Mälardalen University Press Dissertations
No. 55

STRUCTURAL SYSTEM-LEVEL TESTING OF EMBEDDED REAL-TIME SYSTEMS

Daniel Sundmark

2008

MÄLARDALEN UNIVERSITY SWEDEN

School of Innovation, Design and Engineering
STRUCTURAL SYSTEM-LEVEL TESTING OF EMBEDDED REAL-TIME SYSTEMS

Daniel Sundmark

Akademisk avhandling

som för avläggande av Teknologie doktorsexamen i Datavetenskap vid Akademin för innovation, design och teknik kommer att offentligen försvaras fredagen, 15:e februari, 2008, 13.30 i Beta, Mälardalens Högskola, Gurksaltargatan 2, Västerås.

Fakultetsopponent: Professor Insup Lee, University of Pennsylvania, USA.
Abstract

People make mistakes. Software engineers are no exception to this fact. When software engineers make mistakes, these manifest in the form of buggy software - a major problem in today's industry. The existence of bugs is commonly detected using testing, the process of executing the software and checking if its behaviour complies with the specification. As limitations in time make testing of the entire software behaviour impracticable, testers need to make informed decisions on how to test the software to detect as many bugs as possible.

In the realm of real-time systems (RTSs), software testing is made more difficult by non-deterministic factors such as interaction with the surrounding environment, (pseudo)parallelism, and timing requirements. Furthermore, RTS testing suffers from behaviour-altering perturbation from the instrumentation inserted in the system to keep track of test progress (i.e., probe effects).

In our work, we analyse the main test criteria used for traditional software testing in order to see which of these scale to, and assist in, system-level testing of multi-tasking RTSs. We focus on one of these criteria, the all definition-use paths coverage criterion, as it highlights a central aspect of non-deterministic task interaction, and investigate what is specifically required for applying this criterion to testing of multi-tasking RTSs. Further, we examine the possibility of using execution replay for probe effect-free test monitoring. We evaluate this approach in real industrial settings by means of case studies.

The contributions of this thesis are twofold: First, the use of structural test criteria in RTS system-level testing is facilitated by two different analysis methods. Second, the probe effect is handled by recording non-deterministic events during run-time, and by using this recording to create a monitorable deterministic replica of the first execution. By these contributions, this thesis shows how the non-determinism of multi-tasking RTSs can be handled during system-level testing.

ISSN 1651-4238

ISBN 978-91-85485-70-3
We barely remember
Who or what came before
This precious moment
Choosing to be here
Right now
Hold on, stay inside

This body
Holding me
Reminding me that I am not alone in
This body
Makes me feel eternal
All this pain is an illusion

- Maynard James Keenan
To my family
– old and new
Acknowledgements

I didn’t plan to become a PhD student. As a matter of fact, for the better part of my youth, my main interest in computers was enjoying a couple of hours with good friends in front of a compelling adventure game, like The Bard’s Tale or The Last Ninja. As we threw a smoke grenade, conjured a mighty lightning bolt or slained a few Hydras, I never pondered on the amount of skill and dedication that it must have taken to design, develop, and test a commercially successful computer game. However, as life goes, a random set of events (I actually did not realize that my taking a masters in Information Technology at Uppsala University would require me to learn how to write a program), a deep fascination for mathematics, and a heart-felt respect for the competence, dedication and independence of great scientists somehow landed me a PhD candidate position at Mälardalen University.

One of the best things of writing a thesis is that you are allowed to fill an entire section (probably the most read one, too) with nice words about people whose support you highly value and appreciate, without you actually feeling awkward about it. So, here it goes. First, I owe a major thank you to my supervisors Henrik Thane, Andreas Ermedahl, and Hans Hansson. Henrik, you are a brilliant visionary, and unfortunately also eloquent to the degree that it often takes me several hours to realize that I don’t understand what you mean. Andreas, you are supportive, devoted and enthusiastic, and I truly believe that this thesis wouldn’t have been finished if it was not for your efforts. Hans, you are pragmatic, outspoken, and amazingly easy-going. Had it not been for you being so polite, I would have thought of you as the Gordon Ramsay of real-time system research, clearly competent within your field, but even more competent in facilitating activities in your field for others. You all make up a splendid mix!

Also, although not given formal credit for this, I feel that Jukka Mäki-Turja, Mikael Nolin, and Sasikumar Punnekkat deserve a recognition for the de facto
co-supervision they have provided during strategic stages through my studies. Sometimes, a pat on the back or an encouraging word is worth much more than twenty pages of detailed comments on the latest article.

Next, I don’t think there are many department colleagues I haven’t bothered with various questions regarding countless topics, ranging from wedding table seating, publication strategy and very formal takes on program analysis. Thanks for being friendly and supportive all along! I however feel that some of you deserve a special mention. Anders Pettersson, my roommate, PhD student colleague, and primary recipient for general everyday whining. If it wasn’t for you, I probably wouldn’t have finished in a year from now. I would like to thank Thomas Nolte for being a true inspiration and a good friend at best, and an annoying wiseacre at worst, constantly quering me of which decade i plan to present my thesis in. Well, Thomas, the moment is here! Furthermore, Markus Lindgren, thanks a lot for providing that unique atmosphere of support, an uncompromised professionalism, and an infinite series of totally pointless MSN winks. Also, I would like to thank Anders Möller for being a great traveling companion and for teaching me about the boredom of long-distance flight clear air turbulence.

In addition to the above, there are many department colleagues whose path of everyday work periodically seem to cross my own, and whose companionship I highly appreciate. These include (but are not restricted to) Lars Asplund, Mats Björkman, Per Branger, Stefan Bygde, Jan Carlson, Ivica Crnkovic, Radu Dobrin, Gordana Dodig-Crnkovic, Harriet Ekwall, Sigrid Eldh, Cecilia Fernström, Johan Fredriksson, Joakim Fröberg, Jan Gustafsson, Ewa Hansen, Andreas Hjertström, Joel Huselius, Damir Iovic, Helena Jerregård, Johan Kraft, Rikard Land, Stig Larsson, Jörgen Lidholm, Maria Lindén, Björn Lisper, Åsa Lundqvist, Jonas Neander, Christer Norström, Dag Nyström, Hongyu Pei-Breivold, Paul Pettersson, Ingemar Reyier, Larisa Rizvanovic, Christer Sandberg, Anders Wall, Peter Wallin, Monica Wasell, and Gunnar Widforss.

Not surprisingly, I was not able to keep my nose out of the inner workings of the University organisation. In this context, I would like to thank my fellow PhD student council members, and my colleagues from the three years I spent on the Faculty Board of Natural Sciences and Technology (naturally including the members of the UFO staff, who are doing a massive work “backstage”).

I still owe a major thank you to my main lab-partner during my years as an undergraduate in Uppsala – Mattias Sjögren. Besides from being a good comrade, you gave me the first impression of what it means to be a skilled programmer.

It also seems impossible to thank anyone without thanking the notoriously
supportive team of Grabbarna Grus – the foie gras or white truffle version of childhood friends: Anders, David, Kristoffer, Magnus, and Robert. I actually totalled over 7000 mails from you during the course of my PhD studies, averaging nearly 3.5 mails per day (including weekends and vacations). Thank you also to Ammi, Helena, Malin and Petra for all amazing parties and travels. We’ve had great fun – I accept nothing less than to keep it up!

To my mother, father and sister – I can not enough stress how important your support has been to me. It means the world to have a number of people whose appreciation of you is utterly uncorrelated with the acceptance of your latest article. This naturally also goes for Niclas, Malin, Janne, Gittan, Micke, Farmor, Barbro, Sigurd, Helene, Calle, Victoria, and Lovisa. Thank you!

Finally, Kristina, my beautiful wife. Thank you for being there, for being you, and for getting the daily routines working when I was so busy I didn’t even notice you did. I love you. You make me proud!

This work has been supported by the KK Foundation (KKS) and Mälardalen University.

Tack ska ni alla ha!

Västerås, September 2007
Karl Daniel Sundmark
Contents

Contents ix
List of Publications xii

1 Introduction 1
  1.1 Background ................................. 3
    1.1.1 Software Testing .......................... 3
    1.1.2 Concurrent Real-Time System Testing ...... 9
  1.2 System Model ................................. 14
  1.3 Problem Formulation and Hypothesis .......... 15
  1.4 Contributions ................................ 16
  1.5 Thesis Outline ............................... 17

2 Structural Test Criteria for System-Level RTS Testing 19
  2.1 Structural Test Criteria ........................ 19
    2.1.1 Control Flow Criteria ...................... 22
    2.1.2 Data Flow Criteria .......................... 25
    2.1.3 Useful and Non-Useful Criteria ............ 27
    2.1.4 Structural Coverage Criteria Summary ....... 29
    2.1.5 A Note on Feasibility ..................... 29
  2.2 Structural Test Items .......................... 30
    2.2.1 Sets of Test Items ......................... 30
  2.3 Summary ..................................... 34

3 Deriving DU-paths for RTS System-Level Testing 37
  3.1 Introduction .................................. 37
    3.1.1 General Analysis Properties ................. 38
  3.2 Approach 1: Deriving DU-paths using UPPAAL and CO\text{\&ER} . 41
3.2.1 Preliminaries ........................................... 41
3.2.2 RTS Control Flow Modeling in UPPAAL ............ 43
3.2.3 Automatic System Model Generation using SWEET . 45
3.2.4 Deriving $\text{Model}_{DU}(M_S)$ using $\text{COER}$ ....... 46
3.3 Approach 2: DU-path derivation using EOGs .......... 47
3.3.1 Preliminaries .......................................... 47
3.3.2 Shared Variable DU Analysis ......................... 49
3.3.3 Deriving $\text{Model}_{DU}(M_S)$ using EOGs .......... 53
3.4 Discussion .................................................. 53

4 Test Item Monitoring Using Deterministic Replay 61
4.1 Introduction ................................................. 63
4.1.1 The Probe Effect ....................................... 64
4.1.2 Deterministic Replay ................................. 65
4.2 Deterministic Replay DU-path Monitoring ............ 66
4.3 Reproducibility .............................................. 68
4.3.1 Context Issues ......................................... 68
4.3.2 Ordering Issues and Concurrency .................... 69
4.3.3 Timing Issues .......................................... 70
4.3.4 Reproducibility - Summary and Problem Statement . 72
4.4 The Time Machine ......................................... 74
4.4.1 System Model Refinements ........................... 74
4.4.2 The Mechanisms of the Time Machine ............... 74
4.4.3 Deterministic Replay Summary ....................... 80
4.5 Discussion ................................................... 81

5 System-Level DU-path Coverage – An Example 83
5.1 The Process ............................................... 83
5.2 The Example System ...................................... 84
5.2.1 Example System Structure ............................ 85
5.2.2 System Replay Instrumentation ..................... 86
5.3 DU-path Analysis ......................................... 87
5.3.1 Task-Level Analysis ................................... 88
5.3.2 Deriving DU-paths using UPPAAL and $\text{COER}$ .... 89
5.3.3 Deriving DU-paths using EOGs ...................... 90
5.4 System Testing ............................................. 90
5.4.1 Initial Testing ......................................... 92
5.4.2 Replaying Test Cases ................................. 92
5.5 Summary ..................................................... 93
xii Contents

8.2.2 System model .............................................. 139
8.2.3 Test Item Derivation Method Applicability ........ 140
8.2.4 Handling Infeasible Test Items ...................... 140
8.2.5 Approximative Checksums .............................. 141
8.3 Future Work .................................................. 141
8.3.1 Structural System-Level Test Case Generation .... 142
8.3.2 Checksum Unique Marker Precision ................. 142
8.3.3 Facilitating Sporadic Tasks through Servers ....... 142
8.3.4 Expressing Flow Facts in UPPAAL ................. 143
8.3.5 Regression Testing ................................. 143

A Timed Automata Test Item Derivation 147
A.1 Timed Control Flow Modeling using UPPAAL ......... 147
  A.1.1 Clocks ...................................................... 148
  A.1.2 Task CFG modeling .................................... 149
  A.1.3 Modeling of Task Execution Control ............ 153
  A.1.4 FPS Scheduler and Task Switch Modeling ...... 154

B The DUALYSIS Algorithm 159

C Reproduction of Asynchronous Events 165
  C.1 Context Checksums ...................................... 165
    C.1.1 Execution Context .................................. 166
    C.1.2 Register Checksum .................................. 166
    C.1.3 Stack Checksum .................................... 167
  C.2 Adaptations .............................................. 168
    C.2.1 Instrumentation Jitter ............................ 168
    C.2.2 Partial Stack Checksum ................................ 169
    C.2.3 System call markers ................................ 170

Bibliography 171
List of Publications

These publications have been (co-)authored by the author of this thesis:

Publications Related to This Thesis

My contribution: I am the sole author of this thesis.

My contribution: This paper was a joint effort. I wrote the section discussing the Time Machine.

My contribution: This paper was a joint effort. I wrote the section discussing the Implementation, and parts of the Introduction.

My contribution: I am the main author of this paper, even though the case studies described were joint efforts between all authors.

**My contribution:** I am the sole author of this report.


**My contribution:** I am the sole author of this thesis.


**My contribution:** This paper was written by me and Anders Pettersson, under supervision of Henrik Thane.


**My contribution:** I took part in the discussions and wrote parts of the paper, but Anders Pettersson was the main author.


**My contribution:** I am the main author of this paper, and responsible for all parts of the paper except for the task-level shared variable analysis.

### Other Publications


**My contribution:** This paper was a joint effort, but I was the main author.

My contribution: This paper was written by me and Anders Møller, under supervision of Mikael Nolin.


My contribution: This paper was a joint effort. I was not the main author, but took part in the discussions preceding the paper.


My contribution: This paper was a joint effort. I implemented the replay mechanism and wrote the sections regarding the support for replay, but I was not the main author.


My contribution: This paper was mainly written by Sigrid Eldh in cooperation with Hans Hansson and Sasikumar Punnekkat. I contributed in some discussions and in the finalization of the paper.


My contribution: This paper was a joint effort. I was not the main author, but took part in the discussions preceding the paper. Me and Anders Pettersson wrote the parts on the Asterix kernel.
Chapter 1

Introduction

Software engineering is hard. In fact, it is so hard that the annual cost of software errors is estimated to range from $22.2 to $59.5 billion per year in the US alone [71]. One possible explanation for these high costs could be that software does not follow the laws of physics, as the subjects of traditional engineering disciplines do. Instead, software is discrete in its nature, and discontinuous in its behaviour, making it impossible to interpolate between software test results. For example, in solid mechanics, it is possible to test the strength of, e.g., a metallic cylinder under a specific load, and from the result of this test estimate the cylinder’s capability of withstanding a heavier or lighter load. Hence, a bridge that withstands a load of 20 tons could be assumed to withstand a load of 14, 15 or 17 tons. However, if you bear with us and assume a bridge built of software, a test showing that it withstands 20 tons would not guarantee that it could handle 14, 15 or 17 tons. In fact, theoretically we could construct a software bridge that holds for all other loads than exactly 20.47 pounds - and it will only break for that load if it is applied between 14:00 and 14:15 on a Tuesday afternoon. In addition, if a software bridge would fail to withstand a load, it is very hard to foresee in what fashion it would fail. Sure, it could collapse, but it could also implode, move left or fall up in the sky.

Another explanation to the high expenses related to software engineering could be that they are inherent in its young age as an engineering discipline. Methods and tools for aiding developers in their task of delivering bug-free software are still in an early phase of their development. Traditionally, one of the main countermeasures against poor software quality is testing, i.e., the process of dynamically investigating the software behaviour in order to reveal
the existence of bugs. A fundament of software testing is the impracticability of exhaustive testing. In other words, testing of the entire behavioural space of the software is generally impossible. This is often illustrated by means of a simple example: Consider a small software function taking two 16-bit integers as arguments. Further assume that the function is deterministic (i.e., given the same input, it will always exhibit the same behaviour), and that each execution of the function will last 10 milliseconds (ms). An exhaustive testing, where all possible input combinations of this function are tested, will require at least

\[2^{16} \times 2^{16} \times 10 \text{ ms} \approx 497 \text{ days}\]

In addition, this example is overly simplified. The behaviour of most software programs depend on more aspects than input alone, e.g., time or other programs executed on the same computer. More generally, the following statements can be considered fundaments of software testing:

1. Testing cannot be used to prove absence of bugs, but only to show their presence [24].
2. We face a delicate problem of how to perform testing in order to get the most out of our testing efforts [33, 49, 73, 76].

Statement 1 sets the scene of software testing. In today’s software industry, an effective testing method is not a method that proves the software correct, but one that reveals as many (serious) flaws as possible in the least amount of testing time [109]. It is however Statement 2 that states the core problem of this thesis. As, generally, only a fraction of the overall function behaviour can be tested during the assigned time, there is a need to determine how to perform the testing in such an effective (in this case, failure-detecting) manner as possible.

For this purpose, in this thesis, we present a set of methods for enabling the use of a more structured way of testing software systems. Specifically, given a certain type of software system, and an hypothesis on what types of errors we may encounter, we describe how to:

1. Derive information on what parts of the system should be covered by testing using models of the system under test.
2. Extract run-time information on which of these parts have been tested by a certain set of test cases.
3. Establish measurements on how well-tested the system is.

A more detailed description of the contributions of this thesis is given in the end of this chapter.
1.1 Background

Our research rests on three software engineering assumptions:

1. In practice, no industrially relevant software is free from bugs [98].

2. Bugs have a detrimental effect on the quality of the software in which they reside [71, 110].

3. Low software quality is costly and troublesome [71, 74, 82].

Based on these assumptions, it can be concluded that each method or technique that reduces the number of bugs in software (e.g., testing or formal verification) also increases the quality of the software, and saves resources and trouble for the industry (given that the effort for using the methods and techniques is less than the ordeal of coping with the bugs).

1.1.1 Software Testing

Over the years, numerous testing techniques have been proposed, applied and adopted in industry practices [22, 107]. There is no general way of placing these techniques in any strict order of precedence with respect to adequacy or efficiency, since all techniques are specialized in revealing some types of failures and adapted for some types of systems, but fail when it comes to others (even though studies are being undertaken regarding this problem [27, 42, 108]). All these techniques are, however, similar in the sense that they seek suitable abstractions of the software (e.g., flow graphs) in order to focus on aspects that are critical for the software correctness. Other aspects are disregarded, thereby reducing the behavioral complexity of the abstraction.

Functional and Structural Testing

When investigating the underlying properties used to evaluate the adequacy and thoroughness of testing (i.e., the test criteria), two fundamentally different approaches to testing emerge:

- **Structural**, or white-box, test criteria are expressed in terms of the structure of the software implementation.

- **Functional**, or black-box, test criteria are expressed in terms of the specification of the software.
int empty_program (int argc, char* argv[]) { }

Figure 1.1: An empty program.

As structural test criteria are strictly based on the actual software implementation and different inherent aspects of its structure, these are possible to formulate formally. Examples of structural test criteria include exercising of all instructions, all execution paths, or all variable definition-use paths in the software. However, while structural techniques are highly aware of the actual implementation of the software, they are ignorant in the sense that they have no concept of functionality or semantics. In the extreme case, an empty program (like the one in Figure 1.1) could be considered structurally correct; if a program contains no code, its code could contain no faults.

Test case selection based on functional test criteria is, in the general case, ad-hoc in the sense that it depends on the quality, expressiveness, and the level of abstraction of the specification. Basically, a more detailed and thorough specification will result in a more ambitious and thorough functional test suite. Examples of functional test criteria might be exercising of all use cases, or boundary value testing. Functional techniques, that excel in investigating the intended behaviour of the software, would easily detect the missing functionality in Figure 1.1. However, a functional technique does not consider the inner structure of the software, and is weak in detecting the existence of bugs hidden in code or paths that are not explicitly coupled with a certain functionality of the software.

Structural and functional testing techniques complement each other, since they focus on different aspects of the same software. One might say that the strategy of structural testing techniques is to try to cover aspects of what the software could do, whereas the strategy of functional testing techniques is to try to cover aspects of what the software should do. For example, consider Figure 1.2. Each area ($A$, $B$, and $C$) in the figure represents different subsets of the behaviour of an example program. Let the left circle (made up of sets $A$ and $B$) represent the intended behaviour of the program. Further, let the right circle (made up of sets $B$ and $C$) represent the actual implemented behaviour of the program. Hence, $B$ represents a correct partial implementation of the program, $A$ represents what should be implemented, but is not, and $C$ represents program behaviour that was not intended, but is included in the implementation. The behaviours in the latter set could be unnecessary at best, and failure-prone at worst.
Simplified, functional testing will focus on detecting anomalies belonging in set $A$, and structural testing will focus on detecting anomalies belonging in $C$. If only functional or structural testing could be used, one of these sets would be disregarded.

**Levels of Testing**

In the traditional view of the software engineering process, testing is performed at different levels. Throughout the literature, many such levels are discussed, but the most commonly reappearing levels of testing are *unit*, *integration*, *system* and *acceptance testing* [12, 21, 22, 107].

- **Unit testing** is performed at the “lowest” level of software development, where the smallest units of software are tested in isolation. Such units may be functions, classes or components\(^1\). Unit-level testing typically uses both functional and structural techniques [107].

- **Integration testing** can be performed whenever two or more units are integrated into a system or a subsystem. Specifically, integration testing focuses on finding failures that are caused by interaction between the different units in the (sub)system. Integration-level testing typically uses both functional and structural techniques [107].

- **System testing** focuses on the failures that arise at the highest level of integration [21], where all parts of the system are incorporated and executed on the intended target hardware(s). A system testing test case

\(^{1}\)Note that both unit and integration testing sometimes are referred to as *component testing*, depending on the definition of a component.
is considered correct if its output and behaviour complies with what is stated in the system specification. System-level testing typically uses functional techniques [107].

- **Acceptance testing**, like system testing, is performed on the highest level of integration, where all parts of the system are incorporated and executed on the intended target hardware(s). But, unlike system testing, acceptance testing output and behaviour are not checked against the system specification, but rather against what is actually intended by the customer, or required by the end user. Hence, while system testing aims at providing support that the system has been built correctly according to the system specifications, acceptance testing aims at providing support that the system specifications correctly represent the customer intentions or user needs [107].

In this thesis, we will focus on structural testing on system-level, since we argue that this is a neglected area of software testing, and that the possibilities for discovering certain types of failures on system-level would increase from the addition of a structural perspective. A more detailed motivation for our selection of this focus is given in Section 1.1.2.

**Test Criteria**

A test criterion is a specification for evaluating the test adequacy given by a certain set of test cases. A test criterion determines (1) when to stop testing (when the test criterion is fulfilled), and (2) what to monitor during the execution of test cases (to know when the test criterion is fulfilled) [118].

As stated earlier, this thesis focuses on structural testing rather than on functional testing. Structural test criteria are based on the actual software implementation, or on control flow graphs, i.e., abstract representations of the software implementation describing the possible flow of control when executing the software. In general, a control flow graph (CFG) of a function is derived in two basic steps: First, the function is partitioned into basic blocks (i.e., a “sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end” [2]). Second, the basic blocks are interconnected by directed edges representing conditional or unconditional jumps from one block to another. An example function and its corresponding control flow graph is depicted in Figure 1.3.
1.1 Background

Figure 1.3: The structure and CFG of a small program.

*Control flow criteria* [116, 117] are structural test criteria that are based on, or expressed in terms of, the control flow graph of the system under test. Examples of control flow criteria are:

- The **Statement coverage criterion**, which is fulfilled when each statement in the software code are exercised at least once during testing [116]. This test criterion is equivalent to the criteria of visiting all basic blocks. In Figure 1.3, this would correspond to exercising \(A, B, C, D, E, F\) and \(X\) at least once during testing.

- The **Branch coverage criterion**, which is fulfilled when all edges (branches) in the CFG of the software are taken at least once [116]. This test criterion is stronger than the statement coverage criterion (i.e., a test case set fulfilling the branch coverage criterion also fulfills the statement coverage criterion, while the opposite is not true). In Figure 1.3, a full branch coverage would correspond to taking branches \(A \rightarrow B, A \rightarrow C, B \rightarrow D, C \rightarrow D, D \rightarrow E, D \rightarrow F, E \rightarrow X, F \rightarrow X\) and \(X \rightarrow A\) at least once during testing.

- The **Path coverage criterion**, which is fulfilled when all feasible paths through the code are taken during testing [116]. In Figure 1.3, the number of potential paths is infinite, since the \(X \rightarrow A\) branch could be taken an arbitrary number of times. In cases where a full coverage is infeasible, approximations are often done in order to make the criterion applicable. Examples of paths in the figure are \(A \rightarrow B \rightarrow D \rightarrow F \rightarrow X\) and \(A \rightarrow C \rightarrow D \rightarrow E \rightarrow X \rightarrow A \rightarrow C \rightarrow D \rightarrow E \rightarrow X\).
A control flow graph can also be extended to encompass information of accesses to variables (e.g., at which points in the control flow graph assignments and accesses to variables can be made). Test criteria based on such graphs are commonly referred to as data flow criteria [34, 116]. An example of a data flow criterion is:

- The All-DU-paths coverage criterion, which is fulfilled when all paths \( p \) from the definition of a variable (i.e., an assignment of a value to the variable) to a use of the same variable, such that \( p \) contains no other definitions of that variable, are exercised during testing [116]. For example, given the program in Figure 1.3, and that basic blocks \( C \), \( E \), and \( F \) contain the statements \( x = 2 \), \( y = x \), and \( z = x \) respectively, exercising subpaths \( C \rightarrow D \rightarrow E \) and \( C \rightarrow D \rightarrow F \) would correspond to testing two DU-paths.

**Test Items**

*Test items* are the “atoms” of test criteria. A test criterion is generally formulated such that test adequacy (with respect to that criterion) is attained when all test items are exercised during testing. For example, for the statement coverage criterion, statements are the test items. Test items are also called coverage items.

**Coverage**

*Coverage* is a generic term for a set of metrics used for expressing test adequacy (i.e., the thoroughness of testing or determining when to stop testing with respect to a specific test criterion [118]). A coverage measure is generally expressed as a real number between 0 and 1, describing the ratio between the number of test items exercised during testing and the overall number of test items. Hence, a statement coverage of 1 implies that all statements in the software under test are exercised. Stopping rules (e.g., rules for when to stop testing) can be formulated in terms of coverage. For example, a statement coverage of 0.5 (indicating that half of the statements in the software are exercised) may be a valid, if not very practical, stopping rule.

Generic coverage metrics (i.e., coverage metrics that can be applied to all types of software, or a subset of software types) can be formalized for most structural test criteria, and for some formal functional test criteria.
1.1 Background

Monitoring for Software Testing

A necessity for establishing coverage measures (i.e., the ratio between what has been tested and what theoretically or practically could be tested) is the ability of instrumenting and monitoring the test process. For example, in order to establish a statement coverage, there is a need to know (1) the total number of statements in the code, and (2) which of these statements have been exercised during testing. Instrumentation for coverage is most often performed using software probes, automatically inserted by testing tools.

1.1.2 Concurrent Real-Time System Testing

Traditionally, a real-time system (RTS) is a system whose correctness not only depends on its ability to produce correct results, but also on the ability of producing these results within a well-defined interval of time [90]. In the scope of this thesis, we focus on the system-level testing performed after combining a set of sequential program units or components (i.e., tasks in a RTS) to a multi-tasking RTS. Note that we by the term system-level testing refer to any type of testing performed on the system as a whole, where the interaction between integrated subsystems come fully into play – not only the traditional functional acceptance-test type of testing usually performed at this level.

Up until this point, the discussion has implicitly assumed sequential non-preemptive, non-real-time software. However, when looking at system-level testing of RTSs, structural testing is, partly due to complexity, overlooked in favour of functional specification-based testing. This is problematic, not only because the structural aspects of testing are lost at this stage, but also because failures caused by structural bugs in a complex system are often extremely hard to find and correct [71, 103]. A bug that is hard to find is also a bug that consumes much resources in terms of time and money (and also company goodwill). There are two main problems that need to be solved for facilitating structural system-level testing of real-time systems: derivation of test items and overcoming the probe effect.

Deriving Structural Test Items in Multi-Tasking RTSs

To illustrate the problem of structural test item derivation, consider the following example: Given two small programs (in this example, represented by the foo and bar functions in Figure 1.3 on Page 7 and Figure 1.4), assume that these two programs are assembled into a small software system on a single processor, where they are allowed to execute in a pseudoparallel, preemptive
fashion. Further assume that this system is to be tested using the path coverage test criterion. Now, although both programs may produce an infinite number of paths, this could be handled using restrictions or approximations (e.g., bounds on the number of times iterative constructs may be visited during the execution of the program). Still, the combinatory execution of \texttt{foo} and \texttt{bar} might produce an intractable number of paths depending on how the programs execute and preempt each other. Possible paths include

\[
\begin{align*}
p_1 & : \texttt{A} \rightarrow \texttt{G} \rightarrow \texttt{H} \rightarrow \texttt{B} \rightarrow \texttt{I} \rightarrow \texttt{K} \rightarrow \texttt{Y} \rightarrow \texttt{D} \rightarrow \texttt{E} \rightarrow \texttt{X} \\
p_2 & : \texttt{A} \rightarrow \texttt{B} \rightarrow \texttt{D} \rightarrow \texttt{F} \rightarrow \texttt{X} \rightarrow \texttt{G} \rightarrow \texttt{H} \rightarrow \texttt{I} \rightarrow \texttt{K} \rightarrow \texttt{Y} \\
p_3 & : \texttt{A} \rightarrow \texttt{C} \rightarrow \texttt{G} \rightarrow \texttt{I} \rightarrow \texttt{J} \rightarrow \texttt{Y} \rightarrow \texttt{D} \rightarrow \texttt{E} \rightarrow \texttt{X}
\end{align*}
\]

In addition, these are just examples of paths where the programs preempt each other in between basic blocks. For example, \texttt{foo} could just as well preempt \texttt{bar} in the start, the middle or the end of the execution of basic block \texttt{G}. Note also that we disregard the actual execution of the kernel context switch routine at this stage.

The need for considering system-level execution paths becomes evident when there are dependencies between the programs in the system. For example, assume that \texttt{foo} and \texttt{bar} share a variable \texttt{x}, which is assigned a value in blocks \texttt{I} and \texttt{F}, and is read in blocks \texttt{K} and \texttt{X}. Further assume that the programmer of \texttt{bar} assumes that the assignment in block \texttt{I} and the use in block \texttt{K} always will execute in strict sequence, without any interference from any other assignments to \texttt{x}, but fails to design or implement the necessary synchronization for this to be ensured. Regardless of the missing or faulty synchronization, from a program-level perspective, the assumption will always hold.
In a preemptive system, however, there is a possibility that \textit{foo} preempts \textit{bar} in between the assignment in \textit{I} and the use in \textit{K}, just to re-assign \textit{x} in \textit{F} and infect the state of the system (i.e., put the system in an erroneous state, possibly leading to a failure).

This kind of infection (and its potential propagation to output) could never be discovered using only testing of isolated sequential units, such as functions, single-tasking programs or tasks. However, using system-level testing with a correct representation of the data flow, a test criterion like the all DU-paths criterion would easily detect such failures. Note here that at system-level, a definition of a variable may reside in one task, and a use of the same variable may reside in another.

As the above examples however show, multi-tasking systems, when compared to sequential software, exhibit an additional level of control flow. In addition to the path described by traversed and executed basic blocks on task-level, a system-level sequence of task interleavings, preemptions and interrupts will also affect the behavior of the system. Naturally, these sequences are products of the run-time model, the operating system (if any), and the scheduler used. In order for structural coverage to work on system-level, there is a need to be able to represent the possible flow of control through the system. For any concurrent system, the combination of task-level control flow and system-level execution orderings constitute the system-level control flow.

\textbf{Definition 1.} We define an \textit{Execution Ordering} of a system \textit{S} to be an ordered finite sequence of task switches \( E = e_1, e_2, ..., e_k \), such that each task switch \( e_i, i = 1..k \) switches between two tasks \( t_A, t_B \in W_S \), where \( W_S \) is the task set of \( S \), and the sequence \( E \) is achievable by the run-time scheduler of \( S \).

\textbf{Definition 2.} We define a \textit{System-Level Control Flow Path} of a system \textit{S} to be an ordered finite sequence of statements \( s_1, s_2, ..., s_n \), such that each statement \( s_k, k = 1..n \) belongs to a task \( t_A \in W_S \), where \( W_S \) is the task set of \( S \), and any pair of statements \((s_k, s_{k+1})\) in the sequence either is part of the feasible control flow of a task \( t_B \in W_S \), or is a consequence of a task switch in a feasible execution ordering of \( S \) (i.e., \( s_k \) belongs to the preempted task and \( s_{k+1} \) belongs to the preemting task or vice versa).

Hence, the system-level control flow of a system \( S \) is given by the feasible execution orderings of \( S \), and the control flow of each task \( t \) in the task set of \( S \). Note that we, in the definition of a System-Level Control Flow Path, disregard the kernel statements executed during the task-switch routine.

Using these definitions, and the fact that each traditional task-level control flow path also can be seen as a sequence of statements, we are able to
apply task-level definitions of coverage criteria on structural system-level testing. Here, it should be noted that not all structural coverage criteria suffer from concurrency and interleavings. In a multi-tasking system, there is, e.g., no difference between a full system-level statement coverage and a full task-level statement coverage of all tasks in the system. Testing based on such coverage criteria should preferably be performed on task-level and not on system level, since this is less time-consuming. Testing efficiency is also about detecting the right kinds of bugs at the right level of integration. The discussion on which coverage criteria are suitable for system-level testing will be elaborated and formalized in Section 2.1.

