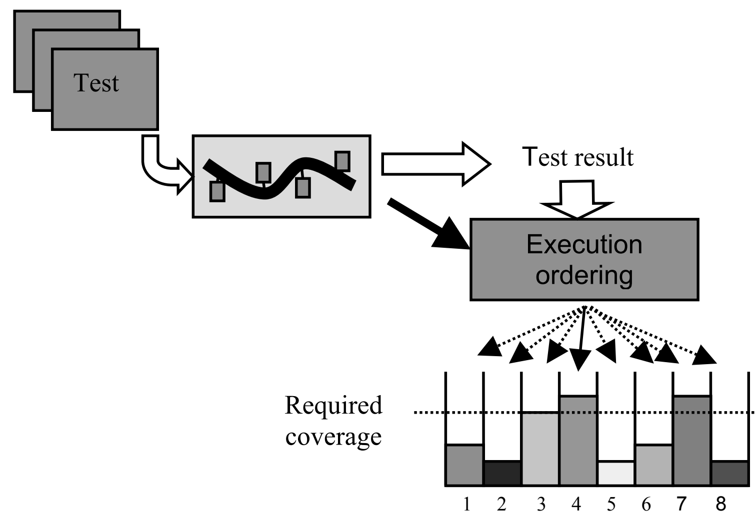


Figure 6-2. The testing procedure. Where each execution ordering must satisfy a certain required level of coverage.

Contribution

In this chapter we present a method for functional integration testing of multitasking real-time systems and distributed real-time systems: This method is to our knowledge the first testing method to fully encompass scheduling of distributed real-time systems and jitter. More specifically we will show how to:

- Identify the execution order scenarios for each node in a distributed real-time system
- Compose them into global execution order scenarios
- Make use of them when testing (the test strategy)
- Reproduce the scenarios.

6.1 The system model

We refine the original system model from chapter 3 to encompass a distributed system consisting of a set of nodes, which communicate via a temporally predictable broadcast network, i.e. upper and lower bounds on communication latencies are known or can be calculated [50][110]. Each node is a self sufficient computing element with CPU, memory, network access, a local clock and I/O units for sampling and actuation of the external system. We further assume the existence of a global synchronized time base [51][27] with a known precision δ, meaning that no two nodes in the system have local clocks differing by more than δ.

We further refine the original system model by assuming that the software that runs on the distributed system consists of a set of concurrent tasks, communicating by message passing. Functionally related and cooperating tasks, e.g., sample-calculate-actuate loops in control systems, are defined as transactions. The relationship between the cooperating tasks with respect to precedence (execution order), interactions (data-flow), and a period time typically define each transaction. The tasks are distributed over the nodes, typically with transactions that span several nodes, and with more than one task on each node. All synchronization is resolved before run-time and therefore no action is needed to enforce synchronization in the actual program code. Different release times and priorities guarantee mutual exclusion and precedence. The distributed system is globally scheduled, which results in a set of specific schedules for each node. At run-time we need only synchronize the local clocks to fulfill the global schedule [50].

Task model

We assume a fairly general task model that includes both preemptive scheduling of statically generated schedules [117] and fixed priority scheduling of strictly periodic tasks [4][75]:

- 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 Least Common Multiple (*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 for all participating nodes.
- Each job $j \in J$ has a release time r_j, worst case execution time ($WCET_j$), best case execution time ($BCET_j$), a deadline D_j and a priority p_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.
- The system is preemptive and jobs may have identical release-times.

Related to the task model we assume that the tasks may have functional and temporal side effects due to preemption, message passing and shared memory. Furthermore, we assume that data is sent at the termination of the sending task (not during its execution), and that received data is available when tasks start (and is made private in an atomic first operation of the task) [108][26][54].

Fault hypothesis

Note that, although synchronization is resolved by the off-line selection of release times and priorities, we cannot dismiss unwanted synchronization side effects. The schedule design can be erroneous, or the assumptions about the execution times might not be accurate due to poor execution time estimates, or simply due to design and coding errors.

Inter-task communication is restricted to the beginning and end of task execution, and therefore we can regard the interval of execution for tasks as atomic. With respect to access to shared resources, such as shared memory and I/O interfaces, the atomicity assumption is only valid if synchronization and mutual exclusion can be guaranteed.

The fault hypothesis is thus:

> Errors can only occur due to erroneous outputs and inputs to jobs, and/or due to synchronization errors, i.e., jobs can only interfere via specified interactions. This hypothesis corresponds to ordering failure semantics as defined in section 3.2.

One way to guarantee the fault hypothesis in a shared memory system is to make use of hardware memory protection schemes, or during design eliminate or minimize shared resources using wait-free and lock-free communication [108][52][13].

6.2 Execution order analysis

In this section we present a method for identifying all the possible orders of execution for sets of jobs conforming to the task model introduced in section 6.1. We will also show how the model and analysis can be extended to accommodate for interrupt interference and multiple nodes in a distributed system by considering clock-synchronization effects, parallel executions, and varying communication latencies.

6.2.1 Execution Orderings

In identifying the execution orderings of a job set we will only consider the following major events of job executions:

- The start of execution of a job, i.e., when the first instruction of a job is executed. We will use *S(J)* to denote the set of start points for the jobs in a job set *J*; $S(J) \subseteq J \times [0, T^{MAX}] \times J \cup \{_\}$, that is *S(J)* is the set of triples (j_1, *time*, j_2), where j_2 is the (lower priority) job that is preempted by the start of j_1 at *time*, or possibly "_" if no such job exists.
- The end of execution of a job, i.e., when the last instruction of a job is executed. We will use *E(J)* to denote the set of end points (termination points) for jobs in a job set *J*; $E(J) \subseteq J \times [0, T^{MAX}] \times J \cup \{_\}$, that is *E(J)* is a set of triples (j_1, *time*, j_2), where j_2 is the (lower priority) job that resumes its execution at the termination of j_1, or possibly "_" if no such job exists.

We will now define an execution to be a sequence of job starts and job terminations, using the additional notation that

- *ev* denotes an event, and *Ev* a set of events.
- *ev.t* denotes the time of the event *ev*,
- *Ev\I* denotes the set of events in *Ev* that occur in the time interval *I*,
- *Prec(Ev, t)* is the event in *Ev* that occurred most recently at time *t* (including an event that occurs at *t*).
- *Nxt(Ev, t)* denotes the next event in *Ev* after time *t*.
- *First(Ev)* and *Last(Ev)* denote the first and last event in *Ev*, respectively.

Definition 6-1. An *Execution* of a job set *J* is a set of events $X \subseteq S(J) \cup E(J)$, such that

1) For each $j \in J$, there is exactly one start and termination event in *X*, denoted $s(j,X)$ and $e(j,X)$ respectively, and $s(j,X)$ precedes $e(j,X)$, i.e. $s(j,X).t \le e(j,X).t$, where $s(j,X) \in S(J)$ and $e(j,X) \in E(J)$.
2) For each $(j_1, t, j_2) \in S(J)$, $p_{j1} > p_{j2}$, i.e., jobs are only preempted by higher priority jobs.
3) For each $j \in J$, $s(j,X).t \ge r_j$, i.e., jobs may only start to execute after being released.
4) After its release, the start of a job may only be delayed by intervals of executions of higher priority jobs, i.e., using the convention that $X \backslash [j.t, j.t) = \emptyset$.

   For each job $j \in J$ each event $ev \in X \backslash [Prec(X,r_j).t, s(j,X).t)$ is either

   - A start of the execution of a higher priority job, i.e. $ev = s(j', X)$ and $p_{j'} > p_j$
   - A job termination, at which a higher priority job resumes its execution, i.e., $ev = (j', t, j'')$, where $p_{j''} > p_j$
5) The sum of execution intervals of a job $j \in J$ is in the range $[BCET(j), WCET(j)]$, i.e.,

   $$BCET(j) \le \sum_{ev \in \{s(j,X)\} \cup \{(j',t,j) | (j',t,j) \in E(J)\}} Nxt(X, ev.t).t - ev.t \le WCET(j)$$

   That is, we are summing up the intervals in which *j* starts or resumes its execution.

We will use $EX_t(J)$ to denote the set of executions of the job set *J*. Intuitively, $EX_t(J)$ denotes the set of possible executions of the job set *J* within $[0, T^{MAX}]$. Assuming a dense time domain $EX_t(J)$ is only finite if $BCET(j) = WCET(j)$ for all $j \in J$. However, if we disregard the exact timing of events and only consider the ordering of events we obtain a finite set of execution orderings for any finite job set *J*.

Using $ev\{x/t\}$ to denote an event ev with the time element t replaced by the undefined element "x", we can formally define the set of execution orderings $EX_o(J)$ as follows:

Definition 6-2. The set of *Execution orderings* $EX_o(J)$ of a job set J is the set of sequences of events such that $ev_0\{x/t\}$, $ev_1\{x/t\}$, ..., $ev_k\{x/t\} \in EX_o(J)$ **iff** there exists an $X \in EX_t(J)$ such that

- $First(X) = ev_0$
- $Last(X) = ev_k$
- For any $j \in [0..(k\text{-}1)]$: $Nxt(X, ev_j.t) = ev_{j+1}$

Intuitively, $EX_o(J)$ is constructed by extracting one representative of each set of equivalent execution orderings in $EX_t(J)$, i.e., using a quotient construction $EX_o(J) = EX_t(J) \backslash \sim$, where $\sim$ is the equivalence induced by considering executions with identical event orderings to be equivalent. This corresponds to our fault hypothesis, with the overhead of keeping track of preemptions and resumptions, although not exactly where in the program code they occur. This overhead means that we can capture more than what ordering failure semantics do, although not as much as needed for interleaving failure semantics, since we do not keep track of exactly where in the program code the preemptions occur. We could thus reduce the number of execution orderings further if we define $EX_o(J) = EX_t(J) \backslash \approx$, where $\approx$ is the equivalence induced by considering executions with identical job start and job stop orderings to be equivalent. In the process of deriving all the possible execution orderings we need however to keep track of all preemptions, i.e., $EX_t(J) \backslash \sim$, but after having derived this set we can reduce it to $EX_t(J) \backslash \approx$. Even further reductions could be of interest, for instance to only consider orderings among tasks that are functionally related, e.g., by sharing data.

In the remainder we will use the terms *execution scenario* and *execution ordering* interchangeably.

6.2.2 Calculating $EX_o(J)$

This section outlines a method to calculate the set of execution orderings $EX_o(J)$ for a set of jobs J, complying with definition 6-2. We will later (in section 6.3) present an algorithm that performs this calculation. In essence, our approach is to make a reachability analysis by simulating the behavior during one $[0, T^{MAX}]$ period for the job set J.

6.2.2.1 The Execution Order Graph (EOG)

The algorithm we are going to present generates, for a given schedule, an Execution Order Graph (EOG), which is a finite tree for which the set of possible paths from the root contains all possible execution scenarios.

But before delving into the algorithm we describe the elements of an EOG. Formally, an EOG is a pair $\langle N, A \rangle$, where

- N is a set of nodes, each node being labeled with a job and a continuous time interval, i.e., for a job set J: $N \subseteq J \cup \{\text{"_"}\} \times I(T^{MAX})$, where {"_"} is used to denote a node where no job is executing 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: $A \subseteq N \times I(T^{MAX}) \times N$.

Intuitively, an *edge*, corresponds to the transition (task-switch) from one job to another. The edge is annotated with a continuous interval of when the transition can take place, as illustrated in *Figures 6-3* and *6-4*.

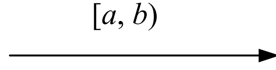


Figure 6-3. A Transition.

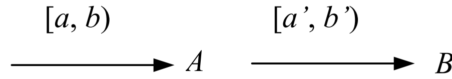


Figure 6-4 Two transitions, one to job A and one from job A to job B.

The interval of possible start times $[a', b')$ for job B, in *Figure 6-4*, is defined by:

$$a' = MAX(a, r_A) + BCET_A \qquad (6\text{-}1)$$
$$b' = MAX(b, r_A) + WCET_A$$

The MAX functions are necessary because the calculated start times a and b can be earlier than the scheduled release of the job A.

A *node* represents a job annotated with a continuous interval of its possible execution, as depicted in *Figure 6-5*.

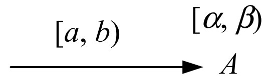


Figure 6-5. A job annotated with its possible execution and start time.

We define the interval of execution, $[\alpha, \beta)$ by:

$$\alpha = MAX(a, r_A) \qquad (6\text{-}2)$$
$$\beta = MAX(b, r_A) + WCET_A$$

That is, the interval, $[\alpha, \beta)$, specifies the interval in which job A can be preempted.

From each node in the execution ordering graph there can be one or more transitions, representing one of four different situations:

1) The job is the last job scheduled in this branch of the tree. In this case the transition is labeled with the interval of finishing times for the node, and has the empty job "_" as destination node, as exemplified in *Figure 6-6*.
2) The job has a *WCET* such that it definitely completes before the release of any higher priority job. In this case we have two situations:
    a) *Normal situation*. One single outgoing transition labeled with the interval of finishing times for the job, [*a'*, *b'*). Exemplified by (1) in *Figure 6-6*.
    b) *Special case*. If a higher priority job is immediately succeeding at [*b',b'*] while *b' > a'*, and there are lower priority jobs ready, or made ready during $[\alpha,\beta)$ then we have two possible transitions: One branch labeled with the interval of finishing times [*a'*,*b*') representing the immediate succession of a lower priority job, and one labeled [*b*', *b'*) representing the completion immediately before the release of the higher priority job. Exemplified by (2) in *Figure 6-6*.
3) The job has a *BCET* such that it definitely is preempted by another job. In this case there is a single outgoing transition labeled with the preemption time *t*, expressed by the interval [*t*, *t*], as exemplified by (3) in *Figure 6-6*.
4) The job has a *BCET* and *WCET* such that it may either complete or be preempted before any preempting job is released. In this case there can be two or three possible outgoing edges depending on if there are any lower priority jobs ready. One branch representing the preemption, labeled with the preemption time [*t*, *t*], and depending on if there are any lower priority jobs ready for execution we have two more transition situations:
    a) *No jobs ready.* Then there is one branch labeled [*a'*, *t*) representing the possible completion prior to the release of the higher priority job. Exemplified by (4) in *Figure 6-6*.
    b) *Lower priority jobs ready.* If $\beta > \alpha$ then there is one branch labeled [*a'*, *t*) representing the immediate succession of a lower priority job, and one labeled [*t, t*) representing the completion immediately before the release of the preempting job. Exemplified by (5) in *Figure 6-6*.

Example 6-1

Figure 6-6 gives an example of an EOG, using the above notation and the attributes in *Table 6-1*. In *Figure 6-6*, all paths from the root node to the "_" nodes correspond to the possible execution order scenarios during one instance of the recurring release pattern.

Table 6-1 A job set for a schedule with a LCM of 400 ms.

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	20	9

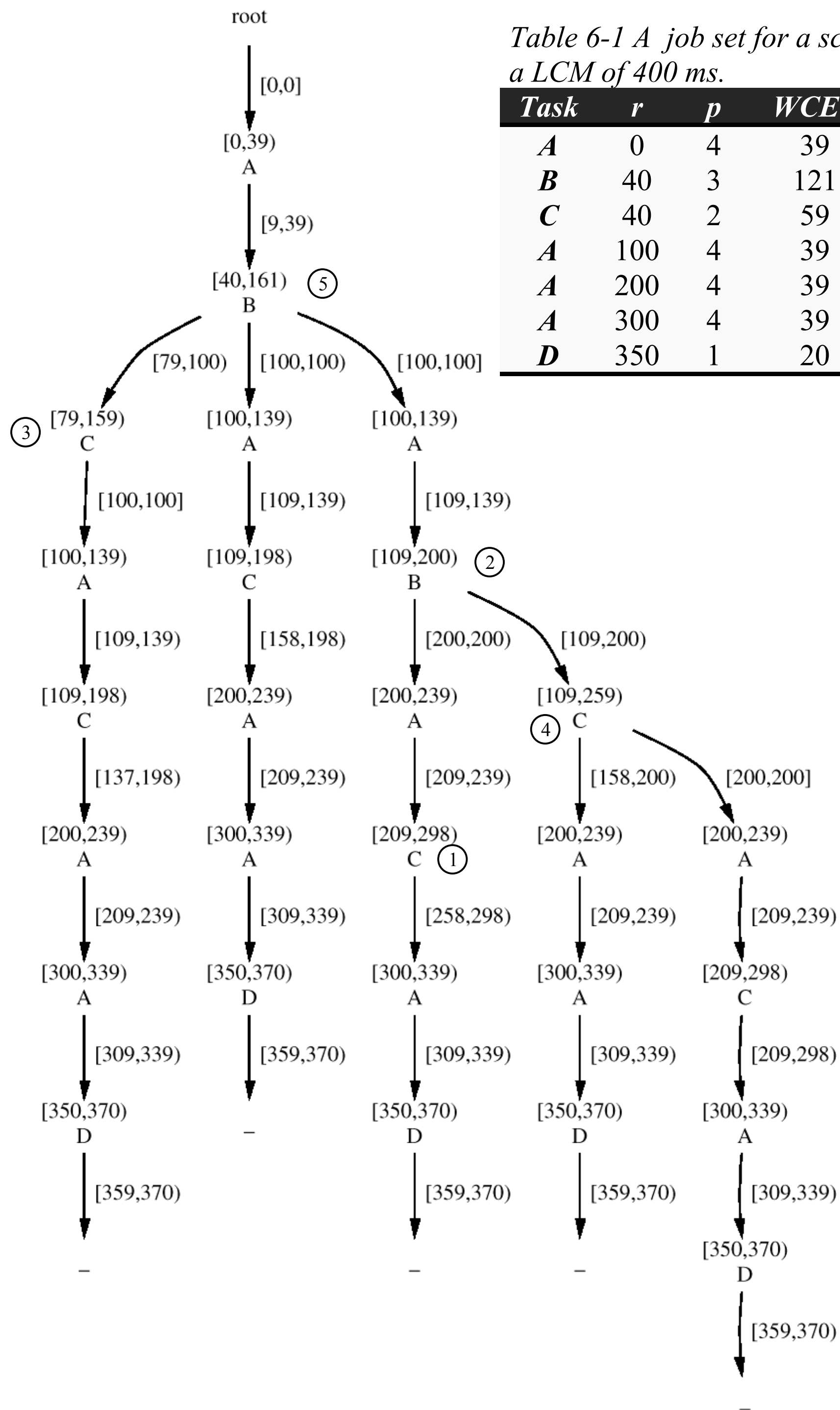


Figure 6-6. The resulting execution order graph for the job set in Table 6-1.

6.3 The EOG algorithm

Here we will define the algorithm for generating the EOG (*Figure 6-7*). Essentially, the algorithm simulates the behavior of a strictly periodic fixed priority preemptive real-time kernel, complying with the previously defined task model and EOG primitives. In defining the algorithm we use the following auxiliary functions and data structures:

1) *rdy* – the set of jobs ready to execute.
2) Next_release(I) – returns the earliest release time of a job $j \in J$ within the interval I. If no such job exists then ∞ is returned . Also, we will use $I.l$ and $I.r$ to denote the extremes of I.
3) P(t) – Returns the highest priority of the jobs that are released at time t. Returns -1 if $t = \infty$.
4) Make_ready(t, *rdy*) – adds all jobs that are released at time t to *rdy*. Returns $\varnothing$ if $t=\infty$, else the set.
5) X(*rdy*) extracts the job with highest priority in *rdy*.
6) Arc(n, I, n') creates an edge from node n to node n' and labels it with the time interval I.
7) Make_node(j, XI) creates a node and labels it with the execution interval XI and the id of job j.

