Automated Selective Test Case Generation Methods for Real-Time Systems

Robert Nilsson

HS-IDA-MD-00-010

Submitted by Robert Nilsson to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the Department of Computer Science.

September 2000

I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has already been conferred upon me.

Robert Nilsson

Abstract

This work aims to investigate the state of the art in test case generation for real-time systems, to analyze existing methods, and to propose future research directions in this area. We believe that a combination of design for testability, automation, and sensible test case selection is the key for verifying modern real-time systems. Existing methods for system-level test case generation for real-time systems are presented, classified, and evaluated against a real-time system model. Significant for real-time systems is that timeliness is crucial for their correctness. Our system model of the testing target adopts the event-triggered design paradigm for maximum flexibility. This paradigm results in target systems that are harder to test than its time-triggered counterpart, but the model improves testability by adopting previously proposed constraints on application behavior. This work investigates how time constraints can be tested using current methods and reveals problems relating to test-case generation for verifying such constraints. Further, approaches for automating the test-case generation process are investigated, paying special attention to methods aimed for real-time systems. We also note a need for special test-coverage criteria for concurrent and real-time systems to select test cases that increase confidence in such systems. We analyze some existing criteria from the perspective of our target model. The results of this dissertation are a classification of methods for generating test cases for real-time systems, an identification of contradictory terminology, and an increased body of knowledge about problems and open issues in this area. We conclude that the test-case generation process often neglects the internal behavior of the tested system and the properties of its execution environment as well as the effects of these on timeliness. Further, we note that most of the surveyed articles on testing methods incorporate automatic test-case generation in some form, but few consider the issues of automated execution of test cases. Four high-level future research directions are proposed that aim to remedy one or more of the identified problems.

Contents

Chapter 1

Introduction

Modern computer systems tend to be increasingly complex. This is particularly true for distributed real-time systems, where the complexity of both distribution and real-time issues need to be considered. Moreover, distributed real-time systems often operate in safety-critical environments where the correctness of the software is imperative, e.g. vehicle control systems or space shuttle life support. These characteristics imply that there is a need for rigorous verification methods for such systems in order to detect and remove faults which otherwise could lead to critical mishaps. Unfortunately, industrial practice often resorts to case specific, ad hoc, solutions in order to verify the correctness of distributed real-time applications [Sch94].

1.1 Testing of Distributed Real-Time Systems

Testing is a method to dynamically verify software by execution in order to detect errors and failures. A problem associated with testing is that the execution of distributed real-time systems is generally non-deterministic and the number of test cases that need to be exercised in order to get a sufficient level of confidence in a system is huge. There are two ways to change the situation. One way to increase the testability of a distributed real-time system is to restrict the non-determinism of the system by introducing constraints on system behavior. Another way is to apply some method for selectively generating test cases that are more likely to reveal faulty behaviors than other and, thus, decrease the need to run redundant test cases [FHLS98]. Our hypothesis is that these two methods can be combined to systematically achieve a sufficiently high level of test coverage for distributed real-time applications.

1.1.1 Automated Selective Generation of Test Cases

This dissertation focuses on the second approach, selective generation of test cases for distributed real-time applications. There exist generally accepted methods for generation of test cases for sequential software, based on, for example, specifications or code structure [Bei90]. Real-time systems are in general parallel or concurrent; this complicates test-case generation and selection because the behavior of the system is dependent on the non-deterministic order in which tasks executes. Further, real-time systems may behave differently depending on time dependencies in program logic, adding another dimension to the test-case search space.

The process of generating test cases should benefit from being automated, which would decrease the test effort for the test engineer and allow automation of the whole test case execution process.

1.2 Motivation of Automated Selective Test Case Generation

Schütz has presented a seminal paper that addresses the issues that arise in the joint area of distributed real-time systems and testing [Sch94]. All the issues presented should be recognized in both the software testing and real-time systems communities. Of these issues, reproducibility, organization and representativity are the most relevant to our work.

According to Schütz, reproducibility is required to be able to do *regression testing*, i.e. to verify that the corrections of detected errors has removed the fault that caused the error and not introduced any new faults. Reproducibility is relevant because it implies requirements on the test cases generated for real-time systems.

Schütz points out that the test process is generally less mature than other phases of the development cycle and that the existing phases and methodologies do not address the issues that are important in real-time and distributed systems. The need for organization is relevant for this work since it motivates research toward a structured test process for real-time systems and, in our opinion, automatic selective test case generation is a part of such process .

Representativity is another problem in this context, because of the additional dimension of time in real-time testing, there will be a combinatorial explosion in the number of possible events and input parameters, making it practically impossible to test all anticipated real-world scenarios [GG75]. Therefore, Schütz introduces *test coverage*, which is defined as the fraction of anticipated real-world scenarios that has been, or can be, covered by generated test scenarios. It is generally impossible to give an absolute value on test coverage under this definition, but it can be specified as a relative measurement between different system architectures and test methods. This motivates a test method where it is possible to selectively generate test cases that get a good measure of test coverage; we believe that automation is desirable for attaining such measure.

1.2.1 Timeliness

Petters and Färber [PF99b] note that state-of-the-art processors have become more unpredictable due to multiple level caches and pipelines, therefore it is hard to produce a complete model of a modern processor in order to calculate worst case execution times. Petters' and Färber's article proposes a method that uses the control flow graph emitted from the compiler during compilation. The graph is analyzed and paths that cannot result in a worst-case execution time are removed. The remaining paths are instrumented and executed. During execution, the time is measured using special hardware. This article is not directly related to testing. However, execution of the system in order to verify the assumptions about time characteristics is relevant to testing, and the unpredictability of modern hardware motivates testing of timeliness instead of relying on models and static analysis.

1.2.2 Test Coverage Criteria

Yang and Pollock [YP97] highlight the problems that arise when you try to apply methods for generating test cases for sequential software on multithreaded programs. Methods for selective generation of test cases for sequential software are often based on a code-coverage criterion such as all-branch testing, which is not applicable for parallel programs. Yang and Pollock note that there is a need for a special coverage criterion for concurrent and time-related testing such as synchronization testing, execution-sequence testing, and communication testing [YP97]. Yang and Pollock's method, for timing-related tests, inserts delays in the code of concurrently executing threads in order to change the timing of different events to uncover potential faults. They also note that the number of timing-related tests that need to be executed is huge and opt for a solution that reduces this number. However, when testing concurrent real-time systems the probe effect associated with the introduction of delays will influence not only the order of tasks, but the behavior of the whole real-time system. A *probe effect*is when the facilities for testing influence the behavior of the system [Gai86]. The probe effect has been shown to be a major problem when testing real-time systems because most test facilities influence the temporal behavior of the system [Sch93], hence we believe that the method of Yang and Pollock is not applicable for concurrent systems with real-time properties. This motivates special test methods for testing concurrent real-time systems, and we agree that there is a need for a special test-coverage criterion for such systems.

Fujiwara et al. [FBK+91] cover a number of different methods for selecting test cases for conformance testing based on finite-state-machine specifications. These methods are based on different levels of transition coverage in finite-state models, e.g. a test coverage criterion is that all transitions between perceivable states should have been tested at least once. Further, issues that impact test case selection, such as architectural structure, interacting input parameters, and non-determinism, are discussed. The article concludes that it would be useful with a tool that not only generates a test-suite automatically, but also is able to determine its 'fault coverage', and based on this allow the selection of additional test cases for testing particular aspects of the specified behavior more thoroughly. This motivates an automated selective test-case generation tool.

Chapter 2

Background

This section presents the context of this work and gives the required background knowledge from the testing and real-time systems domains. First, the test approach taken in this work is discussed in relation to existing approaches. Second, our assumed test case generation model and system model are presented.

2.1 Testing Approaches

Testing is a wide concept that can be performed on different levels and for different purposes. This section contains the most common forms of testing and specifies approaches which are in focus in this work.

The two most common purposes of testing are conformance testing and fault-finding testing [Lap94]. *Conformance testing* aims at verifying that a specific implementation satisfies its specification while *fault-finding testing* aims at finding special classes of implementation faults. This work is not limited to one of these purposes, but will have a focus on conformance testing since test of system properties such as timeliness lies within this category.

Testing is performed at different levels during the software development cycle in order to strengthen the organization of the test process [Bei90]. *Unit testing* refers to testing a module of code with a specified functionality and interface. *Integration testing* is the process of verifying correctness of the interaction between such modules. *System level testing* is done at the implemented system as a whole; obviously, this requires that all parts of the system are completely implemented and integrated [Bei90]. Certain fault types can only be tested in a meaningful way at the system level, e.g. faults in the time domain [Sch94]. Methods that are applicable only to modules and for integration testing are not considered in this work, mainly because this kind of testing only verifies logical correctness.

In conventional software testing, a distinction is commonly made between structural and functional testing methods. In *structural testing*, test cases are based on system design, code structure and specifications of the systems internal architecture. In *functional testing*, the system under test is considered a 'black box' with unknown internal structure. In the latter kind of testing methods, test cases are created based solely on knowledge about system requirements and specification. Structural and functional testing is sometimes also referred to as white-box and black-box testing [Mey79]. This

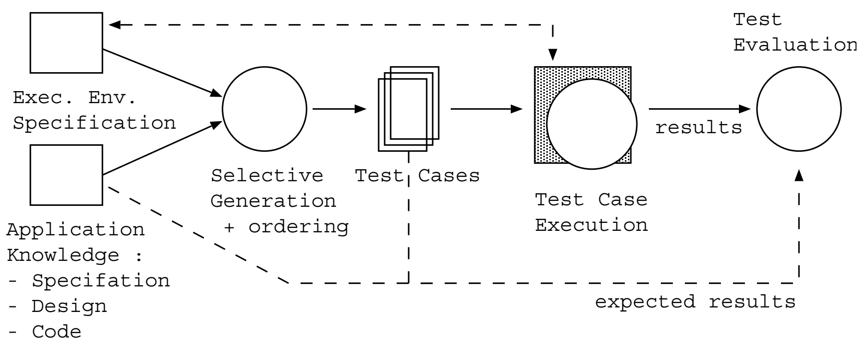

Figure 2.1: High level testing model

work consider methods based on all kinds of application knowledge, paying special attention to methods that exploit formally structured information for generating test cases automatically. Hence, this work does not exclude neither functional nor structural testing techniques.

2.2 Test Case Generation

This section gives a high level model of how test case generation relates to other parts of the test case execution process and the underlying assumptions of how testing is performed (see figure 2.1). Test case generation is assumed to be based on application knowledge such as specifications, code structure, and system design. This information is used to selectively generate a set of test cases, sometimes referred to as a *test suite*, that have high probability of revealing a specific class of errors or detecting deviations from the specification.

An *execution environment*, in this context, is the architecture in which the tested applications will run. This includes real-time operating system services, communication primitives and properties of underlying hardware. The execution environment gives certain constraints on applicability of different test methods, e.g. required contents of the test cases.

Generation of test cases may be done manually by an experienced test engineer or automatically by a test case generation tool if the required application knowledge is sufficiently formalized. Which test cases that are generated is decided in accordance with the test coverage criterion and is strongly related to the specific test method that is being used.

Possibly, the selectively generated test cases need to be prepared for execution on the specific target system and, if possible, sequenced in accordance to some heuristic function to decrease the time (and thereby effort) needed for finding the most critical faults.

When a test suite has been prepared it is executed on the system under test. The test case execution phase requires that the execution environment is sufficiently controllable and observable so that the desired test scenario can be enforced and executed. The term *observability* denote how well the system provides facilities for monitoring or observing the execution of application programs during testing [Sch93]. *Controllability* is associated with how well a tester can control the behavior of the system under test [Sch93]. During the execution of the test cases, the internal events are monitored and logged so that the information can be used for test evaluation and to help regression testing [TH99].

Test evaluation is the process of comparing the attained result of test-case execution with the specified or expected result. The result of a specific test case can be produced in the test case generation process or supplied by a test oracle. The problem with predicting the result of a test case is known as the *oracle problem*. The reason that this is a problem is that, in general the only way to get a result is running the implemented algorithm, that is the subject of testing. Test-case evaluation also incorporates validation against the test coverage criterion to determine if sufficient test coverage has been attained.

