Testing of Computer Software with Temporal Constraints

A State-of-The-Art Report

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

المنارة للاستشارات
www.manaraa.com

Sammanfattning

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

المنارة للاستشارات

www.manaraa.com

Abstract

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

المنارة للاستشارات

www.manaraa.com

Contents

1 Introduction **7**
1.1 Outline . . . 8
1.2 Terminology . . . 8

2 Computer Software Testing **11**
2.1 Planning for Testing . . . 12
2.1.1 Test Plan . . . 12
2.1.2 Fault Hypothesis . . . 13
2.1.3 Test Cases . . . 14
2.1.4 Initial Test Case Selection . . . 14
2.1.5 Test Case Selection for Re-testing . . . 14
2.2 Analysis of Computer Software Execution Behavior . . . 15
2.3 Execution Behavior . . . 17
2.3.1 Synchronization . . . 18
2.3.2 Observability . . . 18
2.3.3 Determinism . . . 19
2.3.4 Controllability . . . 20
2.3.5 Reproducibility . . . 20
2.3.6 Testability . . . 21
2.4 Testing of Sequential Programs . . . 22
2.4.1 Unit Testing . . . 23
2.4.2 Integration Testing . . . 23
2.4.3 System Testing . . . 23

3 Regression Testing **25**
3.1 General Regression Test Assumptions . . . 27
3.1.1 Cost Models . . . 28

المنارة للاستشارات
www.manaraa.com

3.2 Regression Test Techniques 28
3.3 Algorithms for Regression Test Selection 30
3.3.1 *Slicing Algorithm* 30
3.3.2 *Incremental Regression Testing* 31
3.3.3 *Firewall concept for Regression Testing* 32

4 Testing of Concurrent Programs **35**
4.1 Methods 35

5 Testing of Real-Time Systems **37**
5.1 Distributed Real-Time System 38
5.2 Testing of Real-Time Systems 38
5.2.1 Testing for Functional Correctness 40
5.2.2 Testing for Temporal Correctness 41
5.2.3 Test Strategies 42
5.2.4 Test Bed Architectures 43
5.2.5 Environment Simulators 44
5.3 Regression Testing of Real-Time Systems 44

6 Summary **47**

المنارة للاستشارات
www.manaraa.com

المنارة للاستشارات
www.manaraa.com

Chapter 1

Introduction

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

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

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

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

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

المنارة للاستشارات
www.manaraa.com

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

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

In this state-of-the-art report we will discuss software testing and how different test methods can be applied on different types of software: sequential programs, concurrent programs and real-time systems. The focus will be on testing of real-time systems. But we will also discuss testing of non-real-time software to give an introduction to software testing in general.

1.1 Outline

The outline of the rest of this report is as follows: In Chapter 2 we discuss the fundamentals of testing. Testing of sequential programs is discussed in Chapter 2.4. In Chapter 3 we will discuss *regression testing*, i.e., how to test software after the code is modified or needs a retest. In Chapter 4 we will discuss testing of concurrent programs. In Chapter 5 testing of real-time software is discussed, focusing mainly on functional testing of real-time systems.

1.2 Terminology

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

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

المنارة للاستشارات
www.manaraa.com

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

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

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

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

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

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

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

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

المنارة للاستشارات
www.manaraa.com

المنارة للاستشارات
www.manaraa.com

Chapter 2

Computer Software Testing

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

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

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

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

In fault-based testing it may not be sufficient to select test data to meet

المنارة للاستشارات
www.manaraa.com

some coverage criterion, but also chose test data based on to what extent the test is expected to reveal a failure. Based on the approach to reveal failures, testing methods can be divided into: *fault seeding*, *mutation-based* and *fault injecting*.

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

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

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

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

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

2.1 Planning for Testing

2.1.1 Test Plan

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

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

- A strategy for regression testing.

للاستشارات
www.manaraa.com

- A guideline for the test procedure, including a test design strategy, coverage criteria and information on how to handle test cases that do not need to be re-executed.

- Information for identification of test classes, test case execution order, and changes made to the software.

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

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

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

2.1.2 Fault Hypothesis

The fault hypothesis is the definition of what a failing behavior is according to the specification of the software [3]. What a failing behavior is depends on the current failure mode of the system. In Clarke et al. [7] a classification of different failure modes of sequential programs are defined: *control failures*, *value failures*, *addressing failures*, *termination failures* and *input failures*.