The execution order graph for a set of jobs J is generated by a call *Eog*(*ROOT*, {}, [0, 0], [0, T^{MAX}]), i.e., with a root node, an empty ready set, and the initial release interval [0,0], plus the considered interval [0, T^{MAX}].

```
// n- previous node, rdy- set of ready jobs, RI – release interval, SI – the considered interval.
Eog(n, rdy, RI, SI)
{
    //When is the next job(s) released?
    t = Next_release(SI)
    if (rdy = ∅)
        rdy = Make_ready(t, rdy)
        if( rdy ≠ ∅)
            Eog (n, rdy, RI, (t,SI.r] )
        else  Arc(n, RI, _ )
    else
        //Extract the highest priority job in rdy.
        T      = X(rdy)
        [α,β)  = [max(r_T, RI.l), max(r_T, RI.l)+WCET_T)
        a'     = α + BCET_T
        b'     = β
        n'     = Make_node(T, [α,β) ) Arc(n, RI, n')

        // Add all lower priority jobs that are released before
        T's termination, or before a high priority job is preempting T.
        while((t < β) ∧ (P(t) < p_T))
            rdy   =  Make_ready(t, rdy)
            t     =  Next_release((t, SI.r])

        // Does the next scheduled job preempt T?
        if((p_T < P(t)) ∧ (t <  β))
            // Can T complete prior to the release of the next job at t?
            if(t > a')
                Eog ( n', rdy, [a',t), [t,SI.r] )
                if(rdy ≠ ∅)
                    Eog(n',Make_ready(t,rdy),[t,t),(t,SI.r])
                else if(t  = a')
                        Eog(n', Make_ready(t, rdy), [t,t), (t,SI.r])

            //Add all jobs that are released at time t.
            rdy = Make_ready(t, rdy)

            //Best and worst case execution prior to preemption?
            BCET_T = max(BCET_T - (t-(max(r_T, RI.l)),0)
            WCET_T = max(WCET_T - (t-(max(r_T, RI.r)),0)
            Eog( n', rdy + {T}, [t,t], (t,SI.r])

        // No preemption
        else  if(t = ∞ ) //Have we come to the end of the simulation?
                  Eog(n', rdy, [a',b'),[∞,∞]) //Yes, no more jobs to execute

              else // More jobs to execute

                  //Is there a possibility for a high priority job to succeed immediately,
                  while low priority jobs are ready?
                  if(rdy ≠ ∅  ∧  t = β) //Yes, make one branch for this transition
                      Eog(n', Make_ready(t, rdy),[t,t),(t,SI.r])
                      if(a' ≠ b') //And one branch for the low priority job
                          else Eog(n', rdy, [a',b'),[t, SI.r))
                  // The regular succession of the next job (low or high priority)
                  else Eog(n', rdy, [a',b'),[t, SI.r))
}//End
```

Figure 6-7. The Exection Order Graph algorithm.

6.3.1 GEX_o – the Global EOG

In a distributed system with multiple processing units (and schedules) we generate one EOG for each processing unit (node). From these, a global EOG describing all globally possible execution orderings can be constructed. In the case of perfectly synchronized clocks this essentially amounts to deriving the set of all possible combinations of the scenarios in the individual EOGs. In other cases, we first need to introduce the timing uncertainties caused by the non-perfectly synchronized clocks in the individual EOGs.

Since the progress of local time on each node keeps on speeding up, and slowing down due to the global clock synchronization, the inter-arrival time of the clock ticks vary. The effect is such that for an external observer with a perfect clock the start times and completion times change, as the global clock synchronization algorithm adjusts the speed of the local clocks to keep them within specified limits. Locally in the node, where all events are related to the local clock tick, the effect will manifest itself as a differential in the actual *BCET* and *WCET* from the estimated values. As the inter-arrival time between ticks increases, the *BCET* will decrease because of the possibility to execute more machine code instructions due to the longer duration between two consecutive ticks (see *Figure 6-8*). Likewise the *WCET* increases due to the decrease in the inter-arrival time of ticks.

When scheduling the distributed real-time system it is essential to accommodate for the clock synchronization effects by time-wise separating the completion times of preceding, or mutually exclusive, tasks from the release time of a succeeding task with a factor δ, corresponding to the precision of the global time base.

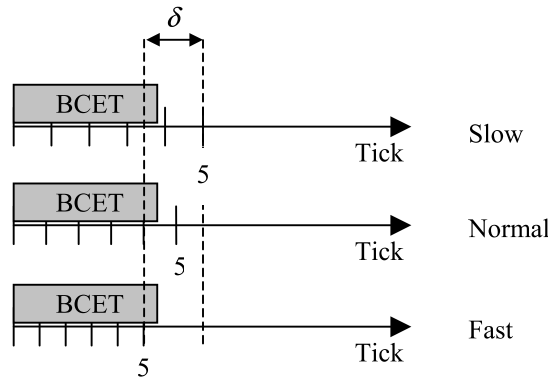


Figure 6-8. The effects of the global clock synchronization on the perceived BCET according to the local tick.

When deriving the execution orderings we need thus change the estimations of the *BCET* and *WCET*:

$$BCET_{new} = \mathrm{MAX}(BCET - \delta/2 * K(BCET), 0) \qquad (6\text{-}3)$$

$$WCET_{new} = WCET + \delta/2 * K(WCET) \qquad (6\text{-}4)$$

Where the function *K*() is derived from the clock synchronization control loop implementation, and where the argument states how much of the clock synchronization interval is under consideration. The *K()* function substantially increases the precision, since it would be overly pessimistic to add, or subtract,

$\delta/2$ when it is possible that the *BCET* and *WCET* will only be fractions of the synchronization interval.

As an illustration, compare the non-adjusted task set in *Table 6-2* and its EOG in *Figure 6-9* with the corresponding adjusted task set and EOG in *Table 6-3*, and *Figure 6-10*.

Table 6-2 The non-adjusted job set for a node in a DRTS, with a clock synchronization precision of δ = 4.

Schedule $\delta = 4$				
Task	*r*	*p*	*BCET*	*WCET*
A	0	1	100	300
B	300	2	100	300

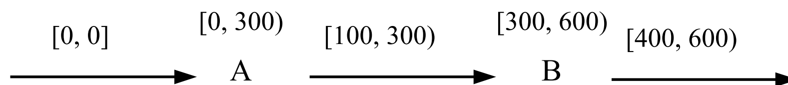


Figure 6-9. The execution ordering for the job set in Table 6-2 without compensating for the precision of the clock synchronization.

Table 6-3 The adjusted job set for a node in a DRTS, with a clock synchronization precision of δ = 4, and K(x) = x.

Schedule $\delta = 4$				
Task	*r*	*p*	*BCET*	*WCET*
A	0	1	98	302
B	300	2	98	302

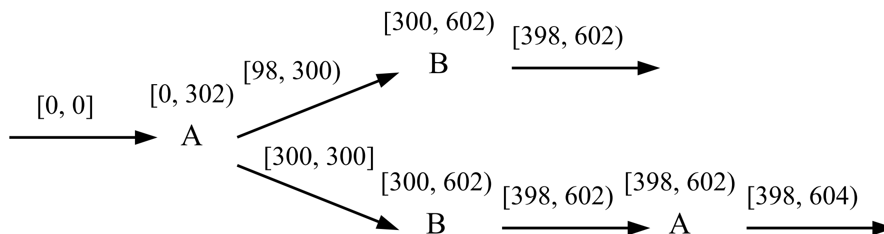


Figure 6-10. The execution ordering for the job set in Table 6-3 where the precision of the clock synchronization has been accounted for.

6.3.1.1 Calculating GEX_o

In testing the behavior of a distributed system we must consider the joint execution of a set of local schedules; one on each node. Typically these schedules are of equal length as a result of the global scheduling. However, if they are not, we have to extend each of them by multiples until they equal the global LCM.

The next step is to generate EOGs for each node where the effects of global clock synchronization have been accommodated for, as presented above. From these local EOGs we can then calculate the set of global execution order scenarios, GEX_o, which essentially are defined by the product of the individual

sets of execution orderings on each node. That is, a set of tuples, $GEX_o = EX_o(node_1) \times \ldots \times EX_o(node_n)$, consisting of all possible combinations of the local execution orderings. This set is not complete, but before delving into that problem, we will give an illustrating example (6-2).

Example 6-2

Assume that we have two nodes, *N1* and *N2*, with corresponding schedules (job sets) *J1* and *J2*, and derived execution orderings of the schedules $EX_o(J1)$ ={*q1, q2, q3*} and $EX_o(J2)$ = {*p1, p2*}. Also assume that task *C* is a calculating task in a control application, using sampled values provided by tasks *A*, *B*, and *E*. Assume further that task *G* is another calculating task using values received from tasks *A,* and *E***.** Tasks *C* and *G* receives all inputs immediately at their respective task starts, and tasks *A*, *B*, *E* all deliver outputs immediately prior to their termination. The resulting set GEX_o, is then equal to {(*q1,p1*)*,* (*q1,p2*)*,* (*q2,p1*)*,* (*q2,p2*), (*q3,p1*)*,* (*q3,p2*)}.

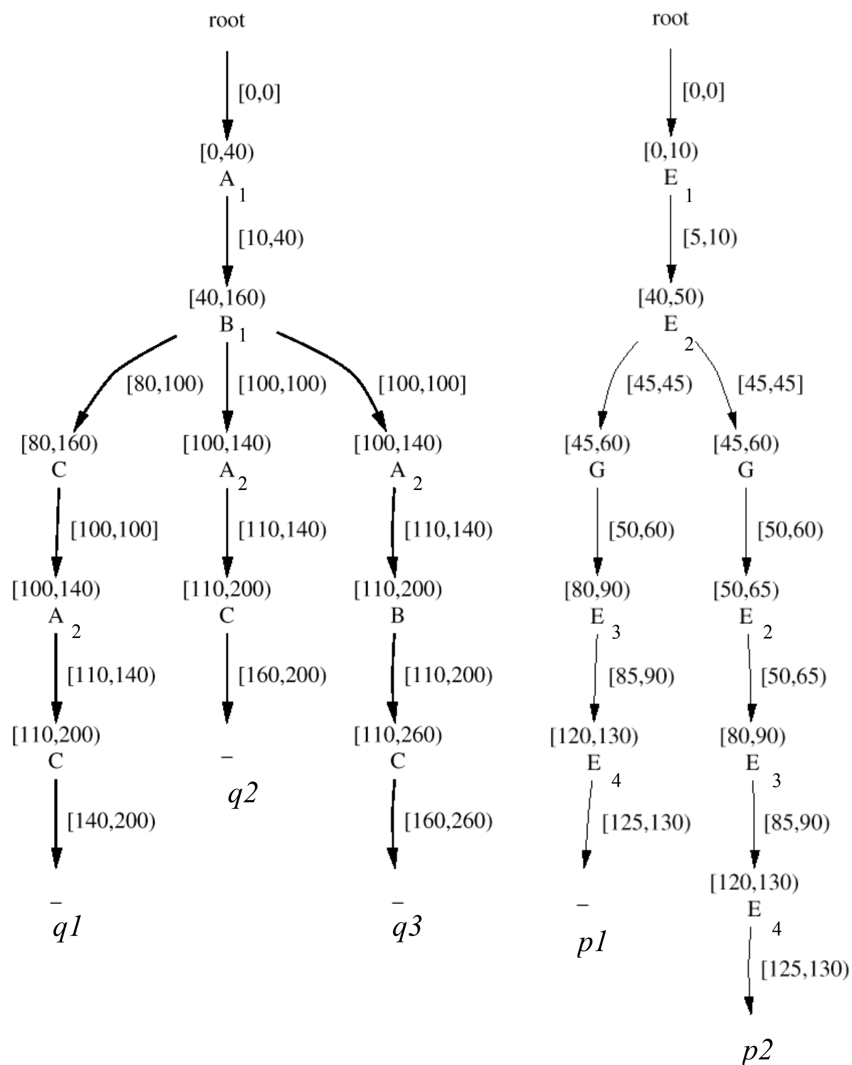


Figure 6-11. The exection orderings for two nodes. The possible global execution orderings are the possible combinations of the individual orderings. for the precision of the clock synchronization.

The set GEX_o = {(*q1,p1*), (*q1,p2*), (*q2,p1*), (*q2,p2*), (*q3,p1*), (*q3,p2*)}, is however not complete. Analyzing the data dependencies between task *C* on node *N1* and task *E* on node *N2*, we see that task *C* can receive data from different instances of task *E*. The same goes for task *G*, on node *N2*, with respect to task *A*, on node *N1*.

For example, during the global scenario (*q1, p1*), task *C* has a varying start time of [80,100) and within that interval task instance E_3, has a completion interval of [85,90]. This means, if we assume communication latencies in the interval [3,6), that task *C* can receive data from task instance E_2 or E_3. In order to differentiate between these scenarios we should therefore substitute the global ordering (*q1,p1*) with the set (*q1*:$C(E_2)$, *p1*) and (*q1*:$C(E_3)$, *p1*). Where "(*q1*:$C(E_2)$, *p1*)" means that during scenario (*q1,p1*), task *C* receives data from task *E*, and that the originating instance of task *E* is of significance.

For task *G*, assuming the same communication latencies, this means that it can receive data from task instance A_0 or A_1, yielding a new set of scenarios, e.g., scenario (*q1,p1*) is substituted with (*q1, p1*:$G(A_0)$) and (*q1, p1*:$G(A_1)$). These transformations necessitates that we, in addition to recording the execution orderings during runtime, need to tag all data such that we can differentiate between different instances, corresponding to different originators.

From the viewpoint of node *N1* the transformed set would look like:

{(*q1*:$C(E_2)$, *p1*), (*q1*:$C(E_3)$, *p1*), (*q1*:$C(E_2)$, *p2*), (*q1*:$C(E_3)$, *p2*),
(*q2*:$C(E_3)$, *p1*), (*q2*:$C(E_4)$, *p1*), (*q2*:$C(E_3)$, *p2*), (*q2*:$C(E_4)$, *p2*),
(*q3*:$C(E_3)$, *p1*), (*q3*:$C(E_4)$, *p1*), (*q3*:$C(E_3)$, *p2*), (*q3*:$C(E_4)$, *p2*)}.

From the viewpoint of node *N2* the transformed set would look like:

{(*q1*, *p1*:$G(A_0)$), (*q1*, *p1*:$G(A_1)$), (*q1*, *p2*:$G(A_0)$), (*q1*, *p2*:$G(A_1)$),
(*q2*, *p1*:$G(A_0)$), (*q2*, *p1*:$G(A_1)$), (*q2*, *p2*:$G(A_0)$), (*q2*, *p2*:$G(A_1)$),
(*q3*, *p1*:$G(A_0)$), (*q3*, *p1*:$G(A_1)$), (*q3*, *p2*:$G(A_0)$), (*q3*, *p2*:$G(A_1)$)}

That is, the new transformed execution orderings are:

- $EX_o(J1)$={*q1*:$C(E_2)$, *q1*:$C(E_3)$, *q2*:$C(E_3)$, *q2*:$C(E_4)$, *q3*:$C(E_3)$, *q3*:$C(E_4)$}
- $EX_o(J2)$={ *p1*:$G(A_0)$, *p1*:$G(A_1)$, *p2*:$G(A_0)$, *p2*:$G(A_1)$}

The new transformed set $GEX_o = EX_o(J1) \times EX_o(J2)$ becomes:

{(*q1*:$C(E_2)$, *p1*:$G(A_0)$), (*q1*:$C(E_3)$, *p1*:$G(A_0)$), (*q1*:$C(E_2)$, *p2*:$G(A_0)$), (*q1*:$C(E_3)$, *p2*:$G(A_0)$),
(*q2*:$C(E_3)$, *p1*:$G(A_0)$), (*q2*:$C(E_4)$, *p1*:$G(A_0)$), (*q2*:$C(E_3)$, *p2*:$G(A_0)$), (*q2*:$C(E_4)$, *p2*:$G(A_0)$),
(*q3*:$C(E_3)$, *p1*:$G(A_0)$), (*q3*:$C(E_4)$, *p1*:$G(A_0)$), (*q3*:$C(E_3)$, *p2*:$G(A_0)$), (*q3*:$C(E_4)$, *p2*:$G(A_0)$),
(*q1*:$C(E_2)$, *p1*:$G(A_1)$), (*q1*:$C(E_3)$, *p1*:$G(A_1)$), (*q1*:$C(E_2)$, *p2*:$G(A_1)$), (*q1*:$C(E_3)$, *p2*:$G(A_1)$),
(*q2*:$C(E_3)$, *p1*:$G(A_1)$), (*q2*:$C(E_4)$, *p1*:$G(A_1)$), (*q2*:$C(E_3)$, *p2*:$G(A_1)$), (*q2*:$C(E_4)$, *p2*:$G(A_1)$),
(*q3*:$C(E_3)$, *p1*:$G(A_1)$), (*q3*:$C(E_4)$, *p1*:$G(A_1)$), (*q3*:$C(E_3)$, *p2*:$G(A_1)$), (*q3*:$C(E_4)$, *p2*:$G(A_1)$}.

6.3.1.2 GEX_o data dependency transformation

We will now more formally define these transformations. Assume the following auxiliary functions:

- *Start*(j), returns the interval of start times for job j, (task instance)
- *Finish*(j), returns the interval of finishing times for job j.
- *ComLatency*(i,j), returns the communication latency interval for data sent by job i to job j.
- *Precede*(j), returns the immediately preceding instance of the same task as job j. Note that this may be a job from the previous LCM.

Data dependency transformation definition

We transform the local execution orderings, $EX_o(node)$, with respect to the set of global execution orderings, $GEX_o = EX_o(node_1) \times \ldots \times EX_o(node_n)$:

1. For each tupel $l \in GEX_o$.
   2. For each ordering $x \in EX_o(n)$, where x is an element in tupel l, and where $K(x)$ is the set of jobs in x consuming data originating from the set of jobs, P, belonging to the other execution ordering elements of l.
      3. For each job $j \in K(x)$.
         4. For each subset $Pt \subseteq P$ representing different instances of the same task producing data for job j.
            5. Let Q be the set of jobs in Pt with finishing times + communication delays, in the start time interval of job j, i.e., $Q=\{u \mid start(j) \cap (Finish(u) + ComLatency(u,j)) \neq \varnothing\}$ The set can be ordered by earliest finishing time. We use q to denote the earliest job in Q.
            6. If $Q \neq \varnothing$ substitute x. That is,

               $EX_o(n) = EX_o(n) \backslash x + x{:}\, j(Precede(q)) + (\forall u \in Q \mid x{:}j(u))$.
   7. Substitute all $x{:}j(y) \in EX_o(n)$ where $j \in K(x)$ and $y \in P$ such that we for the scenario x, construct a substitute of all possible combinations of $\{x{:}j(y_m), x{:}j'(y'_{m'}),\ldots, x{:}j''(y''_{m''})\} \subseteq EX_o(n)$ for consuming jobs, $\{j,\ldots,j''\} \subseteq K(x)$ not belonging to the same task instance, i.e., combinations $x{:}j(y_m){:}j'(y'_{m'}){:}\ldots{:}j''(y''_{m''})$ in order of start time for jobs consuming data produced by jobs $\{y_m,\ldots,y''_{m}''\} \subseteq P$.
8. New transformed $GEX_o = EX_o(node_1) \times \ldots \times EX_o(node_n)$

Example 6-3

Assume the same system as in *Example 6-2*, where the initial $GEX_o = \{ (q1,p1) , (q1,p2), (q2,p1), (q2,p2) , (q3,p1), (q3,p2) \}$.