2.3 System Model

This section presents the system model against which the test generation and selection methods found during the literature survey will be evaluated. First, a more general model of the types of real-time systems under consideration is presented. Second, the problems with this model, from a testing perspective, are discussed and a more specialized model, constrained for testability, is presented.

2.3.1 Distributed Event-Triggered Real-Time Systems

Kopetz [Kop91] describes two ways to design real-time systems, *time-triggered* and *event-triggered*. The primary difference is that the first kind of system observes the environment and performs actions at pre-specified points in time, whereas the latter detects and acts on events immediately.

More precisely, a time-triggered real-time system has observation periods and action periods. At the end of each observation period the system detects all events that have occurred since the previous observation period and react to them by executing tasks in the next action period. Hence, the computations that are made in the action period must be finished before the next observation period starts. Further, scheduling in a time-triggered real-time system is generally made before the system is put into operation in order to guarantee that all tasks meet their deadlines as long as the assumptions about the environment holds.

However, time-triggered real-time systems have some disadvantages. Because of the off-line, static scheduling the system has to be re-analyzed, re-scheduled and re-tested every time something is changed in the real-time system or its environment. Further, time-triggered real-time systems suffer from a resource penalty since the system is designed to always handle the worst-case load, even if this very seldom occurs.

Hence, according to Kopetz [Kop91], event-triggered real-time systems are more flexible and extensible than time-triggered real-time systems. Event-triggered systems are generally less predictable than time-triggered systems because they assume less about the behavior of the environment and often schedule tasks dynamically, at run-time. Further, event-triggered real-time systems are vulnerable to event-showers (a large number of incoming events during a short period), which can cause an overload. An overload in this context occurs when tasks need more resources than the system possibly can provide. There exists methods for predicting and limiting the consequences of overloads, e.g., reject and abort all non-critical requests when the load rise above a specified level.

This work focuses on solutions for event-triggered real-time systems for two reasons. Primarily, we believe that modern real-time system will need to be flexible in order to support the needs of future real-time applications. We believe this because the diversity of services provided by embedded system has increased, e.g., Japanese speaking escalators, and it is impractical and expensive to have special-purpose architectures for each new application service. Second, even though event-triggered real-time systems are more extendible than their time-triggered counterparts, they still need to be re-tested when changes are introduced in the system or its environment. Therefore, we see that there exists a need for simple and time-efficient testing methods in this domain. However, testing of event-triggered real-time systems is much harder than testing time-triggered systems, which make progress in this domain more valuable.

When we refer to real-time systems in the subsequent sections, we mean event-triggered real-time systems, unless the contrary is explicitly stated. Many of the statements about such real-time systems hold for time-triggered systems as well, but are not relevant for this work.

Most real-time systems are inherently distributed, since all sensors, actuators, and computing elements seldom can be placed together. Distribution in this context does not only mean that components are distributed geographically, but a distributed system should consist of several autonomous nodes, each one with processing capabilities for providing services for some part of the overall system. There are several advantages to a distributed architecture, for instance fault-tolerance, scalability, and reduced cabling costs. But distribution generally comes with a cost of complexity because the different nodes need to communicate and synchronize in order provide a correct overall behavior of the system. Communication and synchronization between systems with real-time constraints are common, but not trivial, therefore these kinds of systems need to be effectively tested.

In this work we consider testing of distributed real-time systems interconnected by a network that guarantees timely delivery of messages and provides means for keeping a distributed global system clock.

2.3.2 Issues in Testing Event-Triggered Systems

As mentioned, the fact that we are dealing with event-triggered real-time systems complicates testing. In event-triggered real-time systems, events can be introduced at every observable point in time, and the behavior of the real-time system may be different for each combination of events types and their times of occurrence. Hence, the number of behaviors that can occur in an event-triggered real-time system is theoretically infinite due to the non-discrete nature of time.

A task that executes in an event-triggered real-time system can generally be preempted an unlimited number of times, and each combination of preemptions potentially results in a different state of the overall system. The enormous number of states complicates testing of event-triggered systems, since in order to gain confidence in some aspect, the introduction of an event may need to be tested at each possible system state [Mel98].

Furthermore, if the number of transactions that can be triggered by the same event type in a real-time system is unbounded, the maximum number of transaction-instances in the real-time system at a specific time is also unbounded. This causes problem when testing, since for each state of the system that has been tested, there is another state, with one additional transaction pending, that also needs to be tested.

Most real-time systems have a set of shared resources that many transactions need to access, e.g. communication channels, shared memory buffers, and sensor representatives. Interaction between transactions and shared resources may cause blocking, how this is handled also has an influence on testability [Bir98].

2.3.3 Testable Event-Triggered Real-Time Systems

The goal with the model presented in this section is to keep the flexibility and extendibility of distributed event-triggered real-time systems while solving or limiting the impact of the complex testability issues associated with the event-triggered paradigm.

This model has been proposed and is under development by the distributed real-time systems research group at University of Skövde. In the core of the architecture is a distributed real-time database, this is not a requirement for the applicability of test methods, but it has been shown how this can provide valuable services for application-testing [Mel98].

One of the advantages with the real-time database paradigm is the use of *transactions*. Transaction processing systems provide transactions with properties such as atomicity, isolation and durability. Usually consistency is part of the properties of transactions but since this is the responsibility of the transaction programmer it may be subject to faults and cannot generally be guaranteed [GR93].

Atomicity means that the changes performed by a transaction on the state of the system is atomic.

Isolation means that even if transactions execute concurrently, it appears to each transaction that all transactions is run in series.

Durability means that once a transaction has successfully completed, its changes to the state survive failures [GR93].

These properties simplify application development in the architecture as well as having a positive effect on application testability.

Since we have a real-time database, transactions is assumed to communicate trough protected shared variables, e.g., database objects. The concurrency control policy is assumed to be conservative and pessimistic so that transactions must lock all resources that it need before executing [Bir98]. These assumptions reduce the number of synchronization sequences, e.g., once a transaction has started executing it can not be blocked so sequences where blocking occur during preemption can be excluded. Hence, testability is improved [Mel98].

In order to improve some of the testability issues of event-triggered real-time systems, Mellin [Mel98] has suggested the following constraints on system behavior:

1. The number of times a transaction can be preempted must be bounded.

2. The number of concurrently executing transaction instances, triggered by the same event type, must also be bounded.

3. The number of points in time where the environment can be observed must be constrained.

The constraints put a bound on the test-effort by limiting the number of internal states and possible sequences of events. Hence, by using these constraints an upper bound can be put on the test-effort for an event triggered real-time system.

Further assumptions about the execution of the system and how test case execution is performed in this model are presented in the two following sections.

Execution Model

The execution model assumes a dynamic preemptive scheduling algorithm that can handle sporadic and periodic transactions with both soft and hard deadlines. Trough off-line analysis all hard transactions should be guaranteed to meet their deadlines and the priority of transaction instances is assumed to be unique. This model also assumes that a soft transaction can be aborted from the processor if it has been preempted its maximum number of times and a transaction with a hard deadline arrives. If a soft transaction is aborted it is assumed to be resubmitted if necessary. The time to abort a soft transaction must also be bounded.

Test Case Execution Model

The test case execution sub-model presented by Birgisson, Mellin and Andler [BMA99] has been somewhat refined by Nilsson [Nil99]. According to this model, test-case execution is conducted by first forcing the system into a specific internal state. An *internal state*, in this context, is considered to be an equivalence class of states represented by the state of the active transactions in combination with the state of the shared resources. Then, from this internal state, the system is started and sequences of events (with associated parameters, e.g., oxygen level given by a space shuttle cabin sensor) are supplied by a test driver. During this process, internal state-transitions are logged by a test monitor, these logs may later be used for analyzing the test runs and for detecting errors in, for example the time-domain. In order to prevent the probe effect, the test monitor has to be left in the system during normal operation.

Chapter 3

Problem Definition

This section contains the aims and objectives of this dissertation, the problems under consideration are specified further and the approach taken to pursue the aims and objectives are presented.

3.1 Pursued Aims and Objectives

One of the aims of this work is to survey methods for generating and selecting test cases for distributed real-time systems. The survey is concentrated on distributed real-time systems, but also explores test case generation for conventional concurrent systems. A second aim is to conduct an analysis, based on the survey, to determine which methods can be used when automating the test case generation process for distributed real-time systems. Further, a third aim is to identify problems and prerequisites for automated test case generation for a specific system model and to propose future research directions in this area.

In order to achieve these aims, the following objectives were identified. (i) Articles, books and conference proceedings in the field of test case generation for timed and concurrent systems are systematically searched for appropriate methods. (ii) Terminology used in association with existing test case generation methods is identified and appropriate definitions adopted. (iii) The methods are analyzed from relevant perspectives. (iv) The applicability of the identified methods for the target system model is evaluated, as well as requirements and inherent problems with the methods. (v) The unsolved problems and open issues found during the analysis and evaluation are assembled to a set of promising research directions.

Below, objective (i) is briefly described in section 3.4, while objective (ii) is described in section 3.1. A more extensive review of objective (iii) is made in section 3.3, while objectives (iv) and (v) are summarized in section 3.5.

3.2 Real-Time Testing Terminology

The concepts within the testing area are used in many sub-domains of computer science - but every domain, from hardware testing to graphical user interface design has its own definitions. Therefore, it is important to choose the most appropriate definition of the most commonly used concepts so that inconsistent use of concepts and unambiguous reasoning can be detected and avoided during the analysis.

3.3 Perspectives on Test Case Generation

Selective test case generation can be viewed from different perspectives depending on what problem is currently being considered; this work tries to address some of the most relevant perspectives for problems related to automatic test case generation for distributed real-time applications. These are timeliness, automation, test coverage criteria and reproducibility. Of these timeliness is covered in some detail whereas automation and test selection criterion are only partially covered. Reproducibility is identified as an important perspective but is left as future work.

3.3.1 Timeliness Perspective

Timeliness is one of the most important properties of a real-time system. The goal of testing timeliness is to verify that the temporal characteristics that are assumed during the design are feasible. It has become increasingly complex to model a state-of-the-art processor in order to predict timing characteristics of tasks, e.g. worst-case execution times [PF99b].

Testing of timeliness does not rely on the accuracy of such models; instead the system is executed and monitored so that timing failures can be detected. Since the behavior of the system depends on the test case data, it is interesting to generate test cases that stress the timeliness of the system.

This perspective identifies existing methods to handle this problem.

3.3.2 Automation

From this perspective we explore which of the identified methods are applicable when aiming to automate the test case generation process. There has been previous work on automating test case generation, mainly for sequential and non real-time software. Existing methods incorporate everything from tools for aiding test-engineers to fully automated solutions.

It has been argued that the process of creating test cases can in itself help detect faults in the software, because the designers have to think about the system in another way [Bei90]. This is probably true, but since the number of test cases generally needed to achieve a sufficient level of confidence in safety-critical software is astronomical [Sch93], and therefor the time consumed in manually devising test cases could be better spent correcting the errors detected by an automated test method.

The automation perspective is orthogonal to the other perspectives, and this work will include a judgment of how hard it is to automate the identified methods.

3.3.3 Test Coverage Criteria

The perspective of test coverage criteria is related to test case selection methods. From this perspective it is interesting to determine what coverage criteria that are appropriate for distributed and real-time applications. Generally, it is impossible to perform exhaustive testing of an application because the test process has limited time and resources allocated while the number of possible test cases often is unbounded [GG75]. A test coverage criterion let you set a lower goal of when you have tested an application sufficiently, common test coverage criteria for sequential software are, for example all-branch coverage or statement coverage.

For parallel or concurrent software these criteria are less meaningful, since you have interaction between different threads and processes that influence the results [YP97]. Further, there exist coverage criteria for pure black-box techniques that focus on transition coverage and equivalence classes derived from specifications. An interesting aspect is to evaluate which criteria that are most useful for distributed real-time applications.

3.3.4 Reproducibility

Another important perspective is from the point of view of reproducibility. The contents of the test case should support that the system shows the same behavior when the same test case is executed multiple times. This may seem trivial, but is really a big problem for test case generation and execution due to the non-determinism of concurrent systems [TH99].