**The Probe Effect in Structural RTS Testing Monitoring**

As previously described, testing requires some type of instrumentation in order to measure test progress. Such monitoring is often performed by instrumenting statements inserted in the code of the system. However, these software-based probes are intrusive with respect to the resources of the system being instrumented. Hence, in a multi-tasking or concurrent system, the very act of observing may alter the behavior of the observed system. Such effects on system behavior are known as *probe effects* and their presence in concurrent software were first described by Gait [31].

![Figure 1.5: An execution leading to a system failure.](image)

The cause of probe effects is best described by an example. Consider the two-task system in Figure 1.5. The two tasks ($A$ and $B$) share a resource, $x$, accessed within critical sections. In our figure, accesses to these sections are displayed as black sections within the execution of each task. Now, assume
that the intended order of the accesses to $x$ is first an access by $B$, then an access by $A$. Further assume that the synchronization mechanisms ensuring this ordering are incorrectly implemented, or even neglected. As we can see from Figure 1.5, the intended ordering is not met and this leads to a system failure.

Since the programmers are confused over the faulty system behavior, a probe, represented by the white section in Figure 1.6, is inserted in the program before it is restarted. This time, however, the execution time of the probe will prolong the execution of task $A$ such that it is preempted by task $B$ before it enters the critical section accessing $x$ and the failure is not repeated. Thus, simply by probing the system, the programmers have altered the outcome of the execution they wish to observe such that the observed behavior is no longer valid with respect to the erroneous reference execution. Conversely, the addition of a probe may lead to a system failure that would otherwise not occur.

In concurrent systems, the effects of setting debugging breakpoints, that may stop one thread of execution from executing while allowing all others to continue their execution, thereby invalidating system execution orderings, are also probe effects. The same goes for instrumentation for facilitating measurement of coverage. If the system probing code is altered or removed in between the testing and the software deployment, this may manifest in the form of probe effects. Hence, some level of probing code is often left resident in deployed code.

![Figure 1.6: The same execution, now with an inserted software probe, “invalidating” the failure.](image)
1.2 System Model

The execution platform we consider for this thesis is a small resource-constrained single-processor processing unit running an application with real-time requirements, e.g., an embedded processor in an industrial control system, or a vehicular electronic control unit. The following assumptions should be read with this in mind. Furthermore, this is a basic system model description, which will be refined or relaxed in subsequent chapters of this thesis.

We assume that the functionality of the system ($S$) is implemented in a set of tasks, denoted $W_S$. We assume strictly periodic tasks that conform to the single-shot task model [8]. In other words, the tasks terminate within their own period time. We further assume the use of the immediate inheritance protocol for task synchronization. In the section on future work in Chapter 8, we discuss how to generalise the model to also encompass non-periodic tasks by using server-based scheduling [61, 85, 88].

The tasks of the system are periodically scheduled using the fixed priority scheduling policy [7, 63]. As for restrictions on the system, we assume that no recursive calls or function pointers (i.e., calls to variable addresses) are used. Further, we assume that all iterative constructs (e.g., loops) in the code are (explicitly or implicitly) bounded with an upper bound on the number of iterations.

In this thesis, we represent a task as a tuple:

$$\langle T, O, P, D, ET \rangle$$

where $T$ is the periodicity of the task. Consequently, the release time of the task for the $n$th period is calculated by adding the task offset $O$ to $(n - 1) \times T$. For all released tasks the scheduling mechanism determines which task that will execute based on the task's unique priority, $P$. The latest allowed task completion time, relative to the release of the task, is given by the task’s deadline $D$. In this work, we assume that $D \leq T$. Further, $ET$ describes the execution time properties of the task (i.e., the task’s best- and worst-case execution time).

For each least common multiple (LCM) of the period times of the tasks in the system, the system schedule performs a recurring pattern of task instance releases (jobs). In each LCM, each task is activated at least once (resulting in at least one job per task and LCM). For each release, a job inherits the $P$ and $ET$ properties of its native task, and its release time and deadline are calculated using the task $T$, $O$, and $D$ properties respectively. For example, a task with
$T = 5$, $O = 1$, and $D = 3$ will release jobs at times 1, 6, 11, ... with deadlines 4, 9, 14, ...

1.3 Problem Formulation and Hypothesis

In structural system-level RTS testing, some of the basics of traditional coverage-based testing are not applicable. Specifically, we conclude that performing coverage-based structural testing on multi-tasking RTSs currently lacks (assuming a system under test $S$):

Nec–1 The ability to, by traditional static analysis, derive the actual set of existing test items in $S$.

To meet the above necessity, we state the following research hypothesis (again assuming a system under test $S$):

Hyp–1 By analysing timing, control, and data flow properties for each task in $S$, while also considering all possible task interleaving patterns, it is possible to determine a safe over-approximation of which test items that are exercisable by executing $S$.

In addition, when performing test progress monitoring on system-level, we lose the following necessity:

Nec–2 The ability to, in a resource-efficient manner, instrument and monitor $S$ without perturbing its correct temporal and functional operation.

In order to meet this second necessity, we state the following research hypothesis:

Hyp–2 By recording events and data causing non-determinism during test case executions with a latent low-perturbing instrumentation, it is possible to use the recorded information to enforce the system behaviour in such a way that each test execution can be deterministically re-executed and monitored for test progress without perturbing the temporal correctness of the initial execution.
1.4 Contributions

The contributions of the thesis are directly aimed at providing evidence for hypotheses **Hyp-1** and **Hyp-2**, thereby meeting necessities **Nec-1** (test item derivation) and **Nec-2** (probe effect-free monitoring) for structural testing. In this thesis, we present:

In order to meet structural testing necessity **Nec-1**: 
- A method for deriving test items from a multi-tasking RTS based on timed automata UPPAAL models [10, 59] and the COVER test case generation tool [41].
- A method for deriving test items from a multi-tasking RTS based on execution order graph theory [77, 103].
- An evaluation of the two methods with respect to accuracy, analysis time, and sensitivity to system size and complexity.

In order to meet structural testing necessity **Nec-2**: 
- A replay-based method for probe effect-free monitoring of multi-tasking RTSs by recording non-deterministic events during run-time, and using this recording for replaying a fully monitorable deterministic replica of the first execution.
- A description of how to use the replay method for monitoring test progress (in terms of exercised test items) in structural system-level RTS testing.
- An evaluation of the replay method with respect to run-time recording intrusiveness, and replication accuracy.
- Results and experiences from a number of industrial- and academic case studies of the above method.

The setting in which these contributions are considered is a structural system-level test process for RTSs. The process in its entirety is depicted in Figure 1.7, and works as follows:

1. A set of sequential subunits (tasks) are instrumented to facilitate execution replay, and assembled into a multi-tasking RTS.
2. A timed abstract representation (model) of the system control- or data flow structure is derived by means of static analysis.
1.5 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2 identifies a number of test criteria that are suitable for structural system-level RTS testing. Further, based on these test criteria, this chapter more formally defines the goals of this thesis.

Chapter 3 describes how to, based on a selected test criterion, derive test items using different system-level flow abstractions of the RTS under test.
Chapter 4 shows how to monitor test items for system-level testing without probe-effects, by recording non-deterministic events during run-time, and using these events in order to create a fully monitorable deterministic replica of the initial execution.

Chapter 5 presents a simple system example illustrating how the contributions in this thesis interact for establishing structural coverage in a system-level test process.

Chapter 6 presents experimental evaluations of the methods proposed in this thesis, including numbers on accuracy and performance of the test item derivation methods, industrial and academic case study results, run-time perturbation of the replay recording, and replay reproduction accuracy.

Chapter 7 presents and discusses previous work relevant to this thesis.

Chapter 8 concludes this thesis by presenting a summary and a discussion of the contributions, and by presenting some thoughts on future work.
Chapter 2

Structural Test Criteria for System-Level RTS Testing

When software units (e.g., functions, components, or tasks) are tested in isolation, the focus of the testing is to reveal the existence of bugs in the isolated unit behaviour. As the units are composed into a software system, intended and non-intended interaction between these units will give rise to a new source of potential failures. In a multi-tasking RTS, examples of such interactions could be inter-task memory corruption, race conditions, and use of uninitialized shared variables. As these interactions are undetectable in unit-level testing, they need to be addressed in system-level testing. The purpose of this chapter is to:

- Identify structural test criteria that are effective in finding failures caused by the effects of task interactions, and hence are suitable for usage in structural system-level testing.
- Define the desired sets of test items required in order to calculate coverage with respect to a test suite, a real-time system, and the chosen test criterion.

2.1 Structural Test Criteria

So, which of the structural test criteria defined for unit-level testing can be applied to system-level testing? In their 1997 survey on *Software Unit Test Coverage and Adequacy*, Zhu et al. list and discuss the most commonly used test
criteria for unit testing [118]. In this section, we make use of the definitions in Zhu’s survey to discuss how control- and data flow-based test criteria intended for unit-level testing apply to system-level testing. In doing this, we focus on (1) usefulness for detecting the existence of bugs related to concurrency, and (2) redundancy with respect to unit testing with the same test criterion.

Usefulness will be discussed and shown using examples for each (non-redundant) test criterion. Redundancy for a test criterion in system-level testing is expressed in terms of the globally scalable-property, defined in Definition 3 below. Generally, a test criterion is globally scalable if it can be satisfied equally well by unit-level testing and system-level testing. We will however begin by giving a more informal description of this property.

Traditionally, structural test criteria are defined in terms of a set of execution paths \( P \) and a flow graph of a sequential program (in our case, a task \( t \)) [118]. The definition of a test criterion is formulated such that the test criterion is satisfied if the execution of the paths in \( P \) causes all test items of \( t \) (with respect to the test criterion) to be exercised. Now, consider that we have a set of tasks \( t_1..t_N \) that are intended to be assembled into a multi-tasking RTS. Furthermore, assume that we for each task \( t_k, k = 1..N \) have derived a set of execution paths \( P_k \), such that \( P_k \) satisfies a specific test criterion \( TC \) for \( t_k \) (see (1) in Figure 2.1). Next, these tasks are assembled into a multi-tasking RTS \( S \) (step (2) in Figure 2.1). In the same step, we merge all execution paths of \( P_k, k = 1..N \) into a large set of execution paths \( P \). The main question is as follows: Does the new set of execution paths \( P \) satisfy \( TC \) for \( S \)? If so, the test criterion is globally scalable. Otherwise, it is not. Note here that all execution paths \( p \in P \) can be classified as system-level execution paths according to Definition 2.

Hence, a test criterion that fulfills the property can be tested fully adequately, and with less effort, on unit-level. If a criterion is globally scalable, system-level testing with respect to that criterion is redundant. Formally, we define the property as follows:

**Definition 3.** A test criterion \( TC \) is globally scalable if and only if, for all preemptive multi-tasking real time systems \( S \) with task set \( W_S = \{t_1, t_2, ..., t_n\} \), and a set of sets of execution paths \( P_{WS} = \{P_1, P_2, ..., P_n\} \),

\[
P_k \text{ satisfies } TC \text{ for } t_k, \ k \in \{1..n\} \Rightarrow \left( \bigcup_{P_k \in P_{WS}} P_k \right) \text{ satisfies } TC \text{ for } S
\]

As an example of the use of the property, consider a system \( S \) with a task set \( W_S \) consisting of three tasks \( \{A, B, C\} \). Further assume that a set
of sets of execution paths $P_{WS} : \{\{p_{A1}, p_{A2}, p_{A3}\}, \{p_{B1}, p_{B2}\}, \{p_{C1}, p_{C2}\}\}$ satisfies a certain test criterion $TC$ (i.e., $\{p_{A1}, p_{A2}, p_{A3}\}$ satisfies $TC$ for $A$, etc.). Now, if we assume $TC$ to be the statement coverage criterion, the set $P_S : \{p_{A1}, p_{A2}, p_{A3}, p_{B1}, p_{B2}, p_{C1}, p_{C2}\}$ will also satisfy $TC$ for $S$ (since $S$ will contain no statement that does not belong to any of the tasks in $W_S$, and all statements that belong to a task in $W_S$ will, by definition, be covered by some path in a set in $P_{WS}$).

However, if we assume $TC$ to be the all DU-paths coverage criterion, the system may include a path $p_{D} \notin P_S$ in which a definition of a variable $x$ in, e.g., task $A$ is followed by a subsequent use of $x$ in task $B$. Thus, on system-level new test items that are not covered in unit testing may emerge. Hence, the statement coverage criterion is globally scalable, while the all DU-paths criterion is not.

In the following sections, we will list the most commonly used coverage-based test criteria, and categorize them with respect to redundancy (i.e., if they are globally scalable) and usefulness. These sections will show that many traditional criteria, developed for structural unit-level testing, do not scale to system-level testing of concurrent systems.


Chapter 2. Structural Test Criteria for System-Level RTS Testing

2.1.1 Control Flow Criteria

Control flow criteria are test criteria that are based on, or expressed in terms of, the control flow graph of the software. For sequential programs:

- **Statement coverage criterion**
  
  “A set $P$ of execution paths satisfies the statement coverage criterion if and only if for all nodes $n$ in the flow graph, there is at least one path $p \in P$ such that node $n$ is on the path $p$” [118].

  Since the reasoning in the statement coverage example on Page 21 holds for arbitrary tasks and path sets, the statement coverage criterion is globally scalable, and hence redundant and not useful in system-level testing.

- **Branch coverage criterion**
  
  “A set $P$ of execution paths satisfies the branch coverage criterion if and only if for all edges $e$ in the flow graph, there is at least one path $p \in P$ such that $p$ contains the edge $e$” [118].

  In general, the branch coverage criterion is analogous to the statement coverage criterion with respect to redundancy and usefulness. However, if the transitions from one basic block or statement in one task (via the task switch routine) to another basic block or statement in another task is also considered a control flow edge, such an edge would not be covered in unit-level testing, and the criterion is not globally scalable.

- **Path coverage criterion**
  
  “A set $P$ of execution paths satisfies the path coverage criterion if and only if $P$ contains all execution paths from the begin node to the end node in the flow graph” [118].

  In order to show that the path coverage criterion is not globally scalable, we will make use of a trivial example: Consider two very small tasks $A$ and $B$. Assume that task $A$ consists of two machine code statements $s_{A1}$ and $s_{A2}$ executed in sequence, whereas task $B$ consists of a single statement $s_{B1}$. In order to satisfy the path coverage criterion for the tasks in isolation, we need the following $P_{WS} : \{\{p_A\}, \{p_B\}\}$, where $p_A$ traverses $s_{A1}$ followed by $s_{A2}$, and $p_B$ traverses $s_{B1}$. On system-level, however, there might, e.g., exist an additional path $p_S$, that traverses $s_{A1}$, switches task to $B$, traverses $s_{B1}$, switches task back to $A$, and traverses $s_{A2}$. Hence, the path coverage criterion is not globally scalable.
2.1 Structural Test Criteria

- **Cyclomatic number criterion**

“A set $P$ of execution paths satisfies the cyclomatic number criterion if and only if $P$ contains at least one set of $v$ independent paths, where $v = e - n + p$ is the cyclomatic number of the flow graph” [118].

In the definition, $e$ is the number of edges, $n$ is the number of vertices, and $p$ is the number of strongly connected components in the graph. A strongly connected component is, basically, a maximal subgraph in which for all pairs of vertices $(\alpha, \beta)$, there is a path from $\alpha$ to $\beta$, and a path from $\beta$ to $\alpha$.

Although not explicitly defined, we may think of a system-level control flow graph of a system $S$ as a graph that represents all system-level control flow paths of $S$, and no other paths. Since we, in order to show that the cyclomatic number criterion is not globally scalable, only need to show that there exist a system $S$, and a corresponding system-level control flow graph with a cyclomatic number $v_S$ such that $v_S > \sum_i v_i$, where $i \in W_S$, a very simple example will suffice.

Figure 2.2 depicts two control flow graphs $A$ and $B$. $A$ has a cyclomatic number $v_A = 5 - 4 + 1 = 2$, whereas $B$ has a cyclomatic number of $v_B = 6 - 6 + 0 = 0$. Hence, the programs represented by these control flow graphs could be fully tested according to the cyclomatic number criterion by 2 and 0 independent execution paths respectively. However, consider a RTS $S$, where $A$ and $B$ make up the control flow...
graphs of the system tasks, each statement $a..j$ has an execution time of one time unit, task $A$ has a higher priority than task $B$, and the release time of $A$ and $B$ is 2 and 0 respectively. The system-level control flow graph of $S$ is shown in Figure 2.3. This graph contains two strongly connected components (shaded in the figure), has a cyclomatic number of $v_S = 18 - 14 + 2 = 6$, and would require at least 6 independent execution paths in order to fulfil the cyclomatic number criterion. Hence, the cyclomatic number criterion is not globally scalable.

- **Multiple condition coverage criterion**

“A test set $T$ is said to be adequate according to the multiple-condition coverage criterion if, for every condition $C$, which consists of atomic predicates $(p_1, p_2, ..., p_n)$, and all possible combinations $(b_1, b_2, ..., b_n)$ of their truth values, there is at least one test case in $T$ such that the value of $p_i$ equals $b_i, i = 1, 2, ..., n$” [118].

Even though this criterion is defined in terms of a test set (or test suite) rather than in terms of a set of execution paths, it is informally intuitive to recognize that no new conditions will be introduced in the system by assembling the individual tasks together. Since no new test items will be introduced if no new conditions are introduced, the multiple condition coverage criterion is, even if not formally proven so, globally scalable.

\[1\] Note that there exist alternate definitions of cyclomatic complexity [21, 107], none of which are globally scalable.
2.1.2 Data Flow Criteria

Data flow criteria are based on, or expressed in terms of, control flow graphs extended with information of accesses to data (i.e., data flow graphs). We consider the following data flow criteria:

- **All definition-use (DU) paths criterion**
  
  “A set $P$ of execution paths satisfies the all DU-paths criterion if and only if for all definitions of a variable $x$ and all paths $q$ through which that definition reaches a use of $x$, there is at least one path $p$ in $P$ such that $q$ is a subpath of $p$, and $q$ is cycle-free or contains only simple cycles” [118].

  As shown in the example on Page 21, the integration of tasks into a multi-tasking system may introduce new test items in the form of new DU-paths. Since these test items do not exist at unit level, this criterion is not globally scalable.

- **All definitions criterion**
  
  “A set $P$ of execution paths satisfies the all-definitions criterion if and only if for all definition occurrences of a variable $x$ such that there is a use of $x$ which is feasibly reachable from the definition, there is at least one path $p$ in $P$ such that $p$ includes a subpath through which the definition of $x$ reaches some use occurrence of $x$” [118].

  In terms of redundancy and usefulness, this criterion is analogous to the all DU-paths criterion. The assembly of tasks may cause definitions, that in isolation did not reach any use, to reach a use of the same variable in another task. Hence, this criterion is not globally scalable.

- **All uses criterion**
  
  “A set $P$ of execution paths satisfies the all-uses criterion if and only if for all definition occurrences of a variable $x$ and all use occurrences of $x$ that the definition feasibly reaches, there is at least one path $p$ in $P$ such that $p$ includes a subpath through which that definition reaches the use” [118].

  As above, in terms of redundancy and usefulness, this criterion is analogous to the all DU-paths criterion. Task integration may cause definitions, that in isolation did not reach any use, to reach a use of the same variable in another task. Hence, this criterion is not globally scalable.
• Required k-tuples criterion

The definition of this criterion requires the definitions of *k-dr interaction* and *interaction path*:

"For \( k > 1 \), a *k-dr interaction* is a sequence \( K = [d_1(x_1), u_1(x_1), d_2(x_2), u_2(x_2), ..., d_k(x_k), u_k(x_k)] \) where

(i) \( d_i(x_i), 1 \leq i < k \), is a definition occurrence of the variable \( x_i \);
(ii) \( u_i(x_i), 1 \leq i < k \), is a use occurrence of the variable \( x_i \);
(iii) the use \( u_i(x_i) \) and the definition \( d_{i+1}(x_i) \) are associated with the same node \( n_{i+1} \);
(iv) for all \( i, 1 \leq i < k \), the \( i \)th definition \( d_i(x_i) \) reaches the \( i \)th use \( u_i(x_i) \)[118].

"An interaction path for a *k-dr* interaction is a path \( p = (n_1) * p_1 * (n_2) * ... * (n_{k-1}) * p_{k-1} * (n_k) \) such that for all \( i = 1, 2, ..., k-1, d_i(x_i) \) reaches \( u_i(x_i) \) through \( p_i \)[118].

Using these definitions,

"a set \( P \) of execution paths satisfies the required *k*-tuples criterion, \( k > 1 \), if and only if for all \( j-dr \) interactions \( L, 1 < j \leq k \), there is at least one path \( p \) in \( P \) such that \( P \) includes a subpath which is an interaction path for \( L \)[118].

Since the required *k*-tuples criterion essentially is an extension of the all DU-paths criterion, the same argumentation can be made regarding global scalability. E.g., a use in a preempting task may interfere with an existing interaction path, causing a new interaction path not testable on unit-level. The required *k*-tuples criterion is thus not globally scalable.

• Ordered-context and context coverage criterion

For the last two data flow criteria (The ordered-context coverage criterion and the context coverage criterion), we require the definitions of ordered context and ordered context path:

"Let \( n \) be a node in the flow graph. Suppose that there are uses of the variables \( x_1, x_2, ..., x_m \) at the node \( n \). Let \( [n_1, n_2, ..., n_m] \) be a sequence of nodes such that for all \( i = 1, 2, ..., m \), there is a definition of \( x_i \) on node \( n_i \), and the definition of \( x_i \) reaches the node \( n \) with respect to \( x_i \). A path \( p = (n_1) * p_1 * (n_2) * ... * p_m * (n_m) * p_{m+1} * (n) \) is called an *ordered context path* for the node \( n \) with respect to the sequence \( [n_1, n_2, ..., n_m] \)"
if and only if for all $i = 2, 3, ..., m$, the subpath $p_i * (n_i) * p_{i+1} * ... * p_{m+1}$ is definition clear with respect to $x_{i-1}$. In this case, we say that the sequence $[n_1, n_2, ..., n_m]$ of nodes is an ordered context for $n$ [118].

Further, the ordered-context coverage criterion is defined as:

“a set $P$ of execution paths satisfies the ordered-context coverage criterion if and only if for all nodes $n$ and all ordered contexts $c$ for $n$, there is at least one path $p$ in $P$ such that $p$ contains a subpath which is an ordered context path for $n$ with respect to $c$” [118].

If the order, in which the definitions in the sequence $[n_1, n_2, ..., n_m]$ are performed, is ignored, the ordered context is transformed to a definition context. Using definition contexts and corresponding definition context paths, a weaker condition, the context coverage criterion, can be defined:

“a set $P$ of execution paths satisfies the context coverage criterion if and only if for all nodes $n$ and all contexts for $n$, there is at least one path $p$ in $P$ such that $p$ contains a subpath which is a definition context path for $n$ with respect to the context” [118].

The fact that the context criteria are not globally scalable can easily be shown by a simple example. Consider a system $S$ with two tasks $A$ and $B$, and two sets $P_A$ and $P_B$ of execution paths, such that $P_A$ satisfies the ordered-context criterion for $A$ and $P_B$ satisfies the ordered-context coverage criterion for $B$ (hence, $P_A$ and $P_B$ also satisfy the context coverage criterion for $A$ and $B$ respectively). Further assume that there exists a system-level path $p_S = (n_1) * p_1 * (n_2) * ... * p_m * (n_m) * p_{m+1} * (n)$, such that $p_S$ is an ordered context path, with the ordered context $[n_1, n_2, ..., n_m]$ for the node $n$ in $S$, and $\exists i, j \in [1, m]: n_i \in A \land n_j \in B \land i \neq j$. Hence, in order for the ordered context coverage criterion to be globally scalable, the proposition $p_S \in P_A \cup P_B$ must hold. However, $p_S$ cannot be a path in $P_A$ since it contains at least one node from $B$’s flow graph. Furthermore, $p_S$ cannot be a path in $P_B$ since it contains at least one node from $A$’s flow graph. Thus, the ordered context coverage criterion, and the context coverage criterion, are not globally scalable.

2.1.3 Useful and Non-Useful Criteria

A useful test criterion is efficient (the effort of selecting test cases, and deriving test items is less than the effort of a corresponding exhaustive testing [83]) and applicable (the test criterion does not generally call for intractably large
28 Chapter 2. Structural Test Criteria for System-Level RTS Testing

test suites in order to be satisfied). When discussing usefulness of test criteria for structural system-level testing, we can start by establishing the following fact: No globally scalable test criteria are useful for system-level testing, since they can be met equally well and more efficiently in unit-level testing. For example, if we aim at achieving full statement coverage in a certain task $A$, it is easier to test the task in isolation, than to try to reach that goal once $A$ has been integrated into a large multi-tasking system. For efficiency purposes, it is important to perform the right type of testing on the right level of integration. Hence, we will henceforth only discuss the usefulness of non-globally scalable criteria.

As one might expect, some test criteria are not useful even though they are non-globally scalable. E.g., although the path coverage criterion is non-globally scalable, its usefulness is very low due to its severe complexity problems. Attempting to perform test case selection, or even test case generation, based on all possible sequences of statements that can be executed in a pre-emptive RTS, would inevitably result in infeasibly large test suites. In most systems, the same would apply to the cyclomatic number criterion. Similarly, considering the branch coverage criterion, and assuming that this criterion is not globally scalable, the remaining test cases to execute on system-level would have us covering each possible task switch branch in the system. This is an intriguing thought, but it does not correspond to any known fault hypothesis. In addition, it would likely crave its fair share of test cases in order to be satisfied.

Hence, in the remainder of this thesis, we will focus on test criteria based on definitions and uses of variables, and specifically on definitions and uses of variables shared globally between tasks. From an analysis perspective, there is no significant difference between the all definitions-, all uses-, and the all DU-paths coverage criterion. A full all DU-path coverage subsumes a full all definitions, and all uses coverage [116]. Furthermore, the all-DU-paths criterion is considered highly useful in traditional sequential testing [11]. In addition, we have shown in several examples (see sections 1.1.2 and 1.1.2) the significance of being able to test unsynchronized accesses to shared variables. Note that the required k-tuples, context, and ordered context criteria all subsume and/or extend the all DU-path coverage criterion, and can hence be useful for detecting more complex task inter-dependencies. Even though we focus on all DU-path testing in this thesis, we see no reason why system-level testing using these criteria could not be facilitated using the same approach.

Using our methods for facilitating all DU-path coverage in system-level testing, we will be able to detect failures caused by, e.g., race conditions, uninitialied shared variables, and interleaving failures.
Table 2.1: A summary of structural test criteria and their respective properties.

<table>
<thead>
<tr>
<th>Test Criterion</th>
<th>CF/DF</th>
<th>Useful</th>
<th>Globally scalable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>Control Flow</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Branch</td>
<td>Control Flow</td>
<td>No</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Path</td>
<td>Control Flow</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Cyclomatic-number</td>
<td>Control Flow</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Multiple condition</td>
<td>Control Flow</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>All definitions</td>
<td>Data Flow</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>All uses</td>
<td>Data Flow</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>All DU-paths</td>
<td>Data Flow</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Required k-tuples</td>
<td>Data Flow</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Ordered Context</td>
<td>Data Flow</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Context</td>
<td>Data Flow</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

2.1.4 Structural Coverage Criteria Summary

In Table 2.1, the results of our reasoning is summarized. In the remainder of this thesis, we will make use of the all-DU-paths criterion for exemplification. In any of these examples, however, this criterion could with a minor effort be replaced by most other non-globally scalable test criteria.

In this thesis, we focus on existing test criteria instead of formulating new, specialized criteria for structural system-level testing. The main reason for doing this is that we believe that some of the traditional criteria (specifically the DU criteria) address problems overlooked by current system-level testing practices. It should be noted that other efforts have been made in order to adapt old, and formulate new structural test criteria for system-level testing of concurrent systems [47, 86, 97]. These contributions however disregard the aspect of time and mostly focus on synchronous task communication. Hence, they are not suitable in multi-tasking RTS testing.

2.1.5 A Note on Feasibility

Note that some test items that are feasible when testing a task in isolation might be made infeasible by system integration and vice versa. For example, a definition of a shared variable in one task may influence the flow of control and make control flow paths considered infeasible [32, 35, 37] in other tasks feasi-
ble. This should be considered when performing testing of multi-tasking (and parallel) systems.

2.2 Structural Test Items

A representation of the control- or data flow is a basic requirement for deriving the test items necessary for performing testing based on any structural test criterion. In this section, we provide a more in-depth description of the flow representation problem in structural system-level testing based on test item sets. Further, in the same manner, we describe the desired outcome of the methods we propose.

2.2.1 Sets of Test Items

Once the test criterion to be used for testing is selected, we stumble upon the problem of finding all possible test items with respect the criterion and the system under test. For example, if we plan to measure test thoroughness in a system $S$ by means of visited DU-paths during testing, we need to be aware of the total number of DU-paths in $S$. The set of existing test items in the system under test can be used as a reference when measuring test adequacy, and as a rule for when to stop testing. For this purpose, $r_{TC}$ is a real number in the interval $[0, 1]$ describing the adequacy of a test set with respect to the test criterion $TC$. Ideally, $r_{TC}$ is the ratio between the number of tested elements divided by the total number of possible (feasible) test items. In order to fully satisfy a test criterion, all existing test items must be exercised during testing (i.e., $r_{TC} = 1$), but a stopping rule may state that the system is adequately tested when $r_{TC} = 0.8$. In addition, since the exact contents of the set of existing (or feasible) test items is undecidable in the general case, an approximate set has to be used in order for the metric to be applicable. Note that this over-approximation is not unique for our method, since the exact determination of feasible test cases in the general case is undecidable even for unit-level testing [116].

Example-I As an explanatory example throughout this section, we will consider all-DU-path coverage of shared variables in the concurrent example system in Figure 2.4. The system contains two shared variables ($x$ and $y$) and two local variables ($z$ and $w$). In the figure, each statement corresponding to a CFG node in the code of the system is labeled with a superscript letter for node reference
int x, y;

functionC(funC_params) {
    int z := 0;
    do{
        if [x<10] a
            then [z := x-3] b
        else [y := z+4] c
        if [x>5] d
            then [z := y+3] e
        else [y := z-1] f
    } while [z<10 && y>3] g
}

functionD(funD_params) {
    int w := 0;
    if [y>10] h
        then [x := w] i
    do{
        if [y<5] j
            then [w := y+x+w] k
        else [x := w+x+3] l
    } while [w<6 && x<10] m
}

Figure 2.4: Example system.

(e.g., “c” refers to the statement \( y := z \) in \( \text{functionC} \)). Consequently, in the example program, \( x \) is defined in statements \{i, l\}, and used in statements \{a, b, d, k, l, m\}, while \( y \) is defined in statements \{c, f\}, and used in statements \{e, g, h, j, k\}.

In this example, a DU-path coverage item is defined by an ordered pair \((d_v, u_v)\), where \( d_v \) is a statement with a definition of a variable \( v \), and \( u_v \) is a statement with a use of a variable \( v \). In the example program, e.g., \((b, i)\) is a DU-path test item of variable \( y \).
We will formalize our problem, and the desired outcome, in terms of different sets of test items. Given a generic structural test criterion $TC$, a multi-tasking RTS $S$, a test suite $Q$, and a system model $M_S$ we will define four different sets of test items: $Structural_{TC}(S)$, $Test_{TC}(S,Q)$, $Feasible_{TC}(S)$, and $Model_{TC}(M_S)$. Note that in our explanatory examples, where the test criterion is all-DU-path coverage, $TC = DU$.

Now, we know that our systems of interest are preemptive, pseudo-parallel multi-tasking RTSs. Considering arbitrary preemption (assuming that any task may preempt any other task in the system at any point in time), there is a number of structurally feasible coverage items $Structural_{TC}(S)$ such that each test item $t \in Structural_{TC}(S)$ may or may not be feasible, depending on the task interactions allowed by the run-time system. Basically, the structurally feasible test items in a system $S$ are the test items that can be derived by each task in isolation, plus the test items that can be derived by unrestricted task interaction in $S$. As such, $Structural_{TC}(S)$ is a rough over-approximation of the feasible test items in the system.

Example-II In the case of the all-DU-path criterion, the naive overestimation of all system test items ($Structural_{DU}(S)$) would be given by the set of all possible define-use combinations $(d_v, u_v)$, such that $d_v$ is a statement with a definition of a shared variable $v$, and $u_v$ is a use of the same variable $v$. In our example system, this would correspond to:

$$Structural_{DU}(S) = \{(i,a), (i,b), (i,d), (i,k), (i,l), (i,m), (l,a), (l,b), (l,d), (l,k), (l,l), (l,m), (c,e), (c,g), (c,h), (c,j), (c,k), (f,e), (f,g), (f,h), (f,j), (f,k)\}$$

Further, the set $Test_{TC}(S,Q)$ is defined to be the set of all exercised coverage items during the execution of test suite $Q$ on the system $S$. By definition, this set will always be a subset of $Structural_{TC}(S)$ (since $Test_{TC}(S,Q)$ can contain no coverage items based on definitions and uses not in $S$, regardless formulation of $Q$). Hence:

$$Test_{TC}(S,Q) \subseteq Structural_{TC}(S)$$

Example-III In order to exemplify $Test_{DU}(S,Q)$ on our example system, assume we test the system using a test suite $Q_S$ of three different initial settings to the shared variables $x$ and $y$, such that
(x, y) \in \{(4, 10), (10, 3), (5, 6)\} (since these are the only factors affecting the flow in the system). Further, assume (1) that the priority of the task executing function D (taskD) is higher than the priority of the task executing function C (taskC), (2) that each statement has an execution time of 1 time unit, and (3) that task C is released at time 0, and that task D is released at time 8.