The first tupel would be $l = (q1,p1)$, the consuming task set in scenario $q1$ would be $K(q1) = \{C\}$, the producing set of jobs for $K(q1)$ would be $Pt=P=\{E_{old},E_1, E_2, E_3, E_4\}$, and the set of jobs Q that has a chance of producing data for the sole job in $K(q1)$, C, is $Q=\{ E_2, E_3\}$. We thus substitute $q1$with $q1{:}C(E_2)$ and $q1{:}C(E_3)$.

Likewise we can transform all sets:

- $EX_o(J1)=\{q1{:}C(E_2), q1{:}C(E_3), q2{:}C(E_3), q2{:}C(E_4), q3{:}C(E_3), q3{:}C(E_4)\}$
- $EX_o(J2)=\{ p1{:}G(A_0), p1{:}G(A_1), p2{:}G(A_0), p2{:}G(A_1)\}$

The transformed global execution ordering set $GEX_o = EX_o(J1) \times EX_o(J2) =$

$\{(q1{:}C(E_2), p1{:}G(\mathrm{A}_0)), (q1{:}C(E_3), p1{:}G(\mathrm{A}_0)), (q1{:}C(E_2), p2{:}G(\mathrm{A}_0)), (q1{:}C(E_3), p2{:}G(\mathrm{A}_0)),$
$(q2{:}C(E_3), p1{:}G(\mathrm{A}_0)), (q2{:}C(E_4), p1{:}G(\mathrm{A}_0)), (q2{:}C(E_3), p2{:}G(\mathrm{A}_0)), (q2{:}C(E_4), p2{:}G(\mathrm{A}_0)),$
$(q3{:}C(E_3), p1{:}G(\mathrm{A}_0)), (q3{:}C(E_4), p1{:}G(\mathrm{A}_0)), (q3{:}C(E_3), p2{:}G(\mathrm{A}_0)), (q3{:}C(E_4), p2{:}G(\mathrm{A}_0)),$
$(q1{:}C(E_2), p1{:}G(\mathrm{A}_1)), (q1{:}C(E_3), p1{:}G(\mathrm{A}_1)), (q1{:}C(E_2), p2{:}G(\mathrm{A}_1)), (q1{:}C(E_3), p2{:}G(\mathrm{A}_1)),$
$(q2{:}C(E_3), p1{:}G(\mathrm{A}_1)), (q2{:}C(E_4), p1{:}G(\mathrm{A}_1)), (q2{:}C(E_3), p2{:}G(\mathrm{A}_1)), (q2{:}C(E_4), p2{:}G(\mathrm{A}_1)),$
$(q3{:}C(E_3), p1{:}G(\mathrm{A}_1)), (q3{:}C(E_4), p1{:}G(\mathrm{A}_1)), (q3{:}C(E_3), p2{:}G(\mathrm{A}_1)), (q3{:}C(E_4), p2{:}G(\mathrm{A}_1))\}$.

6.4 Towards systematic testing

We will now outline a method for deterministic integration testing of distributed real-time systems, based on the identification of execution orderings, as presented above. This method is suitable if we want to address ordering failure semantics. We can use it for interleaving failure semantics also, although since we do not keep track of the exact interleavings we cannot guarantee deterministic testing, only partial determinism (as defined in section 3.2.2).

If we are only interested in addressing sequential failure semantics we can make use of regular unit testing [6], and regard each task as a sequential program. However, if we want to take into account the effects that the tasks may have on each other, we need to do integration testing, and assume ordering failure semantics. In any case, we assume that some method for testing of sequential programs is used.

6.4.1 Assumptions

In order to perform integration testing of distributed real-time systems we require the following:

- A feasible global schedule, including probes that will remain in the target system in order to eliminate the probe effect (as discussed in chapter 4).
- Kernel-probes on each node that monitors task-switches. This information is sent to dedicated tasks (probe-tasks) that identify execution orderings from the task-switch information and correlate it to run test cases.
- A set of in-line probes that instrument tasks, as well as probe-nodes, which output and collect significant information to determine if a test run was successful or not. This also includes auxiliary outputs of tasks' state if they keep state between invocations.
- Control over, or at least a possibility to observe, the input data to the system with regard to contents, order, and timing.

6.4.2 Test Strategy

The test strategy is illustrated in *Figure 6-12* and consists of the following steps:

1) Identify the set of execution orderings by performing execution order analysis for the schedule on each node. This includes compensation for the global clock synchronization jitter, and the global LCM, as introduced in section 6.3.1.
2) Test the system using any testing technique of choice, and monitor for each test case and node, which execution ordering is run during the interval $[0, T^{MAX}]$. Where T^{MAX} typically equals the global LCM in the distributed real-time system case.
3) Map the test case and output onto the correct execution ordering, based on observation.
4) Repeat 2-3 until required coverage is achieved.

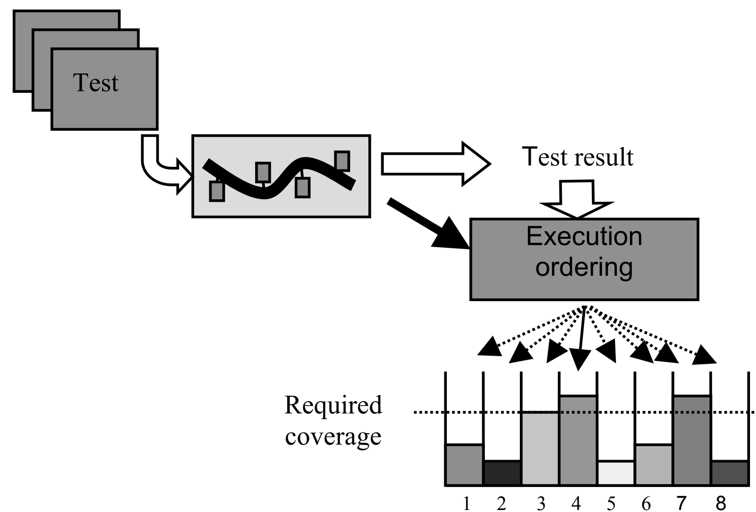


Figure 6-12. The test strategy and testing procedure. The numbers signify the different execution orderings, and the bars the achieved coverage for each ordering.

6.4.3 Coverage

In order to establish a certain level of confidence in the correctness of a system, we need to define coverage criteria, i.e., criteria on how much testing is required. This is typically defined in terms of the fraction of program paths tested [42][118]; paths, which in the multi-tasking/distributed scenarios we consider, are defined by the execution order graphs. The derived execution orderings also provide means for detecting violations of basic assumptions during testing, e.g., the execution of a scenario not defined by the EOG may be caused by an exceeded worst case execution time.

Complete coverage for a single node is defined by the traversal of all possible execution order scenarios, as illustrated in *Figure 6-13*. The coverage criteria for each scenario is however defined by the sequential testing method applied.

Complete coverage for the distributed system would naively be all combinations of all local execution order scenarios (as introduced in section 6.3.1). Depending

on the design, many of the combinations would however be infeasible due to data dependencies, with the consequence that a certain scenario on one node always gives a specific scenario on another node. Reductions could thus be possible by taking these factors into account. Other reductions could also be facilitated by limiting the scope of the tests, e.g., the entire system, multiple transactions (running over several nodes, single transactions (running over several nodes), multiple transactions (single node), single transactions (single node) or parts of transactions. These issues are however outside the scope of this thesis, but will definitely be considered in future work.

The correspondence between observations on each node will be limited by the granularity of the global time base, i.e., the globally observable state has a granularity of 2δ, defined by the precision of the clock synchronization (see section 4.4).

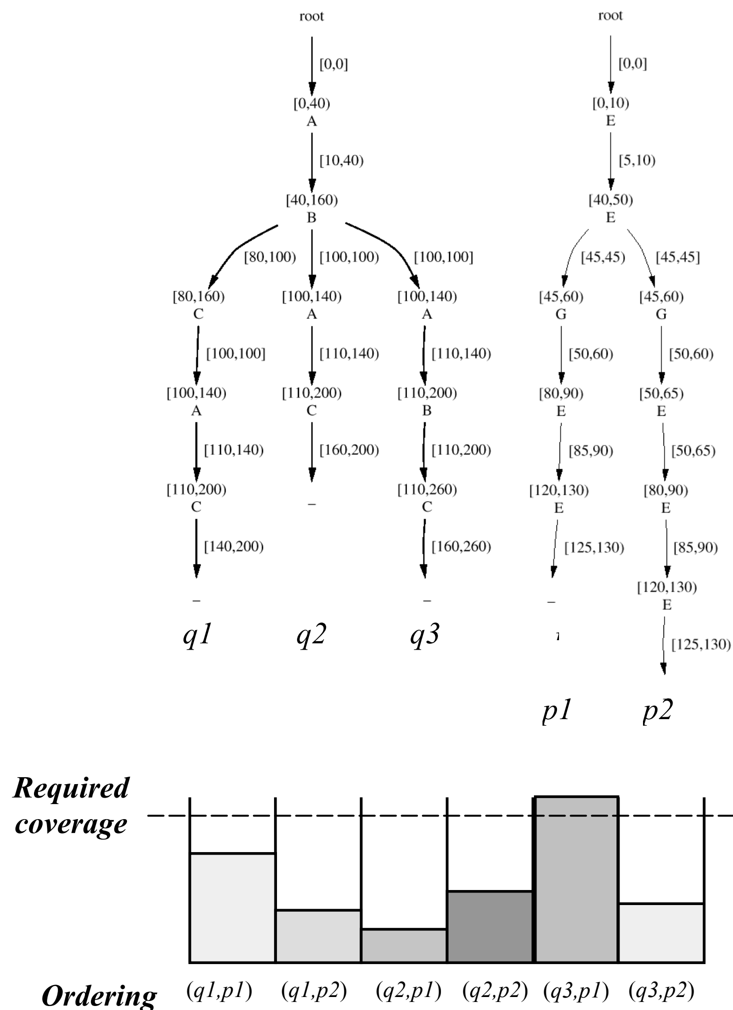


Figure 6-13. The test strategy and testing procedure for a DRTS. The orderings represent the different combinations of the local execution orderings. The bars represent the achieved coverage for each global execution ordering.

6.4.4 Reproducibility

To facilitate reproducible testing we must identify which execution orderings, or parts of execution orderings that can be enforced without introducing any probe effect. From the perspective of a single transaction, this can be achieved by controlling the execution times of preceding and preempting jobs that belong to other transactions. This of course only works in its entirety, if we adhere to ordering failure semantics, that the jobs have no unwanted functional side effects via unspecified interfaces, otherwise we could miss such errors. Control over the execution times in other transactions can easily be achieved by incorporating delays in the jobs, or running dummies, as long as they stay within each job's execution time range [*BCET*, *WCET*].

For example, consider *Figure 6-14*, and assume that task *C* and *A* belong to one transaction, and tasks *B*, *D* to another transaction. Assume that task *C* uses the last five samples provided by task *A*. With respect to tasks *A* and *C* we can reproduce the different scenarios by running a dummy in place of task *B*. By varying the execution time of the dummy we can enforce the different scenarios.

[B_{BCET}, B_{WCET}]:

1) [39,60)
2) [60,60]
3) [121,121]
4) (60,121)
5) (60,121)

Figure 6-14. The resulting execution order scenarios for the job set in Table 6-1.

6.5 Extending analysis with interrupts

We will now incorporate the temporal side effects of interrupts, by regarding them as sporadic jobs with a known minimum and maximum inter-arrival time. If we were interested in addressing the functional side effects of the interrupts, we would have to model each interrupt as a job. This would however make the EOG practically intractable, since we do not know the release times (or phases) of the interrupts, and must therefore consider infinite sets of release times within their inter-arrival times. We will here make use of standard response time analysis techniques [45][4] for calculating the execution- and completion intervals. This is rather pessimistic when the duration between releases of tasks is non-harmonious with the inter-arrival times of the interrupts, but we use this pessimistic (safe) and uncomplicated method in order to simplify the presentation. More refined and less pessimistic analysis can be developed as indicated by Mäki-Turja et al. [82].

Extended task model

We extend the task model in section 6.1 with a set of interrupts *Int*, where each interrupt $k \in Int$ has a minimum inter-arrival time T_k^{min}, a maximum inter-arrival time T_k^{max}, a best case interrupt service time $BCET_k$, and a worst case interrupt service time $WCET_k$.

Execution interval

Considering the interrupt interference, the execution interval $[\alpha, \beta)$ for an arbitrary task *A* (formula 6-2 in section 6.2.2.1) changes to:

$\alpha = MAX(a, r_A)$

$\beta = MAX(b, r_A) + w$, where w is the sum of $WCET_A$ and the maximum delay due to the preemption by sporadic interrupts, given by:

$$w = WCET_A + \sum_{\forall k \in Int} \left\lceil \frac{w}{T_k^{min}} \right\rceil \cdot WCET_k \qquad (6\text{-}5)$$

Hence, we calculate the interrupt interference on the preemption intervals in the same way as Response Time Analysis (RTA) [45] is used to calculate the response times for jobs that are subjected to interference by preemption of higher priority jobs.

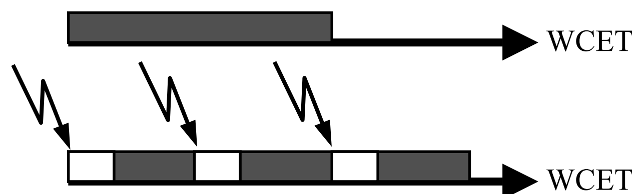


Figure 6-15. The worst case interrupt interference.

Start time interval

When adding interrupts, the upper bound b' of the start time interval $[a', b']$ for task A is still equal to β, whereas the lower bound a' changes to:

$$a' = a + w_a \qquad (6\text{-}6)$$

Where w_a, is defined as:

$$w = BCET_A + \sum_{\forall k \in Int} \left\lfloor \frac{w}{T_k^{max}} \right\rfloor \cdot BCET_k \qquad (6\text{-}7)$$

This equation captures that the minimum interference by the interrupts occur when they have their maximum inter-arrival time and execute with their minimum execution time, and when they have their lowest possible number of hits within the interval. The latter is guaranteed by the use of the floor function (⌊ ⌋).

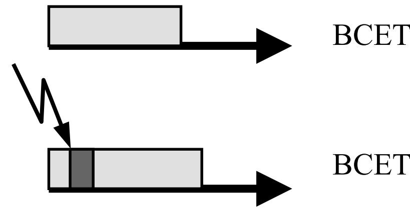


Figure 6-16. The least case interference by interrupts.

Execution times

In the EOG we decrease a preempted job *j's* maximum and minimum execution time with how much it has been able to execute in the worst and best cases before the release time t of the preempting job. Since we are now dealing with interrupts, the effective time that j can execute prior to the preemption point will decrease due to interrupt interference. The remaining minimum execution time $BCET_j'$ is given by:

$$BCET_j' = BCET_j - (t - MAX(a, r_j) - \sum_{\forall k \in Int} \left\lfloor \frac{t - MAX(a, r_j)}{T_k^{max}} \right\rfloor \cdot BCET_k) \qquad (6\text{-}8)$$

Note that the sum of interrupt interference is not iterative, but absolute, because we are only interested in calculating how much the job j can execute in the interval, minus the interrupt interference.

Likewise we can calculate the remaining maximum execution time, $WCET_j'$:

$$WCET_j' = WCET_j - (t - MAX(b, r_j) - \sum_{\forall k \in Int} \left\lceil \frac{t - MAX(b, r_j)}{T_k^{min}} \right\rceil \cdot WCET_k) \qquad (6\text{-}9)$$

Example 6-4

Here we assume that the system is subjected to preemption by interrupts. The attributes are described in *Table 6-4* and *Table 6-5*. The side effects of the interrupts are solely of temporal character. *Figure 6-17* depicts the EOG without interrupt interference, and *Figure 6-18* with interrupts accounted for. We observe that the interrupt may delay the execution of job *A* such that it will be preempted by job *B*, in contrast with the behavior without interrupts, where *A* always completes before *B*.

Table 6-4. Schedule, J

Job	*r*	*p*	*BCET*	*WCET*
A	0	1	1	3
B	3	2	1	3

Table 6-5. Interrupts

Interrupt	T^{max}	T^{min}	*BCET*	*WCET*
I	∞	3	1	1

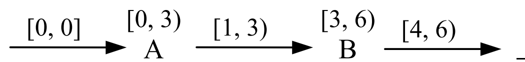


Figure 6-17. Not considering interrupts.

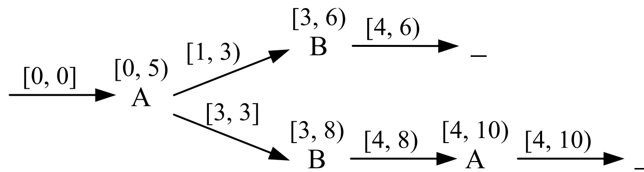


Figure 6-18. Considering interrupt interference.

6.6 Other issues

We will now outline some specifics of the execution order analysis with respect to jitter, scheduling, testability, and complexity.

6.6.1 Jitter

In defining the EOG, and in the presented algorithms, we take the effects of several different types of jitter into account:

- *Execution time jitter*, i.e., the difference between *WCET* and *BCET* of a job.
- *Start jitter*, i.e., the inherited and accumulated jitter due to execution time jitter of preceding higher priority jobs.
- *Clock synchronization jitter,* i.e., the local clocks keep on speeding up, and down, trying to comply with the global time base, leading to varying inter-arrival times between clock ticks.
- *Communication latency jitter*, i.e., the varying data transmission times of data passed between nodes in a DRTS, due to arbitration, propagation, etc.
- *Interrupt induced jitter*, i.e., execution time variations induced by the preemption of interrupts.

Since any reduction of the jitter reduces the preemption and release intervals, the preemption “hit” windows decrease and consequently the number of execution order scenarios decreases. Suggested actions for reducing jitter is to have fixed release times, or to force each job to always maximize its execution time, e.g. by inserting (padding) “no operation” instructions where needed.

Even if the $BCET = WCET$ for all tasks there will still be jitter if interrupts are interfering. Considering the hypothetical situation that we had no jitter what so ever on the nodes locally, then there could still be plenty of different execution orderings if the communication medium contributed to communication latency jitter.

In general, real-time systems with tighter execution time estimates and WCET ≈ BCET, as well as interrupt inter-arrival times of $T_k^{max} \approx T_k^{min}$, low communication latency jitter, and better precision of the clock synchronization yield fewer execution orderings than systems with larger jitter.

6.6.2 Start times and completion times

An interesting property of the EOG is that we can easily find the best and worst case completion-times for any job. We can also identify the best and worst case start times of jobs, i.e., the actual start times of jobs not their release times as dictated by the schedule. For identification of the completion times we only have to search the EOG for the smallest and largest finishing times for all terminated jobs. The response time jitter (the difference between the maximum and minimum response times for a job) can also be quantified both globally for the entire graph and locally for each path, as well as for each job during an LCM cycle. The same goes for start times of jobs. The graph can thus be utilized for schedulability analysis of strictly periodic fixed priority scheduled systems with offsets.