Test case generation for sequential systems is often only concerned with generating sets of input data. When testing a concurrent or distributed real-time system it is not sufficient to generate input data, since the behavior of the system also depends on the current system state, the exact time when input data arrives and properties of the execution environment, e.g. scheduling policy or communication delays. This information must be associated with the test case to get a reproducible test case execution.

3.4 Approach for Survey

The strategy for the survey has been to search scientific databases, journals and conference proceedings for papers covering the selected perspectives (see section 3.3) paying special attention to papers in the overlap area of the two first perspectives. From these papers, references and citations has been followed and evaluated, outlining the area of interest.

3.5 Analysis and Future Research

Based on the survey of methods, an analysis has been conducted to determine which methods are applicable to our specific system model and which issues and problems that need further investigation. The system model under consideration is an event-triggered, distributed, real-time system with some additional mechanisms to improve testability, e.g. transaction-based processing, access to database storage and constrained observation (see section 2.3.3). This system model is part of the result of previous research related to improved testability of distributed real-time systems [Mel98, Bir98, Lin99, Nil99]. We focus on analyzing the methods from the selected perspectives with respect to our specific system model. This analysis results in a specification of unsolved problems so that relevant future research directions in this area can be extracted.

Chapter 4

Generating Tests for Timeliness

This chapter presents the state-of-the art in test-case generation for testing of time constraints in real-time systems. The papers that are relevant from the timeliness perspective have been classified according to a set of criteria; the categories and criteria are explained in this section. This section also presents some of the most significant work in each category and compares it to other similar approaches found during the survey. Further, an analysis of the methods form a timeliness perspective is presented and evaluated against our system model.

4.1 Classification Criteria

The methods and existing works that have been encountered in this perspective are classified in relation to each other, forming taxonomy over test case generation methods for testing temporal properties. From the timeliness perspective, the following classes have been identified:

Specification Type

This class categorizes the methods after the type of specification from which test cases are derived. Four main categories have been identified; temporal logic, finite state machine specification, Petri-net specifications and process algebra. These categories are not disjoint since they often have equivalent expressive power and the specifications can be translated between the different models. There also exist more informal methods for deriving test cases, these methods are categorized as "informal" in this class. The reason for this classification is to identify if a specification type has inherent advantages or problems when used for test case generation.

Test-Case Contents

Different methods supply various degree of test-case detail. Some methods only supply input data and rely on some other mechanism to validate the correctness against a temporal specification. These methods may not seem to be aimed at testing temporal correctness, but when the execution times of a task depends on the input data, the correct input data has to be acquired in some way.

Other methods aim at generating complete execution sequences with series of input events and expected output events, this class of test data will be denoted "event sequences" in this work. A special version of this type of representation is executable test processes, which should be run in parallel with the application, supplying inputs at certain points in the execution and response on outputs from the system. A test process can potentially incorporate both timed input data, and expected results.

Another possible category is test cases which includes the initial internal state of the system in the test case, i.e., a specification of the internal state from which the test should be applied are included in the test case.

Time base

From the timeliness perspective, the time base used in test case generation is interesting. A continuous time base makes it possible to specify time constraints and their relation in a more precise way than in its discrete counterpart. However, this ability makes the number of test cases increase dramatically because between every two tested scenarios, there is another possible scenario.

4.2 Test Case Generation Methods

This section presents the most significant works from the timeliness perspective. The layout of this section follows the categories of the specification type classification.

4.2.1 Methods Based on Process Algebra

A process algebraic language is a way to specify the behavior concurrently running processes. These can be symbolically executed in order to model the behavior of concurrent applications and protocols. Because the process algebra is executable, it can also be used for specifying test processes which can be used both for symbolic execution together with the specification, and executed in parallel with an actual software and hardware system. This is also convenient for test case generation purposes.

Clarke and Lee [CL97] presents a framework for testing timing constraints of real-time systems. The time-constraints are specified in a constraint graph that specify intervals in which events can occur and dependencies between events. These are translated to processes that are specified in the algebra of communicating shared resources (ACSR). Test processes produced from the process algebraic specifications can be used to validate the specification model of the system as well as be converted to test cases for the actual software implementation. The test processes are run in parallel with the system under test and supplies timed inputs to the system and intercept outputs. Further, the article provides taxonomy of timing-constraint faults and a set of test coverage criteria for detecting various types of implementation faults. The test-generation method focuses on deriving tests close to the end-points of the intervals at which events are specified to occur. It is noted that it is desirable to derive test suites that cover as many test scenarios as possible, while avoiding redundancy. It is expressed that some sort of heuristic function is needed to make the test suite derivations tractable.

The method is a functional method since it is based on a constraint graph that is part of the specification. Even if the constraint graphs may appear to be an informal specification, the test cases are derived from and ultimately expressed in ACSR, which is process algebra.

As mentioned above the test cases produced are executable pieces of ACSR specification which supply timed input-events and expect outputs at certain points in time. An advantage with this is a possibility to create tests for long running execution sequences. The target system for this approach is obviously real-time systems and protocols. However, the generated test cases are primarily aimed for testing the execution specifications written in ASCR, but the test cases can then be reused for testing actual software implementations even if this is not explicitly investigated in this paper.

Cleveland and Zwarico [CZ91] propose a method that incorporates a framework for generating "behavioral preorders", which is a specification in process algebra that are suitable for reasoning about real-time characteristics of reactive systems. The main problem with this paper is that it is mainly concerned with the problems arising when describing timed systems with a process algebraic notation. No concrete method for test case generation for software systems is given. But since the method aims to generate tests for the purpose of verifying time constraints, the paper still fits under the timeliness perspective.

4.2.2 Methods Based on Finite State Machines

A finite state machine is a well-known abstraction of computers systems. In general, finite state machines consist of states and transitions. States usually represent some instantiation of variables at a specific point in time. The transitions represent an action or an event that cause the state to change. An interesting property of real-time systems is that the passing of time is an event that may cause a transition in a finite state model. In this section, methods based on three different flavors of finite state machine models, used for test case generation, are presented.

Timed I/O Automaton

Petitjean and Fochal [PF99a] present a test architecture for testing timed systems. In this work, a timed system is described as a timed automaton that can be expressed as a region graph. This work is built on the timed I/O automaton theory proposed by Alur and Dill [AD94]. In their model every automaton has a set of states and a set of clocks. The clocks are represented by real variables that proceed at the same rate and can be reset by any transition in the system. The refinement of this model by Petitjean and Fochal is a clock region definition that can capture the timing constraints in the model. A clock region can be visualized in a graph with one clock variable on the x-axis, another clock variable on the y-axis, and time constraints as vertexes defining regions in the graph. The new definition of the clock regions allows timing constraints to be represented by a region graph where the number of clock regions becomes smaller than in the original model by Alur and Dill. After the transformation, the second step is to use subsets of the clock regions in the region graph and the original automaton to create an untimed labeled transition system. According to the article any conventional test case generation method for untimed labeled transition systems can be used on this model, e.g. [Tre97].

Further, the paper presents a test-architecture, describing how test cases are exercised on the system under test. The architecture takes the management of clocks away from the implementation under test and lifts it to the level of the tester. The reason for this is that if clocks must be managed within the implementation, it must use timers that correspond to the real-valued clocks in the specification in order to use the proposed test method. Instead, real-value clocks are kept within a read only module of the test-driver and these clocks are supposed to be equivalent with the clock representation in the system under test. When the system under test generates output events, the test driver verifies these with the clock zones.

In order to get sufficient test coverage, test cases are selected from each vertex that makes up a clock region in the region graph. This policy guarantees that at least one test case has been exercised for each combination of timing constraints.

The specification is classified as a finite-state-machine specification since an input-output automaton has most in common with this formalism. This method requires the use of special-purpose test drivers for managing clocks that corresponds to the clocks in the specification. The approach is mainly a conformance testing approach since it verifies that the temporal behavior of the system satisfies the timing constraints expressed in the timed input output automata.

Like most finite-state-machine approaches, these test generation algorithms have problems with a state explosion as soon as the number of states in an actual implementation is greater than in the specification model or the modeled system behaves non-deterministically. This is one of the main problems with this approach and the problem is often encountered in other related approaches as well.

Timed Transition Systems

Another method which takes a finite-state-machine approach is Cardell-Oliver and Glover [COG98]. The purpose of their article is to present a formal method for generating conformance tests for real-time systems. The paper starts from a formal-methods perspective, where software specifications is iteratively refined and formally verified at each step. The authors note that the last step from symbol manipulation to an actual hardware and software implementation requires extensive conformance testing since the implementation no longer is a provable description. According to Cardell-Oliver and Glover, a formal test method has four stages,

1. checking the test hypothesis,

2. generating test cases,

3. running the tests,

4. evaluating the test results.

The test hypothesis defines assumptions about the property under test so that correct conclusions from the test results can be drawn. This method requires that the system under test can be viewed, at some level of abstraction, as a deterministic finite state automaton. According to this paper, a test method is complete if "its test generation algorithm determines a finite set of test cases sufficient to demonstrate that the formal model of the implementation under test is equivalent to its specification". Likewise, a test method is practical if the number of test cases is not "too large". The specification language used in this method is Timed Transition Systems (TTS), but the authors claim that the method is adaptable to other formal languages such as timed CSP.

The basic idea of the method is to generate test cases that exercise all transitions between states and compares observable variables with their specification counterparts. Each transition in the TTS specification has an associated clock guard in discrete time units that is used for expressing time constraints. The required input for this method is a specification of the system as a TTS process as well as a specification of its environment. From these models a timed action automaton is derived by symbolic execution. The test case generation algorithm takes the timed action automaton associated with a TTS specification and constructs a finite set of test cases that will test each timed action edge of the automata. Each test is a sequence of timed actions.

The authors note that their method gives a huge number of test cases, but suggest that methods should be applied to eliminate redundancy. They also claim that if a unique sequence of output events can identify each state, the number of required test cases for conformance testing can be decreased. However, even if there are a limited number of test cases with this approach (for a deterministic model), exhaustive testing is only possible if you can divide the system to "manageable pieces".

Extended Timed Input-Output State Machines

The ideas presented above has much in common with an earlier paper by Kone [Kon95] who is one of the first to present a method for designing tests for time constraints. However, the main contributions of his work are a formal approach to model and a theory for testing time-dependencies in system behavior. The approach will not be described in detail here since it is similar to the previous methods and does not handle test case generation explicitly, however the article is mentioned in this context since it is one of the seminal works in the area.

Laurencot and Castanet [LC97] shows different existing methods to integrate time in existing formal modeling languages so that it is possible to test timing constraints. The article also aims to show how the notion of time can be integrated in test cases for distributed systems and protocols. The importance of time in test cases for time-dependent systems is highlighted. Three formal methods from other authors are presented and the notion of time in each of them is discussed.

The different formal methods are the automata of Alur and Dill [AD94], Timed Transition Models [COG98], and Extended Time Input Output State Machine(ETIOSM) [Kon95]. Based on the latest notation, a canonical "tester" that handles time is presented. The canonical tester is aimed for testing the implementation relation, and hence, this is a conformance testing approach. The method decomposes timed formal models, in this case ETIOSM, into sub-models for each real-valued timer. These are then analyzed according to a set of rules and used for building the tester that incorporates time. According to the article, test cases given by this method can be formulated in TTCN test language. An interesting issue is it that the authors claim that their tester is very well suited for testing distributed systems or protocols, because of the use of a transition for each transmission and reception. Also, propagation delay is part of the models.

4.2.3 Methods Based on Temporal Logic

Logic has been used since the days of the ancient Greek philosophers as a fundamental instrument for modeling and reasoning about states, causes, and effects in the world. In order to model behaviors that incorporate time, special temporal-logic languages have been developed. These logic languages can be used to write specifications, requirements on systems, and as a base for test case generation.

Mandrioli et al. [MMM95] present a method for functional, testing of real-time systems based on specifications of system behavior in TRIO. TRIO is an extension of a temporal-logic language defined to deal explicitly with strict time requirements.