The first parameter setting (4, 10) will result in path p1, parameter setting (10, 3) will result in path p2, and parameter setting (5, 6) will result in path p3, as described below:

\[
p_1 : \langle a, b, d, f, g, h, j, k, m, j, k, m \rangle
\]
\[
p_2 : \langle a, c, d, e, g, a, c, d, h, i, j, l, m, j, l, m, j, l, m, j, l, m, e, g \rangle
\]
\[
p_3 : \langle a, b, d, f, g, h, j, k, m \rangle
\]

From these execution paths, the following set of tested DU-paths can be derived:

\[
Test_{DU}(S, Q_S) = \{ (i, l), (l, l), (l, m), (c, e), (c, g), (c, h), (c, j), (f, g), (f, h), (f, j), (f, k) \}
\]

Feasible_{TC}(S) is the set of all feasible coverage items in a system S with regards to the test criterion TC. A test item \( t \in Structural_{TC}(S) \) can be excluded from Feasible_{TC}(S) if the timing of the system does not allow the execution of the test item, regardless of the input parameter settings, or if the execution of \( t \) requires the execution of an infeasible system-level path. For most structural test criteria, the calculation of the set of feasible test items is undecidable in the general case, even in sequential unit testing [118].

The set of feasible test items is always a subset of Structural_{TC}(S), since the latter always is a safe over-approximation of the set of test items contained in S. Still, Feasible_{TC}(S) by definition safely contains all test items that can be executed during testing. Hence:

\[
Test_{TC}(S, Q) \subseteq Feasible_{TC}(S) \subseteq Structural_{TC}(S)
\]

Example-IV Since Feasible_{DU}(S) is undecidable in the general case, we will not explicitly derive it for our example system (even though it probably could be calculated in this particular case). However,
considering the structure of the system, and assuming the timing and priority properties described in the previous example, some test items in $\text{Structural}_{TC}(S)$ are intuitively infeasible. For example, the DU-path $(l,k)$ will not be included in any valid execution path, since the statements are mutually exclusive with respect to a certain value of variable $y$ which will remain unchanged while in function $D$. The DU-path $(i,k)$ is also infeasible with the same motivation.

Basing test coverage and adequacy metrics on a set that is undecidable is naturally a major problem. In order for the metric to be applicable, we need some alternative approximate reference of which test items actually are contained in $S$. The fourth test item set, $\text{Model}_{TC}(M_S)$, is the set derived using an abstract model $M_S$ of the system control and data flow. Basically, the more accurate the model, the smaller the set $\text{Model}_{TC}(M_S) \setminus \text{Feasible}_{TC}(S)$.

Any model $M_S$ useful for our method will model the behaviour of $S$ such that each behaviour in $S$ yielding a test item will be represented in $M_S$. As such, it will yield a set $\text{Model}_{TC}(M_S)$ that is a safe over-approximation of $\text{Feasible}_{TC}(S)$:

$$\text{Test}_{TC}(S,Q) \subseteq \text{Feasible}_{TC}(S) \subseteq \text{Model}_{TC}(M_S) \subseteq \text{Structural}_{TC}(S)$$

Although bounded by the above constraints, the contents of $\text{Model}_{TC}(M_S)$ will be highly dependent on the model used of system representation. For example, bounding the system behaviour to a specific execution order (determined by a run-time or offline schedule) enables us to discard some of the test items in $\text{Structural}_{TC}(S)$. Naturally, a more thorough modeling may further reduce the set.

2.3 Summary

This chapter identified structural test criteria that are suitable for use in system-level RTS testing based on a classification of existing criteria for non-RTS unit-level testing. Furthermore, in the chapter, we chose one of these criteria, the all DU-path coverage criterion, to be the example test criterion for the remainder of this thesis. In addition, this chapter defined four different sets of test items: $\text{Structural}_{TC}(S), \text{Test}_{TC}(S,Q), \text{Feasible}_{TC}(S)$, and $\text{Model}_{TC}(M_S)$, where $\text{Feasible}_{TC}(S)$ describes the the set of test items we wish to derive given a system $S$, $\text{Model}_{TC}(M_S)$ describes its practically derivable approximation,
and $Structural_{TC}(S)$ describes its worst-case over-approximation. Further, $Test_{TC}(S, Q)$ describes the set of test items exercised by executing a certain test suite $Q$. In Chapter 3 we will describe how to derive $Model_{DU}(M_S)$ given two different system models. Chapter 4 describes how to extract $Test_{DU}(S, Q)$ during testing, and Chapter 5 provides a system example, where the above sets are derived and an approximate DU-path coverage is derived by dividing the number of test items in $Test_{DU}(S, Q)$ with the number of test items in $Model_{DU}(M_S)$ (i.e., $r_{DU} = \frac{|Test_{DU}(S, Q)|}{|Model_{DU}(M_S)|}$).
Chapter 3

Deriving DU-paths for RTS
System-Level Testing

In order to calculate coverage for system-level DU-path testing, we need to know (1) the number of unique DU-paths exercised by testing, and (2) the number of exercisable DU-paths in the system under test. As previously described, the former divided by the latter gives us the DU-path coverage achieved by a certain set of test cases. However, the derivation of potentially exercisable DU-paths requires a correct representation of control and data flow on system-level. In this chapter, we seek model representations of the system under test in order to derive the set of potentially exercisable DU-paths $\text{Model}_{DU}(M_S)$. Specifically, we present two different methods for deriving this set. In doing this, we bring together state-of-the-art methods in worst-case execution time (WCET) analysis, model-based testing, and real-time scheduling analysis.

3.1 Introduction

The two methods presented in this chapter make it possible to determine test adequacy in a multi-tasking RTS using the all DU-path coverage criterion with respect to shared variables. Generally, by analyzing the temporal attributes of each access to globally shared data within each task, while at the same time considering the progress of the run-time schedule of the RTS, our methods are able to model and analyze task interaction. Further, the task interaction pattern together with the shared variable access information allow us to derive which
DU-paths are potentially feasible in the execution of the RTS, and which are not. In detail, our contributions in this field are:

- A test item derivation method based on the timed automata modeling and verification tool UPPAAL [10, 59, 78], and UPPAAL’s test case generation extension COVR [41] (Section 3.2).
- A modeling and test item derivation method based on Execution Order Graph (EOG) theory [101] (Section 3.3).
- Theoretical and empirical evaluations and comparisons of the two methods and their respective usefulness (sections 3.4 and 6.1).

The EOG representation was chosen since it, to our knowledge, is the only abstract representation of a preemptive RTS that describes the full execution time complexity, considering the best case-, and worst case execution time of the system. UPPAAL, on the other hand, has some alternatives in, e.g., KRONOS [23] and HyTech [38]. For this thesis, we chose UPPAAL due to the usefulness of its COVR extension (i.e., the ability to derive test items from an UPPAAL model), and for the benefit given by geographical proximity to its originators. The latter also applies for the SWEET tool discussed below.

3.1.1 General Analysis Properties

Before delving into the details of our both approaches, we will discuss the properties, assumptions, and methods that are common for both approaches. These include the structure of the analysis process, the need for run-time system representation, and the use of static program analysis.

Process Overview

Formally, our ambition in this chapter is to, given a generic RTS $S$, and a test suite $Q$, generate a model $M_S$, and to analyse this model for DU-paths, such that the resulting set of test items $Model_{DU}(M_S)$ is as close to $Feasible_{DU}(S)$ as possible (i.e., such that $Model_{DU}(M_S) \setminus Feasible_{DU}(S)$ contains the least number of elements possible). In sections 3.2 and 3.3, we describe two approaches for performing the modeling and analysis for deriving $Model_{DU}(M_S)$. However, both approaches follow the same basic structure:

1. Given a multi-tasking RTS $S$ with a set of tasks $W_S$, each task in $W_S$ is individually analysed with respect to a certain selected test criterion
TC (in this chapter represented by all DU-paths, DU). In this step, the necessary input to the system-level analysis (e.g., best-, and worst-case execution time, variable definitions and uses, etc.) is derived.

2. A model of the system (MS) is created based on the information collected in Step 1 combined with information of the run-time system and the system scheduling parameters (e.g., task periodicity, offset, etc.).

3. Using MS, the set of test items in the model (ModelDU(MS)) is derived.

4. The results from system testing (TestDU(S,Q)) are compared with the results from Step 3. Using these, a coverage thoroughness measure is derived by calculating \( r_{DU} = \frac{|Test_{DU}(S,Q)|}{|Model_{DU}(MS)|} \).

**Run-Time System Modeling**

As shown by several examples in earlier chapters, the system-level control flow has major impact on the execution behaviour of a RTS. Consequently, it also influences the system-level testing of such systems. The system-level control flow, in turn, is dictated by the execution times of the tasks in the system, and the underlying real-time kernel and scheduler. Hence, in order to correctly derive test items for structural testing of a RTS, we also need to consider the impacts of timing as well as the underlying scheduler on the system. In the two approaches presented in this chapter, this is done by explicitly or implicitly modeling or simulating the behaviour of a FPS-scheduler on timed abstractions of tasks.

**Static Analysis**

Both our DU-path derivation approaches require static task-level information regarding the control flow structure of the tasks in the system, the temporal behaviour of the tasks, and shared variable definitions and uses. For this purpose, we use the SWEET (SWEdish Execution time Tool) tool [28], a research prototype tool for worst-case execution time analysis developed at Mälardalen University\(^1\). However, any WCET analysis tool that is modified to provide the data we require could be used for this purpose.

The static analysis we perform consists of three distinguished phases: a flow analysis where bounds on the number of times different entities in the

---

\(^1\)www.mrtc.mdh.se/projects/wcet
code can be executed are derived, a low-level analysis where bounds of the execution times for instructions are derived, and a final calculation phase where the flow and timing information are combined to yield a WCET estimate. Basically, the static analysis answers to Step 1 and parts of Step 2 in the process overview described above. Figure 3.1 depicts the schematic process of how the static analysis is used in the analysis. In (1), the source code is compiled using a compiler that supports the formats required by the analysis tool (in our case, the NIC compiler\(^2\)). Using the intermediate code from the compiler, the analysis tool performs a high-level flow analysis (2), producing a program flow representation (in our case, a scope graph [28]), and corresponding flow information for both our analysis approaches. Next, a low-level analysis (3) produces execution time information for the entities in the scope graph, using information of the assembly code. The flow information and the execution time information is merged with the run-time schedule information to form

\(^2\)See user.it.uu.se/svenolof/wpo. This step in the process assumes an application written in C, but all other steps in the analysis are language-independent.
3.2 Approach 1: Deriving DU-paths using UPPAAL and CO√ER

In our first approach, we present a method for modeling system-level control flow of multi-tasking RTSs using timed automata. In contrast to traditional control flow models or graphs used for testing, our model also expresses: (1) subroutine interaction, (2) timing, such as basic block execution times and time-triggered preemption, (3) certain flow information, such as loop bounds and infeasible paths, and (4) system-level properties, such as concurrency, task-interleaving and race conditions. Consequently, the model may serve as a representation of all possible system-level control flow paths that can be traversed during the execution of a multi-tasking RTS.

3.2.1 Preliminaries

In order to fully describe the contribution of this approach, there is a need to account for the refined system model assumptions, and to give a brief summary of UPPAAL [10, 59], the tool used for modeling.

Refined System Model

To make way for our line of reasoning, we extend the system model defined in Section 1.2. Hence, for the scope of this approach, we define a task using the 6-tuple:

\[ (T, O, P, D, ET, G) \]

The first five items are identical to those in the original definition, where \( T \) is the period time of the task, \( O \) is the activation offset relative to the task period (i.e., a task \( t \) with period time \( T_t \) and offset \( O_t \) is activated at time \( O_t, T_t + O_t, 2T_t + O_t, 3T_t + O_t, \ldots \)), \( P \) is the unique priority of the task, \( D \) is the deadline of the task, and \( ET \) describes the best- and worst case execution time.
times of the task. \( G \), on the other hand, is a new item representing the set of timed control flow graphs (CFGs) describing the control flow of all functions called by the task.

A timed CFG \( g \in G \) is defined by the 5-tuple

\[ \langle V, E, s, X, F \rangle \]

where \( V \) is a set of timed vertices representing the basic blocks of the function, \( E \) is a set of edges connecting the vertices, \( s \) is the unique entry vertex from which all other vertices are reachable, and \( X \) represents the set of exit vertices, from which no other vertices are reachable. \( F \) is a set of flow facts describing the behaviour of the task control flow. A flow fact \( f \in F \) is described as a safe condition on an edge (e.g., a loop bound is expressed as a safe upper or lower bound on the number of times a backward edge may be taken).

In addition, we assume that the execution time of each basic block vertex \( v \in V \) is upper bounded by an integer that is greater than or equal to the actual worst-case execution time of the block. Similarly, we assume that the execution time is lower bounded by an integer that is lower than or equal to the actual best-case execution time of the block. These integers (henceforth referred to as basic block WCET and BCET) describe safe upper and lower bounds on the execution time of that basic block. Hence, a timed vertex \( v \in V \) (representing a basic block) is defined by the 4-tuple

\[ \langle BCET, WCET, D, U \rangle \]

\( D \) is the set of shared variable definitions performed in the basic block represented by the vertex, and \( U \) is the corresponding set of shared variable uses.

As a general note, throughout the chapter we will use subscripted indices when referring to different properties (e.g., \( P_X \) refers to the priority of task \( X \), and \( WCET_Y \) is the worst-case execution time of basic block \( Y \)).

**UPPAAL and Co\(/\)ER**

UPPAAL [10, 59, 78] is a tool for modeling RTSs as networks of timed automata, and for simulating and verifying properties of these models. A thorough formal definition of UPPAAL timed automata is given in the referenced literature, but in short, an UPPAAL model is an extended finite automaton with a set of locations, and a set of directed edges between these locations. Further, a set of real-valued clocks are included in order to keep track of the passage of time in the model. Edges may be guarded with boolean edge guards, allowing
the edge to be taken only if the guard condition is satisfied. Similarly, locations may be restricted by boolean invariants, allowing the location to be visited only if the invariant condition is fulfilled. Different automata may synchronously inter-communicate by means of synchronization channels, expressed as labels on edges, such that a synchronization edge can only be performed if both automata involved in the synchronization are ready to communicate. In addition, UPPAAL locations may be labeled urgent or committed, where an urgent location is a location where no time may pass, and a committed location is an urgent location in which no edges from non-committed location may occur.

UPPAAL supports declaration of global variables (i.e., variables that are accessible by all automata in the model within which they are declared). Hence, a global variable in the UPPAAL model translates directly to a global variable in the system under test. For the method of modeling described in this paper, the 3.4.11 version of UPPAAL\(^3\) is used.

COVER [41] is an UPPAAL-based test case generation tool that generates suites of test cases based on UPPAAL timed automata system specification models, and observer automata.

### 3.2.2 RTS Control Flow Modeling in UPPAAL

In order to use the analysis engine of UPPAAL to derive shared variable DU-paths, we need to correctly model the structural and temporal behaviour of the system under test. In this section, we describe a RTS control flow model made up as a network of three categories of timed automata for this purpose. First, each task is modeled as a set of timed automata (the task main function, and the set of subroutines directly or indirectly called by the main function). Second, the run-time state (ready, running, waiting, etc.) of each task in the system is modeled as an Execution Control Automaton (ECA). Third, the run-time kernel (i.e., the schedule, task switch, and the system clock) is modeled as a set of timed automata. In the following subsections, we will show how these automata form the timed system-level control flow model of the RTS. Using the COVER tool, a search of all possible DU-paths in the model can be performed. These together form \(\text{Model}_{DU}(M_S)\).

Figure 3.2 displays the overall structure of the system-level control flow model. In the figure, the white parts are specific for each system, the grey parts are semi-generic in that they have a generic structure, but need to be parameterized for each specific system, and the black parts are completely generic.

\(^{3}\)www.uppaal.com
This section will focus on this structure and the purpose of our model. For an in-depth description of the model, we refer to Appendix A.

![Diagram of UPPAAL control flow model](image)

Figure 3.2: The structure of the UPPAAL control flow model.

**Task CFG modeling**

From a standard CFG perspective, a task is made up of a set of function-level graphs (i.e., a graph for the task main function, and a graph for each function that is explicitly or implicitly called by the main function). In our model, each function CFG translates to a timed automaton in the UPPAAL format. The basic control flow structure of a function is intuitively modeled in UPPAAL using a one-to-one mapping between basic blocks and UPPAAL locations, and between CFG edges and UPPAAL edges. Each location where a shared variable is accessed (defined or used) is explicitly marked in the model. By using UPPAAL clock guards and invariants, the correct temporal behaviour of each node in the graph can be expressed. Further, subroutine invocation is solved by means of UPPAAL synchronization channels.

Hence, this part of the system-level control flow model ensures that the correct task-level control flow, and shared variable accesses are expressed.
Modeling of Task Execution Control

Even though there are a number of CFG models describing the timing and the internal structure of each task, no task can be allowed to execute unless it is scheduled and released by the run-time scheduler. For each task in the model, a separate task ECA is used in order to keep track of the run-time state of the task. An ECA is generic in the sense that it has an identical structure regardless of the task it controls.

An ECA has five basic locations: waiting, ready, scheduled, executing, and terminated. Basically, a task model is only allowed to consume any time if its ECA is in the executing location. If a task model has not yet been released by the scheduler, its ECA is in the waiting location. If a task model has been released, but there are other released task models with higher priorities, its ECA will be in the ready location. The scheduled and terminated locations are intermediate locations, described further in Appendix A. In summary, the ECA automata restricts the behaviour of the system model, such that no task models make transitions when they are not executing.

FPS Scheduler and Task Switch Modeling

The third part of the system model is the FPS scheduler and the task switch automaton. These models ensure that the run-time schedule of the modeled RTS is correctly represented in the model. The FPS schedule model determines the ordering and timing of task releases (i.e., when certain task ECAs should change location from waiting to ready). Furthermore, the task switch automaton facilitates task model preemption in the system model. When a task model with a higher priority than the currently executing task model is released, the executing task model should be preempted in favour of the newly released task model and only the latter should be able to consume any execution time.

3.2.3 Automatic System Model Generation using SWEET

Using SWEET, the automata for the task models can be automatically generated from the RTS application source code. Basically, in our extension of the SWEET tool for this purpose, timed SWEET scope graph files for each task function are translated into xml-file UPPAAL automata. These automata can thereafter be imported into the main RTS UPPAAL project, initially comprising task switch, schedule, system clock and ECA automata. When all generated task automata are incorporated in the overall model, the test item analysis can be performed.
CovItems=2
Traces=2
TotCost=0
TotLength=4
du<x offset=0, edgeid ctl.a_to_b, edgeid ctl.b_to_c>
du<y offset=0, edgeid ctl.a_to_b, edgeid ctl.b_to_d>

Figure 3.3: Example partial CO✓ER output.

3.2.4 Deriving $Model_{DU}(M_S)$ using CO✓ER

CO✓ER [41] is an UPPAAL-based tool that generates suites of test cases based on UPPAAL timed automata system specification models, and observer automata. Hence, CO✓ER is initially intended for functional testing. In this section, we have described how to express the control flow structure of a FPS-scheduled RTS $S$ as an UPPAAL model $M_S$, hence allowing CO✓ER to perform a structural test case generation. By defining different kinds of observer automata, CO✓ER will generate different kinds of test suites, each of which corresponds to the test criteria represented by the observer automaton\(^4\). For the purpose of this thesis, we use a pre-defined DU-observer\(^5\). Using this observer automaton, and a model $M_S$ of a system $S$, CO✓ER will generate a suite of test cases that, when exercised, satisfies the all DU-path coverage criterion.

As CO✓ER is primarily directed towards usage with functional and temporal models of event-triggered reactive systems, it will generate a test case suite consisting of traces through the model that satisfies the test criterion defined by the observer. In a functional and temporal model of a reactive system, such traces will tell us exactly which inputs to give the system, and when to give them, in order to execute the tests. Using our system model (time-triggered FPS), we will not be able to fully control the execution of the system just by rudimentary user interaction, but we will still be able to derive the set of test items in the system (i.e., $Model_{DU}(M_S)$), as this is part of the CO✓ER test case generation output (see Figure 3.3\(^6\)). In this case, the output tells us that CO✓ER has found two separate DU-paths; one of variable $x$ and one of variable $y$, reachable through two different traces through the model. The discussion of test case generation versus test item derivation is further elaborated in

\(^4\)In the referenced literature on CO✓ER, test criteria are called coverage criteria.
\(^5\)Given at www.uppaal.com/cover.
\(^6\)This output is also in essence taken from the example at www.uppaal.com/cover
3.3 Approach 2: DU-path derivation using EOGs

In our second approach, we present a method for modeling system-level control flow of multi-tasking RTSs using Execution Order Graphs. In contrast to the previous approach, this approach does not explicitly model control flow on task-level, but relies on more abstract representations, such as timing intervals for task execution, preemption, and shared variable access. Hence, the modeling and analysis require less effort than those in the previous approach, but it is also likely to give a slightly less precise result. In other words, $\text{Model}_{DU}(M_S) \setminus \text{Feasible}_{DU}(S)$ for Approach 1 will generally be smaller than $\text{Model}_{DU}(M_S) \setminus \text{Feasible}_{DU}(S)$ for Approach 2.

3.3.1 Preliminaries

As stated in Definition 1 on Page 11, an execution ordering is defined as an ordered list of task switches and interrupts. This list includes all transfers of control from one task to another task, or from a task to an interrupt service routine and back during the execution in question.

The Execution Order Graph

In 1999, Thane and Hansson [101] defined an algorithm for deriving a directed reachability graph of all possible execution orderings from a periodically scheduled set of task instances during a LCM (the Least Common Multiple of

<table>
<thead>
<tr>
<th>Job</th>
<th>Rel</th>
<th>Prio</th>
<th>BCET</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_0)</td>
<td>0</td>
<td>4</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>(A_1)</td>
<td>90</td>
<td>4</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>(B_0)</td>
<td>100</td>
<td>2</td>
<td>7</td>
<td>37</td>
</tr>
<tr>
<td>(C_0)</td>
<td>135</td>
<td>1</td>
<td>13</td>
<td>47</td>
</tr>
<tr>
<td>(D_0)</td>
<td>170</td>
<td>3</td>
<td>5</td>
<td>41</td>
</tr>
<tr>
<td>(A_2)</td>
<td>180</td>
<td>4</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.1: An example 270 ms schedule with job identity, release time, priority, and best- and worst-case execution time.

Section 8.3.1. For a full example of the process of deriving DU-paths using COER from a small multi-tasking RTS, see Chapter 5.
the period times of the tasks; the shortest time before the schedule periodically repeats itself). This graph is called an Execution Order Graph (EOG). Basically, knowing the offset and period time for each task (or release time for each task instance), the task priority, the WCET, and the BCET for each task, all possible execution orderings can be derived and visualized. An example of an EOG (based on the fixed-priority schedule in Table 3.1) is shown in Figure 3.4.

Figure 3.4: An Execution Order Graph (EOG) based on the schedule in Table 3.1.

Subsequent work by Thane and Hansson described how to handle sporadic interrupts in the EOG (in fact, the same technique could be used by our method in order to model functionally non-interactive tasks as temporal interferences, but this is considered future work) [100]. Furthermore, Thane and Pettersson addressed testing of preemptive multi-tasking RTSs [103] where input and
output from a task can be given and produced at the beginning of a task’s execution and at the end of the task’s execution respectively (using a single shot task model [8]). Later, the model was relaxed to cover testing of tasks where input and output could also be given within critical sections guarded with semaphores [77]. In this approach, we extend the EOG-based method to also handle all-DU-path coverage in preemptive multi-tasking RTSs. Hence, we are further relaxing the model, allowing inter-task communication anywhere in the execution of the tasks, however bounded to reads and writes of globally shared variables.

Refined System Model

As with the first approach, we need to make some extensions to the system model for this approach. Assuming a system $S$ with a set of tasks $W_S$ as described in Section 1.2, each task in $W_S$ is extended with a set of global variable definitions $D$ and a set of global variable uses $U$ (hence, in this section, a task is defined as $(T, O, P, D, ET, D, U)$). A definition $d \in D$ is defined as a five-tuple:

$$
\langle id, v, min, max, rdMax \rangle
$$

where $id$ is the unique address of the definition, $v$ is the name of the shared variable, $min$ is the shortest-, and $max$ is the longest execution time from the start of the task within which the statement $id$ can be reached (with no temporal or functional interference from other tasks considered), and $rdMax$ is the longest time $d$ can be live (before it is overwritten by another definition). These temporal properties will be described more in-depth in Section 3.3.2.

A use $u \in U$ is defined as a four-tuple:

$$
\langle id, v, min, max \rangle
$$

where $id$ is the unique address of the use, $v$ is the name of the shared variable, $min$ is the shortest-, and $max$ is the longest execution time from the start of the task within which the statement $id$ can be reached (with no temporal or functional interference from other tasks considered).

As an example, consider the code in Figure 3.5, where uses are enclosed in a selection statement. Figure 3.6 shows an example execution of two concurrently executing tasks, $A$ and $B$, where $B$ consists of the code from Figure 3.5, and $A$ has a higher priority than $B$, and contains a definition of variable $a$. Hence, at task level, $A$ only has a definition and no DU-paths. In $B$ there is a definition ($a:=b+4$) and two uses ($result:=a+1$ and $result:=a\times2$)
1. \( a := b + 4 \)
2. if expression is true then
3. \( \text{result} := a + 1 \)
4. else then
5. \( \text{result} := a \times 2 \)

Figure 3.5: Example of defs and uses.

of the same variable \( a \). In two example executions of the systems (shown in Figure 3.6), \( B \) is allowed to finish before \( A \) starts (Figure 3.6a), alternatively is preempted by \( A \) (Figure 3.6b). Consequently, the result differs between the executions. This is another example of a situation where task execution orderings influence the functional system behaviour, but that is not the prime contribution of this example.

Figure 3.6: Shared variable communication (assuming that the conditional expression is true in task B).

Instead, consider Figure 3.7, where the scenarios from Figure 3.6 are revisited, but here, the focus is on the exact times when the accesses are executed. For example, in Figure 3.7a definition \( a := b + 4 \) is executed at \( \text{def}_1 \) and overwritten at \( \text{rd}_1 \). In Figure 3.7b, \( a := b + 4 \) is executed at \( \text{def}_4 \) and overwritten at \( \text{rd}_4 \). Generally, for each access \( x \), there is an interval with extremal values \( x_{\text{min}} \) and \( x_{\text{max}} \) within which \( x \) can be executed. Furthermore, for each def-
3.3 Approach 2: DU-path derivation using EOGs

3.3.2 Shared Variable DU Analysis

In this section, we will present, in abstract terms, the analysis performed for deriving $Model_{DU}(M_S)$ using EOG models. In Appendix B, the details of the corresponding algorithm is presented. Basically, the analysis consists of two steps:

1. A task-level analysis, deriving $ET$, $D$, and $U$ for each task $w \in W_S$ by means of static program analysis.

Figure 3.7: Shared variable access attributes.

inition $d$, there is a point in time $d_{rdMax}$, where $d$ is safely overwritten. Note here that a definition, that may live after the termination of the task in which it is performed (as is the case with def$_1$, def$_4$ and def$_6$ in our example), will have a $rdMax$ equal to the WCET of that task.
52 Chapter 3. Deriving DU-paths for RTS System-Level Testing

... 1. i := 1;  
2. p := INIT;  
3. do {  
4. i := i + 1;  
5. if(i <= MIN)  
6. g := g + INCR;  
7. p := p * g;  
8. } while(i < MAX)  
9. return;  

Figure 3.8: Example code for timing analysis.

2. A system-level analysis, based on EOG analysis combined with the results from step 1, deriving shared variable DU-paths. In fact, the DU-paths are derived using the same algorithm that builds the EOG. Hence, strictly speaking, the EOG model of the system is not actually used, as it is derived along with the DU-paths.

Task-Level Analysis

The task-level analysis derives the temporal properties for each shared variable access (definition or use) in each task $w \in W_S$. Three properties ($\min$, $\max$, and $\rdMax$) are derived for each definition, and two properties ($\min$ and $\max$) are derived for each use. As $\min$ and $\max$ are analogous for definitions and uses, we will focus on the definition properties.

Assuming a definition $d$ that defines a variable $x$:

- The $d.min$ property is a safe lower bound on the shortest possible execution time from the start of the task to the statement containing $d$.
- The $d.max$ property is a safe upper bound on the longest possible execution time from the start of the task to the statement containing $d$.
- The $d.rdMax$ property is a safe upper bound on the longest possible execution time for any path $p$, starting at task start $s$ and ending at a statement $e$, such that $d$ is on $p$, $e$ contains a statement that redefines $x$, and no other redefinitions of $x$ are made between $d$ and $e$. Intuitively, this property describes the time (relative to the start of a task) where $d$
is safely overwritten. Note that, if \( d \) is not safely overwritten during the execution of the task, \( d.rdMax \) is equal to the worst-case execution time of the task.

**Timing analysis for defs and uses**

For the purpose of DU-path derivation, we have extended SWEET to, except the “normal” program WCET and BCET estimates, also produce estimates upon the above mentioned \( \min \), \( \max \), and \( rdMax \) values. We use SWEET to perform program flow- and low-level analysis, but not the calculation. The result can be seen as a CFG containing both flow- and timing bounds and with two extra \( start \) and \( exit \) nodes. Derived flow constraints are expressed as (lower or upper) bounds on the number of times a certain entity in the graph, (node or edge), can be taken, valid for all possible executions of the program. Derived timing bounds are expressed as a (lower or upper) bounds on the cost for executing a certain entity, valid each time the entity is executed. Figure 3.8 depicts an example code with two globals \( g \) and \( p \). Figure 3.9a illustrates the CFG for the code. The flow analysis has derived a loop bound of 10, expressed as an upper bound on the number of times node \( C \) could be executed. Each node \( i \) has also been given an timing bound \( t_i \) valid each time the node is executed.
Secondly, we perform a reaching definition (RD) analysis for global variables [69]. The analysis derives, for each global variable, where in the program it may be used and defined as well as how far each definition may reach. Since pointers could be used to update globals, SWEET’s RD analysis takes the input of a pointer analysis.

We derive the different estimates using an Implicit Path Enumeration Technique (IPET) calculation [28]. In IPET each node and/or edge in the CFG is given a time \( t_{\text{entity}} \), and a count variable \( x_{\text{entity}} \), the latter denoting the number of times that block or edge is executed. The WCET is found by maximising the sum \( \sum_{i \in \text{entities}} x_i \cdot t_i \), subject to constraints reflecting the structure of the program and possible flows. For example, there are constraints specifying that the start and exit nodes each must be taken exactly once, and constraints specifying that each node must be entered the same number of times as it is exited. The WCET estimate is derived using integer linear programming (ILP). The BCET is found by minimizing the same sum (or, if the statements do not have constant execution times, the corresponding statement BCET sum), subject to the same constraints (using ILP).

Our \( \text{min} \), \( \text{max} \) and \( \text{rdMax} \) analyses start from the above mentioned CFG. Depending on what timing values to derive, we modify the graph by adding extra edges and flow contraints. E.g., the graph for deriving \( \text{min} \), \( \text{max} \) for a use \( u \) is constructed by adding en extra edge from the node holding \( u \) to the exit node. Additionally, for all other edges going to the exit node we add a flow constraint specifying that its source node cannot be taken. Thus, we force the IPET calculation to exit through our newly created exit-edge, thereby deriving the best-case and worst-case estimates for \( u \), instead of the “normal” BCET and WCET. Figure 3.9b shows the CFG for calculating \( \text{min} \) and \( \text{max} \) for the use of \( g \) in node \( F \). For each global use and def derived in the RD analysis, we construct a corresponding graph. The resulting modified graphs are given as input to SWEET to derive the corresponding \( \text{min} \) and \( \text{max} \) values.

To derive \( \text{rdMax} \) for a definition \( d \) we first use the RD analysis to derive the set of nodes which \( d \) may reach. From each of these nodes we add an extra edge to the exit node. Additionally, for all other edges going to the exit node from a node which \( d \) cannot reach, we add a flow constraint specifying that its source node cannot be taken. Thus, we force the IPET calculation to exit through one of the nodes \( d \) may reach. The \( \text{rdMax} \) value is derived by a WCET calculation upon the resulting graph. Figure 3.9c shows the graph for calculating the \( \text{rdMax} \) value of the \( p:=\text{INIT} \) definition in node \( B \).
3.3 Approach 2: DU-path derivation using EOGs

System-Level Analysis

The algorithm for deriving system-level DU-paths is based on the algorithm for deriving EOGs [101]. Consequently, the analysis exhaustively simulates the execution of the system during the time interval of one LCM on an execution ordering level. In this simulation, performed by the recursive algorithm described in Appendix B, all possible task interleaving sequences of the system are traversed. After the time interval of one LCM, the system behaviour is periodically repeated, and no new shared variable definitions and uses will be found. During the analysis, information on the possible orderings of shared variable definitions and uses is collected.

Specifically, given our extended task model (including the set of shared variable definitions, $D$, and the set of shared variable uses, $U$), the analysis derives an overestimation of all DU-paths in the system. In order to derive this, the algorithm needs to keep track of which shared variable accesses have been, will be, or might have been performed at any time in the simulation. For this purpose, throughout the analysis, each definition and each use holds a certain variable state (dead, active or live, see Figure 3.10). An active access has been executed, or can be executed at any time until it has become live or dead. A live access has safely been executed, and not been safely overwritten by another access. A dead access is neither active nor live (i.e., the access has safely not yet been executed, is safely overwritten, or has safely passed the time where it can affect the result of the analysis). As the analysis progresses through the LCM time interval, the state of each access may change according to a set of rules. These rules for making the transitions between these access states constitute the foundation of the algorithm for deriving all system-level DU-paths\footnote{The rules assume that $Model_{DU}(M) = \emptyset$ initially.}:

Definition rules:

1. At $d_{min}$, $d$ makes a transition from dead $\rightarrow$ active.
2. At $d_{max}$, $d$ makes a transition from active $\rightarrow$ live.
3. At $d_{rdMax}$, $d$ makes a transition from live $\rightarrow$ dead.