6.6.3 Testability

The number of execution orderings is an objective measure of system testability, and can thus be used as a metric for comparing different designs, and schedules. It would thus be possible to use this measure, for feedback, or as a new optimization criterion in the heuristic search used for generating static schedules with fewer execution orderings, with the intention of making the system easier to test. We could also use the EOG, as a means for judging which tasks’ execution times should be altered in order to minimize the number of execution orderings.

6.6.4 Complexity

The complexity of the EOG, i.e., the number of different execution orderings, and the computational complexity of the EOG algorithm (section 6.3), is a function of the scheduled set of jobs, J, their preemption pattern, and their jitter. From an $O(n)$ number of operations for a system with no jitter which yields only one scenario, to exponential complexity in cases with large jitter and several preemption points. In the latter case the number of scenarios is roughly 3^{X-1} for X consecutively preempting jobs (as illustrated in *Figure 6-19*), given that each preemption gives rise to the worst case situation of three possible transitions (see section 6.2.2.1).

This complexity is not inherent to the EOG but rather a reflection of the system it represents. In generating the EOG we can be very pragmatic, if the number of execution order scenarios exceeds, say a 100, we can terminate the search, and use the derived scenarios as input for revising the design. If the system has no preemption points, but large jitter there will only be one scenario. If the system has plenty of preemption points but no jitter, there will be only one scenario. If the system has plenty of preemption points, and jitter, but the magnitude of the jitter or the distribution of the preemption points are such that they will not give rise to different execution orderings there would be just one scenario. Knowing this, we can be very pragmatic in generating the EOG and try to tune the system in such a way that the jitter is minimized. Alternatively, release-times of tasks can be offset in such a way that they eliminate forks in the EOG (as long as the schedule or requirements allow).

Event triggered vs. Time triggered

The choice of not trying to model interrupt interference as high priority jobs is the choice of not modeling jobs with arbitrary arrival times. If we were interested in addressing the functional side effects of the interrupts, we would have to model each interrupt as a job. This would force us to consider all possible release times (with a finer granularity than provided by the real-time kernel clock tick) which would result in an (in practice) infinite number of execution orderings. This is exactly why we cannot model interleaving failures deterministically. This could be viewed as a critique against the EOG method but also as a critique against the testability of event triggered systems (system where the release times of tasks are arbitrary). The complexity and the testability of a system is thus strictly dependent on the infrastructure and the assumed failure semantics. That is, if we assume ordering failure semantics as we have done in the EOG method the number of execution orderings would approach infinity if tasks with arbitrary arrival times were allowed.

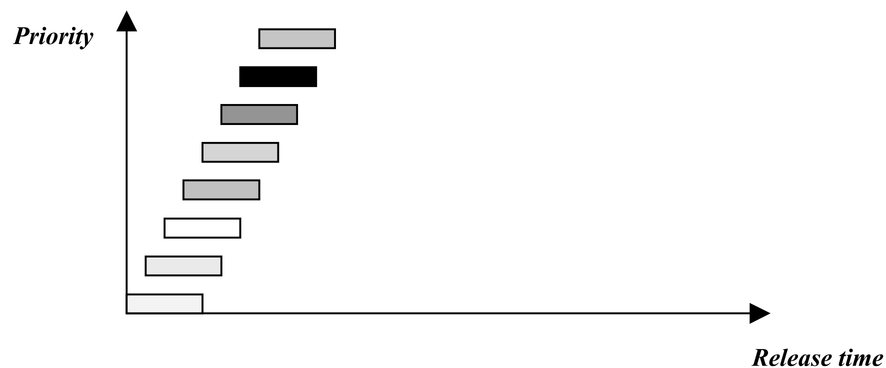


Figure 6-19. Consecutively preempting jobs.

6.7 Summary

In this chapter we have introduced a novel method for achieving deterministic testing of distributed real-time systems (DRTS). We have specifically addressed task sets with recurring release patterns, executing in a distributed system with a globally synchronized time base, and where the scheduling on each node is handled by a priority driven preemptive scheduler. The results can be summed up to:

- We have provided a method for finding all the possible execution scenarios for a DRTS with preemption and jitter.
- We have proposed a testing strategy for deterministic and reproducible testing of DRTS.
- A benefit of the testing strategy is that it allows any testing technique for sequential software to be used for testing of DRTS.
- Jitter increases the number of execution order scenarios, and jitter reduction techniques should therefore be used whenever possible to increase testability.
- The number of execution orderings is an objective measure of the testability of DRTS, which can be used as a new scheduling optimization criterion for generating schedules that are easier to test, or simply make it possible to compare different designs with respect to testability.
- We can use the execution order analysis for calculating the exact best-case and worst-case response-times for jobs, as well as calculating response-time jitter of the jobs. We could thus make use of the execution order analysis for schedulability analysis of strictly periodic fixed priority scheduled systems with offsets.

7 CASE STUDY

We will in this chapter give an example of how we for a given system can add monitoring mechanisms, and how we, using these mechanisms can test the system. We will also show how jitter reduction can increase the system testability.

7.1 A distributed control system

The considered system is a distributed control system (*Figure 7-1*), consisting of two nodes, *N1* and *N2*, which are interconnected by a temporally predictable broadcast network. On these nodes a distributed feedback control application is running. On node *N1* we have one part of the control loop running, consisting of four tasks *A*, *B*, *C* and *D,* and on node *N2* we have the task *E* running. The entire distributed transaction runs with period time, *T* = 400ms. Tasks *A* and *E* over sample the external process and perform some calculations which after some processing are passed on to task *C*, which in turn performs the control calculation. In its calculations task *C* uses the last four values from task *A,* the last five values from task *E,* the last value from task *B,* and the last value from task *D*. All messages are buffered such that when task *C* starts to execute it can atomically read received values without risk of violating consistency.

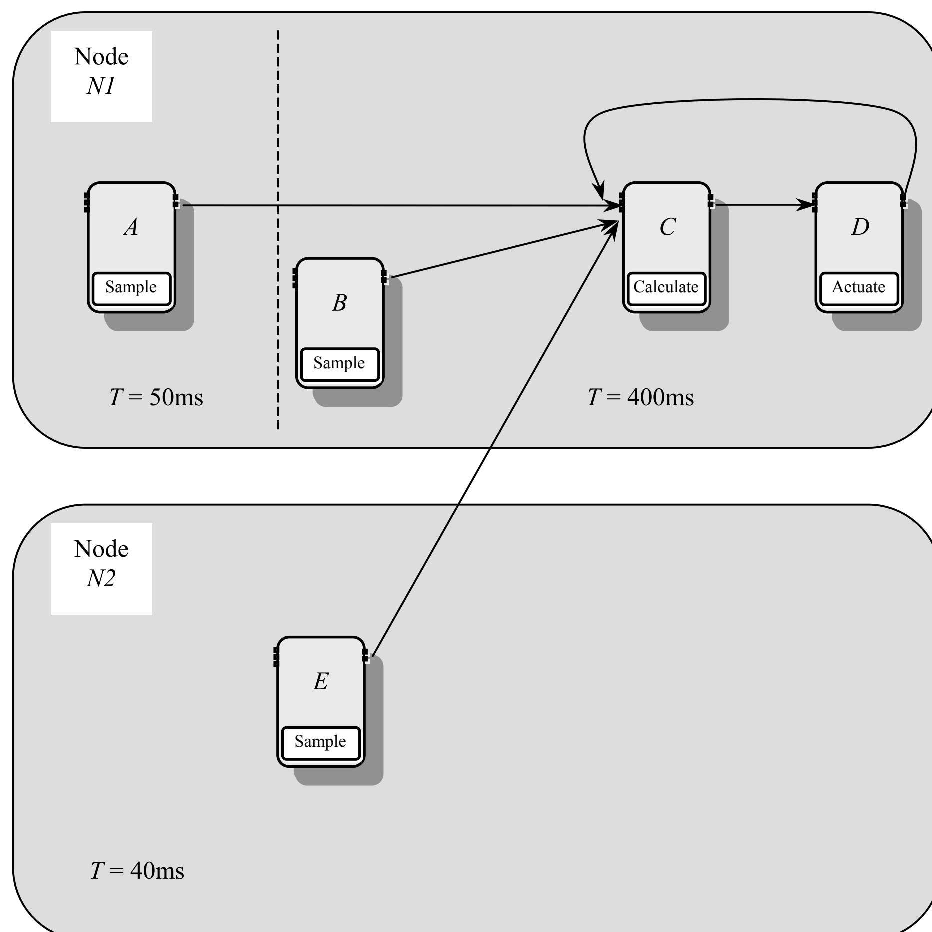


Figure 7-1. The distributed control system, consisting of Tasks A-E and their communication, on nodes N1 and N2.

Table 7-1 The task attributes. The time unit is ms.

Task	***T***	***WCET***	***BCET***	***Precede***	***Offset***	***Communication latency interval [Best, Worst)***
A	50	38	10			
B	400	120	40	*C*		
C	400	58	50	*D*		
D	400	19	10		*350*	
E	40	9	9			[2,5)
Clock synchronization precision						
δ=2 ms/s						

Given the illustrated data dependencies in *Figure 7-1*, and the task attributes in *Table 7-1* we can generate schedules on each node that minimize the sampling and actuation jitter. In *Table 7-2* and *Table 7-3* we display the resulting schedules for node *N1* and node *N2*, respectively, also considering clock synchronization effects.

Table 7-2 The job set for node N1 with a global LCM of 400 ms.

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	20	9

Table 7-3 The job set for node N2 with a global LCM of 400 ms.

Task	***r***	***p***	***WCET***	***BCET***
E	0	3	11	9
E	40	3	11	9
E	80	3	11	9
E	120	3	11	9
E	160	3	11	9
E	200	3	11	9
E	240	3	11	9
E	280	3	11	9
E	320	3	11	9
E	360	3	11	9

The resulting initial execution orderings, before data transformations, are depicted in figures 7-2, and 7-3. For node *N1* the schedule and the execution time jitter result in five different execution orderings, while for node *N2* the result is just one execution ordering. Task *C* on node *N1* is relying on data from tasks *A*, *B*, *E*, and a previous instance of *D*. However, depending on which execution ordering is run, task *C* will receive data from different instances of tasks *A*, and *E,* due to the multi-rate character of the application. We postpone the handling of this to section 7.4.

Figure 7-2. The resulting execution order scenarios for the schedule on node N1.

Figure 7-3. The resulting execution order scenario for the schedule on node N2.

7.2 Adding probes for observation

Having derived all execution orderings, are we all set to start testing the system? No! We need also extract the necessary information for testing, i.e., the inputs, outputs and execution orderings. We can assume that the real-time kernel provides the necessary support for recording context switches, like the real-time kernel Asterix does [108]. In order to collect all information, we need probes on each node. These probes can relay the inputs, outputs, and execution orderings to a test oracle. The test oracle deems if the inputs and outputs from the system comply with the requirements (as illustrated in *Figure 7-4*). The inputs to the system can be produced either by the actual target process environment, an environment simulator, or a test case selection mechanism.

Assuming that the system is receiving inputs from the external process, and that we do not know their values before hand, we need to monitor the sampled values, and pass them on to the test oracle. We need also to provide the test oracle with the outputs; the outputs sent to the actuators.

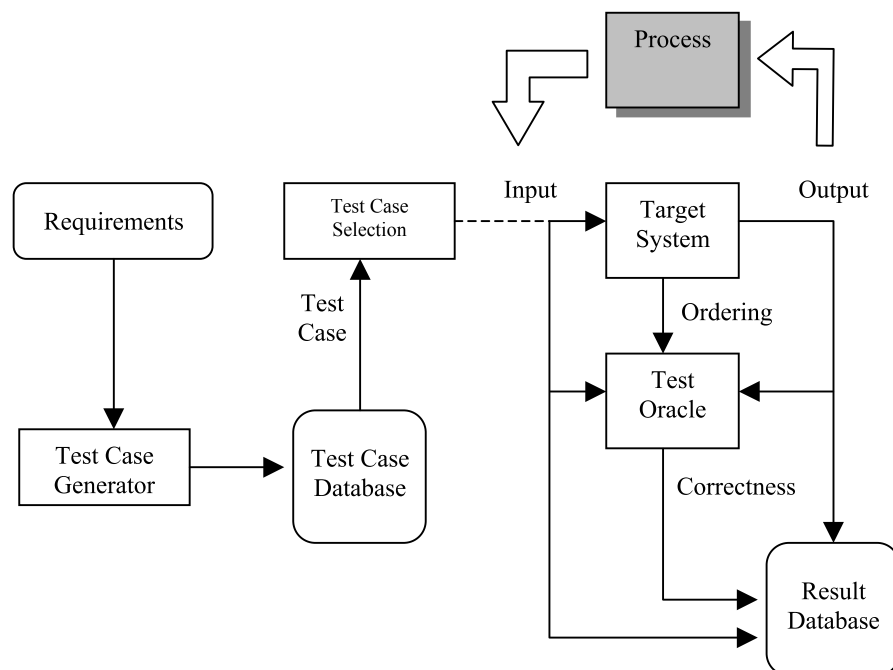


Figure 7-4. Typical test station components for integration testing of real-time systems software.

Observing *Figure 7-1*, we see that task *C* receives all inputs of significance for the control loop, and that task *D* produces the output. We thus need to add an inline probe to task *C,* which relays all inputs it has received, and add one inline probe to task *D,* which relays the output it actuates. We also add a probe task that can relay all the information from the probes in task *C* and *D*, as well as the execution ordering run during the LCM to the test oracle, residing on another node. Since the execution of the schedule on node *N2* only results in one scenario, and that we record the information that task *C* receives from task *E*, we do not need to instrument node *N2* with any probes except for the additional tagging of all messages with the originating task instance, as discussed in section 6.3.1.2.

The resulting system is illustrated in *Figure 7-5*.

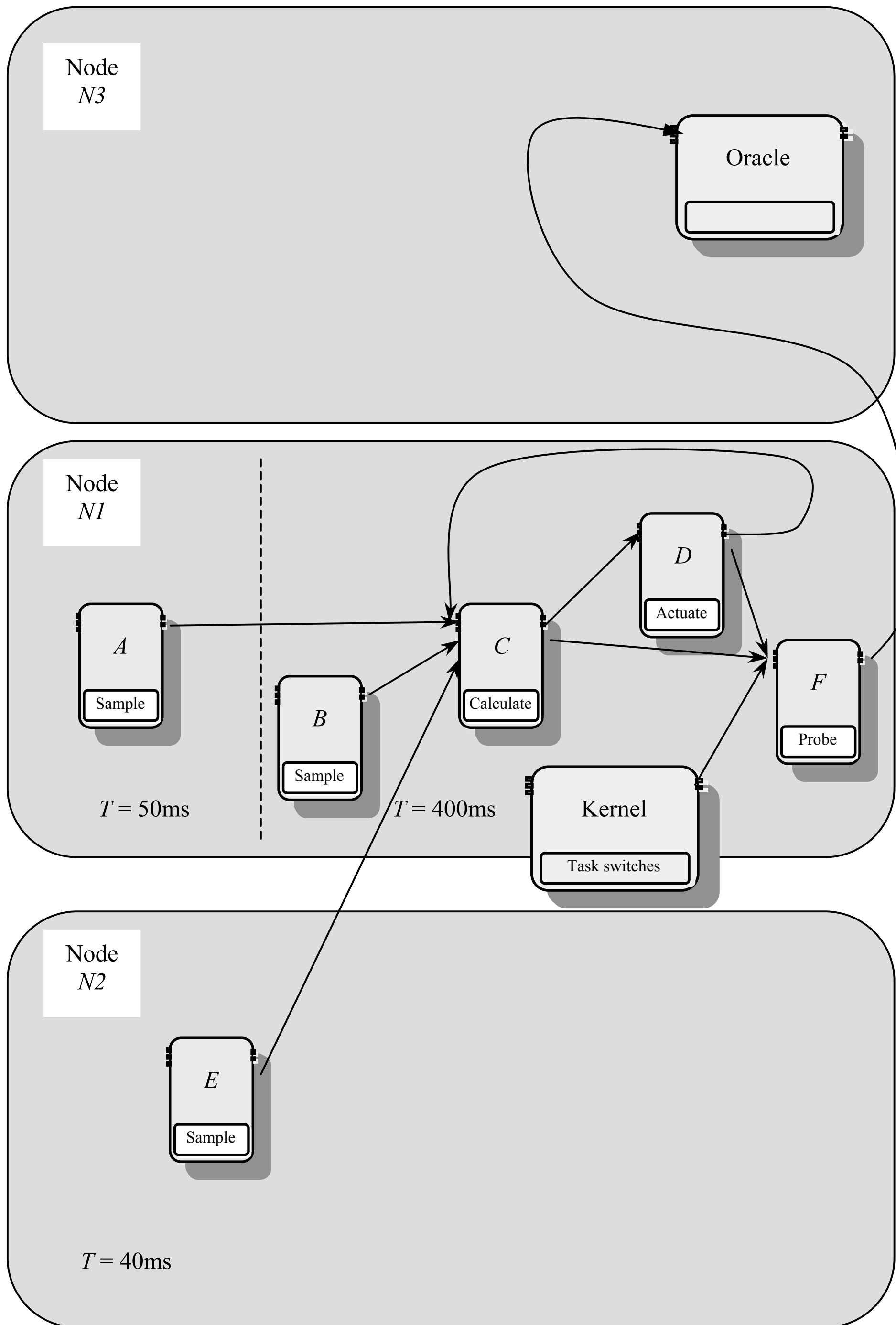


Figure 7-5. The distributed control system with probes in place and the test oracle hooked up.

Assume that we by some optimization of the program code in task *C,* can add the auxiliary outputs without increasing the WCET and BCET of the task with more than 1ms, and that the auxiliary output of task *D* only extends its BCET and WCET by 2ms. The resulting task attributes table can be viewed in *Table 7-4*, and the resulting schedules in tables 7-5, and 7-6.

Table 7-4 The task attributes. The time unit is milliseconds.

Task	*T*	*WCET*	*BCET*	*Precede*	*Offset*	*Communication latency interval [Best, Worst)*
A	50	38	10			
B	400	120	40	*C*		
C	400	59	51	*D*		
D	400	22	12	*F*	*350*	
F	400	14	9		380	
E	40	9	9			[2,5)
Clock synchronization precision						
δ=2 ms/s						

Table 7-5 The job set for node N1 with a global LCM of 400 ms.

Task	*r*	*p*	*WCET*	*BCET*
A	0	4	39	9
B	40	3	121	39
C	40	2	60	50
A	100	4	39	9
A	200	4	39	9
A	300	4	39	9
D	350	1	23	11
F	380	1	15	8

Table 7-6 The job set for node N2 with a global LCM of 400 ms.

Task	*r*	*p*	*WCET*	*BCET*
E	0	3	11	9
E	40	3	11	9
E	80	3	11	9
E	120	3	11	9
E	160	3	11	9
E	200	3	11	9
E	240	3	11	9
E	280	3	11	9
E	320	3	11	9
E	360	3	11	9

In *Figure 7-6*, the new execution ordering scenarios for the job set in *Table 7-5* are illustrated. The multi-rate character of the system and the execution time jitter of the tasks can potentially lead to different behaviors of the control application, even though we have fixed release times of the sampling and actuation tasks.