The basic idea behind the method is to decompose a specification of behaviors into elemental test case fragments that later can be recombined to form test cases consisting of combinations of inputs and expected timed outputs.

The article describes a tool for helping a test engineer to device test cases with this method. The test generation tool includes a history generator and a history checker component. The history generator is used for deriving all elementary test cases satisfying a TRIO formula at a given time instant. The elemental test cases are basically timed input output pairs. These can be combined and shifted in time to create a large number of partial test cases. In order to get a manageable test suite, a heuristic function combined with human guidance is used for selecting which elemental test cases should be combined in order to satisfy some test coverage criterion. The article claims that most coverage criteria can be used with this method. A problem is how to relate actual output events of the system to the specified output events in the test case, in particular for non-deterministic specifications. This problem is solved by a history checker which analyzes the generated history produced by the system under test to determine which test cases that has been satisfied. The paper shows the applicability of their method by applying it to a number of implemented real-time systems. One problem with the method is that it can be hard to model large, complex real-time systems since the behavior has to be modeled on a very detailed level for the test case generation process to work.

The test cases in this method consist of event sequences. The length of these is not specified but we suspect that the number of test cases would grow indefinitely with the length of the sequence. The target model for the test cases are the actual real-time system, the logic specifications in TRIO are executable, but they are only used for extracting parameters for the test cases.

In a more recent work, SanPietro et al. [SMM00] expand the work by Mandriolli et al. to incorporate high-level, structured specifications that can be combined with the low level specifications proposed in the earlier work. The difference in this work is that the language allows modularization and presents a graphical notation of modules and components. The authors point out that test cases must be generated from the part of specification that models the environment even if we only want to produce test data. In this work there is a distinction between test data, which are input values, and test cases, which are a sequence of interactions. The structured model is translated to a graph which is traversed by an algorithm to construct execution sequences for the overall system based on combinations of the existing low level input/output ’‘histories” (cf. [MMM95]). The test case generation semi-automatically constructs ”execution sequences” in accordance with some coverage criterion. The authors points out that some problems arise when the specifications is nested so that the graphs becomes cyclic. A solution to this problem is presented which use known graph-transforming algorithms to remove a minimal set of edges to get at a manageable graph. The removed edges are considered in a subsequent step of the method.

The article also presents the tools that they have developed for supporting their method, as well as a case study with experiences from testing an industrial size application written in a modular specification. Future work is to support generalization and inheritance in specifications and to fully implement tools that support more than one formalism.

4.2.4 Methods Based on Timed Petri-nets

A Petri-net is another notation that has been successfully used for modeling concurrent and real-time systems. It is based on a set of states and transitions with tokens that represent the program control flow. In some of these models, time can be represented, and hence such methods can be used for generation of test cases for real-time systems.

Braberman et al. [BFM97] present a method for generating test cases for real-time systems based on timed Petri-nets. The method aim to extract knowledge not only from the specification or code but from all steps during development, e.g. design. In our opinion, the test cases generated with this method should be able to test timing constraints. The article also aim to present selection criteria and adequacy criterion for testing timing constraints. The article suggests the use of a design notation SA/SD-RT for specifying the behavior of a real-time system. According to the authors [BFM97], SA/SD-RT is an intuitive, graphical design language that can model synchronization primitives, message queues as well as tasks and timing constraints.

For test purposes, this design specification is translated to a high level, timed Petri-net notation from which a timed reachability tree (TRT) can be derived. Each path from root to leaf in this tree represent a "situation" that in itself represent a potentially unlimited number of test cases due to a continuous time base. The "situations" are the base for generating test cases. The authors present different reduction schemes (selection criteria) that can be applied to reduce the number of test cases given from each situation. Hence, the higher-level adequacy criteria presented in the paper are based on "situation" coverage, e.g. "one is enough" is satisfied if one test execution from each situation is executed. Listed future work is to create a tool which helps to manage and automate the method, an extension for applying the method on isolated components, and also to investigate how architectural information, such as scheduling policies, can be taken into account during test case generation. An identified but unsolved problem with this method is that there is no way for deriving actual input test data, only temporal information about sequences of events can be produced, e.g. what timed input sequences are needed to test all situations.

In an earlier paper, Morasca and Pezze [MP90] present a method for testing concurrent and real-time systems that uses high-level Petri-nets both for specification and implementation. The nets proposed in this paper are Er-nets, which can handle concurrency and where time is modeled as a special property associated with the transitions in the model. Problems with testing concurrent and real-time systems are presented, high-lighting the problems with non-determinism and combinations of test-scenarios. Six different adequacy criteria based on the Petri-net models are also presented (these will be discussed in section 6.2).

4.2.5 Informal Methods

Methods that are not based on a specification are generally not applicable for test case generation aimed at testing temporal properties. The reason for this is that in order to determine if some transaction violates a temporal constraint, e.g., misses its deadline, the constraint must be specified in some way. However, informal methods can be used for finding test data which influence the temporal behavior of a transaction in a specific way. For example, there exist statical methods for finding the longest execution path of a transaction [MW98, PF99b]. Other examples of such methods are based on genetic algorithms.

Genetic Algorithms

A *Genetic Algorihm* is basically a heuristic search method which is inspired by Darwinistic evolution theory. The algorithm takes a population of (for example) randomly generated solutions to some problem. These are then evaluated according to a fitness function; e.g., in a shortest path problem, the fitness could be inversely proportional to the length of the path. The solutions with the highest fitness values (selected in a probabilistic way) are changed by crossover and mutation to a set of new solutions. The so-called schema theorem shows that the fitness of the overall population will increase with the number of iterations. The main advantage of this search method is that it effectively avoids local optima in the search space and therefore is applicable in multi-dimensional domains with discontinuous search spaces [Mit96].

The temporal behavior of a software program as a function of the input data is a discontinuous function, since a very small perturbation of some input parameter may result in large changes in the temporal behavior, such as executing another branch with entirely different temporal properties. Hence, genetic algorithms can to advantage be used in this domain to find specified temporal behaviors of programs.

Wegener et al.[WSJE97] propose a method that uses genetic algorithms to generate test data for testing temporal properties of real-time systems. The method aims to find the longest and shortest execution paths of real-time programs. The paper shows that the genetic algorithm method easily outperforms a random method in finding the longest and shortest execution paths, measured in the number of clock cycles of program execution. The fitness function is supplied by a simulation tool, which counts the number of executed machine level instructions and summarizes the associated cycle times. This is only an approximation of the real execution time and the article states that temporal testing must be repeated for each platform where the software will run, and this further emphasize the importance of automation. One problem is that there is no guarantee that a genetic algorithm approach can find the extremes, because the decision on when to stop the search is arbitrary. Further, the genetic algorithms have no knowledge about the specified application behavior, but only aim to supply test data that naively try to satisfy the requirements of the fitness function.

In a later article, Müeller and Wegener [MW98] compares an evolutionary testing (ET) method based on genetic algorithms to static analysis as a mean for estimating and verify the worst and best execution times for transactions. In the static analysis methods used in the experiments, execution paths and hardware characteristics are simulated to make a prediction of the worst-case execution time.

Müller and Wegener points out that most methods for verification of time constraints are based on some sort of formal methods and specification technique, whereas industrial practice seldom use these formal methods. Further, the authors claim that most formal methods are restricted to small systems and that there currently not exists any specialized methods available that are suited for testing temporal behavior of industrial size real-time systems.

The static analysis method used in this comparison is based on an optimizing compiler that has been modified to emit control-flow information and data information from C/C++ source files. This information are used by an Timing Analyzer and a static cache simulator which can help an user produce estimations of both worst and best case execution times.

Evolutionary Testing (ET) uses a standard genetic algorithm for generating inputs; the fitness function is supplied by an off the shelf performance measurement tool. The article argues for evolutionary testing over static analysis (SA), arguing that it is versatile since it considers the system under test as a black box and there is no need to model the behavior of internal components such as caches and pipelines. However, as mentioned previously there are drawbacks of evolutionary testing as well, for example, there is no means for deciding when an optimal path has been found and some other method is needed to decide test coverage.

The different methods are used to predict, and measure the worst and best case execution time for five different real-time algorithms with varying amounts of complexity and input-variables. In all cases the SA method got a more pessimistic estimation of worst-case execution times than the ET method found.

The authors thus believe that the SA method may be too pessimistic. However, since it is inconclusive whether the ET method finds the real worst-case scenario, estimations made by the ET method are unreliable.

The authors suggest that SA methods be used for designing hard real-time system, but that such systems also should be verified using ET. For soft real-time systems, it is possible to use ET for estimating the timing constraints for the design, to get a more optimistic estimation. It is also possible to use a mean-value between the two estimation and measuring methods. Future work includes a method for deciding test coverage metrics, such as path coverage.

4.3 Analysis and Evaluation

In this section, the different approaches will be analyzed according to the timeliness considerations of our target system model.

One conclusion that can be drawn from all the presented methods is that very few consider internal states of the system when generating test cases for testing of timeliness. The only approaches that come close are methods based on finite-state machines or Petri-nets. However, in the case of finite state automata, it is difficult to maintain a relationship between the states of the automata and the internal states of the software system because the state machine models often abstract away from mechanisms in the execution environment. Further, if non-determinism is allowed in the implemented system, finite-state-machine methods seldom apply. The same is generally the case for Petri-net-based methods; the possible exception is the work by Braberman [BFM97] where ”situations” can be considered internal states. However, the situations are based solely on specification about timing constraints and properties of the design. A refinement to include, e.g., constraints in the execution environment, is needed if this method should fulfill the requirements of our test model.

Other methods consider the system as a black box and only verify that the time from when input events are supplied to the time where output is signaled corresponds to the constraints in the specification. This is indeed a relevant problem, but for possibly safety-critical systems with non-deterministic execution times and execution orders, it is generally not sufficient to supply different sets of timed inputs. Parameters and internal behavior is bound to have an impact on timeliness, i.e. different paths in the code are executed for different classes of input data.

Another related observation is that very few of the specification based testing methods supply input parameters. The only input that is assumed is events and timing between events. This may be sufficient in some control applications, but in general a real-time system has to read data values from sensors in its environment and act differently upon the various data values.

An interesting observation is that most methods assume an event-triggered paradigm; none of the test case generation methods assume that input events are constrained in any way. The only constraint on system behavior in this context is that some of the works assume a discrete time base. A disadvantage with this is that it is hard to determine how well the methods work in a constrained environment. However, it would be interesting to see if constraints, such as the ones in our testable real-time system model, could be used in test case generation or selection methods to be able to reduce the number of test cases. For example, some input scenarios may be considered equivalent in a more sparse time base, or in a system model with constrained observation (see section 2.3).

The only method that considers distributed systems explicitly is Laurencot and Castanet [LC97]. This approach is finite state machine based. It is inconclusive how well this method scales to large real-time systems and how the states in the specification can be mapped to internal states of the system. Nevertheless, it is in our opinion a promising research direction. The fact that this is the only test-case generation method found in the survey that considers distribution explicitly is a problem since our target model is distributed and distribution is an inherent aspect of real-time systems.

Chapter 5

Automation Issues in Test Case Generation

This section deals with test-case generation from the automation perspective. This survey covers methods for automatic test case generation. Since there exists a huge amount of methods, mostly for sequential unit testing and modular testing, this part of the survey try to give a representative subset of approaches aimed for automatic test-case generation at the system level.

First, the classification criteria of methods from this perspective are presented. Second, a short summary of some of the most significant works in each category is presented.

5.1 Classification Criteria

From the perspective of automation, it is important that the test case generation methods support automation in various ways; the classification criteria from this perspective are presented here.

Test Case Design Paradigm

The two main categories for automatic test-case generation methods in this dissertation are functional and structural methods. There are different hybrids between these categories but we have chosen to exclude these and categorize methods that use anything else than the specification as structural methods. Naturally, for functional testing, the specification type is important, but since this is included in the classification from the timeliness perspective, it will not be repeated here.

Degree of Automation

In this classification the degree of automation, which is achievable with the method, is specified. The categories are:

1. fully automated,
2. semi-automated,
3. potentially automated,
4. not automated.