Use rules:

1. At $u_{min}$, $u$ makes a transition from dead $\rightarrow$ active.
2. At $u_{max}$, $u$ makes a transition from active $\rightarrow$ dead.
DU-path rules:

1. At $d.\text{min}$, all DU-paths $(d, u)$, such that $u.\text{var} = d.\text{var}$ and $u$ is currently active, are added to $\text{Model}_{DU}(M_S)$.

2. At $u.\text{min}$, all DU-paths $(d, u)$, such that $u.\text{var} = d.\text{var}$ and $d$ is currently live or currently active, are added to $\text{Model}_{DU}(M_S)$.

3.3.3 Deriving $\text{Model}_{DU}(M_S)$ using EOGs

The above seven rules (partially depicted in Figure 3.10) are implemented in the DUANALYSIS algorithm, deriving all system-level DU-paths. The details of the DUANALYSIS algorithm are described in Appendix B.

3.4 Discussion

In this chapter, we have presented two different methods for deriving test items for structural system-level testing. Specifically, we have described how these methods derive the set of DU-paths ($\text{Model}_{DU}(M_S)$) from a time-triggered multi-tasking RTS. Theoretically, there are some basic differences between the two approaches described in this chapter. For example:

- The $\text{COER}$ method has the ability to prune live definitions at resumption [40]. In the EOG-based approach, the set of live definitions propagates through the analysis. If a definition $d$ in a task instance $t_1$ is not safely overwritten by the end of $t_1$’s execution, $d$ is considered live at
the start of the next scheduled task instance $t_2$. To prevent $d$ from being live indefinitely, $t_2$ “inherits” $d$ as a live definition and assigns it a new $rdMax$ value, describing how long into $t_2$’s execution $d$ can be live before it is safely overwritten. This value is derived for each global variable by the task level analysis, by calculating the $rdMax$ property for a “virtual definition” of the variable at the task CFG entry point.

However, should the task switch from $t_1$ to $t_2$ be a resumption of a previously preempted, partially executed task, $d$ cannot be assigned a new $rdMax$ value, since there is no way of knowing how long $d$ will live in $t_2$ from the arbitrary resumption point. In this case, the EOG-based approach will let $d$ live throughout the execution of $t_2$ and make a new decision based on the next scheduled task instance $t_3$.

Using the CoVer based approach, each system-level control flow path is considered (even if not always individually searched), and in the above described scenario of a resumption of a previously preempted task $t_2$, all paths from the resumption point to the end of $t_2$ will be traversed. If all these paths contain at least one statement where $d$ is overwritten, $d$ will not be considered live when scheduling task instance $t_3$. This will result in a tighter over-approximation of $Model_{DU}(M_S)$, and hence a more detailed analysis result.

![Figure 3.11: Example CFGs.](image-url)
Consider the example CFG in Figure 3.11a. If we assume that the left node in the selection consumes 10 time units of execution time, and all other nodes consume 1 time unit of execution time, we will end up with (from x’s point of view) a definition $d_1 (\{x:=3\})$, a definition $d_2 (\{x:=4\})$, and a use $u_3 (\{y:=x\})$ with the properties shown in Table 3.2.

<table>
<thead>
<tr>
<th>Access</th>
<th>min</th>
<th>max</th>
<th>rdMax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1$</td>
<td>4</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>$d_2$</td>
<td>5</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>$u_3$</td>
<td>6</td>
<td>15</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.2: Definition/use properties for the CFG in Figure 3.11a.

Using the EOG based approach, $d_1$ may be live from time 4 (assuming the right branch is taken), but is not safely overwritten until time 14 (assuming the left branch is taken). Since $u_3$ may be reached at time 6, and $14 > 6$, the analysis will consider $u_3$ reachable by $d_1$ (thus yielding a ($d_1, u_3$) DU-path) even though it intuitively is not. The Co/ER-based approach will not experience this problem, since it will detect the $d_2$ intermediate definition of $x$ and accordingly kill $d_1$ before it reaches $u_3$. This problem will however only arise in definitions and uses resident in the same task instance, and might be avoided by a corresponding task-level analysis, or a more explicit representation of task-level definition/use event relations (e.g., BES representations [57]).

UPPAAL-based models for the Co/ER approach can more easily be extended with additional flow facts (similar to those described in [4]). E.g., the CFG in Figure 3.11b may at a first glance appear to contain six DU-paths ({(1,3),(1,4),(1,5),(2,3),(2,4),(2,5)}), but taking into account the condition in statement 3 ($\{x>0\}$), it is obvious that DU-paths (1,4) and (2,5) are infeasible. This is only detectable if the above mentioned condition is considered. In UPPAAL models, such conditions trivially translates to transition guards, whereas they are inherently harder to represent in the EOG based approach. For simplicity’s sake, this problem is only shown in a single task example, but it also applies to inter-task flow facts. That is, a definition of a variable in task $t_1$ may affect the internal flow in task $t_2$. 

Chapter 4

Test Item Monitoring Using Deterministic Replay

Monitoring embedded RTSs using software probes may perturb the correct temporal operation of the software. Still, monitoring is essential for extracting the information required for determining test adequacy, e.g., coverage, during testing. Execution replay is an established method for overcoming the probe effect problem [5, 6, 9, 16, 19, 25, 60, 95, 102, 106]. In this chapter, we describe our RTS-oriented replay method, called Deterministic Replay [94, 105]. We also present the Time Machine, a Deterministic Replay tool implementation, and its three sub-components; the Recorder, the Historian, and the Time Traveler. Further, we describe the use of this replay method to extract the set of exercised test items, i.e., $Test_{DU}(S, Q)$, from a RTS without introducing probe effects. Our method of replay-based DU-path monitoring consists of the following steps (also shown in Figure 4.1):

1. As the system $S$ is designed and built, it is equipped with instrumentation for Deterministic Replay$^1$, i.e., probes, that extract the system events and data causing non-determinism, are introduced. Due to the low perturbation levels for this instrumentation [94, 105], the probes can be placed permanently in the source code of $S$, thereby eliminating probe effects. With the replay-probes embedded, $S$ is tested using some test suite $Q$ (4.1-A and 4.1-B). This yields a set of test reference execution recordings (4.1-A), each of which may span over several system LCMs.

$^1$Naturally, also legacy code systems may be instrumented for replay [94].
Figure 4.1: Replay-based DU-path monitoring - an overview.
2. The replay logs from Step 1 are gathered and analysed (4.1-C). Additional probes are added to \( S \) in order to be able to extract accesses to shared variables. For each test execution, its temporal behaviour is enforced based on the recorded execution, rather than by external events or the passage of time (4.1-D). This creates a monitorable deterministic replica of the test execution. Thus, the execution of added probes in the replay execution will not consume any execution time relevant to the determinism of the execution reproduction. The test suite \( Q \) is then re-executed by replay execution and information of exercised shared variable accesses are recorded (4.1-E). In this two-phase scheme, the amount of information extracted is of no significance to the determinism of the temporal behaviour of the system. Based on the outcome of this step, we derive \( \text{Test}_{DU}(S, Q) \), and are able to determine DU-path coverage by calculating \( r_{DU} = \frac{|\text{Test}_{DU}(S, Q)|}{|\text{Model}_{DU}(M_S)|} \).

4.1 Introduction

Run-time monitoring, e.g., for the purpose of test progress monitoring, is traditionally performed using hardware or software probes, where the hardware probe category includes in-circuit emulators, logic analysers, and custom monitoring devices [26]. Software probes, on the other hand, are implemented using instrumenting statements included in the source code of the system. As for hardware probes, they have the benefit of being able to extract run-time information without temporally or functionally perturbing the execution of the instrumented system. However, hardware probes suffer from other types of drawbacks:

- The support for hardware instrumentation is dependent on the target embedded system target hardware. Small volume, new, and high-end processors may lack state-of-the-art support for hardware instrumentation.

- Hardware instrumentation does in many cases require resources that, due to spatial reasons, may not be shipped with deployed systems [93]. This might not be a problem for testing, but post-deployment support for instrumentation is attractive for other reasons (e.g., debugging [94, 105]).

- Some hardware-based solutions, e.g., some In-Circuit Emulators, are expensive, making them unsuitable for usage in small projects, and a potential bottleneck resource in the development for big projects.
Software probes are more flexible than hardware probes, since they are added, altered and removed with less effort. There are two main problems with software probes:

- Software probes are intrusive in the sense that they consume both execution and memory resources, thereby affecting, and potentially affecting the temporal behaviour of the system. This phenomenon is called the probe effect [31].

- In resource-constrained embedded RTSs, memory and execution time are critical issues, limiting the amount of invasive monitoring allowed at run-time.

Due to the drawbacks of hardware probes, and the hypothesis that we are able to overcome the problems of software probes, we focus on the latter rather than the hardware-based solutions in this thesis. Further, we will show how to overcome the above software probe drawbacks.

### 4.1.1 The Probe Effect

The probe effect [31] describes the fact that it is not always possible to observe without interfering. If probes are added, removed, or altered over time, such that the level of perturbation that they cause varies, the system behaviour may be affected. On function or unit-level in a non-RTS, where execution is performed sequentially and without races or strict deadlines, this intrusiveness may be acceptable. However, in RTSs, software probes may lead to severe discrepancies in execution behaviour when comparing probed and non-probed executions. As an illustrative example, consider the following scenario: Two tasks (\(A\) and \(B\)) share a resource, \(x\), accessed within critical sections. In Figure 4.2, accesses to these sections are displayed as black sections within the execution of each task. Now, assume that the intended ordering of the accesses to \(x\) is first access by \(B\), then access by \(A\). As we can see from Figure 4.2, the intended ordering is not met and this leads to a system failure. When instrumenting the system for testing (e.g., all-du-path based testing), the inserted probes (the white sections in Figure 4.3) will alter the temporal behaviour of the system in which case, no system failure will occur.

Thus, when testing, the execution time of the probes will prolong the execution of task \(A\) such that it is pre-empted by task \(B\) before it enters the critical section accessing \(x\) and the failure is not repeated. Hence, simply by probing the system, the instrumentation has altered the outcome of the execution we
wish to observe and the observed behaviour is no longer valid with respect to the original execution in Figure 4.2.

Figure 4.3: Execution of the same tasks as in Figure 4.2, now with inserted software probes for coverage instrumentation, “invalidating” the failure.

4.1.2 Deterministic Replay

In our work on debugging of embedded RTSs, we have proposed a replay method for recording events and data causing non-determinism during a reference execution and to reproduce the execution in a subsequent replay execution [105]. Similar to other replay debugging methods (described in Chapter 7), the replay execution can be used to reproduce a failure in a debugger environ-
ment if the reference execution fails. Our method allows replay of real-time applications running on top of standard commercial real-time operating systems (RTOS) and uses standard cyclic debuggers for sequential software [16]. There is no need for specialized hardware or specialized compilers for the method to work and the software based instrumentation overhead has so far proven industrially acceptable (in the area of 3% of the total system load [94]). This allows our probes to be left permanently in the system, making post-mortem debugging of a deployed system possible while at the same time eliminating the risk of experiencing probe effects during debugging. We refer to our method as Deterministic Replay.

Considering the full potential of the ability of forcing concurrent executions into a deterministic run-time behaviour, other possibilities than system-level debugging arise. In this chapter, we describe the Deterministic Replay method, the Time Machine tool implementation, and the use of these in order to perform replay-based monitoring of DU-paths (i.e., extracting $Test_{DU}(S, Q)$).

### 4.2 Deterministic Replay DU-path Monitoring

The method for replay-based monitoring that we propose is a two-phase scheme, where phase (1) performs a limited instrumentation of a reference execution in order to facilitate replay, and phase (2) extracts definition and use information during the enforced behaviour of the replay execution. From this information, visited DU-paths in the reference execution can be derived. During phase (1), the software probes may be acceptably intrusive (i.e., they should not consume more resources than affordable in a deployed system, since they should be left permanently in the system). If this requirement is fulfilled, probe effects are eliminated and the technique is also applicable post-deployment [94].

In this section, we describe the process of using Deterministic Replay in order to derive the set of DU-paths exercised by testing the system $S$ with the test suite $Q$ ($Test_{DU}(S, Q)$). The process requires a monitorable deterministic replica of a multi-tasking RTS execution. For now, this replica is assumed to be provided by the Deterministic Replay method. In the later sections of this chapter, we will describe the process of creating this deterministic replica fully. In order to derive exercised DU-paths during the deterministic replica execution, we monitor definitions (assignments of values to shared variables) and uses (read accesses to shared variables) by placing debugger breakpoints on the statements containing the definitions and uses. Using the breakpoint
4.2 Deterministic Replay DU-path Monitoring

commands facilitated, e.g., by a scripting language [89], or by a plug-in API, we are able to extract a sequence of definitions and uses from each test case execution. From these, and the definition of a DU-path, it is trivial to derive the DU-paths. Alternatively, the definition/use access monitoring could be performed by placing debugger watchpoints on the shared variables, but this would have a significant detrimental effect on the performance of the replay execution.

In detail, in order to derive $\text{Test}_{DU}(S, Q)$ given a system $S$ and a test suite $Q$, the following steps are undertaken:

1. Ensure that $S$ is instrumented to facilitate Deterministic Replay.

2. Execute all test cases in $Q$ on $S$. As an outcome of the replay instrumentation, this will yield a set of execution traces $E$, where each execution trace $e_k \in E$ can be used to deterministically replay the execution of test case $q_k \in Q$.

3. For each task $w \in W_S$, statically identify the set $G$ of all statements containing shared variable accesses (actually, this information is a byproduct of the task level analysis described in Section 3.3.2).

4. For each statement $g \in G$, place a breakpoint on that statement. Instead of halting the execution, set the breakpoint to issue a set of commands (an example of breakpoint commands is shown in Section 6.2.3). The commands are set to log the time of occurrence, the variable accessed, the type of access (def or use), and the address of the PC (i.e., the statement itself) before resuming the execution.

5. For each trace $e_k \in E$, perform a replay execution of $S$ in a debugger with all shared variable access breakpoints set. This will yield a new set of traces $E'$, where each $e'_k \in E'$ contains ordered information of the shared variable accesses performed by test case $q_k \in Q$.

6. For each trace $e'_k \in E'$, derive the set of tested DU-paths $\Gamma_k$ in that trace using the following rule: For each shared variable $x$ and each subtrace $\gamma$ of $e'_k$, such that $\gamma$ starts with a definition of $x$, ends with a use of $x$, and contains no other definitions of $x$, $\gamma$ describes a tested DU-path. Consequently, $\text{Test}_{DU}(S, Q) = \bigcup_{e'_k \in E'} \Gamma_k$.

---

2See, e.g., www.iar.com
However, in order for the above scheme to work, and to correctly derive $Test_{DU}(S, Q)$ from a system $S$, it is imperative that we are able to deterministically reproduce the execution of each test case in $Q$. Therefore, in the next section, we will focus on the concept of reproducibility.

### 4.3 Reproducibility

For sequential programs with no external interaction, reproducibility is not a problem. If such a program is started an arbitrary number of times with identical inputs, all invocations of the program will exhibit identical behaviours and produce identical outputs. A program with this property is said to be deterministic [99]. For example, consider the program in Figure 4.4. If the program is executed with an input value of 0, it will eventually come across a division by zero. If the execution is repeated with the same input, we will again end up at the division by zero. Naturally, there is no limit on the number of times this failure will occur. Each time this program is executed with the input 0, we will reach the division by zero.

```c
int avg (int x) {
    int y, ret;
    y = getTotal();
    ret = y / x;
    return ret;
}
```

Figure 4.4: A sequential deterministic program.

In order to be able to monitor DU-paths exercised in an execution by means of execution replay, the system or program needs to be not only deterministic, but also reproducible (i.e., deterministic and controllable with respect to execution parameters [99]). Unfortunately, not many applications or systems today are even deterministic due to factors such as asynchronous I/O, concurrency, and timing.

Starting with the next section, we will try to give a clear picture of the problems involved in deterministic reproduction of non-deterministic system executions.
4.3 Reproducibility

4.3.1 Context Issues

Any useful computer system interacts with its environment. In its most basic form, an example of such interaction could be that of a program taking a command line input, sequentially calculating something based on that input and finishing by returning a command line output. This example system is depicted in Figure 4.5. If the command line input given to this program is the only parameter that can affect the output of this program and we know of this input (after all, we gave it on the command line in the first place), to us, this is a deterministic system. In addition, since we gave the input, we can simply give it again and receive the same behaviour and the same output. Thus, this program is also reproducible.

![Diagram of a basic interaction with a known and controllable external context.](image)

Figure 4.5: A basic interaction with a known and controllable external context.

This might seem somewhat trivial, but as we continue our line of reasoning and consider the system shown in Figure 4.6, we come across a slightly more complex situation. Suppose that the system is part of a mechatronic control system. It can be initialized and terminated by a user, but in between these events it leads a life of its own within a loop. This loop structure is used to periodically sample some external process, calculate a response to these samples and to produce actuating values based on the result of the calculation. As the observed system behaviour is uniquely defined by the inputs read at sampling time, this is a deterministic system. However, without the help of some kind of recording mechanism, it is nearly impossible to reproduce these inputs. Hence, the system in its basic form is not reproducible.

In short: A system, whose behaviour depends on parameters provided by an outside environment, is not reproducible if we cannot control this environment. To make the system reproducible, our only options are to gain control of these parameters or the environment providing them. In this thesis, we concentrate on the former.
4.3.2 Ordering Issues and Concurrency

So far in this chapter, we have only focused on reproducibility of sequential programs, i.e., programs running in a single thread of execution on a single processor. Looking at current state-of-the-practice commercial software, the applications that fulfill these restrictions are easily accounted for. In order to meet requirements regarding efficiency, interactivity and hardware utilization, most applications are pseudoparallel (several threads of executions on the same processor) or parallel (several threads of executions on several different or identical processors). If the interaction with an external context described in the above section seemed troublesome with regard to deterministic reproduction, it is merely a minor problem compared to the problems introduced by the concept of concurrency. Although deterministic reproduction of truly parallel system executions introduces additional complexity compared to deterministic reproduction of pseudoparallel system executions, the main focus of this chapter will be on the latter. In such systems, we face the additional challenge of reproducing the system-level control flow caused by the execution ordering of tasks.

4.3.3 Timing Issues

When discussing the concept of execution orderings, there is an important distinction to be made. Execution ordering events (in essence, events that cause task switches) can be either synchronous or asynchronous. In our view, synchronous events are manifestations of the internal inter-task synchroniza-
4.3 Reproducibility

In many systems, a correct reproduction of the ordering of synchronous events is sufficient for achieving a correct reproduction of the entire system behaviour. However, in systems with real-time requirements, alterations in the temporal behaviour of an execution that do not explicitly alter the ordering of events might nevertheless affect the correctness of the system. An example of a system with this type of behaviour is a multi-tasking RTS with hard deadlines.
nized. If data races exist in an execution, these will go undetected if we focus exclusively on the correct reproduction of the synchronization sequence.

4.3.4 Reproducibility - Summary and Problem Statement

To summarize this section, there are a few issues that have to be resolved when trying to deterministically reproduce multi-tasking real-time programs. At the start of this section, we started with a sequential, non-real-time program with a behaviour depending solely on its command-line inputs.

As systems are incorporated in different temporal and environmental contexts, the assumption of a completely deterministic and reproducible system behaviour will no longer hold. Interactions with external contexts and multi-tasking will lower the probability of traditional execution reproducibility to a negligible level. As depicted in Figure 4.7, events occurring in the temporal external context will actively or passively have an impact on the RTS execution. In this section, we discussed issues related to context, ordering, timing and the embedded nature of embedded systems. Now, let us see if we can derive a less abstract problem formulation from these sections.

![Figure 4.7: Execution of a multi-tasking RTS, dependent on the temporal- and the external context within which it is executing. Grey areas in the tasks represent temporal interaction (e.g., interrupts), and external context interaction (e.g., sampling) respectively.](image)

In order to correctly monitor and derive test items during testing without probe effects using Deterministic Replay, we need to achieve the same level of reproducibility in multi-tasking RTSs as the one we have in non-real-time sequential software. Considering this, there are a number of clarifying steps we
4.3 Reproducibility

can take. First, even though they served a logical- and (hopefully) pedagogical purpose, all issues related to ordering can be ignored. This might seem odd, but consider the conclusion in Section 4.3.3. As timing implies ordering and we need to be able to reproduce timing in order to meet our requirements, the reproduction of system timing will have correct reproduction of execution ordering as a consequence.

Second, from the point of view of the temporal and external contexts, the interaction with the system manifests in two different ways: The context (or rather a peripheral device) can actively interact with the system by means of an asynchronous interrupt. This interrupt most often has its origin in a corresponding event in the context itself (e.g., a reset of a clock register or a completion of an I/O transaction). The other means of interaction is when the context is passively being read by the system (e.g., synchronous reads of sensor values or clock registers). Hence, there are exactly three things required for deterministic reproduction of executions for monitoring on a single-node concurrent system:

- **Starting State**
  We need to be able to provide the exact initial state of the execution we seek to reproduce.

- **Input**
  Any input, initial or intermediate, to the execution must be reproduced, such that it exactly simulates the temporal, external or random context of the execution we desire to reproduce.

- **Asynchronous Events**
  All asynchronous events that occurred within the execution must be reproduced in such a way that their temporal and causal interference with the execution is not altered.

The first item of this list of requirements is no different to that of reproduction of non-real-time sequential software (i.e., command-line input), albeit slightly more difficult to achieve in real-time multi-tasking software. This requirement is equivalent to that of being able to pinpoint the current location on a map in order to use the map correctly. If you do not know where you are, there is no use reading the map in order to get where you want to be.

The second and the third requirement simply reflect the influence of temporal and external means of passive (input) or active (asynchronous events) interaction. However, note that these requirements are based on theoretical assumptions. As we shall see in Section 7.2.1, due to practical reasons, some of
these requirements have been ignored, under-elaborated or over-elaborated in previous work.

4.4 The Time Machine

The contribution of this chapter is a monitoring technique based on Deterministic Replay [105, 94], and we hence need to describe the details of this technique as well as its prototype implementation (the Time Machine). This is the purpose of this section. In our method, information is recorded during runtime with respect to interrupts, task-switches, timing, and data. The system behaviour can then be deterministically reproduced off-line using the recorded history, and inspected to a level of detail, which vastly surpasses what has been recorded since the system is deterministically re-executed such that all calculated data is restored. We will show how entire test executions including interrupts, task-switches and data can be reproduced off-line in order to facilitate the probe effect-free DU-path instrumentation process described in Section 4.2.

4.4.1 System Model Refinements

In addition to the assumptions posed in Section 1.2, we assume access to a debugger with an interface or scripting language such that macros or programs can be invoked conditionally at specified breakpoints, as well as target memory access. As we shall see, the availability of a proper debugger environment is imperative for the extraction of DU-paths from a replay execution. We further assume that the RTOSs have kernel instrumentation support such that task switches can be recorded during runtime. Several commercial RTOSs, e.g., VxWorks\(^3\) and QNX Neutrino\(^4\), do.

4.4.2 The Mechanisms of the Time Machine

We will now in further detail discuss and describe our method for achieving execution replay by describing its Time Machine implementation. The basic elements of the Time Machine are:

1. **The Recorder**, which is an in-target mechanism that collects all necessary information regarding the task-switches, interrupts, and data caus-
ing non-determinism in the system. The function of the Recorder basically answers to Step B in Figure 4.1 on Page 62.

2. **The Historian**, which is the off-target system that automatically analyzes, and correlates events and data in the recording, and composes these into a chronological timeline of breakpoints and break conditions. The Historian analysis corresponds to Step C in Figure 4.1.

3. **The Time Traveler**, which interacts with the debugger and, given the information provided by the historian, allows the recreation of the program state (i.e., state variables, global variables, program counter, etc.) for any given time in the scope of the memory of the Historian. The function of the Time Traveler is shown as Step D in Figure 4.1.

This process is performed without ever changing the target executable code. The same code (including RTOS) that is run in the target, during runtime, is run during the replayed execution in the debugger. DU-path instrumentation is facilitated using watch- or breakpoints with associated commands.

**The Recorder**

To achieve Deterministic Replay of an execution, a sufficient amount of information of the execution must be gathered. The basic idea of replay recording is that we need only record external inputs and internal state variables since we later during replay re-execute the system and consequently recalculate all intermediate variable values and outputs. If we use system start-up replay (i.e., if we do not allow old entries to be overwritten during the recording), all static and global variables are also recalculated, and can be omitted from recording.

The first part of the Time Machine, the Recorder, is basically a collection of kernel and application software probes and buffers for replay recording. In the Recorder, the storage space for monitored data is divided into two parts: the *execution ordering buffer* and a set of *data flow buffers*.

The execution ordering buffer is used to store all kernel events causing non-determinism, such as task switches, interrupts and missed deadlines. It consists of a cyclic buffer of a user-defined number of entries and an index pointing at the next entry available to write. Each entry in the buffer consists of a 7-tuple and is defined as \( \langle Tk, STk, PC, SP, C, Ev, EvT \rangle \) where \( Tk \) and \( STk \) define the time of the event. \( Tk \) is the value of a software-based tick counter, incremented by a timer interrupt. \( STk \), on the other hand can be viewed upon as fractions of a tick and is the value of a free-running counter in the underlying...
hardware, reset at each tick. This means that an event that occurred at $T_k=43$, $ST_k=256$ predates an event occurring at $T_k=43$ and $ST_k=398$. Moving on, we have $PC$, which is the value of the program counter register at the time of the event. $SP$ is the value of the stack pointer and $C$ is the unique marker used to differentiate between $PC$ values in iterative constructs (see, e.g., Figure 4.8).

```c
for (i=0; i<10; i++)
{
    a = a + i;

    b = q * 2 + i;
}
```

Figure 4.8: The PC is not sufficient as a unique marker.

In general, to correctly pinpoint asynchronous events, we need to record where and when they occur by using timestamps and the $PC$ value. However, since $PC$ values can be revisited in loops, subroutines and in recursive calls, additional mechanisms are required in order to define a unique marker for asynchronous events. We make use of a generic marker consisting of the $SP$ value, register-bank checksums, and/or checksums of part of the user-stack. This approach is described in-depth in Appendix C.

Finally, we have $Ev$ and $EvT$. $EvT$ tells us what type of event that occurred and $Ev$ is an identifier of the event. For example, if a hardware interrupt occurs, $EvT$ tells us that an interrupt occurred and $Ev$ tells us which interrupt service routine that was run. Another example can be made from a task switch. If a task switch occurred, $EvT$ tells us that a task switch occurred (and what type of task switch that occurred) and $Ev$ tells us which task that gained control after the task switch. The structure of an event in the execution ordering buffer is shown in Figure 4.9.

The other part of the memory storage space for monitor data is the data flow buffers. This is a set of user-defined buffers, used to record task data instead of kernel events. The data recorded here could be data external to the task, such as inputs to the task, system state, readings of sensors and messages from other tasks or nodes. In other words, data, that cannot be reproduced just by providing the same internal state as in the first run, should be recorded. A thing to keep in mind is that in contrast to the execution ordering buffer, each data flow buffer is tied to a specific task. The execution ordering buffer is global and is used only by the kernel.
An entry in a data flow buffer is defined as a 3-tuple: \((Tk, STk, Dt)\) where the meanings of \(Tk\) and \(STk\) are analogous to those in the control flow buffer. \(Dt\), however, is a set of user-defined data recording entries, that can be used to record data during a run-time execution of the task. The data flow buffers are cyclic and their recording mechanism works in the exact same way as the execution ordering buffer. Note that no full checkpoints of data contents are made (in order to potentially restore a starting state for replay), but individual data are incrementally stored when accessed. This approach requires more effort for instrumenting the system, but has the benefits of storing only the data that needs to be stored, and allowing replay start-up not only from pre-determined checkpoints.

In our basic approach we make use of a separate basic FIFO cyclic buffer for each task data flow, and one for the execution ordering. In these buffers, the oldest entry is always discarded when a new entry has to be recorded. Consequently, the system can only be replayed for the length of the recording in the temporally shortest buffer. Huselius [45] proposed the use of a single buffer and an eviction scheduler for Deterministic Replay recording purposes. The monitoring of this approach is more intrusive than that of cyclic buffers, but generally it provides a longer available replay.

In either case, it is important that the developer in an early state of the design allocates sufficient space to achieve a replay long enough to be able to fulfill the purpose of the replay. Note that for some purposes, e.g., for test item derivation, it is often desirable to replay the entire execution from system start-up.
The Historian

The Historian is an off-line analysis tool with the objective of providing a complete basis for the Time Traveler mechanism. The input to the off-line analysis is the raw data of the execution ordering buffer and the data flow buffers recorded on the target system. The idea is to use the information from system events to create breakpoints in the replay execution where the events occurred in the recorded execution. When halted at a breakpoint, different parts of the system state can be automatically altered in order to simulate an interrupt or a task switch. When the system state variables are set, the execution resumes. In this way, all external system events can be simulated.

Once the execution ordering and the data-flow of the application is recorded, the first job for the historian is to sort the execution ordering entries in order of occurrence and to construct a timeline. An execution ordering entry is either asynchronous (e.g. task preemption or interrupt) or synchronous (e.g. blocking system call). For each asynchronous execution ordering entry, the historian generates a conditional breakpoint, such that for each PC value where asynchronous events occurred, a breakpoint is set. These breakpoints are guarded by the condition of the recorded unique marker, e.g.,

```
break at PC(event) if
  (SP == SP(event) &&
   CHKSUMS_REGS(event) ==
      (R0+R1+R2+...+Rn) &&
   CHKSUMS_STACK(event) ==
      (*(SP)+*(SP+1)+*(SP+2)+...+*(SP+m)))
```

Synchronous events, on the other hand, are not represented by unique individual breakpoints. Instead, the entry point of each blocking system call, that might give rise to a synchronous event, is breakpointed. The control and match of the synchronous unique marker is here managed by the Time Traveler tool, as is the transfer of control from the executing task to the subsequent task. A timeline, similar to that of the execution ordering, is also assembled for the data-flow. To allow a smooth correlation between data and execution ordering, both monitoring activities are closely integrated.

Should the recordings contain the entire execution history from system start-up, the timelines are replayable as-is. Making use of a set of cyclic buffers for recording, where old values can be overwritten, might however raise the problem of having to start the replay from a non-startup state. In this case it is crucial to find a mutually consistent starting point for the replay. In other
words, we need to figure out at what execution ordering log entry and at which data instance to start the replay execution from.

![Diagram](link)

Figure 4.10: Insufficient data-flow between \( t_1 \) and \( t_2 \).

Since the basic idea of Deterministic Replay is to re-execute the application in exactly the same temporal and environmental context as the recorded execution, thus creating a fully monitorable sequential deterministic replica of the reference execution, a basic requirement is that both the execution ordering and the data flow information that constitute the replay context need to be available at the start of the replay. Consider, for instance, the scenario in Figure 4.10. Due to the dimensioning of the buffers, the execution ordering timeline spans from \( t_1 \) to \( t_{\text{end,log}} \), while the shortest data flow timeline spans from \( t_2 \) to \( t_{\text{end,log}} \).

In this case, replay starting points between \( t_1 \) and \( t_2 \) will not be valid since not all data flow information is available. Similarly, Figure 4.11 shows a scenario with an interval where no execution ordering information is available.

![Diagram](link)

Figure 4.11: Insufficient execution ordering between \( t_1 \) and \( t_2 \).

The requirement of available data flow at the replay starting point has to be considered when choosing how to record data. However, how to handle starting point issues are highly dependent on the underlying run-time model, and will be discussed in detail in Section 6.2.3.
Chapter 4. Test Item Monitoring Using Deterministic Replay

The Time Traveler

The Time Traveler is the “engine” driving the replay forward by automatically acting upon breakpoint events and issuing commands to the debugger. Initially, by setting breakpoints at all blocking system calls, we can initialise the replay of the recorded execution. First, the system is reset in the debugger and the timeline index, an index pointing at the current execution ordering entry to be matched, is set to point at the first suitable starting point in the execution ordering timeline. Then, each task is initialised such that its run-time state matches a suitable starting point in the historian-generated timeline. At this point, the recorded data flow of the suspended task is written back into the application. The timeline index is incremented and the next task is set up for execution. Once the data flow of all instrumented tasks has been rewritten into the application, the replay session initialisation phase is complete.

When the initialisation is ready, the replay will step forward as the timeline index is incremented at each execution ordering entry successfully matched. In addition, in the event of a subsequent asynchronous event for the current task in the historian timeline, its corresponding conditional breakpoint is set, making it possible to replay this event as that breakpoint is hit. Once breakpoints representing asynchronous events are hit and successfully matched, they are removed in order to enhance the performance of the replay session. This Deterministic Replay session will behave exactly like a (fully monitorable) regular sequential program mimicking the exact execution of the recorded multitasking real-time application.

4.4.3 Deterministic Replay Summary

In conclusion, we have described how to record and reproduce interrupts and asynchronous task-switches in a multi-tasking RTS. Assuming that significant variables, like state variables, and peripheral inputs like readings of sensor values or events like accesses to the local clock, are identified and recorded, it is possible to reproduce the interference from all sources of non-determinism off-line. Hence, decoupling of the external system (the real-world) and the progression of the system is accomplished, and Deterministic Replay of RTSs is facilitated. During the replay execution, the software behaviour of the system can be investigated to any level of detail. In this thesis, we use this newly achieved observability in order to derive exercised DU-paths during system-level testing.
4.5 Discussion

In this chapter, we have presented our RTS-oriented replay method, called Deterministic Replay [94, 105], and how to use this method in order to extract the set of exercised test items (i.e., $Test_{DU}(S,Q)$) from an embedded RTS without introducing probe effects. We conclude the chapter with a discussion regarding the lessons learned during the development and application of the Deterministic Replay method.