In our system we thus have five different execution orderings on node *N1,* (1-5), and one scenario on node *N2,* (α). That is, these are the original scenarios before taking data dependency relations into account.

7.3 Global execution orderings

The resulting initial set of execution orderings is thus $GEX_o = EX_o(N1) \times X_o(N2) = \{(1, \alpha), (2, \alpha), (3, \alpha), (4, \alpha), (5, \alpha)\}$. But, as task *C* on node *N1* consume data produced by task *E*, on node *N2* we need to analyze if this relationship gives rise to more execution order scenarios, as introduced in section 6.3.1.

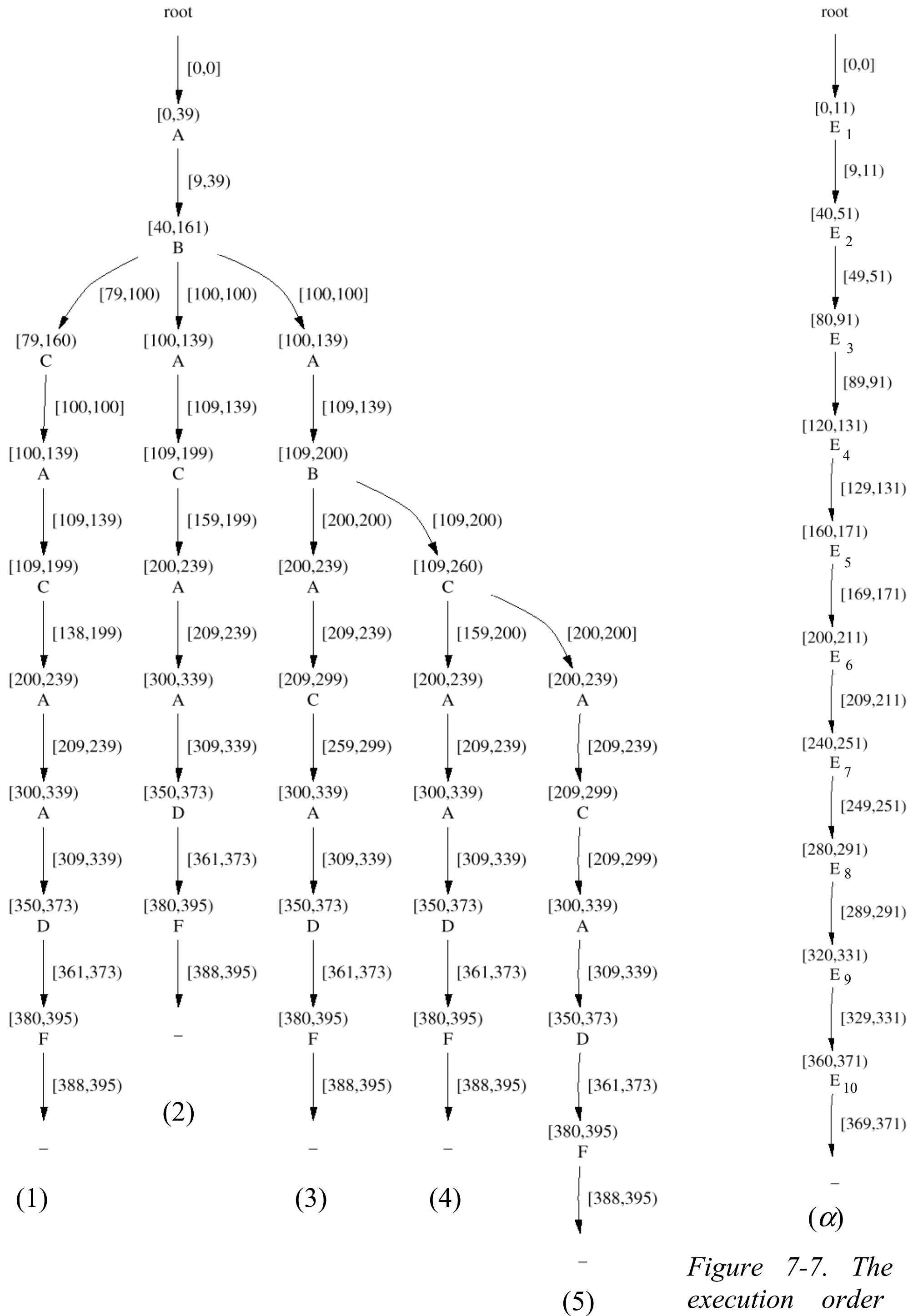


Figure 7-6. The resulting execution order scenarios for the schedule on node N1, with probes in place. All scenarios are indexed.

Figure 7-7. The resulting execution order scenario, (α), for the schedule on node N2. All jobs, i.e., task instances have been indexed.

7.4 Global execution ordering data dependency transformation

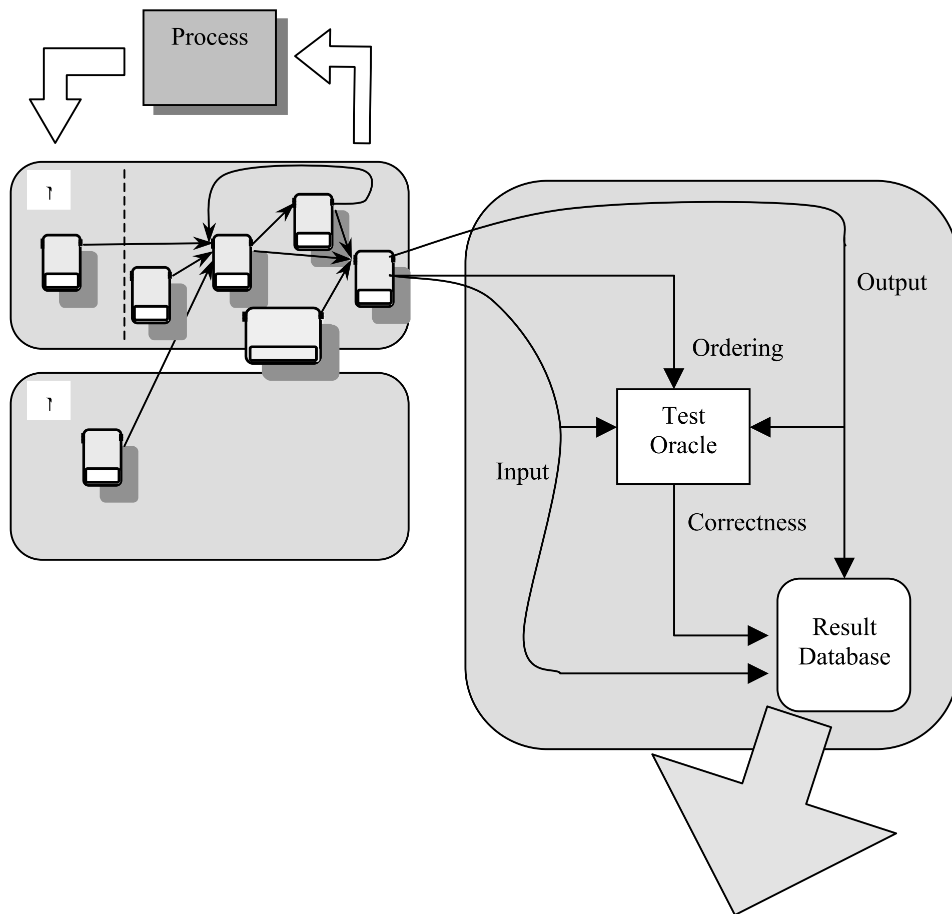
Beginning with scenario (1, α) we see that task C has a start time interval of $start(C)$ = [79,100). Considering the communication latency, [2,5), the only candidates are E_2 and E_3, thus scenario (1, α) transforms into scenario (1:E_2, α), and (1:E_3, α). Likewise:

- Scenario (2, α), where $start(C)$ = [109,139), transforms into scenario (2:E_3, α), and (2:E_4, α).
- Scenario (3, α), where $start(C)$ = [209,239), transforms into scenario (3:E_5 α), and (3:E_6, α).
- Scenario (4, α), where $start(C)$ = [109,150), transforms into scenario (4:E_3 α), and (4:E_4, α).
- Scenario (5, α), where $start(C)$ = [140,200), transforms into scenario (5:E_4 α), and (5:E_5, α).

The resulting global execution orderings set, GEX_o, thus becomes: {(1:E_2, α), (1:E_3, α), (2:E_3, α), (2:E_4, α), (3:E_5, α), (3:E_6, α), (4:E_3, α), (4:E_4, α), (5:E_4, α), (5:E_5, α)}. Which we henceforth rename (1, α) ... (10, α).

7.5 Testing

Assuming that all sampling tasks sample the external process with 8bit A/D converters and that the actuator controls the process with an 8bit D/A converter, we get $(2^8)^4 = 2^{32}$ input combinations, considering the feedback loop. This is a huge state space, and we will thus never in practice be able to do exhaustive testing, since running each LCM takes 400ms. Exhaustive testing would require continuous testing for at least 54 years. One possibility is to resort to statistical black box testing, and decide on a reasonable level of reliability, and confidence in this reliability. As the tasks in the feedback control loop all work on the same physical process it is however very likely that the sampled values by the tasks are dependent, which in practice will reduce the state space. Nonetheless, in this case study we assume that all input combinations are possible. Since we, as testers, cannot control the external process we have to settle for input observations. This means that the basic assumption for statistical testing [6], that all inputs are unique, do not hold. Consequently, we can run the same test case over and over again. The test oracle does thus need to remember if the same test case has been run or not. Having assumed all this we can start testing the system. The test station can be viewed in *Figure 7-8*. *Table 7-7* lists the resulting number of test cases and the time required for testing one scenario and ten scenarios respectively.



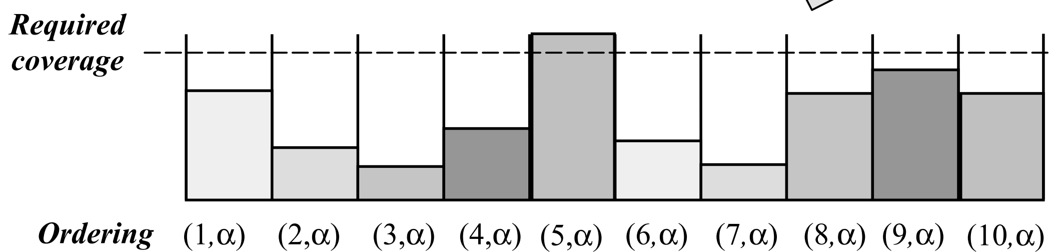


Figure 7-8. The resulting test station with the coverage of the global execution orderings illustrated.

Table 7-7. The relation between reliability, the number of test cases and the time required. Confidence C=0.99. One test case every 400ms.

Failures per tescase	*Number of test cases*	*150 tests per minute and ordering*	*10 orderings*
10^{-1}	44	0,3 minutes	2,9 minutes
10^{-2}	459	3,1 minutes	31 minutes
10^{-3}	4600	31 minutes	5,1 hours
10^{-4}	46,000	5,1 hours	51,1 hours
10^{-5}	460,000	51,1 hours	21,3 days
10^{-6}	4,600,000	21,3 days	213 days
10^{-7}	46M	213 days	5,8 years
10^{-8}	460M	5,8 years	58 years
10^{-9}	4,600M - Full coverage	58 years	580 years

7.6 Improving testability

As the number of scenarios is proportional to the number of preemption points and the jitter in the system, we will now try to improve on the situation by minimizing the jitter. After having analyzed the start times of task *C*, as illustrated in *Figure 7-6*, we give it a new release time corresponding to its latest start time. We also try to extend the BCET for task *B* such that it never can complete before task *A* is released.

Table 7-8 Changed task attributes in order to decrease jitter.

Task	*T*	*WCET*	*BCET*	*Precede*	*Offset*	*Communication latency interval [Best, Worst)*
A	50	38	10			
B	400	120	70	*C*		
C	400	59	51	*D*	240	
D	400	22	12	*F*	*350*	
F	400	14	9		380	
E	40	9	9			[2,5)
Clock synchronization precision						
δ=2 ms/s						

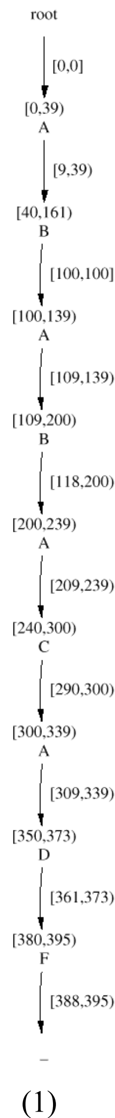
Table 7-9 The job set for node N1 with a global LCM of 400 ms.

Task	*r*	*p*	*WCET*	*BCET*
A	0	4	39	9
B	40	3	121	69
A	100	4	39	9
A	200	4	39	9
C	240	2	60	50
A	300	4	39	9
D	350	1	23	11
F	380	1	15	8

Table 7-10 The job set for node N2 with a global LCM of 400 ms.

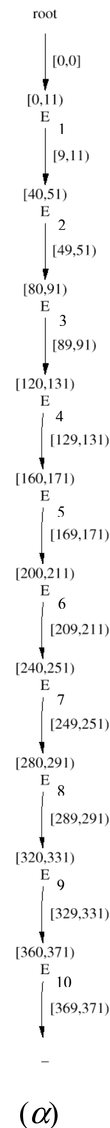
Task	*r*	*p*	*WCET*	*BCET*
E	0	3	11	9
E	40	3	11	9
E	80	3	11	9
E	120	3	11	9
E	160	3	11	9
E	200	3	11	9
E	240	3	11	9
E	280	3	11	9
E	320	3	11	9
E	360	3	11	9

The resulting execution order scenarios can be viewed in *Figure 7-9* and *Figure 7-10*. The global execution orderings set, GEX_o, would thus be $(1{:}E_6, \alpha)$, that is just one scenario. Which for a desired failure rate of 10^{-5} failures per test case would require testing for 51 hours instead of 21 days (according to *Table 7-7*). As illustrated in this example jitter reduction is of great significance. Although we did the analysis and the jitter reduction by hand this could easily be implemented in a tool. The information that can be derived from the execution orderings regarding preemption points, latest start times and jitter, could for example be fed back to a preemptive static scheduler where this information could be used for generating schedules with a reduced number of execution orderings.



(1)

Figure 7-9. The resulting execution order scenario, (1), *for the schedule on node N1.*



(α)

Figure 7-10. The resulting execution order scenario, (α), *for the schedule on node N2. All jobs, i.e., task instances have been indexed.*

7.7 Summary

The case study in this chapter has shown how we from a specified system created a schedule for it, derived its execution order scenarios, instrumented it in order to extract significant information for testing, presented how we could test it and how we could increase the testability of the system. This case study shows the significance of considering testing and monitoring early in the design process.

8 THE TESTABILITY OF DISTRIBUTED REAL-TIME SYSTEMS

What fundamentally defines the testability of RTS and DRTS? We will in this chapter summarize the definitions and results presented in this thesis.

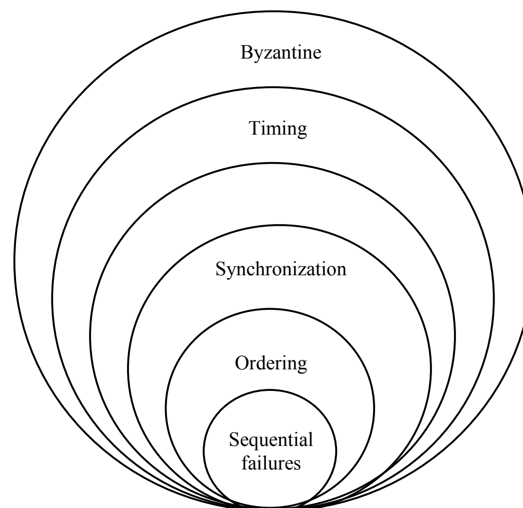
8.1 Observability

As we have previously discussed, there is a least necessary level of observability needed in order to guarantee determinism of observations.

> **Definition**. *Determinism*. (See section 3.2.2). A system is defined as deterministic if an observed behavior, *P*, is uniquely defined by an observed set of necessary and sufficient parameters/conditions, *O*.

In defining the necessary and sufficient conditions for deterministic observations we need to define which failure behavior we are looking for, i.e., we need to define a fault hypotheses (as defined in section 3.2):

1. Sequential failures
2. Ordering failures
3. Synchronization failures
4. Interleaving failures
5. Timing failures.
6. Byzantine and arbitrary failures.



That is, if we intend to deterministically detect ordering failures and interleaving failures, we need to observe more parameters than if we only want to deterministically detect sequential failures. Knowledge of the inputs and state of the tasks are necessary, but not sufficient for deterministically detecting ordering failures. That is, we need also observe the execution

orderings of the tasks. Likewise for deterministic observations of interleaving failures, e.g., non-reentrant code, we need, in addition, to observe the ordering of task interleavings, also observe the exact timing (down to the exact machine code instruction) when task interleavings occur, as done in the deterministic replay in chapter 5. In contrast, if we are just looking for failures pertaining to sequential failure modes, it would be overkill to observe the system to such a level that we can deterministically observe interleaving failures.

However, if the fault hypothesis is interleaving failures, but we have an infrastructure that supports memory protection, we would be able to reduce some, if not all, interleaving failures pertaining to memory corruption and non-reentrant code. We would then be able to reduce the fault hypothesis to synchronization, or even orderings failures, and therefore reduce the information necessary for observations. If we then, with memory protection support, observe, test, and debug the system, we can achieve a certain level of reliability. However, if we later remove the memory protection, more severe failure modes can occur, and we can thus not guarantee the achieved reliability, and as we have reduced the level of observation we cannot guarantee that the observations are deterministic. We will elaborate on this in the future work in chapter 9.

8.2 Coverage

In order to establish a certain level of confidence in the correctness of a system, we need to define coverage criteria, i.e., criteria on how much testing is required. This is typically defined in terms of sets of possible system states that must be explored. Examples are fractions of possible inputs tested, fractions of program paths tested, or in our case the number of execution orderings that needs to be explored. Consequently is the testability of a program a function of the number of states we must explore.

8.3 Controllability

In order to track down errors, achieve sufficient coverage when testing, or to facilitate regression testing, we need to reproduce observations and failures.

> **Definition**. *Reproducibility*. (See section 3.2). A system is reproducible if it is deterministic with respect to a behavior *P*, and it is possible to *control* the entire set of necessary and sufficient conditions, *O*.
>
> **Definition**. *Partial reproducibility*. (See section 3.2). A system is partially reproducible if it is deterministic with respect to a behavior *P*, and it is possible to *control* a subset of the necessary and sufficient conditions, *O*.

8.4 Testability

Based on the above we can conclude that that the testability of a real-time system is a function of:

- the fault hypothesis
- the extent to which the necessary and sufficient parameters for that fault hypotheses can be observed,
- the ability to control these parameters, and
- coverage.

The testability of a system pertaining to this definition is thus dependent on the infrastructure available, i.e., to which extent we can observe the system, which fault hypothesis we can discard, and on the number of behaviors we need to consider (coverage). The latter is facilitated by the infrastructure's support for control over all parameters necessary for achieving sufficient coverage.