A semi-automated method is a method where the user has to interact with a tool or algorithm during test-case generation, i.e. to guide the generation process. A potentially automated method is a method that has not been automated, but where there are no inherent problems with creating a tool that automate the process.

5.2 Automated Test Case Generation Methods

In this section some of the most significant work for automatically deriving test cases is presented. The methods are presented in a thematic way, trying to cluster similar approaches in a more general group. These groups are: labeled transition systems, automation frameworks, evolutionary testing, Z-based methods, and methods based on model checking.

5.2.1 Labeled Transition Systems

A labeled transition system consists of states and transitions (with associated actions) that can be used to model the behavior of processes. Labeled transition systems also serve as a semantic model for many well-known formal languages, e.g. CSP [Hoa85]. Tretmans [Tre97] presents a test-case generation theory based on input-output transition systems, a special kind of transition systems where inputs are always enabled, aimed for automatic conformance testing. According to him, conformance testing based on labeled transition systems has been studied earlier for testing communication protocols, using languages as, for example, LOTOS [ISO89]. The test cases in this work are specified as a labeled transition system that interacts with the system under test. The interaction continues until no more interaction is possible, e.g., this could be when the test case reaches its final state or when a test case expects an output from the implementation that is not produced.

Simplified, the test-case generation algorithm traverses the labeled transition system and recursively makes a choice between:

1. terminating the test case
2. giving input to the implementation
3. checking the next output of the implementation

The method proposed in this paper has no facilities for modeling time or test time constraints. According to Tretmans, an open issue with this approach is the atomicity of actions. Actions in this model are classified as inputs and outputs, which has a one to one correspondence to those of the implementation. Tretmans states that an interesting area for future research is to combine implementation relations, which is a relation that has to be fulfilled for the implementation to correspond with the specification (cf. [Kon95]), with action refinement. Action refinement is when one abstract symmetric action of the specification is implemented using multiple inputs and output with the implementation. Another open issue in this paper is the "well known" test selection problem, where a selection criterion is needed to select a subset of the possible test cases. Further, Tretmans states that the inclusion of data in the test-case generation procedure incurs a state explosion, which must be handled.

From an automation perspective, labeled transition systems are interesting since they are sufficiently formalized to allow automation. Further, there exist tools and languages in the domain of verification of communication protocols that may be reused.

5.2.2 Automation Frameworks

Peleska and Siegel [PM97] describe a framework for automating the testing process of potentially large reactive systems. They also aim to show how to switch between formal verification and testing. The method divides a specification with requirements into normal and acceptable behaviors, where the normal behavior specifies the typical behavior of the system (according to knowledge about the environment) and acceptable behavior specifies how the system behaves when worst-case scenarios occur.

Different testing approaches are suggested for the two cases and it is shown how a large specification can be decomposed into subparts that each can be modeled in a different way. The language used for modeling in the examples is Hoare's CSP [Hoa85]. However, the article points out an interesting aspect, namely that most software implementations cannot faithfully be represented with CSP. When the implementations can be modeled with

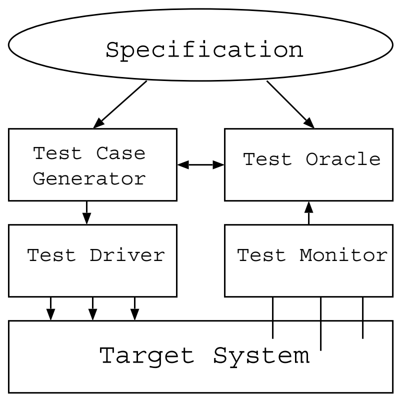

Figure 5.1: Logical components of test case execution

CSP, the authors suggest that model checking is used as a complement to test case execution. It is not explicitly stated how to test timeliness, but in the proposed model a monitor is used to capture execution sequences; we believe it should be possible to analyze these to detect violation of timing constraints.

Further, the article describes the logical components in automated test-case generation and execution and their relations. The logical components are test-case generator, test driver, test monitor and, test oracle (see figure 5.1).

The idea behind the test-case generation method is similar to methods based on labeled transition systems, combined with a method to derive minimal subsets of Hennessys's test classes [Hen88] in order to limit the huge amount of test cases. According to the authors, test-case execution in accordance with this method is capable of detecting safety failures, insufficient requirements coverage, divergence failures (cf. conformance testing), and insufficient robustness. Further, Peleska and Siegel present tools for the automatic generation and execution of test cases in accordance with this model.

5.2.3 Evolutionary Testing

As mentioned in section 4.2.5, test-case generation methods using genetic algorithms are easy to automate. The reason for this is that the only requirement for automatically deriving test cases with this class of methods is a 'good' representation of test data and a fitness function that aims to fulfill the coverage criterion.

Jones et al. [JES98] show that it is possible to use genetic algorithms to automatically generate test sets that cover all branches and supports fault-based testing. The hypothesis is that tests that fall close to a sub-domain boundary, (c.f. equivalence partitioning [Bei90]) are effective in revealing faults in the branch predicate.

The test sets generated are only data inputs; the strategy can be compared to random testing or partition testing. The software considered is sequential and the method is primarily applicable to unit and module testing for faults in the value domain. The genetic algorithms are used to find input data that force the execution along a specific branch of the code. For full branch coverage testing, sets of input data are evolved to cover a specific branch. This is repeated for each branch in the code. The fitness function for branch-coverage is attained by executing the system while monitoring the execution and give fitness values according to how much of the desired branch that was exercised. The article states that this easily could be automated.

Fault-based testing methods can be evaluated via *evolutionary fault injection* [JES98]. The quality of the test data relates to the probability that a specific test set reveals a set of randomly injected faults of the desired fault type (mutations).

The experiments show that ET methods are most applicable for fault-based testing. Hence, evolutionary fault injection can be used to evaluate the fitness function of such methods. The experiments also shows that testing using genetic algorithms outperforms random testing, and in some cases *equivalence partitioning* – a test technique for sequential software where the input space of a program is partitioned into classes which result in an equivalent behavior [Bei90]. One problem associated with this method is that it is time consuming to generate test sets, since the program has to be executed each time fitness should be calculated. The increase in computation time could maybe be justified by the fact that the human effort is minimized compared to other method for branch testing. In the case of fault-based testing the motivation is that the 'quality' of the test cases is so important that the computing overhead is justified.

One conclusion that is drawn in the article by Jones et al. is that ET can be used successfully to derive test data that will give full branch coverage in software. It can give up to two magnitudes fewer tests than simple random testing. It is also a good method for generating test data to probe regions of the input sub-domains where the number of errors are expected to be high. It is also noted that genetic algorithms can be used to test other software properties if these can be included the fitness function.

5.2.4 Z - Based methods

The formal language Z is one of the most commonly applied formal methods for software development today [HB95]. From high-level descriptions of system behavior and specification, the idea is to iteratively refine schemas with state specifications and operators on these states using logical expressions. These schemas can then be refined further into data structures and control flow design, and ultimately into software code using various code generators. However, the most common use is trough the specification and early design phases in order to capture all requirements and conceptual faults. Since these methods provide a formal specification, they can be used as a base for automatic test-case generation as well.

Helke et al. [HNS97] shows how to automatically generate test cases from the formal method language Z using the theorem prover Isabelle. The basic idea of this method is to convert state-specifications, written in Z, into a set of test cases by expanding combinations of state variables to disjunctive normal form. The test method has four steps. First, for each operation in state space, the pre/post relation is divided into sub-relations. Second, for each sub-relation, at least one representative test case is chosen to satisfy the test coverage criterion. In step three the test cases are executed on the testing target. Finally, the test results are verified against the specification. The test methods only test functional properties and only generate “logical” input data. Isabelle is a conventional theorem prover which is highly configurable and refined to operate on Z schemas. The theorem prover processes the operations in the Z schemas by expanding them to state variables; these can then be converted to disjunctive normal form (DNF).

The DNF entries that result in contradictions are eliminated to reduce the number of test cases. The paper does not claim to have invented the test generation method or the deduction methods, but their contribution is in showing the usefulness and feasibility of implementing verification techniques on top of a modern state-of-the-art theorem prover.

Singh et al. [SCS97] suggest a semi-automatic method for generating test cases from Z specification by combining the classification-tree method for partition testing with a disjunctive-normal-form approach. The method uses a formal specification written in Z to derive classification trees; these consist of high-level conditions and input states that are combined in a graphical representation, from which high-level test cases can be derived. According to the authors, the classification tree method are well suited for tool support that allows a user to design large amount of test cases without much effort. A problem with this approach is that the generated test cases are not detailed enough and some aspects may be hidden at first sight of the specification. Hence, the user may never be able to produce test cases that reveal unforeseen classes of faults. Further, this method contains no way of validating results of test cases. The DNF method is purely syntax based and hence, can be automated. The test cases generated with this method contain all information in the formal specification, as well as expected output values. But this method has its shortcomings as well because it produces inconsistent test cases, i.e., test cases that cannot occur, which must be detected or removed manually. Further, the test cases attained from logic conversion are completely unstructured and it is hard to get an overview of the test cases produced. Hence, the authors suggest that the methods can be combined to take advantage of the good properties and try to limit the disadvantages. One problem is that the quality of the test cases produced with this method depends on how the classification tree is built. The construction of such trees cannot be done purely automatically, thus, the user must make clever decisions during the construction. The method already has limited tool-support and the authors state that this will increase with future research.

In this context, the work of Stocks and Carrington must be mentioned [SC96]. Stocks and Carrington do not address automation issues explicitly, rather they present a framework for specification based testing in general, using Z as a test description language. However, this work falls somewhat outside the automation perspective, since their focus is on the framework for specifying tests rather than automatic generation of test cases to be run on the target system.

5.2.5 Methods based on Model-checking

Time constraints can be verified at different levels, one popular field where much research has been done is model checking. The purpose of model checking is to prove properties in a model of the system or specification, often in some executable language. For example, one such property could be to verify that time constraints in an executable model of a system satisfies the temporal specification. This can be done by generating and executing test cases on the model. Some model-checking approaches can also produce test cases that later can be used to verify that an actual software implementation behaves according to the model. This is necessary because even if the model is proven correct, since there is no guarantee that the actual software implementation behaves accordingly (cf. conformance testing).

Callahan et al. [CSE96] propose a method for “formal testing” that is a specification-based testing process that uses model checking techniques to verify, organize, and automatically generate test cases. Unlike many other approaches, this one takes a ”white-box” perspective and considers intermediate values generated from monitoring code inserted into the tested application program. The basic idea of the approach is to verify execution sequences produced by monitoring during testing, using a model checker to determine whether or not a trace corresponds to a path within the computation tree of a model. A computation tree is a conceptual structure that consist of a possible infinite set of all possible execution paths (cf. [TH99]). Most model checkers have some mechanisms for producing counterexamples if some paths in a computation tree exhibit a given property, e.g. test coverage. Coverage criteria in this approach are given by a method similar to equivalence partitioning. In the method of Callahan et al., the paths in the computation tree are partitioned to satisfy combinations of requirements written in a temporal logic language.

Considering this method from a timeliness perspective, one may suspect that it would be possible to incorporate testing of time constraints, since requirements are written in a temporal logic language. However, Callahan et al. do not consider timing requirements in their work and not all aspects of a software implementation are possible to capture using a formal model.

From an automation perspective this method is definitely interesting, since tools for automatic model-checking already exists [CSE96]. An interesting issue with this approach is that internal behaviors are considered. A potential problem with the method is the probe effect of monitoring statements in the code.

5.3 Analysis and Evaluation

As seen in this chapter, there exist many approaches for automating the test-case generation process. Most of these are based on formal methods, which includes formal specifications. The methods presented here are only a subset of all existing approaches for automatically generating test cases, yet several useful methods have been encountered for automating the test-case generation.