- **System Model**
  The Deterministic Replay method is not as sensitive to changes in the assumed system model as the other contributions in this thesis. For the DU-analyses proposed in Chapter 3, a periodically repeated system behaviour is an absolute necessity (for the EOG-based approach), or highly desirable (for the UPPAAL-based approach). Even though some aspects are somewhat complicated (e.g., dimensioning of recording buffers, and the process of finding a consistent starting point), Deterministic Replay is applicable in time-triggered, as well as sporadic and event-triggered real-time systems with terminating or non-terminating tasks.

- **Data Flow Analysis**
  We have successfully applied the time machine approach proposed in this paper in a number of applications running on different operating systems, different hardware, different compilers, and different debuggers (a thorough description of our case studies is given in Section 6.2). What we have learned however from these studies, is that it is necessary to carefully analyze the target system’s data flow with respect to what data is re-executed, re-transmitted and what data has external (process) origin in order not to forego something that may inhibit deterministic re-execution or that we do not record too much.
Chapter 5

System-Level DU-path Coverage – An Example

In the previous chapters, we presented the main technical contributions of this thesis. Specifically, Chapter 3 described how to derive a set of DU-paths from an abstract model of a RTS $S$ ($\text{Model}_{DU}(M_S)$), and Chapter 4 described how to derive the set of DU-paths exercised by testing $S$ by the test suite $Q$ ($\text{Test}_{DU}(S, Q)$).

In this chapter, we show how these contributions are combined in order to determine structural DU-path coverage for system-level testing. For this purpose, we will follow an example system as it propagates through our proposed system-level testing process.

5.1 The Process

First of all, we review the abstract description of our system-level test process. At this stage in the thesis, we are able to introduce our notion of DU-path sets in the process description. Hence, the process, in its entirety depicted in Figure 5.1, works as follows (assuming an all DU-path coverage test criterion):

1. A set of sequential subunits (tasks) are instrumented to facilitate replay, and assembled into a multi-tasking RTS $S$.

2. A timed abstract representation (model) $M_S$ of the system data flow structure is derived by means of static analysis.
3. The system schedule is added to the representation.

4. A suite of test cases $Q$ are selected and executed. During test execution, the non-deterministic events and data of the execution are recorded.

5. The recordings are used to deterministically replay the test suite executions with an enforced timing behaviour, and an added instrumentation, allowing DU-path derivation without probe effects.

6. After the testing phase, the monitored run-time information is extracted, yielding $Test_{DU}(S, Q)$.

7. Using analysis of the model derived in step 2, the theoretical maximum DU-path coverage (i.e., $Model_{DU}(M_S)$) is derived and compared with the results from step 6, allowing an estimate of the system-level coverage to be determined by calculating $r_{DU} = \frac{|Test_{DU}(S, Q)|}{|Model_{DU}(M_S)|}$

5.2 The Example System

We will use a periodic 3-task RTS $S_{ex}$ with 3 shared variables in order to exemplify the test process. The system makes no claim of being functionally sound, reasonably scheduled, or of performing any useful task. Its purpose is solely to exemplify the above test process.
5.2 The Example System

The source code of the tasks of \( S_{\text{ex}} \) are shown in Figure 5.2. Specifically, 5.2a displays the code of the \texttt{modeChange} task, 5.2b shows the code of the \texttt{calculate} task, and 5.2c displays the \texttt{shift} task code. Shared variables used are integers \( x \), \( y \), and \( \text{mode} \).

We assume that the system requirements call for a period time of 800 time units for all tasks, but with \texttt{modeChange} being offset by 100, and \texttt{shift} being offset by 200 time units. In other words, this means that \texttt{calculate} task instances (jobs) will be released at time 0, 800, 1600, ...; \texttt{modeChange} jobs will be released at time 100, 900, 1700, ...; and \texttt{shift} jobs will be released at time 200, 1000, 1800, ... Furthermore, this implies a repeated system behaviour, and a LCM of 800 time units. In addition, we assume that the \texttt{shift} task is given the highest priority, the \texttt{modeChange} task the medium priority, and the \texttt{calculate} task the lowest priority. We further assume that the system is schedulable using these task parameters.
5.2.2 System Replay Instrumentation

The next step in the process is to ensure that any execution of $S_{ex}$ can be deterministically reproduced by means of Deterministic Replay. Specifically, we need to make sure that the execution ordering and the data flow of the system can be reproduced. As the execution ordering instrumentation (e.g., monitoring of interrupts and task switches) is part of the underlying run-time system rather than the application itself, there is no need for inserting any execution ordering probes in any task code. For the purpose of this example, we simply assume that such instrumentation is included in the RTS kernel. The data flow recording, on the other hand, is performed on the application level. The data flow recording consists of (1) recording of intermittent input (e.g., readings of sensor values or the real-time clock), and (2) recording of accesses to shared or static variables (i.e., system state-preserving data). For (1), the application code of $S_{ex}$ does not include any statements that consume external or temporal context input. Hence, no such instrumentation need to be inserted. For (2),
5.3 DU-path Analysis

After the system has been instrumented for replay, we are able to statically derive the models required to find $Model_{DU}(M_{S_{in}})$. Note that, if the model derivation is done before the system instrumentation, the timing of the resulting

---

1Note that any schedulability analysis must also consider the execution time of instrumentation inserted in the system to facilitate execution replay.
model will not consider the temporal effects of the software probes. Since this thesis proposes two different modeling paradigms, we will consider both and derive model $M_N^{S}$ using the UPPAAL-based method, and model $M_E^{S}$ using the EOG-based method. Both model representations are generated using a SWEET-based task-level analysis.

### 5.3.1 Task-Level Analysis

The first step of the task-level analysis is the compilation of the system source code (in our case, by the NIC C compiler). Basically, we require this compilation to produce the format that the static analysis tool (in our case, SWEET) operates on. Next, given the intermediate code format files, the static analysis tool generates the configuration files for both approaches (as described in Sections 3.2.3 and 3.3.2).
5.3 DU-path Analysis

5.3.2 Deriving DU-paths using UPPAAL and COXER

In the UPPAAL-based method, the configuration files generated by the static analysis are made up of native UPPAAL XML-based descriptions of UPPAAL timed automata. For the tasks of the example $S_{ex}$ system, the generated automata are depicted in Figures 5.3 through 5.5. Note that the detailed explanation of our method for task modeling in UPPAAL is given in Appendix A. Further note that the figure displaying the shift task automaton (Figure 5.3) is the only figure that shows complete model annotation as required by COXER, including synchronization labels for task termination (i.e., terminate_shift), statement labels (e.g., shift_BB26), execution control synchronization (go/ready labels), timing guards and invariants (e.g., $\text{bbet} < 9$), and shared variable accesses (e.g., $x := 1$ and local_var := y). For presentation purposes, the modeChange (5.4) and calculate (5.5) figures show solely timing and shared variable access annotation, respectively.
Once generated, the $S_{ex}$ task model automata are imported into the generic UPPAAL run-time system model, comprised of the task switch, system clock, and execution control automata. Model $M'_{S_{ex}}$ is completed by the addition of a tailored schedule automaton (depicted in Figure 5.6), and the definition of task properties, e.g., task period, offset, and priority, in the model.

Using the above $M'_{S_{ex}}$ model, the UPPAAL test case generation extension CoER is capable of deriving $Model_{DU}(M'_{S_{ex}})$. To do this, we simply configure CoER to focus on DU-path coverage, using a standard DU-path observer as described in [41]. Further, we limit the variables of interest in the model to the three shared variables in $S_{ex}$ ($x$, $y$, and $mode$) by the configuration line:

\[
\text{cover duVar \{\{x,y,mode\}\}}
\]

Running CoER under these configurations, the analysis will yield the complete set of DU-paths feasible in $M'_{S_{ex}}$ (i.e., $Model_{DU}(M'_{S_{ex}})$). In our example system, $Model_{DU}(M'_{S_{ex}})$ is a set of 81 DU-paths. As described, this set is an over-approximation of the actual number of feasible DU-paths in $S_{ex}$.

### 5.3.3 Deriving DU-paths using EOGs

For the EOG-based approach, the static task-level analysis will generate a more abstract task representation than in the UPPAAL case. In our case, given the intermediate system code produced by the NIC compiler, SWEET will derive and generate a representation including the BCET and WCET for each task in $S_{ex}$, the min and max times for each shared variable use in $S_{ex}$, and the min, max, and rdMax times for each shared variable definition in $S_{ex}$. Adding scheduling information, like task priority, offset, and periodicity, the $M''_{S_{ex}}$ representation is completed (see Figure 5.7).

Once generated, the $M''_{S_{ex}}$ system representation is analysed for DU-paths using the EOG-based DUANALYSIS algorithm, as described in Section 3.3 and Appendix B. The resulting set, $Model_{DU}(M''_{S_{ex}})$ includes 83 different DU-paths, i.e., two more than considered potentially feasible by the UPPAAL-based approach.

### 5.4 System Testing

Now that the sets of potentially feasible DU-paths have been derived, we are able to test $S_{ex}$ using an arbitrary test suite $Q_{ex}$, monitor the test progress (i.e., extract $Test_{DU}(S_{ex}, Q_{ex})$), and calculate an approximate DU-path coverage...
<table>
<thead>
<tr>
<th>Job</th>
<th>Rel</th>
<th>Prio</th>
<th>WCET</th>
<th>BCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job</td>
<td>0</td>
<td>1</td>
<td>348</td>
<td>77</td>
</tr>
<tr>
<td>Def BB0</td>
<td>Var mode</td>
<td>min 0</td>
<td>max 0</td>
<td>rdMax 348</td>
</tr>
<tr>
<td>Def BB0</td>
<td>Var x</td>
<td>min 0</td>
<td>max 0</td>
<td>rdMax 348</td>
</tr>
<tr>
<td>Def BB0</td>
<td>Var y</td>
<td>min 0</td>
<td>max 0</td>
<td>rdMax 348</td>
</tr>
<tr>
<td>Def BB4</td>
<td>Var x</td>
<td>min 78</td>
<td>max 78</td>
<td>rdMax 120</td>
</tr>
<tr>
<td>Def BB7</td>
<td>Var x</td>
<td>min 87</td>
<td>max 120</td>
<td>rdMax 120</td>
</tr>
<tr>
<td>Def BB12</td>
<td>Var y</td>
<td>min 99</td>
<td>max 194</td>
<td>rdMax 233</td>
</tr>
<tr>
<td>Def BB15</td>
<td>Var y</td>
<td>min 105</td>
<td>max 233</td>
<td>rdMax 348</td>
</tr>
<tr>
<td>Def BB21</td>
<td>Var y</td>
<td>min 123</td>
<td>max 310</td>
<td>rdMax 348</td>
</tr>
<tr>
<td>Use BB1</td>
<td>Var mode</td>
<td>min 25</td>
<td>max 25</td>
<td></td>
</tr>
<tr>
<td>Use BB17</td>
<td>Var mode</td>
<td>min 67</td>
<td>max 254</td>
<td></td>
</tr>
<tr>
<td>Use BB9</td>
<td>Var mode</td>
<td>min 46</td>
<td>max 141</td>
<td></td>
</tr>
<tr>
<td>Use BB15</td>
<td>Var x</td>
<td>min 105</td>
<td>max 233</td>
<td></td>
</tr>
<tr>
<td>Use BB2</td>
<td>Var x</td>
<td>min 45</td>
<td>max 45</td>
<td></td>
</tr>
<tr>
<td>Use BB20</td>
<td>Var x</td>
<td>min 98</td>
<td>max 285</td>
<td></td>
</tr>
<tr>
<td>Use BB4</td>
<td>Var x</td>
<td>min 78</td>
<td>max 78</td>
<td></td>
</tr>
<tr>
<td>Use BB7</td>
<td>Var x</td>
<td>min 87</td>
<td>max 120</td>
<td></td>
</tr>
<tr>
<td>Use BB10</td>
<td>Var y</td>
<td>min 66</td>
<td>max 161</td>
<td></td>
</tr>
<tr>
<td>Use BB12</td>
<td>Var y</td>
<td>min 99</td>
<td>max 194</td>
<td></td>
</tr>
<tr>
<td>Use BB15</td>
<td>Var y</td>
<td>min 105</td>
<td>max 233</td>
<td></td>
</tr>
<tr>
<td>Use BB21</td>
<td>Var y</td>
<td>min 123</td>
<td>max 310</td>
<td></td>
</tr>
<tr>
<td>Use BB22</td>
<td>Var y</td>
<td>min 151</td>
<td>max 338</td>
<td></td>
</tr>
<tr>
<td>Use BB7</td>
<td>Var y</td>
<td>min 87</td>
<td>max 120</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Job</th>
<th>Rel</th>
<th>Prio</th>
<th>WCET</th>
<th>BCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job</td>
<td>100</td>
<td>2</td>
<td>102</td>
<td>66</td>
</tr>
<tr>
<td>Def BB33</td>
<td>Var mode</td>
<td>min 0</td>
<td>max 0</td>
<td>rdMax 102</td>
</tr>
<tr>
<td>Def BB33</td>
<td>Var x</td>
<td>min 0</td>
<td>max 0</td>
<td>rdMax 102</td>
</tr>
<tr>
<td>Def BB33</td>
<td>Var y</td>
<td>min 0</td>
<td>max 0</td>
<td>rdMax 102</td>
</tr>
<tr>
<td>Def BB35</td>
<td>Var mode</td>
<td>min 33</td>
<td>max 33</td>
<td>rdMax 102</td>
</tr>
<tr>
<td>Def BB38</td>
<td>Var mode</td>
<td>min 54</td>
<td>max 66</td>
<td>rdMax 102</td>
</tr>
<tr>
<td>Def BB41</td>
<td>Var mode</td>
<td>min 75</td>
<td>max 99</td>
<td>rdMax 102</td>
</tr>
<tr>
<td>Use BB34</td>
<td>Var mode</td>
<td>min 21</td>
<td>max 21</td>
<td></td>
</tr>
<tr>
<td>Use BB37</td>
<td>Var mode</td>
<td>min 42</td>
<td>max 54</td>
<td></td>
</tr>
<tr>
<td>Use BB40</td>
<td>Var mode</td>
<td>min 63</td>
<td>max 87</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Job</th>
<th>Rel</th>
<th>Prio</th>
<th>WCET</th>
<th>BCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job</td>
<td>200</td>
<td>3</td>
<td>66</td>
<td>66</td>
</tr>
<tr>
<td>Def BB26</td>
<td>Var mode</td>
<td>min 0</td>
<td>max 0</td>
<td>rdMax 66</td>
</tr>
<tr>
<td>Def BB26</td>
<td>Var x</td>
<td>min 0</td>
<td>max 0</td>
<td>rdMax 40</td>
</tr>
<tr>
<td>Def BB26</td>
<td>Var y</td>
<td>min 0</td>
<td>max 0</td>
<td>rdMax 61</td>
</tr>
<tr>
<td>Def BB29</td>
<td>Var x</td>
<td>min 40</td>
<td>max 40</td>
<td>rdMax 66</td>
</tr>
<tr>
<td>Def BB30</td>
<td>Var y</td>
<td>min 61</td>
<td>max 61</td>
<td>rdMax 66</td>
</tr>
<tr>
<td>Use BB30</td>
<td>Var x</td>
<td>min 61</td>
<td>max 61</td>
<td></td>
</tr>
<tr>
<td>Use BB29</td>
<td>Var y</td>
<td>min 40</td>
<td>max 40</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.7: $M''_{\text{Sel}}$ configuration file.
for the test suite. The testing and test monitoring are done in two phases, where phase (1) executes the test cases in \( Q_{ex} \) with no added test instrumentation except for the resident replay instrumentation, and phase (2) uses Deterministic Replay to re-execute all test recordings, this time with an added instrumentation of all shared variable accesses.

5.4.1 Initial Testing

For the system testing, we need to find a proper test suite \( Q_{ex} \). In our example system, the non-deterministic factors are limited (e.g., it contains no statements with variable execution time), and the only thing affecting the flow is the parametrization of the shared variables \( x, y, \) and \( \text{mode} \). Hence, we assume a small test suite comprising three test cases, selected solely for the purpose of this example:

\[
Q_{ex} : \{ (x:0,y:0,\text{mode}:0), (x:0,y:1,\text{mode}:1), (x:1,y:0,\text{mode}:2) \}
\]

We further assume that, for each test case \( q \in Q_{ex} \), testing is performed for the time interval of one LCM (i.e., 800 time units). Under these assumptions, executing the three test cases in \( Q_{ex} \) in the initial testing will result in three different replay execution trace recordings \( e_1, e_2, \) and \( e_3 \).

5.4.2 Replaying Test Cases

In order to derive the number of DU-paths exercised during \( e_1, e_2, \) and \( e_3 \), we deterministically reproduce the executions that these traces describe, together with an added instrumentation of all shared variable accesses. For example, for all uses of variable \( x \), the boldfaced statements in Figure 5.8 are instrumented. Since we use a debugger to facilitate replay, it is close at hand to use breakpoints or similar for the shared variable access instrumentation purpose. Such instrumentation will monitor each shared variable access without changing the executable file.

Using the Deterministic Replay method, the three test cases of \( Q_{ex} \) are re-executed, with correct timing enforced by execution traces \( e_1, e_2, \) and \( e_3 \). Consequently, when exercised, configuration \( (x:0,y:0,\text{mode}:0) \) yields 11 DU-paths, configuration \( (x:0,y:1,\text{mode}:1) \) yields 10 DU-paths, and configuration \( (x:1,y:0,\text{mode}:2) \) yields 14 DU-paths. Combining these results, and disregarding DU-path doublets, we end up with a set \( \text{TestDU}(S_{ex}, Q_{ex}) \) of 21 exercised DU-paths.
5.5 Summary

Once the Model DU ($M_S$) and Test$_{DU}(S, Q)$ sets are found, the DU-path coverage can be calculated as $r_{DU} = \frac{|\text{Test}_{DU}(S, Q)|}{|\text{Model}_{DU}(M_S)|}$. For the $M'_S$ UPPAAL-based model, the coverage is $r_{DU} = \frac{21}{81} \approx 0.259$. For the $M''_S$ EOG-based model, the coverage is $r''_{DU} = \frac{21}{83} \approx 0.253$. We let this example calculation of system-level DU-path coverage conclude our example chapter.

In summary, this chapter described how the contributions of this thesis can be combined to determine structural DU-path coverage in system-level testing. Specifically, we have followed an example 3-task system $S_{ex}$ through replay instrumentation, static DU-path analysis, initial testing, replay testing, and calculation of DU-path coverage.
Chapter 6

Evaluation

This chapter collects the results from the experimental evaluations performed on the techniques and methods proposed in this thesis, and is structured as follows: The first part of this chapter (Section 6.1) will focus on experimental evaluation of the DU-path derivation methods. The second part (Section 6.2) will focus on the case studies and implementations of the Time Machine, and the third part (Section 6.3) will focus on the evaluations of the checksum-based technique for pinpointing asynchronous events, for the purpose of deterministically reproducing them. Our intention is that the evaluations presented in this chapter shall serve as an initial investigation of the feasibility and applicability of the techniques presented in this thesis. Considering this intention, in summary, some of our methods show very promising results, while others need additional future work in order to be fully applicable in real contexts.

6.1 Test Item Derivation Methods

This section presents the experimental evaluation of the EOG-based and the UPPAAL-based DU-path derivation methods presented in Chapter 3. Both methods are evaluated with respect to precision of the analysis result, and analysis run-time for a number of different systems with different properties, size, and run-time complexity. In addition, for the second approach (deriving DU-paths using EOGs), we glance upon the run-time, and analysis result, consequences of varying the execution time jitter (the difference between task BCET and task WCET), since a large execution time jitter is generally said to signifi-
The purpose of the evaluations presented in this section is:

- To show that both the EOG-based analysis and the UPPAAL-based analysis are executable, and able to derive the set of DU-paths given a number of multi-tasking RTSs.
- To compare the precision of the result of both methods with the worst-case over-approximation, and to compare the run-time complexity of the both methods.
- To get a first notion of the limits of both methods in terms of manageable system size and structural system complexity (e.g., number of shared variable accesses, or number of unique task interleaving patterns).

### 6.1.1 Experimental Systems

Ideally, our methods should be evaluated using an established benchmark suite for concurrent system-level testing. Some such benchmarks are available [36, 29], but are unfortunately highly focused on specific run-time environments (e.g., testing of Java synchronization mechanisms).

Instead, as an experimental evaluation of our method, we provide analysis results from eight different control-oriented systems ($S_1$-$S_8$). All systems comprise control-oriented code (e.g., calculation of planet orbits ($S_1$) and a control system for a forklift able to solve the Towers Of Hanoi problem ($S_2$)). Systems $S_3$ and $S_4$ are taken from WCET analysis benchmarks\(^1\), and $S_5$ through

\(^1\)www.mrtc.mdh.se/projects/wcet
6.1 Test Item Derivation Methods

Table 6.2: UPPAAL-based test item derivation evaluation results.

<table>
<thead>
<tr>
<th>Sys</th>
<th>Config 1</th>
<th>Config 2</th>
<th>Config 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$(M_{DU}, S_{DU})$</td>
<td>$(M_{DU}, S_{DU})$</td>
<td>$(M_{DU}, S_{DU})$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$(\cdot, \cdot)$</td>
<td>$(\cdot, \cdot)$</td>
<td>$(\cdot, \cdot)$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$(\cdot, \cdot)$</td>
<td>$(\cdot, \cdot)$</td>
<td>$(\cdot, \cdot)$</td>
</tr>
<tr>
<td>$S_3$</td>
<td>$(\cdot, \cdot)$</td>
<td>$(\cdot, \cdot)$</td>
<td>$(21, 61.8%)$</td>
</tr>
<tr>
<td>$S_4$</td>
<td>$(\cdot, \cdot)$</td>
<td>$(\cdot, \cdot)$</td>
<td>$(\cdot, \cdot)$</td>
</tr>
<tr>
<td>$S_5$</td>
<td>$(\cdot, \cdot)$</td>
<td>$(\cdot, \cdot)$</td>
<td>$(\cdot, \cdot)$</td>
</tr>
<tr>
<td>$S_6$</td>
<td>$(28, 12.6%)$</td>
<td>$(25, 11.2%)$</td>
<td>$(19, 8.5%)$</td>
</tr>
<tr>
<td>$S_7$</td>
<td>$(\cdot, \cdot)$</td>
<td>$(117, 88.6%)$</td>
<td>$(127, 96.2%)$</td>
</tr>
<tr>
<td>$S_8$</td>
<td>$(23, 57.5%)$</td>
<td>$(27, 67.5%)$</td>
<td>$(16, 40.0%)$</td>
</tr>
</tbody>
</table>

$S_6$ are slightly smaller systems that were written by us for the purpose of this evaluation. In Table 6.1, Tasks and Gvars refer to the number of tasks and global variables respectively. $S_{DU}$ refers to the number of DU-paths in $\text{Structural}_{DU}(S)$, considering that each definition $d$ and use $u$ of the same shared variable may naively form a DU-path $(d, u)$. Generally, throughout this chapter, we will use the abbreviations $S_{DU}$ for denoting the number of DU-paths in $\text{Structural}_{DU}(S)$, i.e., $|\text{Structural}_{DU}(S)|$, and $M_{DU}$ for denoting the number of DU-paths in $\text{Model}_{DU}(M_S)$, i.e., $|\text{Model}_{DU}(M_S)|$. Further, $\frac{\text{Avg}_{WCET-BCET}}{WCET}$ describes the average ratio between the task jitter and its worst case execution time for all tasks in the system and, as such, is a measure of the relative system jitter. Thus, a high ratio (as the case in system $S_5$) implies a large difference between task $BCET$ and $WCET$ compared to the task $WCET$ in the system and hence a large execution time jitter. Size is the size of the system in terms of number of machine code statements. Loops and Selections is the number of loops and selections in the system respectively.

For our evaluations, each system is scheduled in three different ways (Config 1-3). As for the configurations, in Config 3, the tasks are completely separated in time and suffer no preemptions. In contrast, Config 1 is scheduled in an effort to maximize the number of task preemptions. Config 2 is an in-between configuration of Config 1 and Config 3. All evaluations were carried out on a 512 MB RAM Dell Precision M70 laptop computer running Windows XP on top of a 1.6 GHz Pentium M processor.

### 6.1.2 UPPAAL-based Test Item Derivation

We first present the evaluation of the method presented as Approach 1 in Chapter 3. In Table 6.2, $M_{DU}$ refers to the number of DU-paths found feasible by
Table 6.3: UPPAAL-based evaluation analysis run-times (in ms).

<table>
<thead>
<tr>
<th>Sys</th>
<th>Config 1</th>
<th>Config 2</th>
<th>Config 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$S_2$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$S_3$</td>
<td>-</td>
<td>-</td>
<td>$7.62 \times 10^6$</td>
</tr>
<tr>
<td>$S_4$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$S_5$</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$S_6$</td>
<td>$8.08 \times 10^4$</td>
<td>$2.47 \times 10^4$</td>
<td>1880</td>
</tr>
<tr>
<td>$S_7$</td>
<td>-</td>
<td>$5.98 \times 10^7$</td>
<td>$1.99 \times 10^4$</td>
</tr>
<tr>
<td>$S_8$</td>
<td>2930</td>
<td>8920</td>
<td>1907</td>
</tr>
</tbody>
</table>

The analysis results vary significantly depending on the modelled system being analysed. At most, the analysis discards 91.5% of all structurally possible DU-paths as being infeasible (system $S_6$, Config 3). On the other end of the scale, for system $S_7$, Config 3, only 3.8% of the DU-paths are discarded. Here, a DU-path being discarded should be interpreted as it is not found potentially feasible by the analysis of this approach, as the analysis does not in practice search for safely infeasible DU-paths, but for potentially feasible DU-paths.

For the systems and configurations that are analysable, the run-times of the analysis are shown in Table 6.3. As can be seen, the run-times range from between just under 2 seconds ($S_8$, Config 3) to well over 2 hours ($S_3$, Config 3). However, the UPPAAL-based test item derivation method is the more detailed of our two approaches, and as such results in a more complex analysis. As a consequence, merely roughly a third of the systems or system configurations can be analysed by the approach. In Figure 6.2 and Table 6.3, the non-analysable configurations are marked "(-, -)" and "-" respectively. The analysis of such a configuration suffers from a state space explosion problem, causing the simulation to run out of virtual memory when executed.

### 6.1.3 EOG-Based Test Item Derivation

The EOG-based approach (the DUANALYSIS algorithm) was evaluated with the exact same set of systems and configurations. Compared to the UPPAAL-based approach, the DUANALYSIS algorithm shows more consistent results.
6.1 Test Item Derivation Methods

### Table 6.4: EOG-based test item derivation evaluation results.

<table>
<thead>
<tr>
<th>Sys</th>
<th>Config 1</th>
<th>Config 2</th>
<th>Config 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>($M_{DU}$, $\frac{M_{DU}}{S_{DU}}$)</td>
<td>($M_{DU}$, $\frac{M_{DU}}{S_{DU}}$)</td>
<td>($M_{DU}$, $\frac{M_{DU}}{S_{DU}}$)</td>
</tr>
<tr>
<td>$S_1$</td>
<td>(157, 72.7%)</td>
<td>(148, 68.5%)</td>
<td>(140, 64.0%)</td>
</tr>
<tr>
<td>$S_2$</td>
<td>(147, 96.7%)</td>
<td>(164, 89.6%)</td>
<td></td>
</tr>
<tr>
<td>$S_3$</td>
<td>(28, 82.4%)</td>
<td>(21, 61.8%)</td>
<td>(24, 70.6%)</td>
</tr>
<tr>
<td>$S_4$</td>
<td>(35, 85.4%)</td>
<td>(35, 85.4%)</td>
<td>(25, 60.1%)</td>
</tr>
<tr>
<td>$S_5$</td>
<td>(204, 89.5%)</td>
<td>(196, 86.0%)</td>
<td>(180, 78.9%)</td>
</tr>
<tr>
<td>$S_6$</td>
<td>(122, 54.7%)</td>
<td>(119, 53.4%)</td>
<td>(27, 12.1%)</td>
</tr>
<tr>
<td>$S_7$</td>
<td>(132, 100%)</td>
<td>(132, 100%)</td>
<td>(132, 100%)</td>
</tr>
<tr>
<td>$S_8$</td>
<td>(26, 65.0%)</td>
<td>(33, 82.5%)</td>
<td>(19, 47.5%)</td>
</tr>
</tbody>
</table>

Table 6.5: EOG-based evaluation analysis run-times (in ms).

<table>
<thead>
<tr>
<th>Sys</th>
<th>Config 1</th>
<th>Config 2</th>
<th>Config 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S_1$</td>
<td>4423</td>
<td>690</td>
<td>13</td>
</tr>
<tr>
<td>$S_2$</td>
<td>-</td>
<td>4229</td>
<td>16</td>
</tr>
<tr>
<td>$S_3$</td>
<td>145</td>
<td>78</td>
<td>3</td>
</tr>
<tr>
<td>$S_4$</td>
<td>125</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>$S_5$</td>
<td>416</td>
<td>265</td>
<td>8</td>
</tr>
<tr>
<td>$S_6$</td>
<td>33</td>
<td>28</td>
<td>3</td>
</tr>
<tr>
<td>$S_7$</td>
<td>488</td>
<td>5932</td>
<td>6</td>
</tr>
<tr>
<td>$S_8$</td>
<td>8</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

(see Table 6.4). As previously stated, $M_{DU}$ in the table refers to the number of DU-paths in $Model_{DU}(M_S)$ (in this case, the model $M_S$ being the implicit EOG system representation), and $\frac{M_{DU}}{S_{DU}}$ describes the ratio between the number of DU-paths found feasible by our algorithm, and the combinatorially feasible DU-paths. Generally, a less complex scheduling results in more DU-paths being found infeasible (except for Config 1 and Config 2 of $S_8$ and Config 2 and Config 3 of $S_3$). Similarly, as shown in Table 6.5, a more complex scheduling renders a longer analysis run-time (the only exceptions here being Config 1 and Config 2 of $S_7$).

As with the previous approach, the main analysis result difference stems from the analysed systems. At most (for system $S_6$, Config 3), 87.9% of the DU-paths are discarded as being infeasible, but for some configurations (i.e., all system $S_7$ configurations), none of the structurally possible DU-paths are discarded. The analysis run-times (see Table 6.5) are significantly lower than in the previous approach, ranging from less than a millisecond in some configurations (Config 3, $S_8$ and Config 3, $S_4$) to nearly 6 seconds (Config 2, $S_7$).
6.1.4 Test Item Derivation Evaluation Discussion

The evaluations presented in this section suggest that our DU-path derivation methods are applicable for small multi-tasking RTSs. As for the precision of the analysis result, Figure 6.1 displays the percentage for all configurations analysable with both test item derivation methods. For all cases, more DU-paths are discarded by the UPPAAL-based approach. Hence, in our evaluations, the UPPAAL-based approach invariably produces a more exact result than the EOG-based approach. In the cases where the CO\textsuperscript{ER} analysis could be performed, it improved over the EOG results by 28.8% in the average case, and 15.8% in the median case.

The obvious drawback of the UPPAAL-based approach is the high complexity of its analysis. Out of our 24 different configurations, the UPPAAL-based approach was merely able to derive a result from 9. For fairness sake, the CO\textsuperscript{ER} tool was not initially intended for test item derivation for structural testing, but for specification model-based functional testing. We however believe that with a more detailed modeling it will be possible to reduce the state
6.1 Test Item Derivation Methods

\[
\begin{align*}
WCET_X & = WCET \\
BCET_X & = BCET + (1 - \frac{X}{100})(WCET - BCET) \\
u_{\text{max}}X & = u_{\text{max}} \\
u_{\text{min}}X & = u_{\text{min}} + (1 - \frac{X}{100})(u_{\text{max}} - u_{\text{min}}) \\
d_{\text{max}}X & = d_{\text{max}} \\
d_{\text{min}}X & = d_{\text{min}} + (1 - \frac{X}{100})(d_{\text{max}} - d_{\text{min}}) \\
d_{\text{rdMax}}X & = d_{\text{rdMax}}
\end{align*}
\]

Figure 6.2: Rules for jitter reduction by \( X \)\%.

As a high level of execution time jitter is generally considered to impair system testability [99], we were interested to see what would happen with the number of feasible DU-paths and the analysis run-time as the jitter in the system was decreased. Specifically, in preemptive RTSs, an increased level of jitter increases the number of task interleaving sequences in the system (i.e., increases the number of paths in the system EOG) [77, 99]. We focused on the EOG-based approach when investigating the implications of varying the level of execution time jitter, as a jitter reduction in the UPPAAL-based approach needs to be represented in the internal control flow of the system’s tasks, something which is disregarded in the EOG-based approach. Further, at this stage, we focused on the larger systems in terms of number of statements and analysis run-time (\( S_1, S_2, \) and \( S_7 \)), as well as the system with the highest level of execution time jitter (\( S_5 \)). For the purpose of this evaluation, the jitter in the system was reduced according to the rules given in Figure 6.2. Intuitively, as \( X \) approaches 100, \( BCET \) approaches \( WCET \), \( u_{\text{min}} \) approaches \( u_{\text{max}} \), and \( d_{\text{min}} \) approaches \( d_{\text{max}} \).
Chapter 6. Evaluation

Figure 6.3: The effect of jitter reduction on the number of DU-paths.