9 CONCLUSIONS

We have in this thesis shown that dynamic verification by testing can never be eliminated when developing *reliable* software. Even if we make use of formal methods or fault tolerance techniques we will never be able to do away with testing, since testing can be used to validate underlying assumptions and the fidelity of models. Software reliability is essential in computer controlled safety-critical systems, all of which (with a few exceptions) are real-time systems, and frequently also distributed real-time systems. Any testing of safety-critical systems has thus to take into account the peculiarities of real-time systems (RTS) and of distributed real-time systems (DRTS).

We have in this paper described, discussed and provide solutions to a number of areas:

Monitoring

We have presented a framework for monitoring single tasking, multi-tasking, and distributed real-time systems. This includes a description of what to observe, how to eliminate the disturbances caused by the actual act of observing (the probe effect), how to correlate observations between nodes (how to define a global state), and how to reproduce the observations. We have given a taxonomy of different observation techniques, and identified where, how and when these techniques should be applied for deterministic observations. We have argued that it is essential to consider monitoring early in the design process, in order to achieve efficient and deterministic observations.

Debugging

We have presented a method for deterministic debugging of distributed real-time systems, which to our knowledge is

- The first entirely software based method for deterministic debugging of single tasking and multi-tasking real-time systems.
- The first method for deterministic debugging of distributed real-time systems.

The method relies on an instrumented kernel to on-line record the occurrences and timings of major system events. The recording can then, using a special debug kernel, be replayed off-line to faithfully reproduce the functional and temporal behavior of the recorded execution, while allowing standard debugging using break points etc. to be applied.

The method scales from debugging of single real-time tasks, to multi-tasking and distributed multitasking systems.

The cost for this dramatically increased debugging capability is the overhead induced by the kernel instrumentation and by instrumentation of the application code. To eliminate probe-effects, these instrumentations should remain in the deployed system. We are however convinced that this is a justifiable penalty for many applications.

Testing

We have in this thesis introduced a novel method for deterministic testing of multitasking and distributed real-time systems. We have specifically addressed task sets with recurring release patterns, executing in a distributed system with a globally synchronized time base, and where the scheduling on each node is handled by a priority driven preemptive scheduler. The results can be summed up to:

- We have provided a technique for finding all the possible execution scenarios for a DRTS with preemption and jitter.
- We have proposed a testing strategy for deterministic and reproducible testing of DRTS.
- A benefit of the testing strategy is that it allows any testing technique for sequential software to be used to test DRTS.
- Jitter increases the number of execution order scenarios, and jitter reduction techniques should therefore be used whenever possible to increase testability.
- The number of execution orderings is an objective measure of the testability of DRTS, which can be used as a new scheduling optimization criterion for generating schedules that are easier to test, or simply make it possible to compare different designs with respect to testability.
- We can use the execution order analysis for calculating the exact best-case and worst-case response-times for jobs, as well as calculating response-time jitter of the jobs. We could thus make use of the execution order analysis for schedulability analysis of strictly periodic fixed priority scheduled systems with offsets.

Testability

We have in this thesis also argued that testability is an essential property of a system, and that it is necessary to try to maximize the testability. The problems of designing and verifying software are fundamental in character: software has a discontinuous behavior, no inertia, and has no physical restrictions what so ever, except for time. Thus, we cannot make use of interpolation or extrapolation when testing, and we conclude that it is essential to design for high testability. If software engineering is going to be established as a real engineering discipline we must take the quest for high testability seriously.

Finally

Any programming language, any formal method, any theory of scheduling, any fault tolerance technique and any testing technique will always be flawed or incomplete. To design reliable software we must thus make use of all these concepts in union. Consequently we will never be able to eliminate testing completely. However, with respect to testing, debugging and monitoring of real-time systems software there has been very little work done. Relating to what was written in the introduction of this thesis, we believe however that the contributions of this thesis increase the depth on the shallow side of the DRTS verification pool of knowledge (no risk of drowning though).

10 FUTURE WORK

In this chapter we will outline some issues that we consider relevant and important to pursue in the future. We will begin with some extensions to the work presented in this thesis, and then widen the scope by discussing the use of components in real-time systems, i.e., reusable software entities, and how these components affect the potential reliability, and verification required.

10.1 Monitoring, testing and debugging

With respect to the results presented in this thesis we suggest the following extensions and pursuits:

- Experimentally validate the usefulness of the presented results. This pursuit will hopefully be facilitated by the use of the real-time kernel Asterix [108] that has specifically been developed to give the necessary support for monitoring, debugging and testing.
- Extend the execution ordering analysis to also encompass critical regions, by for example assuming immediate inheritance protocols for resource sharing. This analysis would however necessitate assumptions about the duration of the critical regions (intervals), and assumptions about the intervals of delay before entering the critical regions. Knowing these parameters we believe it is straightforward to extend the execution ordering analysis.
- Devise an exact analysis (less pessimistic) for interrupt interference on the execution order graph. The method presented in this thesis is rather pessimistic when the duration between releases of tasks is non-harmonious with the inter-arrival times of the interrupts. It is safe but pessimistic. For example, consider a system where we have two tasks, *A* and *B*, where task *B* has the highest priority, where r_A=0, $WCET_A$ = 3, r_B = 4, $WCET_B$ = 3, and an interrupt with a T^{min} = 3, and a service routine with a *WCET* of 1. The response time for both tasks, just considering their execution time and the worst case interference by the interrupt, is 5 (two interrupt hits each). It is however not possible for both tasks to be hit twice by the interrupt during the same scenario. We therefore suggest an extension of the analysis to minimize the pessimism by taking these types of situations into account.
- Complete coverage for the distributed system would naively be all combinations of all local execution order scenarios (as introduced in section 6.3.1). Depending on the design, many of the combinations would however be infeasible due to data dependencies, with the consequence that a certain scenario on one node always gives a specific scenario on another node. Reductions that take these factors into account should be further investigated. One possibility to reduce the required coverage could be to perform parallel composition of the individual EOGs, using standard techniques for composing timed transition systems [101], this would probably reduce the pessimism introduced in the GEXo data dependency transformation.

We will now continue with a discussion on software reuse in real-time systems and what the effects are on verification.

10.2 Formal and probabilistic arguments for component reuse and testing in safety-critical real-time systems

In this section we will discuss how testing of real-time software relates to component reuse in safety-critical real-time systems. Reuse of software components is a new alleged silver bullet that will kill all bugs. This, as for all previously alleged silver bullets is of course just a myth. Experience of software reuse has shown that re-verification cannot be eliminated in general, which for safety-critical systems is a significant problem since verification is the single most cost and time consuming activity during system development (as discussed in chapter 2).

In order to remedy this situation we will introduce a novel framework for formal and probabilistic arguments of component reuse and re-verification in safety-critical real-time systems. Using both quantitative and qualitative descriptions of component attributes and assumptions about the environment, we can relate input-output domains, temporal characteristics, fault hypotheses, reliability levels, and task models. Using these quantitative and qualitative attributes we can deem if a component can be reused without re-verification or not. We can also deem how much and which subsets, of say input-output domains, that need additional functional testing.

In order to be able to design software that is as safe and reliable as the mechanical or electromechanical parts it replaces, and to reduce the time to market, focus of current research and practice has turned to the reuse of proven software components. The basic idea is that the system designer should be able to procure (or reuse) software components in the same manner as hardware components can be acquired. Hardware components like nuts, bolts, CPUs, memory, A/D converters can be procured based on functionality, running conditions (the environment) and their reliability. Hardware components are typically divided into two classes, commercial and military grade components, where military grade electronics can withstand greater temperature spans, handle shock and humidity better than commercial components. However, for software it is not so easy to make the same classification since the metrics of software are not based on physical attributes, but rather on its design, which is unique for each new design. As mentioned in the background of this thesis, software has no physical restrictions what so ever and the sole physical entity that can be modeled and measured by software engineers is *time*.

We will in this section elaborate on this idea of components for safety-critical real-time systems. The arguments for reuse of software (components) are usually arguments for rapid prototyping (reusing code), arguments for outsourcing, and arguments for higher reliability. In the latter case it is assumed that verification of the components can be eliminated and that the reliability of the component can be “inherited” from previous uses. Expensive and catastrophic experiences have however shown that it is not so simple: Ariane 5, and Therac 25. The explosion of the Ariane rocket [43][62], and the accidents due to excessive dosage by the radiotherapy machine Therac 25 [66] were all due to misconceptions, or disregard, of changes in the target environment from earlier uses of the software (Ariane 4 and Therac 20). The believed proven reliability of the components had no bearing in the new environments where they were reused. The belief in the reliability of the components meant in the cases of Ariane5 and Therac 25 that re-verification and re-validation was ignored. One lesson learned from these

accidents is that reuse does not eliminate testing. However, as discussed in chapter 2, verification and maintenance cost is the single most resource consuming activity in software development projects (especially safety-critical such). We will elaborate on this problem during the course of the section and present a framework for determining when components in real-time systems can be reused immediately, when complete re-testing is necessary, or when just parts of the systems need additional verification.

10.2.1 Software components in real-time systems?

There exists yet no commonly agreed upon definition of what constitutes a software component [103][25][98][32][2][15][18]. Most people do however agree that the concept of components is the idea of reusable software entities. In this section, for the sake of enabling analysis of components in safety-critical real-time systems we do not only define a component in terms of software. We define a component also in terms of design documentation, test documents, and in evidence of the reliability achieved from previous uses.

We have chosen a hierarchical/recursive definition of components (*Figure 10-1*). The smallest components are tasks. On the next level, several tasks work together in a transaction where the transaction itself is a component. A transaction is a set of functionally related and cooperating tasks, e.g., sample-calculate-actuate loops in control systems. The relationship between the cooperating tasks with respect to precedence (execution order), interactions (data-flow), and a period time typically define each transaction. Further up in the hierarchy several transactions make up more complex components. The entire system software is for example a component from the complete system's point of view. As this definition eventually forces us to define the universe as a software component, we have to stop the recursion somewhere. Here, in this paper the real-time kernel is not considered to be a component. We define the real-time kernel as part of the infrastructure of the system. It provides the necessary services for the software but it also restricts the possibilities (freedom) of implementing the requirements put upon tasks and transactions. Which in this case is regarded as a good quality since it limits the complexity and also makes formal analysis possible.

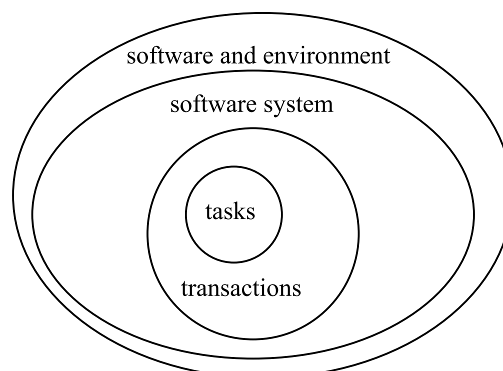


Figure 10-1. Hierarchical definition of components for real-time systems.

10.2.2 Component contracts

A component contract is in many respects analogous to a contract in real life. It defines certain obligations and benefits between the parties in an agreement, in this case the agreement between the provider (the component) and the consumer (the target system). The consumer must ensure that certain assumptions hold before it can use the service offered by the provider, i.e. the component. A component contract essentially specifies:

- The functionality of the component
- The running conditions for which the component has been designed. That is, assumed inputs and outputs, as well as the assumptions about the environment.

Contracts can be described with varying degrees of formality [7][39][41]. The more formal the contracts are, the greater the possibilities are of verifying that the component will work correctly together with the target system. In this section we will settle for a functional description in some natural language, although a formal temporal language such as timed automata [3] could have been used. The main focus in this section is however on the specific demands put on component contracts in safety-critical real-time systems, and we will as such not elaborate on functional descriptions further.

As the temporal behavior is of utmost importance for the correctness of real-time systems, the temporal assumptions have to be specified in the contracts. The temporal attributes make up the task model, which is used when formally verifying the temporal behavior of the system, e.g., when scheduling [4][117]. However, there exist in essence two different task models for real-time systems, one that exists on the design level and one that is provided by the given infrastructure (the real-time kernel). Typically, the design task model describes timing requirements like the periodicity of the transactions, end-to-end deadlines, jitter constraints, etc, and communication and synchronization requirements. These requirements are after successful scheduling implemented by the infrastructure. We will in the rest of this section distinguish between the design task model and the infrastructure task model.

10.2.2.1 The design task model

We view components in real-time systems as single tasks or transactions consisting of tasks and other transactions. Thus, the contract must specify whether or not the component is a transaction or not. If the component is a transaction it has a precedence relation between the cooperating tasks and there is a data-flow between the tasks. Note that a single task is a special case of a transaction without precedence relations or data-flow.

As defined previously the relationship between the cooperating tasks with respect to precedence (execution order), interactions (data-flow), and a period time typically define each transaction.

Precedence

Precedence relations can be represented in a multitude of ways, but we settle for a simple graphical notation called a precedence graph. A precedence graph is a directed graph visualizing the tasks in the transaction by interconnected arrows indicating the direction of the precedence relation. In *Figure 10-2* a simple precedence graph is shown where task *A* precedes task *B* and *C*.

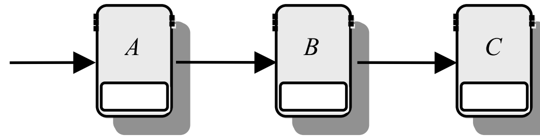


Figure 10-2. A simple precedence graph.

Data flow

Concerning the data-flow in components we identify two different attributes in the contracts namely: *where* and *how*. The attribute *where*, declares which tasks within the component exchange data. The attribute *how*, specifies if the communication between the tasks flows in a synchronous or an asynchronous manner. If the communication is asynchronous, the contract must specify how the data is treated on the receiver side. This is important since data can be consumed either faster or slower than it is produced in a multi-rate transaction (see *Figure 10-3*). If data is produced in a faster pace than it is consumed, the contract must specify whether or not new data should overwrite old data or if data should be buffered to provide history. Finally, concerning data propagation, the size of the data is important since size will restrict the speed and periodicity at which the component can execute.

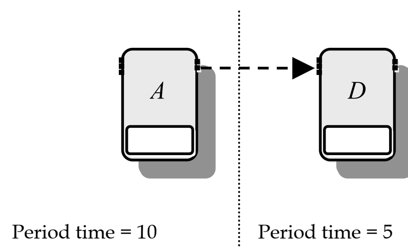


Figure 10-3. The data-flow between two tasks with different period times.

Temporal attributes

The temporal attributes of real-time components specify the required temporal behavior of the components. As discussed, these attributes must not necessarily have their correspondence in the infrastructure, but they must in the end be realized using the infrastructure. As we focus on reuse in this paper, we want our components to be as general as possible in terms of the temporal constraints. The generality is obtained by specifying the temporal constraints as time intervals. Later on in section 10.2.3.2, we will elaborate further on how these intervals are obtained. Transaction components (where a single task is a special case) can have period times, jitter constraints on period times, end-to-end deadlines, etc. (*Figure 10-4*)

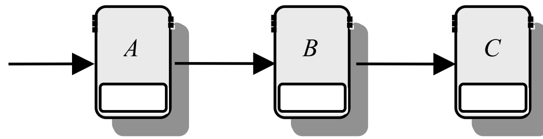


Figure 10-4. A transaction with a period time, precedence relation, and an end-to-end deadline.

For a component consisting of two transactions which run with different period times (multi-rate) we can construct a composite graph representing the Least Common Multiple (LCM) of their period times (*Figures 10-5* and *10-6*).

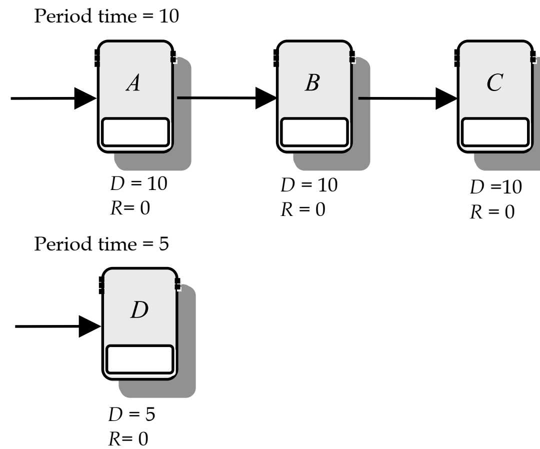


Figure 10-5. A multirate component, where the attributes are D = deadline, and R = release time relative transaction start.

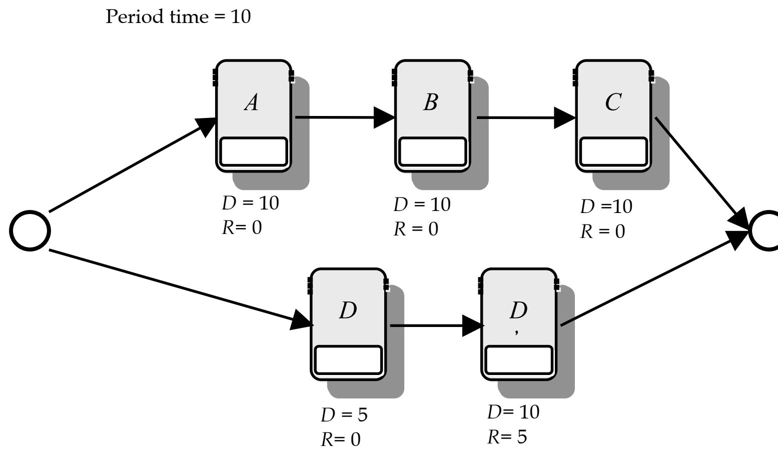


Figure 10-6. The composite transaction with a common period time.

Reuse and the design task model

The period time of a transaction is inherited from the system in which the component will be reused. We therefore suggest that the design task model attributes of the reused components can be parameterized, based on decreased/increased period times of the transactions. That is, parameterized such

that that all precedence relations, and temporal attributes still hold, the only thing that changes is the scale. For example, a 20% decrease in the period time of the transaction in *Figure 10-6* is illustrated in *Figure 10-7*.

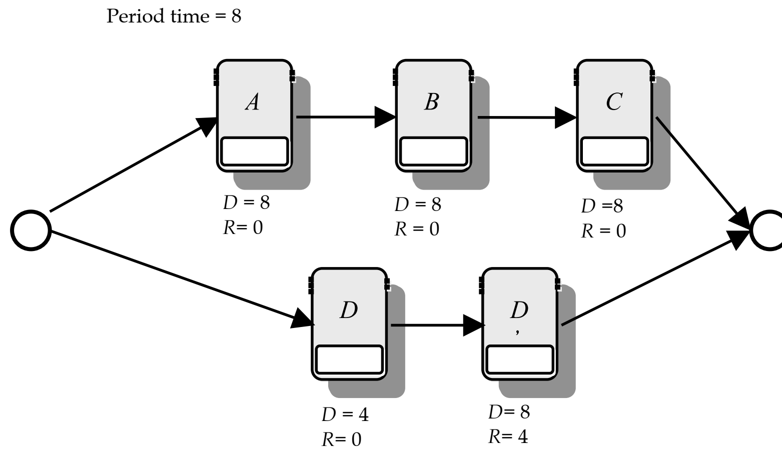


Figure 10-7. A parameterized transaction.

10.2.2.2 Infrastructure task model

Derived from the recursive definition of components and component relations, we can with respect to real-time systems define the components forming the basis for running the real-time tasks as the infrastructure, i.e., the real-time operating system and its attributes. We are now going to give a classification of the infrastructure for real-time operating systems based on their execution strategy, synchronization mechanisms, and communication mechanisms.