For our target model, the test framework presented by Peleska and Siegel is particularly interesting since it covers issues regarding test automation for reactive, safety-critical systems, which have much in common with our requirements. However, when testing timeliness in such a framework, we think that there is a possible probe effect associated with their method that needs to be handled. As mentioned in Schütz [Sch94], the probe effect can be handled by leaving monitoring code or components in the system. Mellin [Mel98] shows a possible solution to this by providing test execution logging facilities within an existing event monitor in a real-time database architecture. In the same way, it would be possible to avoid probe-effects caused by test case execution devices by introducing inputs to the system (e.g. from sensors) trough a combined "test driver" and "input handler" component which is part of the execution environment of the target system, see figure 5.2. It is also possible that an event monitor with rich semantics can provide the required functionality, since such a device would handle incoming external events as well as internal.

Another observation from the automation perspective is that there exist previous work developed within the formal methods community that can be used for automating the generation of test cases, e.g., theorem provers and model checkers. It would be interesting to investigate how existing tools and languages for modeling and specifying real-time and concurrent systems can be used for generating test cases automatically, e.g., UPAAL [BLL$^+$95] and

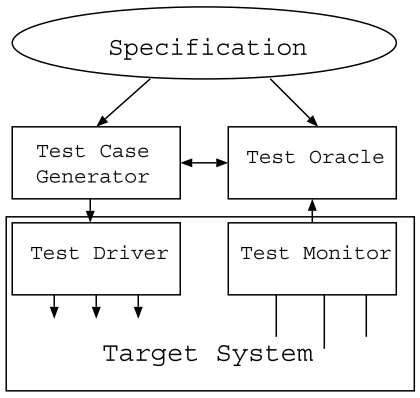

Figure 5.2: Logical components of timed test case execution

SDL .

Many of the methods that are included in the survey from the timeliness perspective also address automation issues, either as part of their method or as identified future work. This indicates that most authors are aware of the need for tool support and automation for test-case generation.

Chapter 6

Test Coverage Criteria for Real-Time

This section discusses the different test coverage criteria found during the surveys presented in chapter 4 and 5. Note that no explicit survey has been conducted from this perspective; hence, there may exist criteria that have not yet been evaluated. Since there exists many related concepts, each section includes the definitions used in the method.

6.1 Clarke's and Lee's Coverage Criteria for Real-Time

As mentioned in section 3.2.1, Clarke and Lee [CL97] presents a set of coverage criteria for testing timing constraints. Clarke and Lee conclude that a test coverage criterion is a rule for evaluating test suites.

"It gives programmers guidance in systematic methods to select test data that gives 'good' covering of the system to be tested".

They also define test-coverage criteria for the behavioral and performance constraints of real-time systems. As mentioned briefly in section 4.2.1, behavioral constraints are associated with the periods of time where certain inputs result in some specific behavior. For example, a signal to raise the landing gear is not accepted until ten seconds after flight takeoff. Performance constraints are associated with the time of output from the computer system; for example, the radar information must be processed before the next radar display update.

Clarke and Lee observes that a criterion that requires every delay in a dense time domain of possibilities to be exercised would be impossible to achieve, and therefore an approximation is necessary. Since Clarke and Lee opt for a domain-based specification of timing constraints, i.e., intervals of time is the domain of certain inputs and outputs, their approximation is to require test cases which exercise the endpoints of each domain. However, since it can be hard to test the exact endpoints of some domains (e.g. in domains with open endpoints), the criterion is relaxed to require inputs occurring within a time period before and after each domain endpoint. This coverage criterion is called *all-input-bounds-approximate.*

A simpler coverage criterion for behavioral constraints requires that each input action must be issued at least once, at any single point in the input time domain. A second criterion, used to complement the first in order to test performance aspects, is that each possible output in the domain should be produced and verified so that it is within the correct time domain. Third, a complement is needed to test that all timeouts in the specifications is implemented correctly; this is fulfilled if the test suite includes test cases so that each timeout in the specification is exercised at least once. Together, these three criteria are referred to as the *all-constraints criterion* .

An interesting note is that All-input-bounds-approximate subsumes the all-constraints criterion when testing behavioral constraints. However, this is not the case when testing performance constraints.

In our opinion, Clarke and Lee's test coverage criteria are sufficient for testing behavioral constraints, but the criterion for performance constraints is somewhat weak, because the same output may occur at different times depending on the internal execution order in the system. A more appropriate criterion in our opinion would be something like "all output-bounds-approximate", exercising the bounds of performance constraints, e.g. outputs occurring near the deadline. However, Clarke and Lee do not consider this since the time of output cannot directly be controlled by input parameters. Clearly, this is not the case for most real-time systems; both the time and order of incoming event influence the temporal behavior, as well as event parameters and states of shared (data) resources.

6.2 Morasca's and Pezze's Adequacy Criteria

According to Morasca and Pezze [MP90] there are two major problems in testing: when to stop, i.e., devising a rule whose fulfillment can ensure a reasonable confidence in the program correctness; and how to select test cases. From this perspective, an adequacy criterion is a predicate that shows if the program has been exercised "enough". They also mention that an adequacy criterion sometimes is called a test-selection criterion and gives a way of choosing test cases.

As mentioned in section 4.2.4, Morascas and Pezzes method is based on a Petri-net specification that, according to the authors, can be seen both as an operational executable specification language and as an abstract machine for the produced code. For this model, adequacy criteria are given that is applicable for concurrent and real-time systems. The test cases that are selected with this method are a sequence of input actions combined with an execution sequence of multiple concurrent threads. The reason that execution sequences are needed is the non-deterministic behavior of concurrent threads.

The adequacy criteria are divided into coverage of transitions and firings. The most exhaustive test criterion is the firing sequence test criterion where all possible firing sequences (i.e. all execution scenarios of the model) should be part of the test suite. Below this criterion in the test-criterion hierarchy is the "firings testing". The firings test-criterion states that instead of testing each possible sequence of firings, it suffice that each firing that is reachable from the set of initial states is exercised in some test case. Basically, the difference is that two firings in different order requires two test cases in the former criterion but it suffice with one in the latter. The firing-sequence criterion subsumes the firing-testing criterion. As indicated above, the firings testing criterion dramatically decreases the number of required test cases. However, the numbers of test cases needed may still be infinite, because the number of firings reachable from the initial states is potentially infinite.

The next step in reducing the number of test cases is to consider transitions instead of firings. When considering transitions, two firings of the same transition are considered the same entity, from a coverage point of view. The adequacy criteria for transitions then, again, consists of the choice between all combinations of transitions sequences (*transitions-sequence criterion*), or testing of all transitions (*transition-testing criterion*) in the same way as for the firings criterion. Due to the presence of iterations in ER-nets, the transition-sequence criterion can result in unbounded number of sequences that needs to be executed. However, the number of test cases resulting from the transition-testing criterion is bounded and hence, this is the first practically applicable adequacy criterion. Unfortunately, this coverage criterion fails to detect many types of failures, that are common in concurrent and real-time systems, namely failures that occurs when the order of writing to shared resources have impact on the final result. Morasca and Pezze conclude that a more sophisticated adequacy criterion is needed.

The (obvious) solution is to stick with the transition sequence adequacy criterion and handle the occurrence of loops so that the number of needed execution sequences is bounded. The authors do this by introducing the *N-times transition testing* adequacy criterion, which limits the number of times a transition is fired to *N*. Thus, a loop only iterates *N-1*times. A further optimization to this is proposed, where sequences that result in equivalent behavior, independent on the order of transition firings, needs only one test case (cf. Isolation in section 2.3.3).

This criterion is interesting since it considers adequacy criteria for multiple concurrent threads. Even if the criteria considered here relate to Petri-net models, many of the problems with concurrency and test selection is similar to our target model.

6.3 Criteria for Specification-Based Tests

Offut et al. [OXL99] present a set of criteria for generating test cases from state-based specifications. State based specifications in this context are similar to labeled transition systems as described in section 5.2.1. The test criteria are not aimed for testing timed systems, but are included here as a comparison with criteria for automatic generation of non-timed specification based tests.

The article defines a test case as a collection of inputs that cause one execution of a software program; this includes a set of prefix values that is all inputs necessary to reach pre-state and to give the triggering events their before-values.

A testing criterion is defined as a rule, or collection of rules, that imposes requirements on a set of test cases. Test coverage is the percent of requirements that are satisfied.

The article presents four different levels of software testing criteria. At the first level; every transition in a state-graph specification should be traversed at least once (*transition coverage*). At the second level, every clause of a predicate associated each transition the state-graph specification should be exercised independently (*full predicate coverage*). Informally, a clause is an undividable element of a logic predicate. At the third level, every combination of transitions over a state should be tested (*transition-pair coverage*). To fulfill the last testing criterion (*complete sequence*), it is required to test series of 'realistic' state-transitions. The authors judge that this is the hardest level to automate and that the expertise of a test engineer is required to device test cases. The article gives a case study of using some of these testing criteria to test a simple car control system, showing that the full predicate coverage criterion, outperforms both random testing and the transition coverage criterion.

It is interesting to see the definition of a test case in this work, since it clearly incorporates some notion of state from which the test runs should start. But a disappointing aspect is that the effect of concurrency and parallelism on the target system is not covered.

Chapter 7

Results

In this chapter, the results of this work are summarized. First the survey and analysis of test-case generation methods from the timeliness perspective is presented. This includes the classification of methods and the analysis of applicability on the target model. Then, results and observations of the survey from the automation perspective are summarized. Adopted terminology is suggested that aims to clarify multiple definitions and inconsistencies that have been found during the survey. Also, possible future research and open issues are presented where appropriate in this chapter and summarized in the following chapter.

7.1 Survey of Test Case Generation Methods for Testing Timeliness

This work have presented a survey of all existing methods aimed at generating test cases for testing temporal properties of real-time computer systems. A classification of methods relating to how timing constraints are specified, the structure of resulting test cases, and the nature of the assumed time base has been performed (see Table 7.1). This classification complements the already existing classification for test-case generation methods used in the dependability community [Lap94].

7.1.1 Classification from a Timeliness Perspective

Our survey focuses on methods that are aimed for testing of time constraints in real-time systems. These methods have been classified as described in section 4.1. The result of the classification is shown in Table 7.1. An observation that can be drawn from this classification is that the type of specification has influence on the contents of produced test cases. For example, temporal logic and Petri-net specifications generally generates test cases that consist of event sequences whereas test cases generated from some type of labeled transition system or process algebra often are specified as test processes.

The time base used in these methods seems unrelated to the specification language or test case-type. Another observation is that a majority of the methods assume a discrete time base. This assumption simplifies testing

Authors [Reference]	Specification type	Test Case Contents	Time Base
Laurencot and Castanet [LC97]	ETIOSM	Test Process	Continuous
Petitjean and Fochal [PF99]	Timed Input Output Automata	Test Process	Continuous
Kone [OK95]	ETIOSM	Test Process	Continuous
Cardell-Oliver and Glover [COG98]	Timed Transition Systems	Test Process	Discrete
Braberman et al. [BFM97]	Petri - net Specifications	Event Sequence	Continuous
Morasca and Pezze [MP90]	Petri - net Specifications	Event Sequence	Continuous
Clarke and Lee [CL97]	Process Algebra	Test Process	Discrete
Cleveland and Zwarico [CZ91]	Process Algebra	Test Process	Discrete
Mandrioli et al. [MMM95]	Temporal logic	Event Sequence	Discrete
SanPietro et al. [SMM00]	Temporal logic	Event Sequence	Discrete
Wegener et al.[WSJE97]	Informal	Test Data	Discrete
Müeller and Wegener [MW98]	Informal	Test Data	Discrete

Table 7.1: Classification of Methods from a Timeliness Perspective

issues and it may be a valid assumption, since all currently used computer systems operate with a discrete time base at the lowest level, i.e., at the level of hardware clock cycles. A note to this classification is that in the case of the methods that use genetic algorithms for deriving input data [WSJE97, MW98], a discrete time base have been chosen. This is in general not relevant to these methods, since they rely on some other mechanism to verify time constraints and are only concerned with forcing a task to specific temporal behaviors.

In retrospect, another interesting classification would be to look at the fault types each method can handle or to classify methods according to their relation to behavioral and performance time constraints as in [CL97]. However, this is left as future work.

7.1.2 Results from Evaluation and Analysis

The methods in our survey have been analyzed according to the timeliness considerations of our target system model (see section 2.3.3). This section presents the resulting observations and conclusions.