The results of the reduced jitter analysis in terms of DU-paths are shown in Figure 6.3. As suspected, a reduced jitter results in a reduced, or, in one case, unchanged, number of potentially feasible DU-paths in the system. This reduction was most significant in the system with the highest level of execution time jitter. At a full execution time jitter, the analysis of Config 1 of system $S_5$ rendered 204 potentially feasible DU-paths. Without jitter (resulting in a deterministic scheduling of tasks), the same configuration rendered 135 potentially feasible DU-paths, a reduction of 34%. The other configurations, however, showed less significant reductions, ranging from 0% to 22%.

The analysis run-time, on the other hand, was more significantly reduced (see Figure 6.4). All configurations showed a reduction of analysis time of over 75%, with the three largest systems showing an analysis run-time reduction of over 90%. Surprisingly, the system configuration with the highest level of execution time jitter shows the smallest analysis run-time reduction when eliminating the jitter. On the other hand, this could be explained by the number of DU-paths found feasible in this configuration, which at a full jitter reduction...
Figure 6.4: The effect of jitter reduction on the run-time of the DUANALYSIS algorithm.
still is the second largest of all configurations. In addition, the run-time of this configuration at a full execution time jitter is more than ten times smaller than the run-time of the next fastest configuration.

Considering the results in Figure 6.3 and 6.4, it should be noted that, given our specific test criterion (the all-DU-path coverage criterion), and these two specific system configurations, the number of test items to test do not dramatically decrease when significantly reducing (or even eliminating) the execution time jitter. On the other hand, the analysis performed for finding these test items is made less complex. In addition, the scheduling determinism caused by zero jitter is highly beneficial for system behaviour predictability and reproducibility of bugs.

6.2 Time Machine Case Studies

The Time Machine is the tool implementation of the Deterministic Replay method proposed in this thesis for test item monitoring, and for extracting the set of exercised test items during the execution of a test suite \( Q, \text{Test}_{DU}(S,Q) \). Throughout the course of our work, we have successfully implemented the Time Machine for replay-based monitoring and debugging on a number of different platforms, including:

1. Processors, among them the 8/16 bit CISC Hitachi H8, the 32 bit RISC NEC V850, and the 32bit CISC Intel Pentium.

2. Compilers, among them GNU GCC (Hitachi H8, Intel x86), IAR Systems (Hitachi H8, NEC V850).

3. Real-time operating systems, among them VxWorks, and Asterix [104].

In this section, we describe aspects of the implementation of the Time Machine and its three subcomponents (1) The Recorder, (2) The Historian, and (3) The Time Traveler learnt from three case studies. Specifically, we present the adaptations required for using the Time Machine with a small experimental open-source real-time kernel, as well as with full-scale commercial industrial applications. For a more detailed description of these case studies, see [94] and [92].
6.2 Time Machine Case Studies

6.2.1 Introduction

For our Time Machine case studies, we had the opportunity to work with three different types of systems, all of which operate under hard real-time requirements:

1. A small real-time application running on top of the Asterix open source RTOS [104].
2. A state-of-the-art robotics system for industrial automation running on top of the commercial VxWorks RTOS.
3. A state-of-the-art military aircraft radar warning system, also running on top of VxWorks.

The systems were chosen as subjects based on their high level of software- and overall technical complexity. In addition, the systems operate (or, in the Asterix case, are intended to operate) in safety-critical and high availability environments, where failures might be very costly, making system validation and verification even more important.

The Asterix RTOS

The Asterix real-time kernel [104] is a small RTOS developed at the Department of Computer Science and Electronics at Mälardalens University. The Asterix RTOS comes together with a highly configurable compiling distributed real-time kernel, a single-shot task model supporting state-of-the-art scheduling theory [8], wait- and lock-free interprocess communication and support for monitoring and replay. All these features are collected in an open-source framework, called the Asterix framework. For the purpose of this thesis, the Asterix real-time kernel is configured according to the system model described in Section 1.2.

In short, the Asterix replay system is an implementation of the Time Machine, consisting of a set of generic software probes, analysis tools and system add-ons. In our work, we have implemented replay functionality in the 16-bit NEC V850 and the 8-bit Renesas H8/300 microcontroller ports of Asterix.

The VxWorks RTOS and Applications

To validate our ideas in an industrial setting, we had the opportunity to work with a robotics system that is state-of-the-art in industrial manufacturing automation built by ABB Robotics, a world leading manufacturer of industrial
robots for industrial automation, especially for car manufacturing. Further, we were able to work with a military aircraft radar system built by SAAB Avionics, a major supplier of electronic warfare technology on the international market. SAAB Avionics products can for example be found in the Swedish fighter aircraft Gripen, the American F-15 and the NH-90 helicopter. Similar for both systems is that they run on top of the commercial VxWorks real-time operating system$^2$. 

6.2.2 Implementing The Recorder

The Recorder is basically a collection of kernel and application software probes for recording execution ordering- and data flow information, in order to facilitate replay of the system. In the Recorder, the storage space for monitored data is divided into two parts: the execution ordering buffer and a set of data flow buffers.

In our case study implementations, we have made use of cyclic buffers for both execution ordering and data flow.

The Asterix Recorder

In the Asterix kernel there are four different entries into the kernel, causing a task switch: timertick, return, irq and yield. The first one, timertick is called when a timer interrupt is generated. The second, return, is called each time a task terminates and wants to hand over the control to the kernel. The third, irq, is a task switch generated by an external interrupt and the forth, yield, is a task switch generated by a semaphore race. In addition to these event types, EvT can also indicate a missed deadline. Furthermore, the unique marker, C, is a 8-bit or 16-bit register checksum (depending on the hardware platform).

The VxWorks Recorder

In order to avoid the somewhat impossible mission of analyzing and instrumenting an approximate of 2.5 million lines of code in an academic project, we focused on instrumenting five central parts in the Robotics system:

- The operating system.
- The application's operating system abstraction layer.

$^2$www.windriver.com
• The inter-process communication abstraction layer.

• The peripheral I/O abstraction layer.

• The state preserving structures of each individual task.

A common denominator for the Robotics system and the Avionics systems is that they both run on top of the Wind River VxWorks RTOS. Therefore, we instrumented the VxWorks kernel entries, including semaphore semTake and semGive operations, message queue blocking operations msgQSend and msgQReceive, task sleep function taskDelay as well as preemptive scheduling decisions. In the VxWorks Recorder, these correspond to the EvT event types.

To instrument the blocking task delay-, semaphore- and message queue primitives, we added an instrumentation layer on top of the VxWorks system call API. This layer replaces the ordinary primitives with wrappers including the added functionality of instrumentation. However, in order to see which of the invoked calls actually lead to task interleavings, we need to be able to probe the scheduling mechanisms of the kernel. Fortunately, VxWorks provides a TaskSwitchHook, implemented as an empty callback function included in kernel mechanisms. This hook can be used for instrumentation purposes, making it possible to monitor and log sufficient information of each task switch in order to be able to reproduce it. Similarly, asynchronous events can be monitored.

In contrast to the execution ordering instrumentation, which is handled on the VxWorks RTOS level, some of the data flow instrumentation needs to be handled on the application level. Such data include static and global variables and structures holding information that can be altered during the process of system execution. To be able to reproduce interaction with an external context and inter-task communication, the peripheral I/O and the inter-task communication message queues are instrumented in two operating system abstraction layers. This solution gives the instrumentation a quality of transparency, making it less sensitive to changes in the application code.

A summarizing overview of system instrumentation can be viewed in Figure 6.5. In the figure, the gray area represents the instrumentation layer, which is slightly integrated into different parts of the RTOS and some application abstraction layers.

Since both the Robotics and the Avionics system run on top of VxWorks and the RTOS-level instrumentation is application independent, the instrumentation of the Avionics execution ordering was implemented in a very similar fashion. However, one aspect had to be taken into account. In contrast to the
Robotics system, the Avionics system was implemented in Ada. As the Ada runtime environment is added as a layer on top of VxWorks, this layer had to be altered in order to be able to monitor rendezvous and other Ada synchronization mechanisms.

### 6.2.3 Implementing The Historian

The Historian is the off-line tool that analyses the recordings from the recorder, and prepares the Time Machine for replay. This is basically done by sorting the execution ordering- and data flow buffers, creating breakpoints, and, if necessary, finding a consistent starting point for replay.

#### The Asterix Historian

The debugger used in the Asterix replay system is the GNU source-level C-debugger, gdb [89]. The current version of the Asterix replay system uses gdb-4.17 configured with a h8300-renesas-hms microcontroller software simulator (which is included in the standard gdb distribution). Gdb supports script command files, which means that the breakpoints and settings of system variables can be defined as macros in script files, automatically generated by the Asterix off-line analysis tool. A gdb macro breakpoint can have different commands attached to it and will follow the syntax seen in Figure 6.6.

As previously stated, an execution ordering buffer entry is made up of a 7-tuple and represents an event in the history of the recorded execution. To reproduce these events, each preemptive (asynchronous) entry in the buffer has to be transformed into a conditional breakpoint macro. This is done in
break [BREAKPOINT] if [BREAK_CONDITION]
commands
[COMMAND_1]
[COMMAND_2]
.
[COMMAND_N]
end

Figure 6.6: Generic structure of a conditional breakpoint with commands in gdb.

```c
break *0x8845 if (sp==0x91ae &&
register_checksum==0xc2)
```

Figure 6.7: Specific conditional breakpoint example.

a number of steps. When the control flow buffer and the data flow buffers are uploaded from the recorded execution, they are represented as a bunch of unsorted raw data. Therefore, the first step is to separate the execution ordering buffer and the different data flow buffers from each other. Next, they are sorted in a chronological order. After that, each entry in the execution ordering buffer is translated into a gdb conditional breakpoint macro.

This conditional breakpoint will be slightly different for different types of events, but the principle is the same. For a fictional event \( e_1 \):

\[
\{12(Tk), 235(STk), 0x8845(PC), 0x91ae(SP), 0xc2(C), 3(Ev), tt(EvT)\}
\]

the corresponding breakpoint is depicted in Figure 6.7.

The break condition is there to make sure that the execution is not halted at the wrong iteration of a loop or at the wrong instance of a recursive call, where the same program counter value is visited a number of times, but almost never with the same stack pointer or register checksum values.

To make sure that the system behaves in exactly the same way as in the recorded run, we must see to it that tasks are invoked at the right clock ticks. This is done by setting the task’s length-to-next-period (LDP)-variable [2] at the different breakpoints. In the conditional breakpoint macro, this is shown in Figure 6.8. Here, \( N \) is the number of tasks - 1 and \( Lx \) is the LDP calculated
break *0x8845 if (sp==0x91ae &&
            register_checksum==0xc2)
commands
set tasklist[0].LDP = L0
set tasklist[1].LDP = L1
.
.
set tasklist[3].LDP = 0
.
.
set tasklist[N].LDP = LN
jump timertick
end

Figure 6.8: Conditional breakpoint macro

for task x. At each breakpoint, the \textit{LDP} for all tasks is calculated and set.

Creating a replay basis of the data flow buffers is a lot easier than creating
the conditional breakpoints from the execution ordering buffer. The data flow
buffers are just a set of sorted data arrays, which are written back in the system
by gdb command files. In order to facilitate replay, the Historian writes back
the data to the same storage in memory where it once was recorded. Then
the index is set to the initial value and when the replay execution starts, the
same kernel mechanisms that once stored the data can be used to retrieve data
instead. In other words, this time the data is read from the recording area into
user tasks variables instead of the other way round.

The last step of the off-line analysis that has to be performed before we
can start reproducing the execution by deterministic replay is the task of setting
a valid replay starting point. Consider the recorded execution ordering in
Figure 6.9, representing an execution history.

The first task executing at starting point \(s\) in our execution history is task B.
After task B has terminated, task A is scheduled. However, we know nothing of
this invocation of task A. Is it a resumed preempted execution, or is it an initial
task start? We don’t know. The easiest way around this problem is to find the
first point in the execution history where no user-tasks are in the ready-queue.
We could call this an idle point. At such a point, no task will be in a preempted
state and therefor all task executions will be reproducible. In Figure 6.10, the
first idle point in the recorded execution is identified and set to starting point \(s\).
The VxWorks Historian

In the VxWorks case, we differ between local starting points, which are suitable task-level starting points, and global starting points, which are suitable system-level starting points [46]. To be able to perform a replay of the reference execution, the data flow and the execution ordering logs need to be correlated. For example, a local starting point \( p \) is made up of a log point where the execution ordering and data flow entries for that task coincide.

Consider, for instance, the example code in Figure 6.11. The potentially blocking system call \( \text{msgQReceive} \) is followed by a software probe, storing (or retrieving) the contents of the received message. In addition, the value of
TaskA()
{
    int gvar = 0;
    while (FOREVER)
    {
        msgQReceive(msgQId, &msg);
        probe(MSG_PROBE);
        ...
        subr(gvar);
        gvar++;  
        ...
        semTake(sem);
        ...
    }
}

Figure 6.11: Probed Code Example.

the global variable gvar should be stored due to the fact that it helps define the state of the task in each iteration of the loop. If the global variable is not stored, the replay execution will always start with a gvar-value of zero, corrupting the correlation between data- and execution ordering of the replay.

In the case of an empty message queue, the task will make a transition to a waiting state and thus cause a task switch, which will be logged as an entry in the execution ordering log. When a message arrives to the queue, the task will be awakened and the software probe will execute, storing the received message in the data flow log. This is an example of a situation where control- and data flow log entries coincide, producing a potential local starting point for this task. On the other hand, consider the next potentially blocking system call, semTake. When executed, if the semaphore is taken, this call will cause a running- to waiting-state transition for this task as well. This transition will be stored in the execution ordering log and will be essential for the deterministic reproduction of the execution. However, since no data flow is stored in conjunction with this, the task state will not be restored during the replay execution at this location and the execution ordering log entry is not part of the set of potential local starting points.

By viewing each task in the replay execution as a fairly autonomous and isolated entity, we ensure that the state consistency of the global starting point
does not depend on any irreproducible communication. In our implementation, a message sent by a task $A$ to a subsequent task $B$ is logged using a data flow probe in the execution of task $B$. Using this approach, tasks $A$ and $B$ operate in isolated environments during the replay execution and task $B$ needs not rely on the correct deliverance of messages from task $A$ in order to be reproduced deterministically.

### 6.2.4 Implementing The Time Traveler

As the replay execution is started, the Time Traveler interacts with the debugger and, given the information provided by the Historian and the breakpoints visited in the program, allows recreation of the system state for any given time in the scope of the replay execution [105].

**The Asterix Time Traveler**

For Asterix, once all gdb command files are created by the Asterix off-line analysis tool, gdb can be started. A static initiation file is read into gdb, which in turn reads all dynamic command files created by the Historian. This sets all conditional breakpoints in the system. The breakpoints causing the simulated events are stepped through and the replay execution will run exactly the same path as the recorded run did.

**The VxWorks Time Traveler**

Even though the Tornado 2 Integrated Development Environment (IDE) features a VxWorks-level simulator, both recording and replay execution are performed on the actual target system. The VxWorks replay execution is initiated by breakpoints being set at all potential local starting points in the code of the system. In VxWorks, this is done by issuing breakpoint commands to the on-target debug task. Once these breakpoints are set, the system application can be started and executed up until all tasks have hit their first breakpoint. This will leave the entire application in a suspended state, from which we are able to chose, from local starting points in the log, which task to release for execution first. The chosen task is released and deterministically executed up until its next breakpointed location of task interleaving (blocking system call, preemption or interrupt) in the log. An asynchronous event occurrence is always checked for correctness with respect to its logged unique marker (e.g., to minimize the risk that a task is preempted during the wrong iteration of a loop during replay). At
each new event breakpoint a new selection is made, based on the log sequence, regarding which task to release for execution next.

6.2.5 IDE Integration

Depending on the purpose of replay, the integration of the technology in an IDE may face different requirements. We have currently implemented three different IDE integrations: gdb, Embedded Workbench, and Tornado 2.

Asterix IDE Integration

The original gdb-based version of the Asterix Time Machine was based on a command-line interface and was not the most user-friendly application. In order to improve applicability, we integrated the Asterix Time Machine in the IAR Embedded Workbench (EW) IDE\(^3\), both for H8/300 and NEC V850 ports (see Figure 6.12). EW is an IDE for small, embedded applications, and includes a text editor, a compiler, a debugger, and a cycle accurate instruction-level simulator. The Time Machine was made into an interactive EW plugin, with an execution timeline as primary user interface (see bottom left window of Figure 6.12). Regardless of how the replay log was collected, the replay execution is performed on the instruction-level simulator on the host machine. Conditional breakpoints, hidden from the user, make sure of the progress of the replay. As the replay execution progresses, this is shown on the execution timeline. Furthermore, a double-click on any job instance on the in the timeline will cause the replay to execute (or start over and re-execute) up until that point. This interactive solution is primarily developed for concurrent debugging purposes, but could naturally also be used for test instrumentation.

VxWorks IDE Integration

WindRiver Tornado 2\(^4\) is an IDE for VxWorks applications, including a text editor, a project/workspace handler, a gcc compiler, a target simulator and a gdb-based debugger, capable of performing multi-threaded debugging with no requirements on deterministic reproduction of execution orderings. We use the Tornado 2 IDE as this is the standard IDE for developing VxWorks real-time applications.

\(^3\)www.iar.com
\(^4\)www.windriver.com
Debugging in the Tornado 2 environment is performed by means of remote debugging. That is, the debugging session is executed on-target and controlled from the development environment via communication with an on-target task.

To provide tool vendors with a possibility to create their own add-ons to the Tornado 2 IDE, a programming interface is provided. The Wind River Tool Exchange (WTX) API enables developers to create tools that communicate and interact directly with the VxWorks target server. For the implementation of our Time Machine system, the Historian and the Time Traveler were integrated and implemented as a WTX tool, able to connect with a running target server and to force a replay execution upon the system running on that target. The structure of the Tornado 2 IDE, the Time Machine and the target system interactions is depicted in Figure 6.13.

To handle the on-target debugging operation, VxWorks provides a dedicated task, the Wdb task. This task handles requests of breakpoints, watches, single-stepping and other debugging functions from the IDE. These functions
are used by the Time Traveler via the WTX interface and the target server in order to control the replay execution.

Breakpoints play a central role in the interaction between the time machine and the target system. Breakpoints are set at each point of possible task interleaving and as they are encountered in the target system, their occurrence is echoed from the Wdb task through the WTX and into the event handler of the Time Traveler. Based on the individual features of each breakpoint, the state of the replay execution can be deduced and the Time Traveler replay engine will force the desired behavior on the target system.

### 6.2.6 Instrumentation Load

One of the main reasons for the industrial Time Machine case studies was to investigate whether the overhead incorporated by system-level execution ordering- and data flow instrumentation was acceptable in a full-scale complex industrial real-time application (in our case, a robotic control system from ABB Robotics and an avionic radar warning system from SAAB Avionics). In order to resolve this issue, we performed benchmarks measuring instrumentation mechanism CPU load and memory usage in both systems. Since the instrumentation is yet to be optimized and the benchmark tests are performed
6.2 Time Machine Case Studies

<table>
<thead>
<tr>
<th>ABB Robotics system</th>
<th>SAAB Avionics system</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Ordering</strong></td>
<td><strong>Data Flow</strong></td>
</tr>
<tr>
<td>CPU</td>
<td>Mem</td>
</tr>
<tr>
<td>0.05%</td>
<td>72.0 kbps</td>
</tr>
<tr>
<td><strong>Total instrumentation load</strong></td>
<td><strong>Total instrumentation load</strong></td>
</tr>
<tr>
<td>CPU</td>
<td>Mem</td>
</tr>
<tr>
<td>3.55%</td>
<td>2.19 MBps</td>
</tr>
</tbody>
</table>

Table 6.6: Replay instrumentation processor and memory bandwidth utilization in the ABB Robotics and the SAAB Avionics case study systems.

under worst-case scenario conditions, many results might be rather pessimistic.

In the ABB Robotics system case, we timestamped the entry and the exit of all instrumentation and recording mechanisms. This gave us a possibility of extracting the execution time of the software probes. In addition, we instrumented the frequency and recording size of the data flow instrumentation mechanisms. The system was then executed in a simulated environment under heavy user interaction. The results of the ABB Robotics system evaluations are shown in the left column of Table 6.6.

As the data instrumentation in the Avionics system is performed solely on message queues, the data flow benchmark was made in a per-queue fashion. As with the robotics system, the evaluations were performed on the instrumented system in a simulated environment. The results of the SAAB Avionics system evaluations are shown in the right column of Table 6.6.

6.2.7 Time Machine Case Studies Discussion

In this chapter, we presented the experiences from the implementation of the Time Machine on three different platforms. We will end this chapter with a discussion of lessons learned from numerous hours spent designing and implementing the Time Machine replay technology on several hardware platforms and real-time operating systems.

- **Full Kernel Source Code Access**

  Asterix is an open-source compiling real-time kernel, whereas VxWorks is a pre-compiled library linked together with the application tasks in order to form a target executable. From a replay point of view, full source code access is an obvious advantage when probing the kernel code, especially significant for execution ordering instrumentation. On kernels
like VxWorks, we rely on kernel hooks (i.e., pre-declared instrumentation functions, defined by the user and called from within the kernel) for instrumentation.

An even more disabling disadvantage of is the lack of knowledge of the inner structure of the kernel. E.g., investigating the ready queue, or which semaphores are taken at a certain point, calls for advanced and time consuming reverse engineering. Hence, introducing replay in a real-time kernel should preferably be done in cooperation with the real-time kernel developers.

- **Task Model**

  Asterix uses a single-shot task model [8], in which tasks are terminated after each task instance. Hence, each task instance is started with an entirely new scope and the system state is solely dependent on static or global variables. VxWorks, on the other hand, uses a more traditional task model, where tasks are implemented using infinite loops and each iteration in the loop is guarded by a blocking system call (see, e.g., Figure 6.11). Hence, data flow instrumentation in the VxWorks case has to consider local state-preserving variables as well. Another aspect of this is that in order for the register checksums to operate properly on non-terminating tasks, it is required that tasks periodically reset their registers each iteration. Concerning stack checksums, the compiler-generated code will always initialise the stack space to zero.

  In addition, the Asterix tasks have only one distinct entry point and one distinct exit point. When no tasks are active, the system is completely idle. This significantly decreases the effort of finding consistent starting points for replay.

- **Replay-Based Debugging versus Test Item Instrumentation**

  The Time Machine case studies were initially aimed towards concurrent real-time system debugging [94]. In the scope of this thesis, we use replay for avoiding probe effects in test item instrumentation. Regardless of the objective, the underlying technology is identical in both cases. The usage of the techniques are however different. Replay Debugging poses high demands on interactivity, since debugging is a highly interactive process. Details of an execution is investigated in a repetitive fashion until the causes of a failure are revealed. Replay-based test item instrumentation is more of a batch job and calls for a higher level of automatisation. All implemented versions of the Time Machine could
do automated test item instrumentation at some level, but a well-defined plug-in API (e.g., that of IAR EW) would significantly help the automation process.

Concerning the instrumentation load, the main thing one notices when comparing the industrial studies is the difference between the data flow memory instrumentation load in the robotics and the avionics system. The robotics system, with its memory load of 2.18 MBps, is nearly 200 times more heavily loaded by the data flow instrumentation than the avionics system. The main reason for this is that, in the data flow instrumentation, the ABB Robotics system state preserving structures for each task are also probed. With the downside of causing a significantly heavier load, this facilitates an execution replay that need not be started from system start-up (see The VxWorks Historian, Section 6.2.3). In the SAAB Avionics case, only the queue input to the tasks are probed, and each replay execution has to start from system start-up.

Naturally, the applicability of the approach in a general system depends on how heavily loaded the system is prior to the replay instrumentation. Very high utilization systems might struggle to fit in another 4% of CPU utilization. Vendors of both systems in our case study, however, conceded that the instrumentation perturbation was well within an acceptable range [94].

6.3 Checksums

Our method for pinpointing asynchronous interrupts is briefly mentioned in Chapter 4 and thoroughly described in Appendix C. As an approximation of the execution context is used, and the method is not exact, we performed a number of tests where we evaluated the approximation accuracy and the level of perturbation of our method. For the accuracy tests, a tailor-made program $P$ was written, executed and preempted by an interrupt $I$. Our test platform was the IAR EW commercial IDE for embedded systems. We used the NEC V850 (a RISC architecture processor) version of EW and our tests were performed using the Deterministic Replay implementation on the EW V850 target simulator. Using the cycle counter of the simulator, we were able to simulate interrupts after a fixed number of clock cycles. In our experiments, both $P$ and $I$ were implemented as real-time tasks running on top of the Asterix real-time operating system [104]. By varying the time $t$ (or rather the number of clock cycles)...
cycles during $t$), we can cause $I$ to preempt $P$ at different states of its execution (see Figure 6.14).

The perturbation tests were performed on the ABB Robotics system described in Section 6.2. In a way, it would be desirable to use the full-scale industrial application for the checksum accuracy tests as well. However, using a tailor-made program instead, it is possible to force execution scenarios upon the method in such a way that it is tested more thoroughly. In addition, due to the current implementations, applications running on top of Asterix are significantly more manageable with respect to interactive debugging. This is an invaluable property when examining and comparing program states during run-time.

### 6.3.1 Approximation Accuracy

To be able to test the accuracy of the context checksum methods under different circumstances, the program $P$ was written such that its properties could be easily changed. Over all tests, however, $P$ preserved its basic structure (depicted in Figure 6.15). In short, $P$ consists of a main function and four subroutines $sr_1$..$sr_4$. In a loop with 100 iterations, $P$ main calls $sr_1$, which in turn performs some calculations and calls $sr_2$. The $sr_2$ subroutine, in turn, calls $sr_3$ in two loop iterations. From $sr_3$, $sr_4$ is called four times. In its structure, $sr_4$ has a loop that iterates eight times.

On the lowest subroutine level, this yields 6400 iterations ($100 \ast 1 \ast 2 \ast 4 \ast 8$) in the $sr_4$ loop for each execution of $P$, meaning that each instruction in this loop is visited 6400 times. As a consequence, in an execution of $P$, every instruction of the $sr_4$ loop will result in 6400 states in $E_P$. All of these
states will have identical program counter values, but different environments. Hence, $P$ is well suited for examination of how well our method will perform regarding differentiation between states with identical program counter values. In our tests, we used four different allocation schemes in order to investigate the performance of our method. These schemes were modeled such that the allocation of variables were placed in different parts of the execution context:

1. In the **Stack Allocation 1** scheme, all variables (loop counters and calculation variables) were allocated on the stack of each subroutine. No parameters were passed.

2. The **Stack Allocation 2** scheme is allocated like Stack Allocation 1, but also passed parameters explicitly between subroutines.

3. In the **Heap Allocation 1** scheme, all variables were allocated globally. No parameters were passed.

4. In **Heap Allocation 2**, all variables were allocated globally and parameters were passed between subroutines.

Each of these schemes were tested using the register checksum, the stack checksum and the partial stack checksum. In our test, we used the scope of the current subroutine as a delimiter of the partial stack interval (corresponding to the $[SP, BP]$ interval in Figure C.1). All in all, this yielded 12 different test scenarios. These scenarios were tested by executing $P$ during $t$ time units. At time $t$, $P$ is preempted by an interrupt and the unique marker and the program counter are sampled. Then, $P$ is deterministically re-executed until the program counter and the unique marker matches those that were sampled. At this point, we compare the current state of execution with the original interrupt state by comparing the values of a set of globally defined loop counters. If the states match, we consider the test successful. By varying $t$, we can cause the
interrupt to preempt the program at different states each time. In our tests, we started at a $t$ value of 10000 clock cycles and in increments of 200, we raised it to 18000 cycles. This produced 41 test cases in each of the 12 test scenarios. The reason for the sparse number of tests is that they had to be performed by hand. Each test case yields a binary outcome (either the interrupt is successfully pinpointed, or it is not).

In Table 6.7, the results of the accuracy simulations are shown. Naturally, since the partial stack checksum and the stack checksum are complementary techniques to the register checksum, these always exhibit a better accuracy. If all variables are allocated on the stack, the stack checksum techniques will outperform the register checksum techniques by far. However, if all variables are allocated on the heap, the difference is not that significant. It should also be noted that a checksum of parts of the stack in many cases performs nearly as well as a full stack checksum.

### 6.3.2 Perturbation

As the unique marker checksums need to be sampled during run-time at the occurrence of an interrupt, this imposes a perturbation to the execution of the context switch. However, contrary to the perturbation of the software instruction counter (SIC) [65], further discussed in Chapter 7, the size of this perturbation is not proportional to the number of branches in the code, but to the number of interrupts in an execution.

The perturbation tests were performed on the ABB Robotics system described in Section 6.2. While letting the system perform some robot arm movements, we sampled the unique markers at interrupt occurrences as well as the execution time of the unique marker code. The upper and lower boundaries of the instrumentation execution time is presented in Table 6.8. The instrumentation jitter of the stack checksum execution is several magnitudes larger than that of the register checksum. The alterations in stack checksum execu-
6.3 Checksums

<table>
<thead>
<tr>
<th></th>
<th>Register checksum</th>
<th>Stack checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCET</td>
<td>1.15 µs</td>
<td>0.11 ms</td>
</tr>
<tr>
<td>BCET</td>
<td>0.37 µs</td>
<td>0.40 µs</td>
</tr>
<tr>
<td>Overall perturbation</td>
<td>0.002 – 0.007 %</td>
<td>0.003 – 0.37%</td>
</tr>
</tbody>
</table>

Table 6.8: Perturbation levels for stack- and register checksum.

... and the register checksum execution time alterations are due to cache effects (since the registers are sampled from the task control blocks rather than the actual hardware registers).

Regarding the level of perturbation, this could be compared with that of the SIC [65], which requires approximately 10 % of the overall CPU utilization.

6.3.3 Checksum Evaluation Discussion

When considering the results of this evaluation, we would again like to stress that the program \( P \) used for this evaluation was designed to be malignant to our method. For example, both heap allocations assume that all variables affecting the program flow of control are allocated globally, a practice neither recommended nor common in real systems.

However, as the checksum markers proposed in this chapter for reproducing asynchronous events are approximate, and not always truly unique, we face a problem of handling the situations when the markers ambiguously pinpoint program locations. This basically occurs when the execution context of the location \( l_i \) of an interrupt \( i \) generates the same checksum as another location \( l_2 \), such that \( l_i \) and \( l_2 \) are on the same PC address and \( l_2 \) predates \( l_i \) in the execution. Luckily, this problem can be handled in a number of ways:

First, we believe that the precision of the checksum markers can be significantly improved by combining them with the system call markers described in Section C.2.3, and with debugger cycle counters\(^6\), by mapping the recorded event timestamps to small cycle counter intervals in which the event is feasible. As for now, this is listed as future work. Second, if the ambiguity is discovered (by detecting dissimilarities between the reference and the replay execution), the replay execution can be reset and restarted with an added skip count on \( l_2 \). Third, in the worst case, the Determinstic Replay method can be facilitated using hardware- or software instruction counters [6, 15, 48, 52, 65], however

\(^6\)See, e.g., www.iar.com
inherit the drawbacks inherent to these methods.

6.4 Summary

This chapter presented a collection of experimental evaluation results from the techniques proposed in this thesis. These experiments included an evaluation of the test item derivation methods with respect to analysis precision, run-time and sensitivity to jitter variation, which showed that the UPPAAL-based method produces a less pessimistic result than the EOG-based method at the cost of a significantly higher run-time complexity. Further, a set of case study reports, and an evaluation of the Time Machine with respect to instrumentation mechanism CPU load and memory usage was presented. The case studies showed that the Time Machine is applicable in an industrial setting, but also that the hands-on effort of instrumenting the application can be quite significant. In addition, the chapter presented an evaluation of the approximate checksum method for pinpointing asynchronous events. This evaluation showed that the method in some cases is unsuccessful in pinpointing these events correctly. Hence, some ideas on improving the technique was also presented.
Chapter 7

Related Work

Naturally, the problems considered in this thesis have not gone by undetected or uninvestigated. This section describes related work performed and published within structural RTS testing (Section 7.1), and monitoring for testing (Section 7.2). Furthermore, we relate and compare previous efforts to our own work with respect to similarities and differences.

7.1 Structural Testing

Testing based on structural test criteria focuses on the actual software implementation and different inherent aspects of its structure. Classic structural test criteria (e.g., statement coverage, edge coverage, and all-DU-path coverage) are based on graph representations of the software implementations, such as control flow graphs or data flow graphs and are thoroughly examined and applied [21, 22, 107]. Related to our work, Hong et al. [43] showed how to perform data flow-based test case generation using model checking temporal logic properties of program flow graphs.

As for structural testing on system-level, most effort has been put into testing of non-preemptive systems (or testing of preemptive systems without considering the preemptive aspects of the system). However, a few methods have also considered the effects of asynchronous preemption when performing structural system-level testing. In the following two subsections, previously proposed non-preemptive system testing, and preemptive system testing techniques will be discussed, respectively.
7.1.1 Concurrent Systems Structural Testing

For structural testing of concurrent systems, many approaches combine the internal control flow structure of concurrent threads with the possible synchronizations between the threads. By doing this, a system-level control flow representation for structural testing can be achieved.

Wong et al. proposed a method for deriving test sequences from a concurrent program’s reachability graph (including synchronization selections) [111]. Similarly, Katayama et al. proposed a method for automatically deriving unit-level control flow graphs from a concurrent program, and their interaction sequences [51]. In their article, they also present an algorithm for deriving test cases in order to cover all possible interaction sequences. Furthermore, C. D. Yang et al. describe how to combine thread-level control flow graphs at synchronization points in order to build a concurrent program-level graph [112, 114, 113]. This graph is then used to automatically extract test items for all-uses and all-du-paths coverage testing, and to exercise them using “temporal testing” (a testing method where execution time is added to program segments in order to enforce certain paths).