Execution strategy

The execution strategy of a real-time system defines how the tasks of the system are run. A usual classification is event and time triggered systems [55]. An event-triggered system is a system where the task activations are determined by an event that can be of external or internal origin, e.g., an interrupt, or a signal from another task. There are usually no restrictions what so ever on when events are allowed to arrive, since the time base is dense (continuos). For time-triggered systems events are only allowed to arrive into the system, and activate tasks, with a certain minimum and maximum inter-arrival time; the time-base is sparse [53]. The only event allowed is the periodic activation of the real-time kernel, generated by the real-time clock.

Another characteristic of an execution strategy is the support of single tasking or multitasking. That is, can multiple tasks share the same computing resource (CPU), or not? If the infrastructure does support multitasking does it also support task interleavings, i.e., does it allow an executing task to be preempted during its execution by another task, and then allow it to be resumed after the completion of the preempting task?

A very common characteristic of an execution strategy, especially for multi-tasking RTS, is the infrastructure task model. The task model defines execution attributes for the tasks [4][75][89][117]. For example:

- **Execution time**, C_j. Where $C_j \in [BCET_j, WCET_j]$, i.e., the execution time for a task, *j*, varies depending on input in an interval delimited by the task's best case execution time ($BCET_j$) and its worst case execution time ($WCET_j$).
- **Periodicity**, T_j. Where $T_j \in [T_j^{min}, T_j^{max}]$, i.e., the inter-arrival time between the activations of a task, *j*, is delimited by an interval ranging from the minimum inter-arrival time, to the maximum inter-arrival time. For strictly periodic RTS, $T_j^{min} = T_j^{max}$.
- **Release time**, R_j. Where R_j defines an offset relative the period start at which the task *j* should be activated. Release time also go by the name Offset.
- **Priority**, P_j. Where P_j defines the priority of task *j*. When several tasks are activated at the same time, or when an activated task has higher priority than the currently running task, the priority determines which task should have access to the computing resource.
- **Deadline**, D_j. Where D_j defines the deadline for the task, *j*, that is, when it has to be finished. The deadline can be defined relative task activation, its period time, absolute time, etc.

For different infrastructures the task model vary with different flavors of the above-exemplified attributes.

Synchronization

Depending on the infrastructure we can either make necessary synchronizations between tasks off-line if it is time triggered and supports offsets, or we can synchronize tasks on-line using primitives like semaphores. In the off-line case we guarantee precedence and mutual exclusion relations by separating tasks time-wise, using offsets.

Communication

Communication between tasks in RTS can be achieved in a multitude of ways. We can make use of shared memory which is guarded by semaphores, or time synchronization, or we can via the operating system infrastructure send messages, or signals between tasks. Depending on the relation between the communicating tasks, in respect to periodicity, the communication can vary between totally synchronous communication to totally asynchronous communication. That is, if task *i* sends data to task *j*, and both tasks have equal periodicity, $T_j = T_i$, we can make use of just one shared memory buffer. However, if $T_j > T_i$ or $T_j < T_i$ the issue gets more complicated. Either we make use of overwriting semantics (state-based communication), using just a few buffers, or we record all data that has been sent by the higher frequency task so that it can be consumed by the lower frequency task when it is activated. There are several approaches to solving this problem [26][52][13].

10.2.3 Component contract analysis

We are now going to introduce a framework for formal and probabilistic arguments of component reuse in safety-critical real-time systems. The basic idea is to provide evidence, based on the components contracts and the experience accumulated, that a component can be reused immediately, or if only parts can be reused - or not at all. That is, we want to relate the environment for which the

component was originally designed and verified, with the new environment where it is going to be reused. Depending on the match with respect to input-output domains and temporal domains we want to deem how much of the reliability from the earlier use of the component can be inherited in the new environment. Faced with a non-match we usually are required to re-verify the entire component. However, we would like to make use of the parts that match and only re-verify the non-matching parts.

Specifically, we are now going to introduce two analyses, one with respect to changes in the input-output domain, and one with respect to changes in the temporal domain.

10.2.3.1 Input-output domain analysis

We are now going to establish a framework for comparative analysis of components input-output domains, i.e. their respective expected inputs and outputs, as defined by the components contracts (interfaces). But, we are also going to relate the input-output domain to its verified and experienced reliability, according to certain failure semantics. We can represent the input-output domain for a component, *c*, as a tupel of inputs and outputs; $\Pi(c) \subseteq I(c) \times O(c)$. Where the input domain, $I(c)$, is defined as a tupel of all interfaces and their respective sets of input; $I(c) \subseteq I(c)_1 \cup \ldots \cup I(c)_n$. Further, the output domain, $O(c)$, is defined as a set of all interfaces and their respective sets of output; $O(\mathrm{c}) \subseteq O(c)_1 \cup \ldots \cup O(c)_n$.

The basic idea is that, given that we have a reliability profile, $R(c)$, of a component's input-output domain, $\Pi(c)$ and that the attributes for the real-time components are fixed, we can deem how well the component would fare in a new environment. That is, we would be able to make comparative analysis of the experienced input-output domain and the domain of the new environment.

For example, assume that we have designed and verified a component, *c*, that has an input domain $I(c)$ corresponding to a range of integer inputs, $I(c)$ = [40,70], to a certain level of reliability, $R(c)$, as illustrated in *Figure 10-8*.

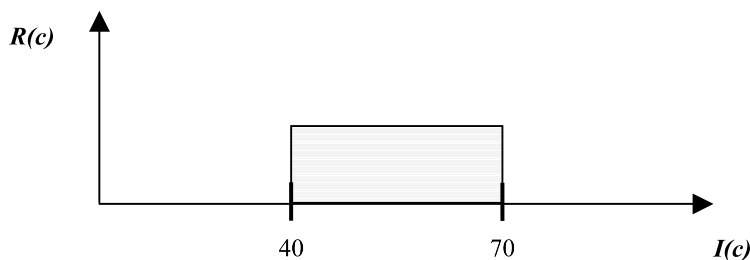


Figure 10-8. The reliability for input domain [40,70].

Consider now that we reuse this component in another system, where all things are equal except for the input domain, $I_2(c)$ = [50,80], and that we by use and additional verification have achieved another level of reliability. We can now introduce a new relation called the experienced input-output-reliability domain, $E(c) \subseteq (\Pi_1(c) \times R_1(c)) \cup \ldots \cup (\Pi_n(c) \times R_n(c))$, which represent the union of all input-output domains and the achieved reliability for each of these domains. This union is illustrated in *Figure 10-9*. Using this $E(c)$ we can deem if we can reuse the component immediately in a new environment $\Pi_{new}(c)$ and inherit the

experienced reliability, or we can put another condition on the reuse, a reliability requirement.

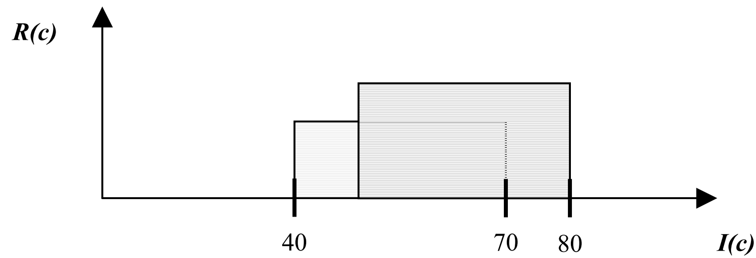


Figure 10-9. The reliability for joint input domain.

In *Figure 10-10*, the new environment and the area named *X*, illustrates the reliability requirement. If it can be shown that $\Pi_{new}(c) \times R_{new}(c) \subseteq E(c)$ then the component can be reused without re-verification. However if the reliability requirement cannot be satisfied as illustrated in *Figure 10-10*, reuse cannot be done without additional verification.

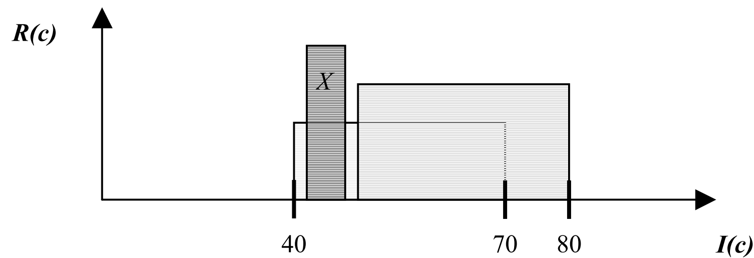


Figure 10-10. Reliability cannot be guaranteed in new environment.

If the $\Pi_{new}(c) \times R_{new}(c) \subset E(c)$, then we cannot immediately reuse the component without re-verification, however we need not re-verify the component entirely. It is sufficient to verify the part of the input domain that is non-overlapping with the experienced one, i.e., $(\Pi_{new}(c) \times R_{new}(c)) \setminus E(c)$.

By making this classification we can provide arguments for immediate reuse, arguments for re-verification of only cut sets and non-reuse. For example, assume that the $\Pi_{new}(c)$ (the area named *Y* in the *Figure 10-11*) has an $I_{new}(c)$ = [35,45] then we can calculate the cut set for $I_{new}(c)$ and $I(c)$ to be $I_{cut}(c)$ = [35,40]. This limits the need for additional verification to only encompass the cut set $I_{cut}(c)$, see *Figure 10-11*.

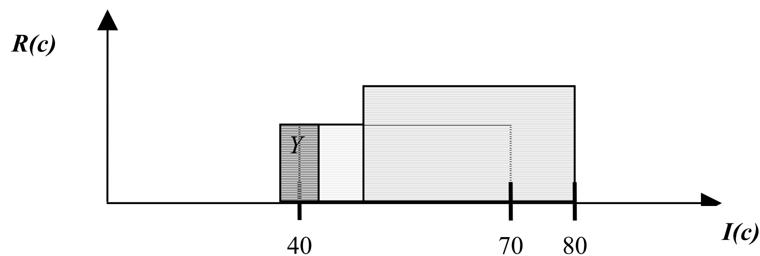


Figure 10-11. Verification only necessary for a limited area.

10.2.3.2 Temporal analysis

Whenever a new component is to be used in a real-time system, the temporal behavior of the system must be re-verified, i.e. we need to re-verify that all tasks are schedulable. In order to schedule the system the components have to be decomposed into their smallest entities, i.e., their tasks. Thus, the level of abstraction provided by the component concept is violated.

In this section we will discuss the impact of changing some of the temporal attributes of a component when reusing it. The following situations are dealt with:

- The same infrastructure and the same input-output domain
- The same input-output domain but a different infrastructure

A change in the temporal domain is always initiated by the system in which the component will be used. For instance, if the new system executes on different hardware, the execution times might vary (a faster or slower CPU). The changes can also originate from the requirements of the new system, for example that the component has to run with a higher frequency than originally designed for.

The same infrastructure and input-output domain

In this case, the infrastructure is exactly the same in the new environment as the one previously used. Consequently, we have an exact match between the sets of services provided (denoted as *S* below). The services required by the design task model for the component have been satisfied earlier which implicates that they still are satisfied in the new environment. Thus, $S(c) \subseteq S_{old}(infrastructure)$ and $S_{new}(infrastructure) = S_{old}(infrastructure)$ holds, where $S(c)$ is the set of services required by component *c*.

Moreover, the new input-output domain for the new instance of component *c*, $\Pi_{new}(c)$, is completely within the verified range of input-output for the component $\Pi(c)$. That is, the following must hold $\Pi_{new}(c) \subseteq \Pi(c)$.

The only alteration is in one or several of the temporal attributes for the component. Such a change requires the system to undertake a schedulability analysis [5][75][117], where the component is decomposed into its smallest constituents, the tasks. The component can be considered to fit into the new system if the system is schedulable and the relations between the tasks in the component, as well as the tasks in the new system, are not violated.

The same input-output domain but different infrastructure

There exist two different types of infra-structural changes:

- One where the infrastructures are different, but where the infrastructure parts pertaining to the component are the same. That is, if $S(c) \subseteq S_{new}(infrastructure) \cap S_{old}(infrastructure)$, then the necessary services are provided by the new infrastructure. If this is the case, and a correct mapping of the services required by the component to the new infrastructure is performed, we can reuse the component with the same confidence in its reliability as in the original environment.

- One where the infrastructures are different and the infrastructure parts pertaining to the component are non-compliant. That is, $S(c) \setminus S_{new}(infrastructure) \neq \emptyset$.

In the latter case where the new infrastructure does not enable the same implementation of the design task model, a new mapping from the design must be performed. This mapping is a matter of implementing the new infrastructure model using the services provided in the new infrastructure. If this mapping is not possible then we cannot reuse the component at all. If the mapping is possible, we can still argue if this is reuse at all, since major parts of the component must be modified. Here we make a clear distinction between modify and parameterize, where modifications are changes in the source code and parameterizations leave the source code unchanged, but its behavior can be changed dynamically.

As an example, consider a component that has been proven to have a certain reliability in an infrastructure that provides synchronization between tasks using time-wise separation. If now the component is reused in an infrastructure that handles synchronization using semaphores the synchronization strategy has to be changed. Consequently, we cannot assume anything about the reliability regarding synchronization failures. We have to assume weaker failure semantics since the preconditions for which the reliability estimates regarding synchronization failures are no longer valid. That is, we cannot guarantee anything regarding synchronization failures when reusing the component.

Figure 10-12 illustrates the different reliabilities for the fault hypotheses (assumptions of failure semantics as introduced in section 3.2), 1 through 5 for component *c*.

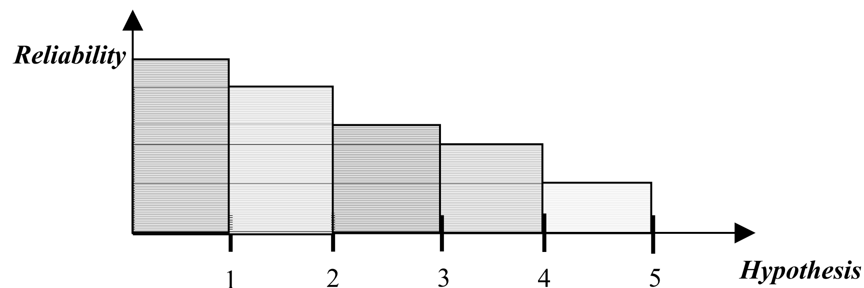


Figure 10-12. The reliability/fault hypothesis before reuse in a new infrastructure.

If the assumptions made for the reliability measure of fault hypothesis 3 is changed, the reliability is inherited from fault hypothesis 2 (see *Figure 10-13*), and we cannot say anything about the reliability regarding more severe failure semantics.

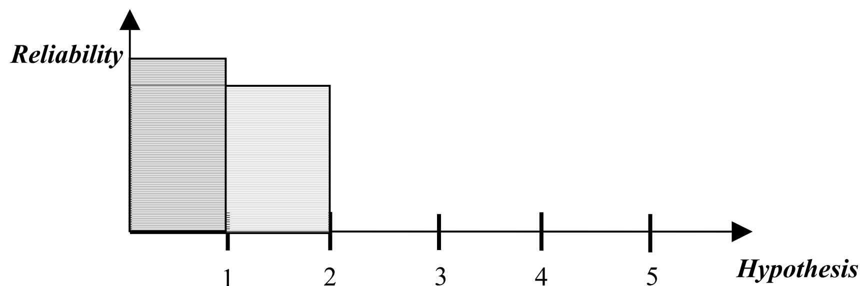


Figure 10-13. The new reliability/fault hypothesis after reuse in a new infrastructre.

Just as in the previous case where the infrastructure was not changed, the schedulability analysis must be performed all over again, ensuring that no temporal constraints are violated in the component as well in the rest of the system.

A measure of the reusability of components

For every new environment in which a component is successfully reused, the usability increases. That is, if a component has been proven in practice to work in many different environments it is highly reusable. *Figure 10-14* illustrates successful reuses of a component, with respect to different period times and the achieved reliability (*R*). The greater the number of period times covered by the diagram, the more reusable the component is.

However, can one argue that a component, which has been reused in a lot of different environments but where every reuse resulted in a low reliability, is reusable? Probably not. Consequently, the reusability is a combination of the number of environments where the component has been used, and the success of every reuse.

A diagram like the one illustrated in *Figure 10-14* can be generated for every type of attribute in the component contract. For instance, the different types of real-time operating systems for which the component has been reused. In this case, the distribution is also a measurement of the portability for the component.

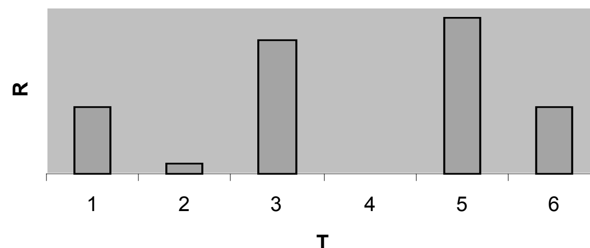


Figure 10-14. Distribution of period times for which the component has been reused, with reliability, R.

10.2.4 Summary

In this section of future work we have presented a novel framework for arguments of reuse and re-verification of components in safety-critical real-time systems. The arguments for reuse of software (components) are usually arguments for rapid prototyping (reusing code), arguments for outsourcing, and arguments for higher reliability. In the latter case it is assumed that verification of the components can be eliminated and that the reliability of the component can be "inherited" from previous uses. Expensive and catastrophic experiences have however shown that it is not so simple, e.g., Ariane 5, and Therac 25 [43][62][66]. In this framework, we formally described component contracts in terms of temporal constraints given in the design phase (the design task model) and the temporal attributes available in the implementation (the infrastructure). Furthermore, the input-output domain for a component is specified in the contract. By relating the input-output domain, fault hypotheses, probabilistic reliability levels and the temporal behavior of the component, we can deem if a component

can be reused without re-verification or not. Of great significance is that we can deem how much and which subsets, of say input-output domains, that need additional *testing* based on reliability requirements in the environment in which the reuse is intended. Faced with complete re-verification the possibility of decreasing the testing effort is attractive, and essential for safety-critical real-time systems.

11 REFERENCES

[1] Abdalla-Ghaly A. A. and Chan a. B. L. P. Y. *Evaluation of competing reliability predictions.* IEEE Transactions on Software Engineering, pp. 950-967, 1986.

[2] Alexander C. *The Timeless Way of Building*. Oxford University Press, 1979.

[3] Alur R. and Dill D. *A theory of timed automata.* Theoretical Computer Science vol. 126 pp. 183-235, 1994.

[4] Audsley N. C., Burns A., Davis R. I., Tindell K. W. *Fixed Priority Pre-emptive Scheduling: A Historical Perspective.* Real-Time Systems journal, Vol.8(2/3), March/May, Kluwer A.P., 1995.

[5] Audsley N. C., Burns A., Richardson M.F., and Wellings A.J. *Hard Real-Time Scheduling: The Deadline Monotonic Approach.* Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software, pp. 127-132, Atlanta, Georgia, May, 1991

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

[7] Beugnard, Antoine, Jézéquel, Jean-Marc, Plouzeau, Noël and Watkins, Damien. *Making Component Contracts Aware*. IEEE Computer July 1999, pp. 37-45.

[8] Brantley W.C., MeAuliffe K.P. and Ngo T.A. *RP3 performance monitoring hardware*. In M. Simmons, R. Koskela, and I. Bucher, eds. *Instrumentation for Future Parallel Computing Systems*, pp. 35-45. Addison-Wesley, Reading, MA, 1989.