In our test-case-execution model, internal states of the system during execution are considered. One observation is that only approaches that are based on finite-state machines or Petri-nets fulfill the requirement of representing internal states. However, problems exist with finite-state-machine solutions found in the survey, since they seem to require that the internal states correspond to states of the specification. Further, non-determinism is hard to handle with these representations. Petri-net solutions, however, seems better equipped to handle the problems with concurrency and internal states, but methods based on either of these specification languages have problems with non-determinism.

Another issue is that many methods for testing time constraints abstract away from is providing input data; the input to a system is considered as events at specific points in time. This may be a problem since we believe that the state of shared resources and event parameters have an impact on the timeliness of real-time systems; this view is shared by methods based on genetic algorithms. On the other hand, many input values can be considered equivalent from a timeliness point of view, so methods for supplying event parameters and state of shared resources may be combined with methods for supplying event sequences for conformance testing. Further, we believe that some sort of conformance testing is necessary for testing timeliness, since otherwise it would be impossible to know when a time constraint has been violated [Nil99]. This problem will be discussed further in the context of future work.

In the target model, constrained observation of the environment is assumed to reduce the number of possible input scenarios. During the survey, no methods consider constrained observation explicitly and the only constraint on system behavior in this context is that some of the methods assume a sparse time base. We believe that constrained observation has influence on test-selection criteria and test-case generation methods. However, it would be a useful future research area to investigate the effects of including constrained observation in the target model of existing methods, e.g. the method presented by Mandrioli et al. [MMM95].

Further, we have found only one work, which explicitly cover distribution aspects in relation to generating test cases for real-time systems. We believe that this is a problem that needs further investigation.

7.2 Survey of Automation Issues in Test Case Generation

This work has investigated methods for automatically generating test cases in areas similar to real-time systems. A classification has been conducted of the methods that have been part of the survey, both those from the timeliness perspective that address automation, and the methods that are interesting solely from the automation perspective. The classification is performed according to the degree of automation that the method currently offers, and the test case design paradigm used. Further, some observations and possible future research areas has been identified.

7.2.1 Classification from an Automation Perspective

Table 7.2 shows the classification made from the automation perspective. In this table, the methods that consider both timeliness and some degree of automation have been included.

From the table, it seems as if functional methods are more common than structural methods for automatic generation of test cases. We doubt that this is a correct conclusion; the impression comes from the fact that the number of methods aimed at testing of timeliness is over-represented in this survey. Another possible reason is that this survey mainly focuses on test-case generation for system level testing and in this specific sub-field functional methods seem to be more common.

Authors [Reference]	**Deg. of Automation**	**Timed**	**Design Paradigm**
Callahan et al. [CES96]	Automated	no	Functional
Pelska and Siegel [PS96]	Automated	no	Functional
Tretmans [Tre97]	Automated	no	Functional
Jones et al. [JES98]	Automated	no	Structural
Cardell-Oliver and Glover [COG98]	Automated	yes	Functional
Clarke and Lee [CL97]	Automated	yes	Functional
Müeller and Wegener [MW98]	Automated	yes	Structural
Wegener et al.[WSJE97]	Automated	yes	Structural
Morasca and Pezze [MP90]	Potentially Automated	yes	Functional
Petitjean and Fochal [PF99]	Potentially Automated	yes	Functional
Helke et al. [HNS97]	Semi-Automated	no	Functional
Singh et al . [SCS97]	Semi-Automated	no	Functional
Laurencot and Castanet [LC97]	Semi-Automated	yes	Functional
Mandrioli et al. [MMM95]	Semi-Automated	yes	Functional
SanPietro et al. [SMM00]	Semi-Automated	yes	Functional
Braberman et al. [BFM97]	Semi-Automated	yes	Structural

Table 7.2: Classification of Methods from an Automation Perspective

Another observation is that there exist four methods that are 'fully automated' and are aimed at testing of timeliness. Two of these are based on genetic algorithms and can only be used as a complement to other methods. The other methods are based on process algebra and finite state functional specifications.

Further, there exist six semi-automated and potentially automated methods, which consider time. A possible future research project is to benchmark these methods against each other to find out the quality of the test suites and the associated test effort.

7.2.2 Results from Evaluation and Analysis

This work has described some existing approaches for automating the test case generation process. An automation framework for safety critical systems has been analyzed from an automation and timeliness perspective. The observation from the analysis is that the conceptual model of test related components may be refined to reflect the need to avoid a probe effect, which is a requirement for testing time constraints.

Other existing methods based on labeled transition systems and the formal language Z have been described. It is noted that much benefit can be drawn from using a notation from the formal methods community, since tools often exist, which can help automating test-case generation.

The benefit of evolutionary test methods over other methods for generating test data from an automation perspective has been highlighted.

7.3 Analysis of Test Coverage Criteria for Testing of Real-Time Systems

This work analyzes three different sets of test-coverage criteria from a real-time perspective.

The first set of coverage criteria is the one presented by Clarke and Lee [CL97]. The coverage criteria presented in their work are directly aimed at testing time constraints in real-time systems. Even if the method is promising, we find that it lacks certain properties. For example is there no mechanism for influencing and testing performance constraints in the tested system. In order to test timeliness we believe it is necessary to be able to test performance constraints as well as behavioral constraints. This issue will be further discussed in the context of future work.

The second set of coverage criteria, presented by Morasca and Pezze [MP90], is interesting since they take internal states and concurrency into consideration in their test coverage criterion. Timing constraints is not explicitly addressed in their coverage criterion; instead, the test criterion is associated with coverage of a certain type of transition in a Petri-net model. We conclude that this model has many properties in common with our target system and that it is interesting to further evaluate how the criterion apply in our model. Hence, this is a potential future work.

The third set of coverage criteria is presented by Offutt et al. [OXL99] and comes from specification-based testing; the method is included as an example of other applicable test coverage criteria at the system level. The coverage criterion is based on a state-specification where degrees of transition coverage should be achieved. From our perspective, it is interesting that test cases specify from which state testing starts, which is similar to our test execution model even if the nature of the states is different.

7.4 Adopted Terminology

This section presents terminology for which multiple definitions have been encountered in the literature. Each subsection begins with presenting the definition that we have adopted and the relevant arguments and counter-arguments for the definition in a dialectic style.

7.4.1 Testability

Testability is a concept that has many definitions. The definition that is adopted is based on Schütz [Sch93] definition. Schütz uses three testability criteria, namely test coverage, observability, and controllability (see section 2.2 to define testability. Test coverage is defined as the number of anticipated real-world scenarios that have been, or can be, covered by generated test scenarios. Note that, given that the time and resources for testing some application is limited, this also incorporate test effort, since a higher test effort will result in lower test coverage.

Some authors only use observability and controllability as part of the testability concept. In our opinion, this is inappropriate since test coverage, as defined by Schütz is very important. Even if it is possible to observe and control everything that occurs inside a system, it is impossible to increase confidence in a system under test unless you can fulfill a meaningful coverage criterion in finite time.

7.4.2 White-Box and Black-Box Testing

The definition that we adopt in this work is from the community of safety critical software development, where black-box and white-box testing is used to denote the degree of observability during testing. In this definition, *black-box testing* is performed by supplying input data to the system without observing the internal state of the system during execution, whereas *white-box testing* take internal states into consideration [PM97, CSE96].

However, in the software testing community, black-box and white-box testing is considered equivalent to structural and functional testing as defined in section 2.1, namely that white-box testing is testing where the implemented code, design and architecture are used for deriving tests whereas black-box testing is when only specifications are used to derive tests, e.g. [Mey79, Bei90]

The reason for our choice of the former is that the latter is equivalent to the definition of functional and structural testing and in our opinion this is redundant. However, our main motivation is that it is more useful for us to be able to denote the consideration of internal states with one concept.

7.4.3 Test Case Selection Criterion

As already mentioned in chapter 6 there exist multiple definitions of a test selection-criterion. Many related or similar concepts exists, e.g. adequacy criterion, coverage criterion.

In this work *test coverage criterion* denote the level of ambition for a test method, i.e., test coverage criterion sets a goal of when an application has been tested sufficiently. A *test case selection criterion*is related to this – a criterion for selecting tests that eventually will fulfill the test coverage criterion. Adequacy criterion is equivalent to test-coverage criterion in this work. However, many definitions have been found during the survey.

For example, according to Braberman et al. [BFM97] a selection criterion is a way to select test cases from an often infinite number of possible test cases whereas an adequacy criterion is used for determining when testing has reached an adequate level.

Clarke and Lee [CL97] define that a test coverage criterion is a rule for evaluating test suites which gives programmers guidance in systematic methods to select test data that gives 'good' covering of the system to be tested.

Offut et al. [OXL99] use a different terminology for this and states that a 'testing criterion' is a rule or collection of rules that impose requirements on a set of test cases. 'Test coverage' is the percent of requirements that are satisfied.

According to Morasca and Pezze [MP90] an adequacy criterion is a predicate that shows if the program has been exercised "enough". They also state that an adequacy criterion is equivalent to a test selection criterion.

Chapter 8

Conclusions

This chapter contains a discussion of the conclusions that can be drawn from the result of this work. Four high-level research directions that address the problems identified in this work are described. The contributions of this dissertation and related work in the area are presented.

8.1 Discussion

The area of testing of real-time system are slowly beginning to get more attention in academia, the reasons for this we believe is that real-time systems are becoming increasingly complex and the static analysis methods and formal verification methods have troubles keeping up with the flexibility, diversity of applications, immense parallelism, and distribution requirements of new real-time systems. Hence, people turn to testing in hope of getting a simpler way to verify overall correctness. Unfortunately, testing cannot provide this, the number of test cases needed to formally verify a medium-sized real-time system with these properties is to big to even consider it an option, and the methods required would be anything but simple. However, if we take testing in consideration when designing systems by avoiding mechanisms that result in explosions in test complexity, we believe that it is possible to construct test methods and test criteria that give confidence in correctness while not loosing the properties mentioned above.

Test case generation is a very broad term, and there is significant difference of what is meant by a test case at different levels and in different communities. In common for all test case generation schemes is the problem of selecting what test cases to actually run. The approaches taken for this is strongly connected to what property that is decided to be ensured, and what type of system that are tested.

A general comment is that many works that aim to address these problems tend to abstract away from the actual test case execution on software systems, test cases are generated from models, expressed by models and is sometimes executed on models. Very few works considers the constraints in the actual execution environment, the impact of operating systems and hardware mechanisms. It is possible that by observing the actual test case executions, we can find new coverage criteria and exploit properties that we usually abstract away from. By this we do not claim that the models are unnecessary, but we think that in order to get a sound test method, all available knowledge and constraints must be exploited.

8.2 Future Research Directions

This section summarizes the, in our opinion, most important of the identified problems and proposes future research directions in this area, which aim to solve these problems.

8.2.1 Combination of Specification based methods and Genetic algorithms

This work have identified that the existing methods for test case generation produce event sequences or input data, but there exist very few that can supply both. One idea for a future work is to combine conformance-testing techniques, where sequences of events and associated deadlines are generated from a formal specification, with a method for generating test data, such as genetic algorithms, static analysis or equivalence partitioning. The advantage with genetic algorithms is that these are easy to automate, but there may also exist other algorithmic approaches for enforcing temporal behaviors of software.

A possible advantage with this approach is that it is possible to specify a coverage criterion in the specification-based method and use, e.g. genetic algorithms to search for input data that cause the selected internal behavior to occur.

Under the resource model specified in section 2.3.3, it may also be possible to consider locked shared resources as input parameters, if testing is performed form a specific state. However, if resources depend on each other, it may result in inconsistent overall states.

8.2.2 More Appropriate Coverage Criteria for Real-Time

This work summarizes some of the coverage criterion found during the survey for addressing real-time systems. However, we find that there is a gap in coverage criteria that take constraints in the execution environment and internal states of the system into consideration.

In our opinion, a coverage criterion for real-time must take timing constraints into consideration, both for testing that input events are handled in a correct way and that the output of a system always is within a specified interval. To control the second issue, testing must be conducted on two levels. On a higher level, the tested event scenario must cause the worst-case (or best-case) internal behavior in blocking, synchronization, and arrival time of events. On a lower level the input parameters and shared resources must cause the worst-case (or best-case) execution time of transactions.