R. D. Yang et al. use knowledge of the synchronization constructs of the Ada language and runtime environment in order to derive SYN-sequences of concurrent Ada programs [115]. These sequences are used for concurrent program-level path coverage, by (1) running tests on the program, (2) looking for the SYN-sequence, (3) examining coverage on SYN-sequence level, and (4) enforcing remaining paths by controlled execution.

7.1.2 Preemptive RTS Structural Testing

Although the major portion of preemptive RTS testing research has focused on testing of system timeliness [13, 18, 70], where the aim is to detect, enforce, and test the worst case response times of the system tasks, some efforts have been made in control- and data-flow-based testing of RTSs. Specifically, Thane [103] and Pettersson [77] defined and elaborated an algorithm for deriving a directed graph of all possible system-level control flow paths from a preemptive, FPS-scheduled set of task instances during a LCM (the Least Common Multiple of the period times of the tasks; the shortest time before the schedule periodically reappears). This graph is called the Execution Order Graph (EOG). The EOG has been proposed for usage in integration- and system-level testing by treating and testing each of the EOG paths in a system as a regular sequential program. However, the EOG paths do not provide suffi-
cient information in order to establish control flow coverage on a more detailed level than that of execution orderings.

7.1.3 Program Analysis for Concurrent System Testing

There are several concurrent program analysis methods not aimed directly towards testing that are of interest to our work. Chen et al. [17] survey methods for dependence analysis of concurrent programs, specifically different techniques for representing inter-task communication.

Masticola and Ryder [64], on the other hand, focus on the determination of non-concurrent sets of statements (i.e., statements that can never happen together). This type of analysis is helpful in concurrent system analysis in general, since it may be used to isolate parts of the concurrent program that are not actually concurrent.

Ranganath and Hatcliff [80] describe how to perform data flow analysis in concurrent Java programs using program slicing, and Knoop et al. [53] propose the use of bitvector analysis for performing data flow analysis of truly parallel programs.

7.1.4 Race Detection

Race Conditions are data flow-based software errors caused by synchronization defects in shared data accesses. In a journal article from 1992 [68], Netzer and Miller formalize and categorize race conditions into two different types of races:

- **General Races** General races occur in improperly synchronized programs intended to be completely deterministic. Such a program could for instance be a parallelization of a sequential program, where the parallel program is forced into the same semantics as the sequential program by means of explicit synchronization. If this synchronization fails, the program will exhibit general races.

- **Data Races** Data races describe the situation where critical-section accesses, intended to be atomic, are non-atomic due to erroneous use- or lack of mutual exclusion mechanisms in the program. In other words, a data race requires a possibility for a process or a task to execute a critical section access during the same time interval that another process or task accesses that very critical section.
Races in concurrent systems can be detected by either dynamic or static analysis. Dynamic methods generally focus on detecting actual or potential race conditions in specific system executions during or after the execution [20, 67, 72, 81], whereas static methods focus on detecting race conditions by investigating the source code of the system [66, 79, 84]. Since we focus on pre-deployment testing of non-deterministic systems, static detection of data races is the type of race detection most related to our work. Using these techniques, an over-approximation of all data races in the system can be derived and analysed based on the output of the race detection algorithm.

7.1.5 Model-Based Testing

Model-based testing is a growing area within the field of software verification. Basically, the main idea of model-based testing is to derive a model $M_S$ of a system $S$, where $M_S$ is an abstraction of $S$. I.e., $M_S$ only thoroughly represents the aspects of $S$ that are needed for testing purposes (given a certain test criterion). Once $M_S$ is (manually or automatically) derived, this model can be used for test case selection and generation with less effort than if the entire system was used.

Model-Based Testing of Concurrent and RTSs

Using modeling for RTS testing is established within the testing community, but most contributions to this field describe methods for performing functional testing [30, 39, 50, 58, 87, 96]. Most often, the system specification is modeled using some formal modeling tool or method. The model is then analysed with respect to different test criteria, and through searches of the possible state space, test suites (i.e., sets of test cases aimed at fulfilling a specific test criterion) are generated.

One inherent drawback of functional model-based testing is that the quality and thoroughness of the generated test suite are highly dependent of the quality and thoroughness of the system specification model. If the models are manually generated, the skills of the person modeling the system will have serious impact on the quality assuring abilities of the test suite generated by the model.

There are also model-based system-level testing techniques that are structural in nature, e.g., [54], but none to our knowledge that take the timing aspects of RTSs into account.
void bar()
{
    wait(LIST);
    list=removeItem(list);
    signal(LIST);
}

void foo()
{
    bool newlist;
    wait(LIST);
    if(list==NULL);
    {
        newlist=TRUE;
        list=createNewListHead();
    }
    else
    {
        newlist=FALSE;
    }
    signal(LIST);
    ... if(newlist)
    {
        wait(LIST);
        setName(list,'"1ST"');
        signal(LIST);
    }
}

(a) bar task
(b) foo task

Figure 7.1: Stale-value error.

7.1.6 Relation to Our Work

While the above methods for synchronous concurrent program testing ([51, 111, 112, 113, 114, 115]) excel in revealing ordering failures related to erroneous synchronizations, they fail in capturing the behaviour of asynchronous task interleaving in preemptive systems. Hence, they are not sufficient for structural system-level testing of preemptive RTSs. This is also the case for the model-based techniques with similar restrictions [54]

The contribution described in Section 3.3 is a direct extension of the work of Thane [103] and Pettersson [77], where the strict limitations on inter-task communication in the original work are loosened. Further, although no other approaches for concurrent system analysis consider time as a factor [17, 53, 64, 80], aspects of these could possibly be used in order to perform more efficient test item derivation.

It could be argued that static race detection could be used instead of our DU-path based testing, as both types of methods focus on, and find, the same types of concurrency-related bugs. There are, however, some problems with
static data race detection: First, since race detection generally produces false positives, a zero tolerance towards statically reported race conditions might be hard to enforce. Second, there are benign races that do not cause system failure, and opinions vary whether allowing such races are good software engineering practice. Regardless of the opinion, the truth of the matter is that many industrial control systems are built without eliminating races. Particularly, some types of systems, e.g., multitrate control systems, are intuitively implemented such that data races exist in the system. Furthermore, very small RTSs might not afford the execution time cost of explicit synchronization to protect shared variables. We consider this a solid ground for not only focusing on race detection and elimination, but also on testing the actual consequences of different shared variable access patterns. In addition, considering an example system (given in Figure 7.1), where two concurrently executing processes foo and bar perform protected accesses of a linked list list, one might imagine the following execution scenario: foo initially executes the first critical section, finding the list empty, thus creating a new list head and setting the local newlist boolean to FALSE. Next, foo is pre-empted by bar, which removes the new list head and terminates. The scenario is concluded by the resumption of foo, which tries to name the newly created (and removed) list head. This type of concurrency-bug is not actually a race condition, but a stale-value error [14], and can accordingly not be found by traditional race detection methods. Since models used for concurrent system testing ([30, 39, 50, 58, 87, 96]) generally focus on the functional aspects of the system, these approaches have a specific advantage over our methods. Not only can these models perform a valid test case selection with respect to a selected test criterion, many are also able to derive entire test suites that are guaranteed to fulfil the test criterion selected once executed on the system under test. Most often, such test suites are sets of test cases consisting of (timed) sequences of inputs that, when fed to the system, enforce a desired behaviour. Although functional and structural testing aim at finding fundamentally different types of errors, this is a current drawback of our approach. We further address this issue, and propose an idea for a solution, in Section 8.3.1.

7.2 Monitoring for Testing

Instrumentation and monitoring are absolute necessities for determining test progress and accuracy. In sequential software without real-time requirements on execution time, monitoring for testing can be performed by introducing
7.2 Monitoring for Testing

software probes at suitable locations in the source- or assembly code without perturbing the functional operation of the software. As described in this thesis, this is not the case for concurrent or RTSs.

7.2.1 Monitoring using Execution Replay

Monitoring (and also debugging) by means of execution replay was first proposed in 1987 by LeBlanc and Mellor-Crummey [60]. Their method was called *Instant Replay* and focused on logging the sequence of occurrence of significant events in parallel programs. This log sequence could then be used to reproduce the program behavior with respect to the correct ordering of events. The method was simple in that it did not include any support for reproduction of, e.g., interrupts or auxiliary input.

Many subsequent proposals have been extensions to the work of LeBlanc and Mellor-Crummey. Audenaert and Levroux have proposed methods for minimizing recordings [5] and replaying interrupts [6] and Chassin de Ker-gommeaux and Fagot included support for threads [16] in Instant Replay.

In addition, some methods have been proposed that use constructs in the run-time environment of specific programming languages, such as Ada [95] and Java [19], for execution recording and monitorable reproduction.

Replay Monitoring of RTSs

As for RTS replay-based monitoring, results have been very scarce and all contributions known to us are at least ten years old. In 1989, Banda and Volz proposed a method for reproducing the execution behavior of RTSs using a non-intrusive hardware-based recording and replay mechanism [9]. In addition to the dedicated monitoring hardware required, this method also called for specialized compiler support. Similarly, Tsai et al. proposed a non-intrusive hardware-based monitoring- and replay mechanism [106], craving a highly specialized hardware platform.

In contrast, Dodd and Ravishankar [25] proposed a software-based replay method for the distributed multiprocessor-node RTS HARTS. The logging is software-based in that it is performed by intrusive software probes. However, in order for the method to work, a dedicated processor needs to handle the monitor processing on each node.

In 2000, Thane and Hansson proposed the use of a method called *Deterministic Replay* for replay-based monitoring and debugging of distributed RTSs
The method was similar to previous replay methods in that it used a two-pass scheme (recording execution and replay execution). However, the method made use of software probes and required no specialized hardware to collect execution information. In addition, the method included a novel approach for using breakpoints to reproduce asynchronous events as well as a possibility of reproducing only parts of long-running executions. The contributions on monitoring described in this thesis are primarily implementations, extensions and generalizations of the original Deterministic Replay method.

Recently, execution replay tools have also made their debut on the commercial scene. ZealCore Embedded Solutions\(^1\) offer an execution replication tool (the BlackBox Replicator) that basically is a commercialization of the above work by Thane and Hansson [102]. After the proposal of our Time Machine tool, Green Hill Software Inc.\(^2\) released a commercial execution replay tool for embedded RTTs with the same name. Furthermore, the Silicon Valley-based Replay Solutions\(^3\) offer the ReplayDIRECTOR tool, primarily aimed at facilitating replay in entertainment software and computer game platforms.

### Reproduction of Asynchronous Events

A correct reproduction of the occurrence of asynchronous events is an absolute necessity for reproducing executions in preemptive systems. The main problem of monitoring asynchronous events is the difficulty of pin-pointing their exact location of occurrence (i.e., the correct machine code instruction with the correct data state). As we describe in this thesis, the program counter alone is not sufficient for marking the location of occurrence of the event. Machine code instructions can be revisited in loops or recursive calls, with different data states. For the purpose of pinpointing events, instruction counter techniques have been proposed.

The simplest unique marker technique is the instruction counter. In its basic form, this is a mechanism that counts machine code instructions as they are executed. When an asynchronous event occurs, the instruction counter is sampled along with the program counter in order to pinpoint the exact location of occurrence. In order to be able to count each instruction as it is executed, there is an obvious need for some kind of specialized hardware [15, 48]. Doing this in software would call for an instrumentation of the assembly code, adding an incrementing instruction to each instruction. Not only would this significantly

---

\(^1\)www.zealcore.com
\(^2\)www.ghs.com
\(^3\)www.replaysolutions.com
slow down overall system performance, it would also double the size of the code part of the program (assuming all instructions are of the same size).

The hardware required to count instructions is not a complicated mechanism, but a simple counter incremented on each instruction execution cycle. Even though this technique solves the problem of uniquely pinpointing interrupts, the method has drawbacks. One of the more significant problems is lack of the hardware needed in modern state-of-the-practice embedded microprocessors. Even though some of the larger processors available today have registers capable of performing instruction counting [1], this is no standard component in smaller or embedded processors. In addition, for many existing hardware platforms, there is reason to doubt the accuracy of the provided instruction counters [75]. Another problem is the sampling of the instruction counter. For applications using an Operating System (OS) or a Real-Time Operating System (RTOS), this may call for OS- or RTOS support. For example, consider an interrupt occurring at time $t_{evt}$. At time $t_{evt} + \delta$, the hardware instruction counter is sampled. Obviously, we will receive the instruction counter value of the latter, giving us a sampling error equal to the number of instructions executed during $\delta$. This might be problematic, especially if $\delta$ varies from time to time, due to interrupt service routine- or kernel jitter.

In 1989, Mellor-Crummey and LeBlanc proposed the use of a software instruction counter (SIC) [65], suitable for applications and systems running on top of platforms not equipped with the instruction counter hardware. As stated in the previous section, a software-based instruction counter performing the same task as a traditional hardware instruction counter would incur an intolerable overhead on the system. Thus, the software counterpart had to be much more restrictive when selecting upon which instructions to increment.

The SIC idea is based on the fact that only backward branches in a program can cause program counter values to be revisited. For instance, in a sequential program without backward branches, no instruction will be executed more than once. In such a system, the program counter is a unique marker, defining unique states in the execution. However, using structures such as loops, subroutines and recursive calls will require backward branches. Due to performance reasons, the implementation of the SIC not only increments on backward branches, but also on forward branches. In short, the SIC is implemented as a register-bound counter, requiring special compiler support. In addition, a platform-specific tool is used to instrument the machine code with incrementing instructions before each branch. According to the authors, the SIC incurs an execution overhead of approximately 10% in the instrumented programs.

The problem of getting the correct instruction counter value at sampling
time $t_{evt} + \delta$ is not solved using software instruction counters, although we are only interested in the number of backward branches rather than the number of instructions during $\delta$.

In 2002, Kim et. al. proposed an “enhanced software instruction counter” \cite{52}, which operated similarly to the original instruction counter. The enhanced software instruction counter managed to achieve an approximate 4 – 22% overhead reduction rate (i.e., a resulting CPU utilization overhead of about 9%) by analysing the machine code and separating deterministic scopes from non-deterministic scopes, and only instrumenting the non-deterministic scopes.

In 1994, Audenaert and Levrouw proposed the use of an approximate software instruction counter \cite{6}. Their method was called Interrupt Replay. In Interrupt Replay, the run-time logging of system interrupts is done by recording interrupt ID and an approximate SIC value. This version of SIC is incremented at entry- and exit points of interrupt service routines and can therefore not be used to pinpoint exact locations (program states) of occurrence of interrupts. As a consequence, Interrupt Replay is able to reproduce orderings of interrupts correctly, but the interrupts will not be reproduced at the correct instruction.

### 7.2.2 Hybrid and Hardware-Based Monitoring

Monitoring for testing does not have to be performed using software probes. Instrumentation can be performed in different ways, ranging from intrusive-free hardware and usually immobile techniques, to intrusive but mobile software techniques. We describe two alternative monitoring approaches, where the appropriateness depends on the resources available, the architecture of the target system, and whether or not black-box functionality is required (i.e., if leaving the probes in the deployed system serves a purpose).

**Non-Intrusive Hardware Recorders** use in-circuit emulators (ICE) with dual port RAM. An ICE replaces the ordinary CPU; it is plugged-in into the CPU target socket, and works together with the rest of the system. The difference from an ordinary CPU is the amount of auxiliary output available. If the ICE (like those from e.g., Lauterbach, and AMC) has RTOS awareness, this type of history recorder needs no instrumentation of the target system. The only input needed is the location of the data to monitor. The hardware monitor category also includes hardware probes using specialized microcontroller debug and trace ports, such as JTAG \cite{91} and BDM \cite{44} ports\(^4\). Certain microcontrollers also provide trace ports that allow each machine code instruction to

\(^4\)See, e.g., www.ghs.com
be recorded. While hardware-based recorders in general have the potential be non-intrusive, since they do not steal any CPU-cycles or target memory, due to price and space limitations these cannot usually be delivered with the product. The application of this type of recorder is consequently best suited for pre-deployment lab testing and debugging.

**Hybrid Recorders** have hardware support and a minimum of target software instrumentation. Software probes write history data to specific addresses, and a hardware component snoops the bus at these addresses, typically in the form of a modern logic analyzer (example manufacturers are Agilent, HP, Lauterbach, VisionICE, and Microtek.) This type of recording system could also be intrusive free if all data manipulations and states were reflected in the system’s external memory, and we had RTOS and data awareness (knowledge of the kernels data structures, and variable locations). However, many micro-controllers and CPUs have on-chip memory and caches, wherefore changes in state or data of the system are not necessarily reflected in the external memory. As a consequence it is necessary to perform instrumentation such that events and data monitored are recorded and stored in external memory, temporarily bypassing the cache and on-chip memory. Data-flow monitoring is similar to software recorders, with the additional penalty of a computing slowdown due to cache write-throughs and access to slower external memory. This type of history recorder is cheaper than ICE:s, but the same argumentation for not leaving the monitoring hardware in the target system still applies. There are however System on Chip (SoC) solutions [26] which can be permanently resident.

### 7.2.3 Relation to Our Work

Most replay techniques require properly synchronized systems and are not able to reproduce erroneous unsynchronized accesses to shared variables [5, 6, 16, 19, 60, 95]. Others are able to correctly reproduce a preemptive execution, but require specialized hardware solutions for this purpose [9, 25, 106]. The technique upon which the replay method in this thesis is based ([102]) requires, in its original form, support from a specially tailored real-time kernel and is not able to perform replay if not a full log (from system start-up) is available.

The recent commercial replay tools mentioned above are more generally applicable. In fact, the BlackBox tool is an extension and commercialization of the work presented in [102] and Chapter 4 of this thesis.

Compared to hardware probing techniques, our method leaves an option of leaving software probes in the deployed system, with the equivalent benefit of a black-box functionality, similar to what is employed in airplanes.
Chapter 8

Discussion, Conclusion and Future Work

This chapter concludes this thesis by reviewing and discussing its contributions, and by providing directions for future work. Specifically, Section 8.1 summarizes the contributions of this thesis, Section 8.2 provides a concluding discussion, and Section 8.3 lists ideas for future work.

8.1 Summary

In this thesis, we have described how to apply traditional structural unit-level testing techniques to system-level testing of real-time systems. Particularly, we have focused on how to perform test item derivation and probe effect-free monitoring for structural testing in this setting.

In Chapter 3, the test item derivation problem was solved, partly by a method for deriving test items from a multi-tasking RTS based on timed automata UPPAAL [10, 59] models and the COER test case generation tool [41], and partly by a method for deriving test items from a multi-tasking RTS based on EOG theory [77, 103]. In our case, both approaches use the SWEET worst-case execution time static analysis tool\(^1\) in order to create the system analysis configuration files. We also provided an evaluation of the above methods with respect to accuracy, analysis time, and sensitivity to system size and complexity. This evaluation showed that, in all applicable cases, the UPPAAL-based

\(^1\)www.mrtc.mdh.se/projects/wcet
method provided a more detailed analysis result at the cost of a more complex analysis. Furthermore, the evaluations suggest that an increased system jitter will significantly decrease the analysability, and, to a lesser extent, also the number of test items found in the system.

In Chapter 4, the probe effect problem was solved by a method for probe effect-free monitoring of multi-tasking real-time systems by recording non-deterministic events and data during test case execution run-time, and using this recording for replaying a fully monitorable deterministic replica of the original reference execution. We call this method Deterministic Replay and its prototype implementation the Time Machine. With this method, we provided results and experiences from two industrial and one academic case study. The case studies not only showed that our method is applicable, but also that it is applicable in a more general system model than the one assumed for the general scope of this thesis, and that the perturbation caused by the latent in-system instrumentation can be held within an acceptable range for a full-scale industrial real-time control system.

In Chapter 6, we also provided an evaluation of the replay method with respect to execution replication accuracy. This evaluation showed that in the worst-case scenario, additional mechanisms are required in order for the method to work. We have also suggested several examples of such additional mechanisms, e.g., taking timestamps, debugger cycle counters and system call markers into consideration when pinpointing asynchronous events, e.g., hardware interrupts.

### 8.2 Discussion

In the starting chapter, we began this thesis with an ambitious statement of the problem of software quality assurance in a complex multi-tasking real-time system setting. By the contributions of this thesis, have we solved the problem? Truthfully, no, not in the general case, but hopefully we have provided results, insights, and angles of approach that takes us one step closer to the elusive goal of bug-free RTS software. This part of the concluding chapter provides a self-reflecting discussion on the contributions of this thesis and their generality, starting by revisiting the research hypotheses of this thesis.
8.2 Discussion

8.2.1 Research Hypotheses Revisited

In Chapter 1, two hypotheses were posed regarding the research presented in this thesis:

**Hyp–1** By analysing timing, control, and data flow properties for each task in $S$, while also considering all possible task interleaving patterns, it is possible to determine which test items are exercisable by executing $S$.

**Hyp–2** By recording events and data causing non-determinism during test case executions with a latent low-perturbing instrumentation, it is possible to use the recorded information to enforce the system behaviour in such a way that each test execution can be deterministically re-executed and monitored for test progress without perturbing the temporal correctness of the initial execution.

Evidence for **Hyp–1** is given in the form of the two test item derivation methods presented in Chapter 3, and the evaluations in Section 6.1. Evidence for **Hyp–2** is given by the Deterministic Replay method presented in Chapter 4, and the evaluations and the case studies presented in sections 6.2 and 6.3. As stated, however, the solutions are not generally applicable. The following sections discuss specific aspects of the generality and the limitations of our approach.

8.2.2 System model

In this thesis, we assume a FPS-scheduled system with periodic tasks. Also, in Section 8.3.3, we give an indication on how this system model can be extended to a model with a server-based scheduling, also allowing sporadic tasks with a deferred preemption without violating the basic requirements of our methods. For the test item derivation analyses proposed in Chapter 3, those requirements encompass a periodically repeated system behaviour. A purely event-triggered system, without bounds on event inter-arrival times can thus not be handled by our approach.

The Deterministic Replay method is not as sensitive to changes in the assumed system model as the test item derivation methods. Even though some aspects are somewhat complicated (e.g., dimensioning of recording buffers, and the process of finding a consistent starting point), Deterministic Replay is applicable in time-triggered, as well as sporadic and event-triggered real-time systems with terminating or non-terminating tasks. This is shown by our case studies, where Asterix uses a single-shot task model [8], in which tasks are
terminated after each task instance, and VxWorks uses a more traditional task model, where tasks are implemented using infinite loops and each iteration in the loop is guarded by a blocking system call (see, e.g., Figure 6.11).

8.2.3 Test Item Derivation Method Applicability

The applicability of the test item derivation methods primarily depend on two different aspects. First, since both methods rely on SWEET for deriving static system properties, and SWEET requires full application and library source code access for this analysis, this is a requirement for our the current implementation of our methods as well. Naturally, for the time being, this is a major problem for analysing systems where source code is not fully available. Second, the inherent complexity of the system being modeled is crucial. As shown by the evaluations, the EOG-based approach is able to handle larger and more complex systems than the UPPAAL-based approach. However, even the EOG-based method is in its current state only able to handle relatively small systems.

The applicability of the Deterministic Replay test item monitoring method is technically not as dependent of system or schedule complexity as it is of sheer system size. Since the method requires a selective monitoring of the data flow, and this process for the time being is not automated, large systems are currently a problem simply for the effort of analysing and inserting the data flow probes by hand. For example, the ABB Robotics case study system consisted of 2.5 million lines of code, a matter that probably caused a massive over-pessimistic instrumentation. Static analysis could be a potential solution to this problem, and something which we intend to investigate further.

8.2.4 Handling Infeasible Test Items

As both our test item derivation methods potentially report false positives (i.e., test items considered feasible by the analysis, but actually being infeasible in the system execution), a test strategy including our test item derivation methods requires some way of dealing with infeasible test items that have been found (potentially) feasible.

Ideally, our method for test item derivation should, given a certain test criterion, and a multi-tasking real-time system, generate a set of test cases that, when executed, would satisfy the test criterion. For the time being, this is not the case (although an idea for solving this problem is sketched in Section 8.3.1), and hence, a different approach must be taken to test case generation. However, once the test case set \( Q \) is generated, executed and monitored
(yielding $Test_{TC}(S,Q)$), we are able to determine the coverage as the ratio $r_{TC} = \frac{|Test_{TC}(S,Q)|}{|Model_{TC}(M_S)|}$. If $r_{TC} < 1$, there are some items in $Model_{TC}$ that have not been exercised during testing. Does this mean that these test items do not actually exist in the system (i.e., are they infeasible?), or that the test suite simply has not been able to enforce them? Just as absence of bugs is practically impossible to prove by testing, it is impossible to show that a test item is infeasible by exhaustive testing. However, it could be considered that a more detailed analysis, with respect to these non-exercised test items, could be performed, in cases where a more general analysis fails.

8.2.5 Approximative Checksums

The fact that the checksums we use for asynchronous event reproduction during replay are approximative might result in ambiguous replay executions. Specifically, we face a problem of handling the situations when the execution context of the location $l_i$ of an interrupt $i$ generates the same checksum as another location $l_2$, such that $l_i$ and $l_2$ are on the same $PC$ address and $l_2$ predates $l_i$ in the execution. Luckily, in these cases, this problem can be handled in a number of ways:

First, we believe that the precision of the checksum markers can be significantly improved by combining them with the system call markers described in Section C.2.3, and with debugger cycle counters\(^2\), by mapping the recorded event timestamps to small cycle counter intervals in which the event is feasible. As for now, this is listed as future work. Second, if the ambiguity is discovered (by detecting dissimilarities between the reference and the replay execution), the replay execution can be reset and restarted with an added skip count on $l_2$. Third, in the worst case, the Deterministic Replay method can be facilitated using hardware- or software instruction counters [6, 15, 48, 52, 65], however inheriting the drawbacks inherent to these methods.

8.3 Future Work

Based on the conclusions and discussions in this chapter, this section provides directions for future work in the area of structural system-level testing of multitasking real-time systems.

\(^2\)See, e.g., www.iar.com
8.3.1 Structural System-Level Test Case Generation

Functional model-based testing can not only perform a valid test case selection with respect to a selected test criterion. Many such methods are also able to derive entire test suites that are guaranteed to satisfy the test criterion selected once executed on the system under test [30, 39, 50, 58, 87, 96]. For example, a functional test case (of, say, a coffee vending machine) could be a sequence of system interactions: (1) insert a coin, (2) wait 3.5 seconds, (3) press button ‘A’, (4) insert another coin, whereas a structural test case would be a sequence of internal selections: (1) execute task $C$ along control flow path $p_C$ for 14 ms, (2) switch to task $D$ and execute along path $p_D$ until interrupt $i$ preempts the execution 22 ms later. Intuitively, the latter is a test case less easily enforced. Although functional and structural testing aim at finding fundamentally different types of errors, this is a current drawback of our approach.

However, given that we are able to synthetically create system-level execution traces, and enforce these traces on the system under test by means of the replay method proposed in this thesis, this would facilitate work on test case generation techniques for the structural test criteria described here. This would be truly novel work, and strongly enhance the ease of use of the methods proposed in this thesis.

8.3.2 Checksum Unique Marker Precision

The precision of the checksum markers used for pinpointing of asynchronous events can be significantly improved by combining them with (1) the system call markers described as a part of the implementation of the Time Machine in VxWorks (see Section C.2.3), and (2) debugger cycle counters\(^3\), by mapping the recorded event timestamps to small cycle counter intervals in which the event is feasible. We predict that this method will significantly reduce the problem of the Time Traveler invoking interrupts prematurely in the replay execution (since that state has the same program counter value and its execution context produces the same checksum).

8.3.3 Facilitating Sporadic Tasks through Servers

Task periodicity is an important property for the termination of our DU-path analyses, as it is required for the behaviour space of the analysis to be finitely bounded. As a consequence, DU-path analysis of systems with aperiodic tasks

\(^3\)See, e.g., www.iar.com
are not facilitated by our methods. However, through the use of a periodic server task [85, 61, 88], even aperiodic tasks can be facilitated. The basic idea of server-based scheduling is that all aperiodic tasks are executed by a periodically invoked server task. Note that not all types of periodic servers would work for facilitating DU-path analysis of aperiodic tasks. For example, high-utilization servers, like the Deferrable Server or the Priority Exchange Server [61], are not well-suited for our methods, as they allow restricted but arbitrary preemption by asynchronous tasks. However, simpler server-based methods, like the Polling Server [88] suit our method well, as they give any aperiodic tasks of the system the periodic qualities required for bounding the analysis behaviour space. We aim at extending the EOG- and the UPPAAL-based DU-path derivation methods, such that they support the use of aperiodic tasks executed by a polling server.

### 8.3.4 Expressing Flow Facts in UPPAAL

The original UPPAAL control flow models described in this thesis do not include information on boolean selections, exact conditions for loop termination, etc. In SWEET, the worst case analysis time analysis tool used to automatically generate these models from system source code, such information is represented as flow facts [4]. SWEET includes a fairly detailed language for describing quite complex flow fact relations in a single control flow graph. In our work of automatically generating UPPAAL models from SWEET representations, we have translated a small subset of the SWEET flow facts into UPPAAL semantics (essentially the facts needed for expressing loop bounds). However, if the entire set of flow facts is translated from the SWEET representation, this would aid in producing a more detailed analysis result in the UPPAAL-based approach, since inter-task infeasible paths will be considered. In addition, the analysis runtime and complexity would be reduced, since a number of infeasible paths would not have to be searched for test items.

### 8.3.5 Regression Testing

As a piece of software evolves during its life cycle, the set of test cases used to verify the software cannot remain static. Each change to the software requires a corresponding change in the test set. An initial set of test cases \( T_0 \) should evolve to a modified test set \( T_1 \) following software changes. Hence, after \( n \) changes, the software is verified by the test cases in \( T_n \) (see Figure 8.1). We assume that \( E_k \) is an exhaustive set of test cases, including all possible test
cases of our system. The smaller set $T_k \subset E_k$ contains all test cases selected by the test selection method of our choice. Furthermore, $t_k \in T_k$ is a specific test case, which results in an error. To correct this error, the source code of the software is changed. This change yields a new system, and hence a new exhaustive test set $E_{k+1}$. Finally, the set $C_{k+1}$ defines all test cases that are affected by program changes. From the above sets, we can derive four subsets of particular importance for test set maintenance (see Figure 8.2):

1. $T_k \setminus (E_{k+1} \cap T_k)$ contains test cases that no longer are part of the behavior of the software. Note the possibility of $t_{k+1} \notin E_{k+1}$. By changing the system, we may have prohibited execution of $t_k$.

2. $(E_{k+1} \cap T_k) \setminus C_{k+1}$ is the set of unaffected, still valid test cases. These require no re-testing at this stage.

3. $C_{k+1} \setminus (T_k \cap C_{k+1})$ holds untested test cases that should be tested due to the software changes.

4. $T_k \cap C_{k+1}$ contains still valid test cases, possibly affected by software changes. These need to be re-tested.

In short, set $1$ needs to be identified, such that no effort is spent trying to exercise test cases that cannot be exercised. Similarly, a tight identification of
set 2 saves us the effort of testing test cases that *need not* be tested. Identification of set 3 is also required in order to keep the test set up to date with the changed system. For the purpose of regression testing, however, set 4 is of highest importance. This set contains all test cases that are possibly affected by software changes, and must be re-verified by means of regression testing. In the next iteration of software and test set evolution, $(E_{k+1} \cap T_k) \cup C_{k+1}$ will serve as $T_{k+1}$.

To our knowledge, no work has been done on investigating the impact of changes on the system-level control-flow. We intend to tackle the test selection problem for system-level regression testing by performing static analysis of the system prior to, and after system changes. By comparing the analysis results, we hope to be able to determine which parts of the system need to be re-tested. Further, we aim at enforcing these tests using synthesized replay traces.

![Figure 8.2: Derived test case subsets.](image-url)
Appendix A

Timed Automata Test Item Derivation

In order to use the analysis engine of UPPAAL to derive shared variable DU-paths, we need to correctly model the behaviour of the system under test. In this appendix, we describe the UPPAAL-based real-time system control flow model for this purpose. In the model, each task is modeled as a set of timed automata (the task main function, and the set of subroutines directly or indirectly called by the main function). Further, the run-time state (ready, running, waiting, etc.) of each task in the system is modeled as an Execution Control Automaton (ECA). In addition, the run-time kernel (i.e., the schedule, task switch, and the system clock) is modeled as a set of timed automata. Together, these automata form the system-level control flow model of the real-time system. Figure A.1 describes the overall structure of the system-level control flow model.

A.1 Timed Control Flow Modeling using UPPAAL

This section will in depth describe the three different categories of timed automata included in the model. First, however, the clocks of the model are described.
A.1.1 Clocks

Clocks are essential to the operation of the system model, since they enforce and constrain the behaviour of the model during simulation and analysis. To keep the memory consumption of the analysis down it is however important to restrict the number of clocks in the model as much as possible. In order to express the system-level control flow behaviour, the model makes use of four kinds of clocks:

- **Free running counter clock (frc)** The frc clock models the behaviour of the free-running counter of the assumed underlying hardware. In other words, its only purpose is to generate strictly periodic interrupts, used to drive the system.

- **Basic block execution time clock (bbet)** The bbet clock is used to constrain the execution time of basic blocks in the model. The clock is reset each time the simulation of the model enters a basic block $X$. In addition, the simulation is not allowed to leave the basic block when $bbet$ is outside the interval $[BCET_X, WCET_X]$. Since all basic blocks in the
entire model are mutually exclusive (i.e., if a basic block $X$ in a task is visited during simulation, no other basic blocks can be visited until the simulation leaves $X$), there is no need for individual clocks for each task, but a single clock can be used for all tasks.

- **Kernel execution time clock (ket)** This clock keeps track of the time spent in the kernel during task switches.

- **Task scheduling clocks ($cT$)** Each task $T$ in the system keeps a clock $cT$ for keeping track of the release time of the task.