[9] Brooks F. P. *No silver bullet – essence and accidents of software engineering*. In information processing 1986, the proceedings of the IFIP 10th world computing conference. H-J Kugler ed. Amsterdam: Elsevier Science, 1986, pp. 1069-1076.

[10] Butler, R.W. and Finelli, G.B. *The infeasibility of quantifying the reliability of life-critical real-time software.* IEEE Transactions on Software Engineering, (19): 3-12, January, 1993.

[11] Calvez J.P., and Pasquier O. *Performance Monitoring and Assessment of Embedded HW/SW Systems.* Design Automation for Embedded Systems journal, 3:5-22, Kluwer A.P., 1998.

[12] Chandy K. M. and Lamport L. *Distributed snapshots: Determining global states of distributed systems*. ACM Trans. On Computing Systems, 3(1):63-75, February 1985.

[13] Chen J. and Burns A. *Asynchronous Data Sharing in Muliprocessor Real-Time Systems Using Process Consensus*. 10th Euromicro Workshop on Real-Time Systems, June 1998,

[14] Chodrow S.E, Jahanian F., and Donner M. *Run-time monitoring of real-time systems*. In Proc. of IEEE 12th Real-Time Systems Symposium, San Antonio, TX, pp. 74-83, December 1991.

[15] Ciupke O., Schmidt R. *Components as Context-independent Units of Softwar*e. In Proc. of ECOOP, 1996.

[16] Clark G. and Powell A. *Experiences with Sharing a Common Measurement Philosophy*. In Proceedings International Conference on Systems Engineering (INCOSE'99), Brighton, UK, 1999.

[17] Clarke S.J. and McDermid JA. *Software fault trees and weakest preconditions: a comparison and analysis.* Software Engineering Journal. 8(4):225-236, 1993.

[18] D'Souza D. F and Wills A. C. *Objects, Components and Frameworks with UML. The Catalysis Approach*. Addison-Wesley, 1999.

[19] Dahl O.J, Dijskstra E.W., and Hoare C.A.R. *Structured Programming*. Academic Press, 1972.

[20] DeMarco T. *Structured Analysis and System Specification*. Yourdon Press 1978. ISBN 0-917072-07

[21] DeMillo R. A., McCracken W.M., Martin R.J., and Passafiume J.F. *Software Testing and Evaluation.* Benjamin/Cummings Publications. Co., 1987.

[22] Dodd P. S., Ravishankar C. V. *Monitoring and debugging distributed real-time programs*. Software-practice and experience. Vol. 22(10), pp. 863-877, October 1992.

[23] E. Loyd and W. Tye. *Systematic Safety: Safety Assessment of Aircraft Systems*. Civil Aviation Authority, London, England, 1982. Reprinted 1992.

[24] Ellis A. *Achieving Safety in Complex Control Systems*. Proceedings of the Safety-Critical Systems Symposium. pp. 2-14. Brighton, England, 1995. Springr-Verlag. ISBN 3-540-19922-5

[25] Ellis W. J., Hilliard R.F., Poon P.T., Rayford D., Saunders T. F., Sherlund B., Wade R. L., Toward a Recommended Practice for Architectural Description, Proceedings 2nd IEEE International Conference on Engineering of Complex Commuter Systems, Montreal, Quebec, Canada, October, 1996.

[26] Eriksson C., Mäki-Turja J., Post K., Gustafsson M., Gustafsson J., Sandström K., and Brorsson E. *An Overview of RTT: A design Framework for Real-Time Systems.* Journal of Parallel and Distributed Computing, vol. 36, pp. 66-80, Oct. 1996.

[27] Eriksson C., Thane H. and Gustafsson M. *A Communication Protocol for Hard and Soft Real-Time Systems.* In the proceedings of the 8th Euromicro Real-Time Workshop, L'Aquila Italy, June, 1996.

[28] Ferrari D. *Consideration on the insularity of performance perturbations.* IEEE Trans. Software Engineering, SE-16(6):678-683, June, 1986.

[29] Fidge, C. *Fundamentals of distributed system observation*. IEEE Software, (13):77 – 83, November, 1996.

[30] G J. Myers. *The Art of Software Testing*. John Wiley and Sons. New York 1979.

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

[32] Gamma E, Helm R., Johnson R., and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[33] Glass R. L. *Real-time: The "lost world" of software debugging and testing*. Communications of the ACM, 23(5):264-271,May 1980.

[34] Gorlick M. M. *The flight recorder: An architectural aid for system monitoring*. In Proc. of ACM/ONR Workshop on Parallel and Distributed Debugging, Santa Cruz, CA, pp. 175-183, May 1991.

[35] Graham R. L. *Bounds on Multiprocessing Timing Anomolies*. SIAM journal of Applied Mathematics, 17(2), March, 1969.

[36] Haban D. and Wybranietz D. *A Hyrbid monitor for behavior and performance analysis of distributed systems.* IEEE Trans. Software Engineering, 16(2):197-211, February, 1990.

[37] Hamlet R. G. *Probable Correctness Theory*. Information processing letters 25, pp. 17-25, 1987.

[38] Hatton L.. *Unexpected (and sometimes unpleasant) Lessons from Data in Real Software Systems*. 12th Annual CSR Workshop, Bruges 12-15 September 1995. Proceedings, pp. 251-259. Springer. ISBN 3-540-76034-2.

[39] Helm R., Holland I. and Gangopadhyay D. *Contracts: Specifying Behavioral Compositions in Object Oriented Systems.* In Proc. of the Conference on Object Oriented Programming: Systems, Languages and Application, 1990.

[40] Hetzel B.. *The Complete Guide to Software Testing*. 2nd edition. QED Information Sciences, 1988.

[41] Holland I. *Specifying reusable components using contracts.* In Proc. of ECOOP, 1992.

[42] Hwang G.H, Tai K.C and Huang T.L. *Reachability Testing: An Approach to Testing Concurrent Software.* Int. Journal of Software Engineering and Knowledge Engineering, vol. 5(4):493-510, 1995.

[43] Inquiry Board. ARIANE 5 – Flight 50l Failure. Inquiry Board report, http://wxtnu.inria.fr/actualirdsfra.htrnl (July 1996), l8 p.

[44] Jelinski Z. and Moranda P. *Software reliability research*. In Statistical Computer Performance Evaluation (Frieberger W., ed.). New York: Academic Press.

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

[46] Joyce J., Lomow G., Slind K., and Unger B. *Monitoring distributed systems*. ACM Trans. On Computer Systems, 5(2):121-150, May 1987.

[47] Keiller P. A. and Miller D. R. *On the use and the performance of software reliability growth models*. Reliability Engineering and System Safety, pp. 95-117, 1991.

[48] Knight J. C. and Leveson N. G. *An experimental evaluation of the assumptions of independence in multiversion programming.* IEEE Transactions on Software Engineering, vol. SE-12, pp. 96-109, Jan. 1986.

[49] Knight J. C. and N. G. Leveson. *A reply to the criticism of the Knight and Leveson experiment*. ACM SIGSOFT Software engineering Notes, 15, p. 25-35, January 1990.

[50] Kopetz H. and Grünsteidl H. *TTP - A Protocol for Fault-Tolerant Real-Time Systems*. IEEE Computer, January, 1994.

[51] Kopetz H. and Ochsenreiter W. *Clock Synchronisation in Distributed Real-Time Systems*. IEEE Trans. Computers, 36(8):933-940, Aug. 1987.

[52] Kopetz H. and Reisinger J. *The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem*. In Proceedings of he 14th Real-Time Systems Symposium, pp. 131-137, 1993.

[53] Kopetz H. *Sparse time versus dense time in distributed real-time systems*. In the proceedings of the 12th International Conference on Distributed Computing Systems, pp. 460-467, 1992.

[54] Kopetz H., Damm A., Koza Ch., Mulazzani M., Schwabl W., Senft Ch., and Zainlinger R.. *Distributed Fault-Tolerant Real-Time Systems: The MARS Approach*. IEEE Micro, (9):25-40, 1989.

[55] Kopetz H.. *Event-Triggered versus Time-Triggered Real-Time Systems*. Lecture Notes in Computer Science, vol. 563, Springer Verlag, Berlin, 1991.

[56] Kopetz, H. and Kim, K. *Real-time temporal uncertainties in interactions among real-time objects.* Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems, Huntsville, AL, 1990.

[57] L Barroca and J McDermid. Formal Methods: Use and Relevance for Development of Safety-Critical Systems. The Computer Journal, Vol. 35, No. 6, 1992.

[58] Lamport L. *Time, clock, and the ordering of events in a distributed systems*. Comm. Of ACM, (21):558-565: July 1978.

[59] Laprie J.C. *Dependability: Basic Concepts and Associated Terminology*. Dependable Computing and Fault-Tolerant Systems, vol. 5, Springer Verlag, 1992.

[60] Larsen K. G., Pettersson P. and Yi W. *Uppaal in a Nutshell*. In Springer International Journal of Software Tools for Technology Transfer 1(1+2), 1997.

[61] Lauterbach emulators. Lauterbach GmbH Germany. http://www.lauterbach.com/.

[62] Le Lann, G. *An analysis of the Ariane 5 flight 501 failure - a system engineering perspective.* Proceedings., International Conference and Workshop on Engineering of Computer-Based Systems, pp. 339 - 346, 1997.

[63] LeBlanc T. J. and Mellor-Crummey J. M. *Debugging parallel programs with instant replay*. IEEE Trans. on Computers, C-36(4):471-482, April 1987.

[64] LeDoux C.H., and Parker D.S. *Saving Traces for Ada Debugging*. In the proceedings of Ada int. conf. ACM, Cambridge University press, pp. 97-108, 1985.

[65] Lee J.Y., Kang K.C.,Kim G.J., Kim H.J. Form the missing piece in effective real-time system specification and simulation. In proc. IEEE 4th Real-Time Technology and Applications Symposium, pp.155 – 164, June 1998.

[66] Leveson N. and Turner C. *An investigation of the Therac-25 accidents*. IEEE Computer, 26(7):18-41, July 1993.

[67] Leveson N. G. *Safeware - System, Safety and Computers*. Addison Wesley 1995. ISBN 0-201-11972-2.

[68] Leveson N. G. *Software safety: What, why and How*. ACM Computing surveys, 18(2),1986.

[69] Littlewood B. and Keiller P. A. *Adaptive software reliability modeling*. In 14th International Symposium on Fault-Tolerant Computing, pp. 108-113, IEEE Computer Society Press, 1984.

[70] Littlewood B. and Strigini L. Validation of Ultrahigh Dependability for Software-based Systems. Com. ACM, 11(36):69-80, November 1993.

[71] Littlewood B. *Stochastic reliability-growth: A model for fault-removal in computer programs and hardware designs.*' IEEE Transactions on Reliability, pp. 313-320, 1981.

[72] Littlewood B.and Verrall. *A Bayesian Reliability Growth Model For Computer Software*. Journal of the Royal Statistical Society, Series C, No. 22, p 332-346, 1973

[73] Liu A.C. and Parthasarathi R. *Hardware monitoring of a multiprocessor systems.* IEEE Micro, pp. 44-51, October 1989.

[74] Lozzerini B., Prete C. A., and Lopriore L. *A programmable debugging aid for real-time software development*. IEEE Micro, 6(3):34-42, June 1986.

[75] Lui C. L. and Layland J. W.. *Scheduling Algorithms for multiprogramming in a hard real-time environment*. Journal of the ACM 20(1), 1973.

[76] Lutz R. R.. *Analyzing software requirements errors in safety-critical, embedded systems*. In software requirements conference, IEEE, January 1992.

[77] Malony A. D., Reed D. A., and Wijshoff H. A. G. *Performance measurement intrusion and perturbation analysis*. IEEE Trans. on Parallel and Distributed Systems 3(4):433-450, July 1992.

[78] McDowell C.E. and Hembold D.P. *Debugging concurrent programs*. ACM Computing Surveys, 21(4), pp. 593-622, December 1989.

[79] Mellor-Crummey J. M. and LeBlanc T. J. *A software instruction counter*. In Proc. of 3d International Conference on Architectural Support for Programming Languages and Operating Systerns, Boston, pp. 78-86, April 1989

[80] Miller B.P., Macrander C., and Sechrest S. *A distributed programs monitor for Berkeley UNIX*. Software Practice and Experience, 16(2):183-200, February 1986.

[81] Mink K., Carpenter R., Nacht G., and Roberts J. *Multiprocessor performance measurement instrumentation*. IEEE Computer, 23(9):63-75, September 1990.

[82] Mäki-Turja J., Fohler G. and Sandström K. *Towards Efficient Analysis of Interrupts in Real-Time Systems*. In proceedings of the 11th EUROMICRO Conference on Real-Time Systems, York, England, May 1999.

[83] Netzer R.H.B. and Xu Y. *Replaying Distributed Programs Without Message Logging.* In proc. 6th IEEE Int. Symposium on High Performance Distributed Computing. Pp. 137-147. August 1997.

[84] Neuman P G. *Computer Related Risks*. ACM Press, Adison-Wesley, 1995. ISBN 0-201-55805-x.

[85] Parnas D.L., van Schouwen J., and Kwan S.P. *Evaluation of Safety-Critical Software*. Communication of the ACM, 6(33):636-648, June 1990.

[86] Plattner B. *Real-time execution monitoring.* IEEE Trans. Software Engineering, 10(6), pp. 756-764, Nov., 1984.

[87] Poledna S. *Replica Determinism in Distributed Real-Time Systems: A Brief Survey*. Real-Times systems Journal, Kluwer A.P., (6):289-316, 1994.

[88] Powell D. *Failure Mode Assumptions and Assumption Coverage: In Proc. 22nd International Symposium on Fault-Tolerant Computing.* IEEE Computer Society Press, pp.386-395, July, 1992.

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

[90] Raju S. C. V., Rajkumar R., and Jahanian F. *Monitoring timing constraints in distributed real-time systems*. In Proc. of IEEE 13th Real-Time Systems Symposium, Phoenix, AZ, pp. 57-67, December 1992.

[91] Reilly M. *Instrumentation for application performance tuning: The M3l system*. In Simmons M., Koskela R., and Bucher I., eds. Instrumentation for Future Parallel Computing Systems, pp. 143-158. Addison-Wesley, Reading, MA, 1989.

[92] Rothermel G. and Harrold M.J. *Analyzing regression test selection techniques*. IEEE trans. Software Engineering. 8(22):529-551. August 1996.

[93] Rushby J. *Formal methods and their Role in the Certification of Critical Systems.* Proc. 12th Annual Center for Software Reliability Workshop, Bruges 12-15 Sept. 1995, pp. 2-42. Springer Verlag. ISBN 3-540-76034-2.

[94] Rushby J., *Formal Specification and Verification for Critical systems: Tools, Achievements, and prospects*. Advances in Ultra-Dependable Distributed Systems. IEEE Computer Society Press. 1995. ISBN 0-8186-6287-5.

[95] Sandström K., Eriksson C., and Fohler G. *Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System*. In proceedings of the 5th Int. Conference on Real-Time Computing Systems and Applications (RTCSA'98). October 1998, Japan.

[96] Schütz W. *Fundamental Issues in Testing Distributed Real-Time Systems*. Real-Time Systems journal, vol. 7(2): 129-157, Kluwer A.P., 1994.

[97] Schütz W. *Real-Time Simulation in the Distributed Real-Time System MARS*. In proc. European Simulation Multiconference 1990, Erlangen, BRD, June 1990.

[98] Shaw M., Clements P., *A field guide to boxology: preliminary classification of architectural styles.* In proc. 21st Annual International Computer Software and Applications Conference, 1997 (COMPSAC '97). ISBN: 0-8186-8105-5.

[99] Shimeall T. J. and Leveson N. G. *An empirical comparison of software fault-tolerance and fault elimination.* IEEE Transactions on Software Engineering, pp. 173-183, Feb. 1991.

[100] Shin K. G. *HARTS: A distributed real-time architecture.* IEEE Computer, 24(5), pp. 25-35, May, 1991.

[101] Sifakis J. and Yovine S. *Compositional specification of timed systems.* In Proc. 13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96, Grenoble, France, Lecture Notes in Computer Science 1046, pp. 347-359. Springer Verlag, February 1996.

[102] Sommerville I. *Software Engineering.* Addison-Wesley, 1992. ISBN 0-201-56529-3.

[103] Szyperski C. *Component Software - Beyond Object-Oriented Programmin*g. Addison-Wesley, 1997.

[104] Tai K.C, Carver R.H., and Obaid E.E. *Debugging concurrent Ada programs by deterministic execution*. IEEE transactions on software engineering. Vol. 17(1), pp. 45-63, January 1991.

[105] Thane H. and Hansson H. *Handling Interrupts in Testing of Distributed Real-Time Systems*. In proc. Real-Time Computing Systems and Applications conference (RTCSA'99), Hong Kong, December, 1999.

[106] Thane H. and Hansson H. *Towards Systematic Testing of Distributed Real-Time Systems*. Proc. 20th IEEE Real-Time Systems Symposium, Phoenix, Arizona, December 1999.

[107] Thane H. and Hansson H. *Using Deterministic Replay for Debugging of Distributed Real-Time Systems*. In proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS'00), Stockholm, June 2000.

[108] Thane H. *Asterix the T-REX among real-time kernels. Timely, reliable, efficient and extraordinary.* Technical report in preparation, Mälardalen Real-Time Research Centre, Dept. Computer Engineering, Mälardalen University, *2000*

[109] Thane H. *Design for Deterministic Monitoring of Distributed Real-Time Systems.* Technical report, Mälardalen Real-Time Research Centre, Dept. Computer Engineering, Mälardalen University, 1999.

[110] Tindell K. W., Burns A., and Wellings A.J. *Analysis of Hard Real-Time Communications.* Journal of Real-Time Systems, vol. 9(2), pp.147-171, September 1995.

[111] Tokuda H., Kotera M., and Mercer C.W. *A Real-Time Monitor for a Distributed Real-Time Operating System*. In proc. of ACM Workshop on Parallel and Distributed Debugging, Madison, WI, pp. 68-77, May, 1988.

[112] Tsai J.P., Bi Y.-D., Yang S., and Smith R.. *Distributed Real-Time System: Monitoring, Visualization, Debugging, and Analysis.* Wiley-Interscience, 1996. ISBN 0-471-16007-5.

[113] Tsai J.P., Fang K.-Y., Chen H.-Y., and Bi Y.-D. *A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging.* IEEE Trans. on Software Eng. vol. 16, pp. 897 - 916, 1990.

[114] Voas J M and Freidman. *Software Assessment: Reliability, Safety, testability*. Wiley Interscience, 1995. ISBN 0-471-01009-x.

[115] Voas J. and Miller. *Designing Programs that are Less Likely to Hide Faults*. Journal of Systems Software, 20:93-100, 1993,

[116] Voas J. *PIE: A dynamic Failure Based Technique*. IEEE Transactions on Software Engineering, vol. 18(8), Aug. 1992.

[117] Xu J. and Parnas D. *Scheduling processes with release times, deadlines, precedence, and exclusion, relations*. IEEE Trans. on Software Eng. 16(3):360-369, 1990.

[118] Yang R-D and Chung C-G. *Path analysis testing of concurrent programs*. Information and software technology. vol. 34(1), January 1992.