Also, no work encountered in the survey use constraints in the execution environment for limiting the number of test cases. We believe that it is possible to eliminate many redundant test cases by taking such constraints into consideration.

Hence, a future work is to further refine these ideas into more formal coverage criteria for real-time systems. Preferably, this should result in a hierarchy of coverage criteria attaining increasing degrees of test coverage. Such criteria must be evaluated against other approaches on real-life application or formally proven to give a high test-coverage.

8.2.3 Tool Support and Visualization Aid

From the automation perspective the survey has indicated that it is important to have tools for automating the test case generation process. Further, if a semi-automated solution is used, then it is important to support visualization of the specification and their temporal implications to help testers to device test suits. The survey has encountered formal languages that handle time and concurrency. If some of these are used as a base, then there may be tools and theory that can be reused in this context. Another advantage with using an existing notation is that people easier can learn and relate to the notation if it has been used before.

Further, we believe that it be valuable to be able to visualize the test case execution process. This is interesting both for debugging purposes and for giving confidence in the test method and coverage criterion. Constraints in the execution environment and assumptions about the environment should ultimately be visualized and enforced in such tool.

One problem with this vision is that the tools most probably would have to be specialized for execution environments with certain properties and restricted to support a specific modeling language, which may restrict the usability of such tools.

8.2.4 Practical Issues with Test Case Execution

There are several practical issues relating to test case generation and execution that have not been handled in the literature. For example, there are requirements on the execution environment to support the test case execution, e.g. [Sch93, PM97, Mel98]. Especially in the test case execution model presented in section 2.3.3 where state enforcement is assumed there exist many practical issues that needs to be solved. An investigation whether it is a realistic test case execution model and if there exist easy solutions to the problems associated with it would be valuable.

Another issue that may need consideration is how to handle caches and pipelines and how such mechanisms influence test case execution. In the worst case it may be realistic to turn of such mechanism, or assume maximum number of page faults during testing, but it is inconclusive whether these worst case scenarios really occurs and how much penalty in wasted resources it is to have these extreme safety margins. If caches are left on during execution, it may need special test cases and test coverage criterion.

Another issue that is related to this from reproducibility perspective is support to 'replay' test cases, even if it is possible to restart the test case from the same state, there is additional requirements for obtaining a equivalent execution [Sch94].

8.3 Contributions

This work has identified four important perspectives, relevant to test case execution in real-time systems. The timeliness perspective, which is concerned with problems associated with testing that deadlines are held and no timing constraints are violated. The automation perspective that is associated with approaches and facilities for automating test case generation. The test coverage criteria perspective that aims to find appropriate test coverage criteria for real time systems and the reproducibility perspective which is concerned with facilities for making test case execution reproducible.

The main contribution of this dissertation is a survey of methods that address the verification of time constraints in real-time systems. A classification of these kinds of methods has been proposed, relating to their specification formalism, their resulting test cases, and their assumptions about time. We believe that new methods, which address testing of time constraints in real-time systems, can be classified into these categories. We also hope that our survey can serve as an introductory source of knowledge for people interested in this area.

This work also gives examples of how test case generation can be automated. The benefits of automation, and problems encountered when achieving automation are highlighted. The described methods have been classified according by their supported degree of automation and their adopted test case design paradigm.

Ambiguous concepts and terminology of this area have been identified and the best definition, in our opinion has been adopted. The result of this is a set of definition that we hope to be commonly accepted. Even if these definitions cannot be commonly accepted we have gained knowledge of different existing viewpoints, and can hopefully prevent misconceptions in the future.

Further, this work has revealed problems with existing methods for generating test cases for a real-time system model that adopts the event-triggered design paradigm. The identified problems relates to testing of timeliness, automation, and test case selection for this model.

The identified problems are summarized into four high-level research directions, which aim to remedy one or more of the identified problems.

8.4 Related Work

This section presents other works that has similar aims as this, i.e., other surveys in the testing domain and related areas are presented.

8.4.1 Roper's Selected Annotated Bibliography of Software Testing

Marc Roper presents a annotated bibliography with 96 references from the area of software testing [Rop92]. The purpose of this bibliography is to provide a list of "essential" references for readers new to the area of software testing. This survey contains references that are included in Ropers list, but since we aim for a more specialized domain, i.e., real-time systems and automatic test case generation, the works are quite different. Ropers survey is a very good first look of what has been done in the software testing area and almost all seminal papers from this time is included in his bibliography. Further, Roper focus on the contents of the papers whereas in our work, focus is towards the methods.

8.4.2 Carringtons and Stocks' Tale of Two Paradigms

Carrington and Stocks [CS94] presents the viewpoint of two different fields, the formal methods community and the advocates for traditional software testing. Their paper aim to highlight the advantages of the two different paradigms and the benefits that can be attained if they are combined. They also include a list of players in the field of integration of software testing into formal methods or vice versa. The difference between their work and this is that their focus is in the formal specifications and mainly sequential software testing, whereas our work primarily cover papers which address problems related to concurrency. The common factor of the two works is that we both agree that a formal specification is beneficiary for automation of test case generation.

8.4.3 Glass' Lost World of Software Debugging and Testing

In 1980, R. L. Glass wrote an article where he denoted real-time the lost world of software debugging and testing to reflect that testing of real-time systems is much less mature than testing of other software [Gla80]. The article examine state of the art in the area and note that there is a lack of tools and methodologies for testing real-time systems, testers are often left to examination of machine language code. Execution on the target system is most primitive part of the lost word according to Glass, there exist very few facilities for testing at the target system. This dissertation has also examined the joint area of real-time systems and testing to reveal open issues and unsolved problems. However, Glass' work focused on testing and debugging of real-time systems in general, our focus has been on test case generation and execution on such systems. Furthermore, much has happened in the real-time and software testing communities in the last twenty years.

Acknowledgements

First I want to thank my supervisor Sten F. Andler for tips, valuable discussions and encouragement throughout this project.

I also want to thank my mentor Jonas Mellin, and the other members of the 'test project' group; Birgitta Lindström and Mats Grindal for reading my work and for valuable critique and discussions.

Other people that have made life easier during the writing of this dissertation and deserve to be mentioned are; Sanny Gustavsson, Johan Carlsson, Hanna Andersdotter, Simon Wetterlind, Erik Hedström, and George Lucas.

Bibliography

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[Bei90] B. Beizer. *Software Testing Techniques.* Von Nostrand Reinhold, 1990.

[BFM97] V. Braberman, M. Felder, and M. Marré. Testing timing behavior of real-time software. International Software Quality Week, 1997.

[Bir98] R. Birgisson. Improving testability of applications in acitve real-time database environments. Master's thesis, University of Skövde, 1998. HS-IDA-MD-98-001.

[BLL$^+$95] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[BMA99] R. Birgisson, J. Mellin, and S. F. Andler. Bounds on test effort for event-triggered real-time systems. Department of Computer Science, University of Sk¨vde, 1999.

[CL97] D. Clarke and I. Lee. Automatic generation of tests for timing constraints from requirements. In *Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems*, Newport Beach, California, February 1997.

[COG98] R. Cardell-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. *Lecture Notes in Computer Science*, 1486:251–, 1998.

[CS94] D. A. Carrington and P. A. Stocks. A tale of two paradigms : Formal methods and software testing. In *Proceedings of the 8th Z User Meeting*. Springer-Verlag, 1994.

[CSE96] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. DRAFT NASA Software Research Laboratory West, 1996.

[CZ91] R. Cleveland and A. E. Zwarico. A theory of testing for real-time. *IEEE Computer Society Press*, pages 110–119, 1991.

[FBK$^+$91] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on software engineering*, 1991.

[FHLS98] P. G. Frankl, R. G. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 24(8):586–601, August 1998.

[Gai86] J. Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, Mar 1986.

[GG75] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, Se 1(2):156–173, 1975.

[Gla80] R. L. Glass. Real-time: The "lost world" of software debugging and testing. 1980.

[GR93] J. Gray and A. Reuter. *Transaction Processing, Concepts and Techniques.* Morgan Kaufmann, 1993.

[HB95] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods.* Prentice Hall, 1995.

[Hen88] M. C. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[HNS97] S. Helke, T. Neustupny, and T. Santen. Automating test case generation from z specifications with isabelle. Technical report, GDM first International Conference of Z users, 1997.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs NJ, 1985.

[ISO89] ISO. *LOTOS - International Standard IS-8807*, 1989.

[JES98] B.F. Jones, D.E. Eyres, and H.-H. Sthamer. A strategy for using generic algorithms to automate branch and fault-based testing. 1998.

[Kon95] O. Kone. Designing tests for time dependent systems. Seoul, South Korea, 1995. IFIP International conference on Computer Communications.

[Kop91] H. Kopetz. An engineering approach to hard real-time system design. pages 166–188, 1991.

[Lap94] J. Laprie, editor. *Dependability : Basic Concepts and Terminology*. Springer-Verlag for IFIP WG 10.4, August 1994.

[LC97] P. Laurencot and R Castanet. Integration of time in canonical testers for real-time systems. California, 1997. 1997 Workshop on Object-Oriented Real-time dependable Systems, IEEE Computer Society Press.

[Lin99] B. Lindström. Reducering av testfall vid testning av punktlighet i realtidsystem. Technical Report HS-IDA-EA-99-115, Department of Comupterscience, University of Skövde, 1999.

[Mel98] J. Mellin. Supporting system level testing of applications by active real-time databases. ARTDB97, Springer-Verlag, 1998.

[Mey79] G. J. Meyers. *The Art of Software Testing.* John Wiley and Sons, 1979.

[Mit96] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[MMM95] D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems*, 4(13):365–398, Nov 1995.

[MP90] S. Morasca and M. Pezze. Using high level petri-nets for testing concurrent and real-time system. *Real-Time Systems : Theory and Applications*, pages 119–131, 1990. Amsterdam North-Holland.

[MW98] F. Müeller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. 1998.

[Nil99] R. Nilsson. Autmated test case execution for real-time systems that are constrained for improved testability. Technical Report HS-IDA-EA-99-118, Department of Computersience, University of Skövde, 1999.

[OXL99] J. A. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. Las Vegas, 1999. ICECCS99.

[PF99a] E. Petitjean and H. Fochal. A realistic architecture for timed testing. Las Vegas, 1999. ICECCS99.

[PF99b] S. M. Petters and G. Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. Hong Kong, 1999. RTCSA99.

[PM97] J. Peleska and M.Siegel. Test automation of safety critical reactive systems. *South African Computer Journal*, 16:53–73, 1997.

[Rop92] M. Roper. Software testing: A selected annotated bibliography. 1992.

[SC96] P. Stocks and DA. Carrington. A framework for specification based testing. *IEEE transactions on SOFTWARE ENGENEERING*, 22(11):777–793, nov 1996.

[Sch93] W. Schütz. *The Testability of Distributed Real-Time Systems.* Kluwer Academic Publishers, 1993.

[Sch94] W. Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, September 1994.

[SCS97] H. Singh, M. Conrad, and S. Sadeghipour. Test case design based on Z and the classification-tree method. pages 81–90, Hiroshima, Japan,, November 1997. First International Conference on Formal Engineering Methods, IEEE Computer Society.

[SMM00] P. SanPietro, A. Morzenti, and S. Morasca. Generation of execution sequences for modular time critical systems. *IEEE Transactions on Software Engineering*, 26(2), feb 2000.

[TH99] H. Thane and H. Hansson. Towards deterministic testing of distributed real-time systems. Swedish National Real-Time Conference SNART'99, August 1999.

[Tre97] J. Tretmans. Test generation with inputs outputs and quiescene. In T.Margaria and B. Steffan, editors, *Tools and Algorithms for the construction and analysis of systems*, volume 1055 of *Lecture Notes in Computer Sience*. University of Twente, Springer-Verlag, 1997.

[WSJE97] J. Wegener, H. H StHammer, B.F Jones, and D.E Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.

[YP97] C. D. Yang and L. L. Pollock. The challenges in automated testing of multithreaded programs. 1997.