Note that the first three clocks are one-per-system clocks, whereas the number of $cT$ clocks are given by the number of tasks in the system. E.g., a system with four tasks will be modeled using a total of seven clocks.

### A.1.2 Task CFG modeling

From a standard CFG perspective, a task is made up of a set of function-level graphs (i.e., a graph for the task main function, and a graph for each function that is explicitly or implicitly called by the main function). In our model, each function translates to a timed automaton. The basic control flow structure of a function is intuitively modeled in UPPAAL using a one-to-one mapping between basic blocks and UPPAAL locations, and between CFG edges and UPPAAL edges. An UPPAAL model of a very small example task is shown in Figure A.2. The details of UPPAAL task modeling are described below.

**Definitions and Uses** belonging to a vertex $v$ are modeled using assignments on all outgoing edges from $v$. This should be interpreted as once the basic block is visited, the definition or use is executed. Definitions are modeled using the `update` functionality in UPPAAL (e.g., a definition of a shared variable $x$ to 5 is modeled as an update with $x$ on the left hand side: “$x := 5$”). Similarly, a use of a shared variable $y$ is modeled either using the same functionality (e.g., a use of a shared variable $y$ in an assignment to another variable $z$ is modeled as an update with $y$ on the right hand side: “$z := y$”). Note that, if the left hand side variable in such a statement is not a shared variable, we will use updates to local dummy variables in order to express the shared variable usage. An example of a definition of $y$ and a use of $x$ can be found in Figure A.2.

**Best case execution time (BCET)** of a basic block is modeled using a clock guard on each transition from the node representing the basic block. The guard makes sure that no transitions are made before the requirement of the
guards is fulfilled. For example, the BCET guards of basic block \( j \) are formulated as \( \text{bbet} \geq BCET_j \). The semantics of the guard could be expressed as “it is not possible to leave this basic block before \( BCET_j \) time units have passed”.

**Worst case execution time (WCET)** of a basic block is modeled using an invariant in the node representing the basic block. For example, the WCET invariant of a basic block \( i \) is formulated as \( \text{bbet} \leq WCET_i \). The semantics of the invariant could be expressed as “it is not possible to stay in this basic block longer than \( WCET_i \)”. It should be noted that WCET could be expressed as guards on transitions (as BCET is), but this would result in a semantics that does not enforce transitions within the WCET, but only restricts transitions to be made within the WCET.

Upper **loop bounds** are essential to our method of modeling (and to time analysis in RTSSs in general [35]). An unbounded loop may iterate e.g., 2, 5, 127, or infinitely many times, causing an unbounded WCET of the function including the loop. In the model, upper loop bounds are expressed in the form of guard/assignment pairs on backward branches. Each transition representing
a backward branch (i.e., a prerequisite for an iterative construct) will have a variable associated with it. The variable is initialized to the value of the upper loop bound. The transition may be taken if the value of the variable is above 0. Each time the branch is taken, the variable is decremented. When the variable reaches 0, the guard will prevent the backward branch from being taken.

Lower loop bounds are not essential to the operation of the model, but if they can be derived, they help reducing the complexity of the model by restricting the number of derivable test items. In addition, existing lower loop bounds that are not explicitly expressed in the system may result in infeasible test items being added to the result. Both upper and lower loop bounds are stated per entry to the loop. Hence, a loop bound is reset each time the loop is entered.

Since the control flow of a task is represented by the main function graph and a set of subroutine CFGs, there is a need for expressing the mechanisms of subroutine invocation and subroutine return. For this purpose, we use UPPAAL synchronization channels. As a subroutine is called by another function automata, this is modeled using a synchronization channel by the name of the called subroutine. While the called subroutine automaton executes, the callee is suspended in an intermediate location (see Figure A.3), waiting for the called function to terminate and synchronize back on the same channel. Note that this way of working with subroutine automata imposes two restrictions on
our model. First, if several tasks use a common function, each task needs to have a unique automaton instance of that function. Second, no recursive calls are allowed (since this assumes that a subroutine is allowed to call itself).

In order to enable a preemptive behaviour in the model, there is a need for the model to be able to suspend and preempt an executing task. Since a truly preemptive behaviour is troublesome to model and analyse using timed automata, we approximate a preemption such that it always occurs in between basic blocks. Note that the negative effects of this approximation can be avoided if each statement is modeled as a separate basic block (as is done in the evaluation in Section 6.1). As the system is strictly time-triggered, all task preemption will be caused by clock interrupts. If a clock interrupt is signaled by the system clock automaton (described in Section A.1.1) during the execution of a basic block, the interrupt is deferred to preempt the task instantaneously after the execution of the basic block. In the model, this is expressed by adding extra locations in between all basic block locations. When a basic block has finished executing, the task synchronizes on the ready channel with the task’s ECA. The ECA checks whether an interrupt has occurred during the execution of the basic block. If so, the task is preempted. If not, the task may continue executing the next basic block (modeled using the go synchronization channel). In other words, the task model “asks” the ECA if it is allowed to execute the next basic block. As an example of this, consider the transitions in Figure A.4. Scenario I describes a transition from a basic block $A$ (with best-case execution time $BCET_A$ and worst-case execution time $WCET_A$) to a basic

Figure A.4: The construct for facilitating in-between-basic block preemption.
A.1 Timed Control Flow Modeling using UPPAAL

Figure A.5: An example Execution Control Automaton for a task “A”. A “C” denotes a committed state, where no time may pass and no transitions from non-committed states may occur.

Scenario II describes the same scenario with the added in-between location and synchronizations of our model. If a clock interrupt would occur somewhere during the execution of basic block $A$, the task would be suspended in the intermediate state between $A$ and $B$ until it is resumed by the scheduler.

A.1.3 Modeling of Task Execution Control

For each task in the model, a separate task ECA (depicted in Figure A.5) is used in order to keep track of the state of the task. An ECA is generic in the sense that it has identical structure regardless of the task it controls. The initial state of the automaton (WAITING) represents a waiting state of the task (i.e., the task has not yet been put in the ready-queue).

The ECA needs to be able to handle task activation. From the initial state, the task can be made ready by an synchronization on the moveToReady channel. The new state (the top rightmost in the figure) represents the task being in the ready-queue, where it might be selected for execution. The transition from WAITING to READY also adds the unique task priority to the current global system priority. Since each task in the underlying system model has a unique priority $P$ enumerated from 0 (idle task) and up, task priorities in the
UPPAAL models are expressed as $2^P$. Consequently, if a task with a priority $P_1$ is scheduled, the global task priority will be in the interval $[2P_1, 2P_1+1]$ (unless no other task with a higher priority is also scheduled). As for task resumption, using FPS scheduling, the task with the highest priority in the ready-queue will always be selected for execution. In the model, this corresponds to the ECA of the task selected for execution by the schedule (described in Section A.1.4) synchronizing on the resume channel, causing a transition to the SCHEDULED state (the top middle in the figure). Note that each task ECA checks whether the current system priority is within the interval $[2P, 2P+1]$ before making the transition to SCHEDULED. If this is the case, no higher prioritized task can be in READY.

In the running state, an ECA has three options: (1) If an unacknowledged clock interrupt has occurred, the ECA will synchronize on the preempt channel in order to hand over control to the scheduler automaton. (2) If no event has occurred and there still are basic blocks to execute before task termination, the ECA will synchronize on the go channel, indicating that the task is allowed to execute one full basic block unpreempted. Once the basic block is executed, the task synchronizes back on the ready channel. (3) If no basic blocks are left for execution in the task (or if the task selects the termination edge in favor of looping back), the ECA synchronizes on the terminate channel.

The first termination channel (terminate) is used to check whether the task is finished with its execution and ready to terminate, the second termination channel (terminated) is used to communicate the termination to the task switch automaton, in order for it to dispatch the next task for execution. When the ECA is in fully terminated position (back in the initial state), the task is ready for re-scheduling. The second task termination edge also subtracts the task priority from the system priority, allowing lower prioritized tasks to resume their execution.

It should be noted that the running state is a committed state. In other words, no time can pass while in this state (i.e., a task must be immediately executed, preempted or terminated) and no other transactions from non-committed states can be performed while in this state.

A.1.4 FPS Scheduler and Task Switch Modeling

The task switch and scheduler mechanisms of the system are modeled using three different types of automata: The system clock automaton maintains the overall passage of time in the system, the task switch automaton facilitates the transfer of control from one task automaton to another, and the schedule
A.1 Timed Control Flow Modeling using UPPAAL

automaton determines which task automaton to gain control at a given point in the simulation of the model.

The system clock automaton is a rudimentary automaton with a fundamental function. This automaton (depicted in Figure A.6) is responsible for driving the entire system by providing it with strictly periodic “heartbeat” interrupts (i.e., timerticks). The frequency of these interrupts is determined by the resolution parameter. Basically, resolution represents the length, in terms of time units, in between each timertick.

Note that the system clock automaton synchronizes on different channels depending on the current state of the system. If no tasks are currently executing, the timertickIdle synchronization channel edge is taken. Otherwise, the timertick synchronization channel edge is taken. The reason for differentiating between these is that an idle system is instantly preempted for task switching, while a busy system has to wait until the executing task leaves its current basic block. Regardless if the system is idle or not, each timertick interrupt invokes the task switch automaton, thereby enforcing the release and execution of tasks.

The task switch automaton (depicted in Figure A.7) is the generic mechanism for facilitating preemption and task termination (i.e., the two basic mechanisms for task switching included in our run-time model). As discussed in the above section, for practical reasons, a preemption of an idle system is considered different from a task to task preemption. While non-busy (i.e., in the TASK_EXEC or the SYS_IDLE state), the task switch automaton listens for system clock timerticks or task terminations. Once a timertick or a termination is received, the automaton

1. calls the scheduler automaton in order to release all tasks that should be released according to their $T$ and $O$ properties.

Figure A.6: The system clock automaton.
2. waits for \([K_{\text{MIN}}, K_{\text{MAX}}]\) time units (i.e., consumes the time actually spent in the task switch routine).

3. releases the next task for execution by synchronizing with the ECA automaton of the highest prioritized released on the \texttt{resume} channel, or returns to \texttt{SYS_IDLE} if no task is released for execution. Note that only tasks with their ECA in a \texttt{READY} state (i.e., tasks that are moved to ready by the schedule) may be scheduled for execution, and that the priority guards in the ECA prohibits lower prioritized tasks from being scheduled.

The \texttt{schedule} automaton represents the run-time schedule of the RTS (in our case a time-triggered fixed-priority schedule). The automaton has an initial waiting state, and an activation channel called \texttt{schedule} (the activation of the schedule automaton is prompted by the task switch automaton as a response to the occurrence of a timertick). Once the schedule automaton is activated, it steps through all scheduling decisions based on the current value of the local task clocks. If a task’s period is passed, the task is released and the clock is reset. Furthermore, if the task is previously released and the task’s offset is passed, the task is moved to ready (by synchronizing with the task’s ECA over the corresponding \texttt{moveToReady} channel). Once the schedule is traversed,
A.1 Timed Control Flow Modeling using UPPAAL

control is handed back over to the task switch automaton over the scheduled synchronization channel. Figure A.8 depicts an example time-triggered FPS schedule with four tasks.

This method of modeling presupposes that all task $T$ and $O$ parameters are multiples of the RESOLUTION parameter driving the timerticks, which is a natural limitation since time-triggered systems typically only release tasks at timertick invocations.

Note that other scheduling mechanisms also could be adapted to this model. E.g., in [3], Amnell et al. describe a tool for formally modeling different scheduling mechanisms using timed automata. In this paper, however, we stick to the run-time model described in Section 1.2.
Appendix B

The DUANALYSIS Algorithm

The DUANALYSIS algorithm (Figure B.1) is a slight variation of the original EOG algorithm [103], built upon the manipulation of two data structures. Throughout the analysis, an abstract state of type State propagates through the execution of the system. For each execution of a job, the abstract state is changed according to the StateChange created by executing the job. State represents the current abstract state of the execution, and contains information regarding currently live definitions, active definitions, active uses, and encountered DU-paths. Hence, State is defined as:

\[
\text{State} : \{\text{liveDefs}, \text{activeDefs}, \text{activeUses}, \text{duPaths}\}
\]

StateChange represents the change of state incurred by the execution of a certain part of a job. As such, StateChange contains new active definitions, killed live definitions, killed active definitions, killed active uses, and the WCET of the executed job. Hence, StateChange is defined as:

\[
\text{StateChange} : \{\text{newActiveDefs, killedLiveDefs, killedActiveDefs, killedActiveUses, wcet}\}
\]

Intuitively, the combination of a State \(a_1\) and a StateChange \(a'_1\) yields a new State \(a_2\), representing the original state affected by the changes in \(a'_1\). E.g., if \(a_1\) contains a set of liveDefs \(\{d_1, d_2, d_3\}\), and \(a'_1\) contains a set of killedLiveDefs \(\{d_2\}\), then \(a_2\)’s set of liveDefs will look as follows: \(\{d_1, d_3\}\).

Two more structures (Ivl and Job) are used in the analysis. Ivl defines a time
interval by its extremal values \( l \) and \( r \). Job represents a task instance and contains definitions, uses, job priority, release time, BCET and WCET. Job is defined as:

\[
\text{Job} : \{D, U, P, R, BCET, WCET\}
\]

In the algorithm, the process of task execution and task switch is formalized by the functions EXECUTE and SWITCHTASK, where

\[
\text{EXECUTE} : \text{State} \times \text{Job} \times \text{Ivl} \times \text{Ivl} \to \text{StateChange}
\]

\[
\text{SWITCHTASK} : \text{StateChange} \times \text{State} \times \text{Ivl} \to \text{State}
\]

In essence, the EXECUTE function produces a change of abstract state (StateChange) incurred on original abstract state (State) by executing a certain job (Job), executed during an execution window (first Ivl), and released during a release interval (second Ivl). The SWITCHTASK function describes the impact of a task switch occurring during a time interval (Ivl), and produces a new abstract state (State), based on the original abstract state and the changes described by StateChange. The implementation of these functions are directly based on the Definition, Use, and DU-path rules described in Chapter 3.

Roughly, the DUANALYSIS algorithm starts with an empty State at time 0 by scheduling the highest prioritized ready job \( j \). Using the EXECUTE function, \( j \)'s StateChange is derived. Next, if \( j \) is always finished before the next higher priority job is released, SWITCHTASK combines the StateChange with the old State to a new State. The algorithm increments the time and schedules the next job. Else, if \( j \) is safely preempted by a higher priority job, SWITCHTASK combines the StateChange with the old State - but only regards the events that predate the preemption, stores the remainder of \( j \), increments the time, and schedules the higher prioritized job. Else, if \( j \) might be preempted, the algorithm splits into two recursive branches, one of which considers the case with a preemption, and the other considers the case with no preemption. This behaviour is repeated until all jobs in the LCM are analysed.

In order to derive the StateChange created by executing a job \( j \), the EXECUTE function and its auxiliary functions, EXECDEF and EXECUSE (Figure B.2) work through all shared variable accesses in \( j \) in a chronological order. Each access is treated according to its corresponding definition or use rule (definition rules are implemented in EXECDEF and use rules are implemented in EXECUSE).

SWITCHTASK and its auxiliary functions REVISEDEF and REVISEUSE (Figure B.3) create a new State by adding the changes in \( j \)'s StateChange
to the State prior to the execution of \( j \). If \( j \) is not preempted, all changes in StateChange are considered when creating the new State. Otherwise, only those changes prior to the preemption time are considered. In order to add new DU-paths to the result of the analysis, SWITCHTASK uses the two DU-path rules given in Page 53. Specifically, SWITCHTASK implements DU-path Rule 1 in REVISEDEF, and DU-path Rule 2 in REVISEUSE.

Note that, compared to the original EOG algorithm, DUANALYSIS uses a number of additional auxiliary functions (for a description of all auxiliary functions, we refer to [103]):

1. PREEMPT\((j, t_0, t_1)\) - re-calculates the shared variable access times (i.e., \( \min \), \( \max \), and \( \text{rdMax} \)) for all definitions and uses in the preempted job \( j \), given that task \( j \) has been allowed to execute somewhere in the time interval of \([t_0, t_1]\) prior to the preemption.

2. GETNEXTDUEVENT\((t, j)\) - returns the time \( t_s \) of the next shared variable access in job \( j \), such that \( t_s > t \).

3. GETNEXTDUCHANGE\((t, trs)\) - returns the time \( t_s \) of the next abstract state change in transition \( trs \), such that \( t_s > t \).
DU\textsc{Analysis}(state, state\textsc{Change}, rdy, RI, SI)

// When is the next job(s) released?
1. \( t := \text{NextRelease}(SI) \)
2. if \( rdy = \emptyset \)
3. \( rdy := \text{MakeReady}(t, rdy) \)
4. if \( rdy \neq \emptyset \)
5. \( \text{DU\textsc{Analysis}}(state, state\textsc{Change}, rdy, RI, (t, SI, r)) \)
6. else \( state := \text{SwitchTask}(state\textsc{Change}, state, RI) \)
7. else
   
   // Extract the highest priority job in rdy.
8. \( J := \text{Dispatch}(rdy) \)
9. \([\alpha, \beta] := [\max(J.R, RI.l), \max(J.R, RI.l) + J.WCET] \)
10. \( a' := \alpha + J.BCET \)
11. \( b' := \beta \)
12. \( state := \text{SwitchTask}(state\textsc{Change}, state, RI) \)
13. \( state\textsc{Change} := \text{Execute}(state, J, [\alpha, \beta], RI) \)

// Add all lower priority jobs released before J's termination,
// or before a high priority job is preempting J.
14. while \((t < \beta) \land (\text{Prio}(t) < J.P) \)
15. \( rdy := \text{MakeReady}(t, rdy) \)
16. \( t := \text{NextRelease}((t, SI, r)) \)
17. // Does the next scheduled job preempt J?
18. if \((t < \beta) \land (\text{Prio}(t) > J.P) \)
19. \( \text{DU\textsc{Analysis}}(state, state\textsc{Change}, rdy, [a', t], [t, SI, r]) \)
20. if \( rdy = \emptyset \)
21. \( \text{DU\textsc{Analysis}}(state, state\textsc{Change}, \text{MakeReady}(t, rdy), [t, t], (t, SI, r)) \)
22. else if \( t = a' \)
23. \( \text{DU\textsc{Analysis}}(state, state\textsc{Change}, \text{MakeReady}(t, rdy), [t, t], (t, SI, r)) \)

// Add all jobs that are released at time t.
24. \( rdy := \text{MakeReady}(t, rdy) \)

// Best and worst case execution time prior to preemption?
25. \( J.BCET := \max(J.BCET - (t - (\max(J.R, RI.l))), 0) \)
26. \( J.WCET := \max(J.WCET - (t - (\max(J.R, RI.l))), 0) \)
27. \( \text{Preempt}(J, (t - (\max(J.R, RI.l))), (t - (\max(J.R, RI.l)))) \)
28. \( \text{DU\textsc{Analysis}}(state, state\textsc{Change}, rdy \cup \{J\}, [t, t], (t, SI, r)) \)

// No preemption.
29. else if \( t = \infty \) \# Have we come to the end of the analysis?
30. \( \text{DU\textsc{Analysis}}(state, state\textsc{Change}, rdy, [a', b'], [\infty, \infty]) \) \# Yes
31. else // More jobs to execute.
32. \( \text{DU\textsc{Analysis}}(state, state\textsc{Change}, rdy, [a', b'], [t, SI, r]) \)
33. if \( \text{rdy} \neq \emptyset \land t = \beta \)
34. \( \text{DU\textsc{Analysis}}(state, state\textsc{Change}, \text{MakeReady}(t, rdy), [t, t], (t, SI, r)) \)
35. if \( a' \neq b' \) \# And one branch for the low priority job.
   // The regular succession of the next job
36. \( \text{DU\textsc{Analysis}}(state, state\textsc{Change}, rdy, [a', b'], [t, SI, r]) \)

Figure B.1: The DU\textsc{Analysis} algorithm.
EXECDef \((t, D, \text{stateChange}, \text{state})\)
1. for each \(d \in D\) // For all defines in the job.
   2. if \(t = d.\text{min}\)
   3. \text{stateChange}.newActiveDefs := \text{stateChange}.newActiveDefs \cup \{d\}
   4. if \(t = d.\text{max}\)
   5. if \((d \notin \text{state}.\text{liveDefs}) \land (d \notin \text{stateChange}.\text{newActiveDefs})\)
   6. \text{stateChange}.newActiveDefs := \text{stateChange}.newActiveDefs \cup \{d\}
   7. \text{stateChange}.killedActiveDefs := \text{stateChange}.killedActiveDefs \cup \{d\}
   8. if \(t = d.\text{rdMax}\)
   9. \text{stateChange}.killedLiveDefs := \text{stateChange}.killedLiveDefs \cup \{d\}
10. return \text{stateChange}

EXECUse \((t, D, \text{stateChange}, \text{state})\)
1. for each \(u \in U\) // For all uses in the job.
2. if \(t = u.\text{min}\)
3. \text{stateChange}.newActiveUses := \text{stateChange}.newActiveUses \cup \{u\}
4. if \(t = u.\text{max}\)
5. if \((d \notin \text{state}.\text{activeUses}) \land (u \notin \text{stateChange}.\text{newActiveUses})\)
6. \text{stateChange}.newActiveUses := \text{stateChange}.newActiveUses \cup \{u\}
7. \text{stateChange}.killedActiveUses := \text{stateChange}.killedActiveUses \cup \{u\}
8. return \text{stateChange}

EXECUTE \((\text{state}, \text{job}, EW, RI)\)
1. \text{stateChange} := \{\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, EW, r\}
2. // Update the execution time properties of the defines
3. // and uses such that they are relative to the release
4. // time of the job rather than the job start.
5. for each \(d \in \text{job}.D\)
6. \(d.\text{min} := d.\text{min} + \text{max}(EW, l, RI, l)\)
7. \(d.\text{max} := d.\text{max} + \text{max}(EW, l, RI, r)\)
8. \(d.\text{rdMax} := d.\text{rdMax} + \text{max}(EW, l, RI, r)\)
9. for each \(u \in \text{job}.U\)
10. \(u.\text{min} := u.\text{min} + \text{max}(EW, l, RI, l)\)
11. \(u.\text{max} := u.\text{max} + \text{max}(EW, l, RI, r)\)
12. \(t := \text{getNextDueEvent}(0, \text{job})\)
13. while \(t < EW, r\) // Within the execution time of the job.
14. // For each du-event occurring at t, handle it.
15. \text{stateChange} := \text{EXECDef}(t, \text{job}.D, \text{stateChange}, \text{state})
16. \text{stateChange} := \text{EXECUse}(t, \text{job}.U, \text{stateChange}, \text{state})
17. \(t := \text{getNextDueEvent}(t, \text{job})\)
18. // Return the change of state.
19. return \text{stateChange}

Figure B.2: The EXECUTE functions.
REVISED\(\tau\), stateChange, state\)
1. for each \(u \in\) state.activeUses // For all currently live uses
2. for each \(d \in\) stateChange.newActiveDefs // and new active defs.
   3. if \((t = d.\text{min}) \land (d.\text{var} = u.\text{var})\)
      // New DU-path found. Add it!
      state.duPaths := state.duPaths \cup \{(d, u)\}
4. for each \(d \in\) stateChange.killedActiveDefs // For killed active defs.
6. if \(t = d.\text{max}\)
      // Remove d from the set of active defs.
5. state.activeDefs := state.activeDefs \\{d\}
7. for each \(u \in\) stateChange.killedLiveUses // For killed live uses.
8. if \((t = u.\text{max})\)
      // Remove u from the set of live uses.
9. state.activeUses := state.activeUses \\{u\}
10. return state

REVISEUSE\(\tau\), stateChange, state\)
1. for each \(d \in\) state.liveDefs // For all currently live defs
2. for each \(u \in\) stateChange.newActiveUses // and new live uses.
3. if \((t = u.\text{min}) \land (d.\text{var} = u.\text{var})\)
      // New DU-path found. Add it!
4. state.duPaths := state.duPaths \cup \{(d, u)\}
5. for each \(u \in\) stateChange.killedActiveUses // For killed active uses.
7. state.activeUses := state.activeUses \\{u\}
8. return state

SWITCHTASK\(s, stateChange\), TSI\)
1. \(t := \text{GETNEXTDUCHANGE}(0\), stateChange\)
2. while \(t < \text{TSI}._r\)
3. state := REVISED\(t\), stateChange, state\)
4. state := REVISEUSE\(t\), stateChange, state\)
5. \(t := \text{GETNEXTDUCHANGE}(t\), stateChange\)
6. state.activeUses := \(\emptyset\)
7. if \((\text{TSI}._l \neq \text{TSI}._r) \land (\text{TSI}._r < \text{stateChange.WCET})\)
      // A non-preempted non-wcet branch.
8. for each \(d \in\) stateChange.killedActiveDefs
9.   if \(d.\text{max} > \text{TSI}._r\)
       // Remove d from the set of active defs.
10. state.activeDefs := state.activeDefs \\{d\}
11. for each \(d \in\) stateChange.killedLiveDefs
12.   if \(\text{stateChange.WCET} \geq d.\text{max} > \text{TSI}._r\)
       // Remove d from the set of live defs.
13. state.liveDefs := state.liveDefs \\{d\}
14. return new abstract state.

return state

Figure B.3: The SWITCHTASK functions.
Appendix C

Reproduction of Asynchronous Events

Interrupts are a major source of non-determinism in program executions. As a part of the Time Machine, we present a novel method for pinpointing interrupts, suitable for embedded real-time systems.

As an interrupt occurs, the ongoing CPU activity is halted, the state of the executing program is stored and the interrupt is handled by an interrupt service routine. To reproduce this scenario, we need to be able to correctly reproduce the occurrence of the interrupt. In other words, we must make sure that the interrupt preempts the execution of the CPU activity at the exact same state during the replay as it did in the first execution. Traditionally, this is achieved using hardware instruction counters [15, 48] or software instruction counters (SICs) [6, 52, 65]. However, few embedded microcontrollers are equipped with sufficiently accurate hardware instruction counters [75], and software instruction counters require approximately 10% of the overall execution time, a hard to meet requirement in resource constrained systems. Consequently, a different approach, more adapted to the requirements of embedded systems and able to perform using standard development tools, is needed.

C.1 Context Checksums

In our method, we use an approximation of the state of the preempted program at the time of the interrupt as a marker. This approximation is represented in the
form of the SP value and a checksum of the execution environment of the pro-
gram, such as the registers, or the registers combined with (part of) the program
stack. Our method imposes an execution time overhead of a small fraction of
that of the SIC and requires no additional hardware support to function.

The basic idea of our method is to reproduce interrupts by recording the
PC values and unique markers of their occurrence. In order to reproduce these
interrupts, a debugger breakpoint is set at each interrupted PC address and
the program is restarted in the debugger. As a breakpoint is hit, the unique
marker of the current execution is compared to the recorded unique marker
value. If these markers match, we consider this interrupt to be pinpointed and
an interrupt is forced upon the system.

C.1.1 Execution Context

If we look back upon the example shown in Figure 4.8, where a PC value is in-
distinguishably revisited a number of times, a good solution in theory would be
to make use of the loop counter and the PC value as a unique marker. Unfortu-
nately, not all loops have loop counters. In general, it is very hard to determine
exactly which parts of the program execution context that differentiate between
specific loop iterations, subroutine calls or recursive calls. Ideally, we would
base our unique marker on the entire content of the execution context in or-
der to be able to differentiate between loop iterations. However, considering
the amount of data used to represent this context, we face a practical problem
when recording it during execution due to the massive perturbation to the sys-
tem. Consequently, we need to derive a subset of the execution context suitable
for unique marker use.

The program execution context is basically a set of data stored in registers,
on the stack and/or on the heap. Since the processor registers are small and
very fast, these hold the most current parts of the execution context. And, since
they are small and very fast, they make excellent candidates as a basis for the
execution context-based unique markers.

C.1.2 Register Checksum

One solution would be to store the contents of each processor register. How-
ever, in most embedded systems, computing- as well as memory resources
are scarce. Storing all registers at each interrupt might incur an intolerable
overhead on the memory usage of the system. In our method, we handle this
problem by separately storing the SP value, as it is invaluable for differentiating between recursive calls, and calculating and storing a checksum of the contents of the remaining registers. By doing this, we destroy information, but still preserve an approximative representation of the register contents from the time of the interrupt.

The register checksum operation is a simple addition of all processor registers. Overflow of the accumulated checksum is ignored. Hence, if the processor is equipped with eight general-purpose 16-bit registers ($R_0$...$R_7$), the register checksum $C_R$ is calculated as follows:

$$C_R = (R_0 + R_1 + ... + R_7) \mod 2^{16}$$

Due to the modest size and the ease of access of processor registers, the computational cost of calculating a register checksum is very small. However, since the register checksum is based solely on the processor registers, its main disadvantage is that it only covers a minor subset of the execution context. If an interrupt occurs within a loop and the actual parameters differentiating between iterations are not included in this subset, we will not be able to uniquely pinpoint the occurrence of the interrupt.

C.1.3 Stack Checksum

In order to capture those interrupt occurrences not successfully pinpointed by the register checksum, we must expand the interval of execution context included in the context checksum. As we already used the registers, the remainder of the execution context is located on the stack and on the heap. In our method, we chose to work with the program stack contents. This has two reasons: First, implementing a stack checksum calculation in an instrumentation probe [94] is significantly easier than implementing a heap checksum in the same probe. The stack area is well defined, continuous and often easily accessed from within the probe. Second, without having extensive proof of this, we assume that variables influencing the program control flow, such as loop counters, are more often allocated on the stack than on the heap.

The checksum operation of the stack checksum is identical to the one performed in order to calculate the register checksum. Here, the subset of the execution context included in the checksum is bounded by the boundaries of the stack of the executing program. Hence, on a 16-bit architecture, the stack checksum $C_S$ is calculated using the following formula:

$$C_S = (S_{SP} + S_{SP+1} + ... + S_{SB}) \mod 2^{16}$$
In the above formula, \( S_X \) denotes the byte at stack address \( X \). \( SP \) denotes the value of the stack pointer at the time of the interrupt and \( SB \) denotes the value of the stack base of the interrupted program.

The stack checksum should be viewed upon as a complement to the register checksum rather than a stand-alone solution for pinpointing interrupts. The reason for this is the fact that the execution overhead of the stack checksum exceeds the overhead of the register checksum to such an extent that the perturbation of the latter becomes negligible. In addition, discarding the option of using a register checksum when choosing a stack checksum solution will eliminate the possibility of detecting changes in register-bound variables over loop iterations.

**C.2 Adaptations**

Apart from the sheer size issue, there is another property that separates the register checksum from the stack checksum. In the case of the register checksum, we always use the same number of elements in order to calculate the checksum. If the processor has eight registers, the checksum will always calculate the register checksum by accumulating the values stored in these eight registers. Hence, we can guarantee a constant execution time as far as number of instructions are concerned. Using a stack checksum, the situation is different. The stack base of a program is constant whereas the SP varies over time. This implies a variable size of the program stack and thus a variable execution time of the stack checksum calculation, depending on the size of the stack at the time of the interrupt.

**C.2.1 Instrumentation Jitter**

Variations in execution time of software are usually referred to as *jitter*. In multi-tasking systems (such as most embedded RTSs), designers try to keep the jitter to a minimum, since it comprises the testability and analyzability of the system [99]. Therefore, jitter introduced by instrumentation activities (such as the stack checksum calculation) may complicate testing of sensitive systems, even though the instrumentation was included in order to increase the analyzability.
C.2.2 Partial Stack Checksum

To reduce the execution time perturbation and the problem of large instrumentation jitter when using the stack checksum technique, the developer has the option of not including the entire program stack in the stack checksum. A partial stack checksum $C_P$ would be calculated similarly to the original stack checksum (once again on a 16-bit platform):

$$C_P = (S_{SP} + S_{SP+1} + ... + S_{SX}) \mod 2^{16}$$

However, the upper boundary $SX$ of the stack interval to be included in the checksum is chosen such that:

$$SP \leq SX \leq SB$$

By using this formula, we once again reduce the percentage of the execution context included in the stack checksum, thereby reducing the accuracy of the unique marker approximation. In turn, we obtain the following benefits:

- **Eliminating instrumentation jitter**
  
  By defining $SX$ in terms of a constant positive offset to $SP$ (denoted $x$ in Figure C.1) such that the interval $[SP, SX]$ delimits a constant number of bytes on the stack, we make sure that the stack checksum will be calculated using a constant number of instructions. This will eliminate the instrumentation jitter of the checksum calculation (not considering cache effects or similar). In the case where the size of the fixed interval $[SP, SX]$ exceeds the size of the actual program stack (i.e., when $SX > SB$), this can be detected and the remaining instructions can be simulated using additions of zero to the checksum or similar.

- **Reducing instrumentation overhead**
  
  Intuitively, reducing the percentage of the stack included in the stack checksum will reduce the execution time of the stack checksum calculation. If total elimination of instrumentation jitter is no major requirement, a reasonable candidate for $SX$ may be the base of the stack for the current subroutine. Many processors are equipped with a dedicated register holding the value of this base pointer ($BP$ in Figure C.1) to the current scope of execution.
Figure C.1: Different delimiter alternatives for the stack checksum. The \( g() \) and \( h() \) intervals represent stack intervals for subroutine execution scopes for interrupted example functions \( g \) and \( h \) respectively.

## C.2.3 System call markers

Synchronous events, like blocking system calls, require a less elaborate approach. We make use of a per-task counter, incremented each time a potentially blocking system call is invoked by that task. If the call actually blocks the task, the value of the counter is recorded as a unique marker and the counter is reset. This way we will keep track of at which system call invocation the task actually blocked.
Bibliography