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

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

المنارة للاستشارات
www.manaraa.com

2.1.3 Test Cases

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

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

2.1.4 Initial Test Case Selection

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

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

2.1.5 Test Case Selection for Re-testing

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

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

المنارة للاستشارات
www.manaraa.com

modified so the test case no longer test the program construct, or the test case no longer contribute to the structural coverage.

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

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

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

2.2 Analysis of Computer Software Execution Behavior

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

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

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

المنارة للاستشارات
www.manaraa.com

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

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

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

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

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

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

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

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

المنارة للاستشارات
www.manaraa.com

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

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

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

2.3 Execution Behavior

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

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

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

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

المنارة للاستشارات
www.manaraa.com

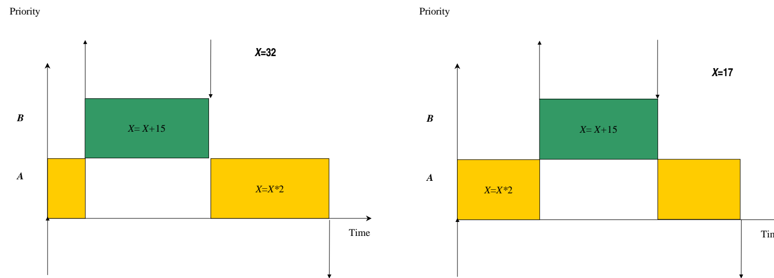

Figure 2.3 (a): Example 1 Figure 2.3 (b): Example 2

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

2.3.1 Synchronization

Multi-tasking programs may have requirements that restrict the order of interleaving between programs; such requirements may be due to data dependencies between programs. Without these constraints *race situations* can occur.

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

2.3.2 Observability

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

Observing the input and output in sequential programs is straightforward,

المنارة للاستشارات
www.manaraa.com

that is if the program does not include any non-deterministic statements [33].

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

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

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

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

Yann-Hang et al. [20] propose a tool suite for testing real-time ADA applications. The tool suite includes an instrumentation tool implemented as an ADA run-time library. Output generated from the analysis and the instrumentation are flow graphs and trace files that are used to determine the code coverage criteria of the ADA-program. The analysis tool can handle different kinds of coverage criteria, e.g., basic blocks coverage, c-use coverage, and p-use coverage. However, in their paper the proposed test analysis do not consider the temporal behavior of the application.

2.3.3 Determinism

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

المنارة للاستشارات
www.manaraa.com

of such statements are random statements or dependencies of a clock readings in sequential programs [32].

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

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

2.3.4 Controllability

Controllability is the ability to force the program into a desired state. For sequential programs it is sufficient to give the input to the program and set a break-point at the desired program statement to achieve controllability. For multi-tasking programs controllability is achieved at a coarser scale than for sequential programs. Here synchronization sequences are derived by statically analyzing the concurrent program and then forcing the program to traverse the same trajectory as the derived synchronization sequence.

2.3.5 Reproducibility

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

After errors have been corrected the tester wants to assure that the error have been removed and that no new errors have been introduced. Therefore it is necessary to test the system repeatedly. During repeated test runs with the same test cases, the same outputs must be observed in order to determine if the software is correct [26]. If test executions are not reproducible re-testing cannot

المنارة للاستشارات
www.manaraa.com

determine that corrections have removed the errors. For concurrent programs and real-time systems, in which *races* have impact on the execution path, the program is not usually reproducible.

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

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

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

The implementation based approach requires an event history. The behavior of a program is logged during run-time in a history log. The information in the history log is then used to reproduce the behavior of the execution. In Thane [36] an implementation based method for creating a history log and reproducing the behavior of real-time systems by deterministic replay is introduced.

2.3.6 Testability

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

المنارة للاستشارات
www.manaraa.com

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

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

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

2.4 Testing of Sequential Programs

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

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

المنارة للاستشارات
www.manaraa.com

2.4.1 Unit Testing

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

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

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

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

2.4.2 Integration Testing

Integration testing is the phase when the units are integrated with each other and tested. Approaches for integration testing are *incremental*, *top down*, *bottom up* or the *big-bang* approach. Incremental integration testing is to stepwise integrate the program unit for unit. Top down integration testing is to integrate the program by starting with the main unit and then integrate the units as they are called from the units above in the hierarchy. Bottom up approach is the opposite to top down approach, the ingration is started from the units that is in the lowest level of the call hierarchy. In both the top down and the bottom up approach it can be neccessary to use stubs, dummy units, for those units, which are not yet subject for testing. The big bang apprach for integration testing means that all units functionality are implemented and then all units are intgrated at the same time.

2.4.3 System Testing

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

Despite that there exists several phases the different types of testing are not isolated activities; testing is an iterative process. For example, system testing

المنارة للاستشارات
www.manaraa.com

can be done several times in a project because we have subsystems that will be put together into the final system, and during maintenance faults are corrected and new functionality are added or removed.

المنارة للاستشارات
www.manaraa.com

Chapter 3

Regression Testing

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

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

One issue is that although the re-test all strategy is costly and time consuming, it is not always desirable to find a subset of test cases. Especially for those companies that must use retest-all method because of certain constraints such as safety-critical programs, etc [27]. Examples of other issues can be the use of tools for automation when regression testing are used extensively and frequently. A drawback of regression testing is that the suite of test cases increases when the software is maintained and this makes testing even more time consuming.

المنارة للاستشارات
www.manaraa.com

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

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

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

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

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

Leung et al. [22] divides the regression testing problem into two subproblems: the test selection problem (1) and the test plan update problem (2). In selection of test cases to test a modified program, select test cases from an existing test plan, and create new test cases based on made modifications. For (2), see Section 2.1.3, where the update problem is included.

Most of the existing regression test techniques are using structural based test cases (white box testing). This because most of them compare the structure

المنارة للاستشارات www.manaraa.com

of the old program with the new program, and then the tester tries to get the same coverage percentage as the previous test.

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

3.1 General Regression Test Assumptions

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

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

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

The second and third assumptions are more difficult to overcome. Here are cost/benefit trade-offs important and should be considered. If the program is too simple then the set of test cases may be small and the cost for selecting test cases is more than re-execute all the tests.

Leung [21] have examined the fundamentals of selective regression testing when divided into two strategies: *selective regression unit testing* and *selective regression function testing*. Where function testing addresses integration and

المنارة للاستشارات
www.manaraa.com

system test. Unit testing refers to strategies where structural coverage criteria must be met. For unit testing Leung discuss *proper scope assumptions*. The approach is to choose a scope such as that all faults in a modified program are revealed. This can however lead to unnecessary testing if the scope is large. They claim the problem is to design a selective regression unit testing strategy to minimize the scope and code to re-test.

3.1.1 Cost Models

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

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

3.2 Regression Test Techniques

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

- *retest-all* techniques
- *random/ad-hook* techniques
- *minimization* techniques
- *safe* techniques
- *data-flow/coverage* techniques
- *prioritization* techniques

المنارة للاستشارات
www.manaraa.com

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

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

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

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

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

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

المنارة للاستشارات
www.manaraa.com

3.3 Algorithms for Regression Test Selection

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

3.3.1 *Slicing Algorithm*

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

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

- *Directly affected*
- *Indirectly affected*

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

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

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

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

المنارة للاستشارات
www.manaraa.com

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

3.3.2 *Incremental Regression Testing*

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

(1) to fix faults.

(2) the specification is changed.

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

The authors propose three methods for incremental regression testing:

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

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

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

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

المنارة للاستشارات
www.manaraa.com

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

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

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

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

3.3.3 *Firewall concept for Regression Testing*

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

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

للاستشارات المنارة
www.manaraa.com

the number of modules inside the firewall is not increased.

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

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

المنارة للاستشارات
www.manaraa.com

www.manaraa.com

Chapter 4

Testing of Concurrent Programs

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

4.1 Methods

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

During the program execution of the guaranteed synchronization sequence

www.manaraa.com

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

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

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

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

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

المنارة للاستشارات
www.manaraa.com

Chapter 5

Testing of Real-Time Systems

Real-time systems are software and hardware that in cooperation with their environment, and based on inputs from the environment, produce and deliver results within specified time intervals. The time intervals are determined by the temporal constraints derived from the temporal properties of the environment. Because of the temporal constraints on the interaction between the real-time system and its environment the date of the data (inputs and outputs) is important [33]. Below is a definition of a real-time system:

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

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

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

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

المنارة للاستشارات
www.manaraa.com

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

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

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

5.1 Distributed Real-Time System

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

5.2 Testing of Real-Time Systems

Hassan Gomaa [11] propose a software development approach for real-time systems that incorporates tools for automated testing of real-time systems. The development approach and the tools have been evaluated in a case-study in the development of a robot controller. The development approach is based on the software design method DARTS (Design Approach for Real-Time Systems) [10]. The design of a real-time system is to decompose the system into

المنارة للاستشارات
www.manaraa.com

task and defining the tasks' interfaces according to the requirements in the specification. Thus, it is important to formally review the design specifications and to verify that the task decomposition conforms to the specification. After a detailed design specification has been accomplished the functionality of the tasks are implemented.

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

The unit testing and the initial functional integration testing are exercised on the host computer. The reason for this is that there are often more and better tools on the host computer than for the target system. Also, it is much more efficient to test on the host computer than on the target platform because testing can be more easily automated on the host.

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

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

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

المنارة للاستشارات
www.manaraa.com

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

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

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

in which the three first phases are similar to the phases of test execution of sequential programs. The fourth step is the testing of correctness of the control of devices attached to the system, i.e. the environment which the real-time system are controlling. In practice, the test execution in each phase is often performed in two steps [33]. The first step consists of execution of the application while recording the behavior. Then in the second step, the recorded behavior is analyzed.

5.2.1 Testing for Functional Correctness

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

The test approach is divided into three iterative steps:

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

المنارة للاستشارات
www.manaraa.com

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

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

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

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

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

5.2.2 Testing for Temporal Correctness

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

المنارة للاستشارات
www.manaraa.com

ral constraints. The graphs created are *Timed Process Interaction Graph* and *Dedicated Timed Process Interaction Graph*.

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

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

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

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

5.2.3 Test Strategies

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

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

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

المنارة للاستشارات
www.manaraa.com

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

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

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

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

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

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

5.2.4 Test Bed Architectures

Kopetz et al. [17] propose a architecture for running distributed fault-tolerant real-time systems. The architecture is called Maintainable Real-Time Systems (MARS) architecture and supports statically scheduled hard real-time systems.

المنارة للاستشارات
www.manaraa.com

MARS consist of clusters that can be interconnected by an arbitrary network topology. Tasks that have functionality relation are allocated to the same cluster. There are no tools for automating the allocation of tasks to a cluster so the designer itself is responsible for the appropriateness of task allocations on clusters.

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

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

5.2.5 Environment Simulators

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

5.3 Regression Testing of Real-Time Systems

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

المنارة للاستشارات
www.manaraa.com

efficient way and to remove the error-prone manual handling. The framework is developed based on Onomas [27] regression testing process.

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

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

The framework consists of four different layers *network layer*, *support layer*, *task layer* and *interface layer*. In the network layer existing communication mechanisms provided by the operating system are used. The support layer have three responsibilities: connection for access of test database, transportation of files between the three components and remote control of method invocations. The task layer is a set of programs that performs tasks such as test case submission, test case selection and test case execution. For easy use of the framework for test engineers the interface layer provides visual interfaces.

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

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

للاستشارات المنارة
www.manaraa.com

www.manaraa.com

Chapter 6

Summary

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

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

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

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

- an analysis is performed in order to chose a subset of test cases that must be exercised,

- for each iterative step in the regression testing new test cases are added

للاستشارات المنارة

www.manaraa.com

that increases the number of test cases to run, and

- by not running test cases there is a potential risk of faults being present in the software.

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

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

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

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

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

المنارة للاستشارات
www.manaraa.com

Bibliography

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

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

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

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

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

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

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

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

المنارة للاستشارات

www.manaraa.com

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

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

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

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

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

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

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

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

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

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

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

للاستشارات المنارة
www.manaraa.com

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

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

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

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

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

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

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

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

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

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

المنارة للاستشارات

www.manaraa.com

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

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

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

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

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

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

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

[37] H. Thane and H. Hansson. Towards Systematic Testing of Distributed Real-Time Systems. In *Proceedings of The 20th IEEE Real-Time Systems Symposium*, pages 360–369, 1999.

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

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

المنارة للاستشارات
www.manaraa.com

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

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

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

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

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

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

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

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

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

المنارة للاستشارات

www.manaraa.com