Automatic test vector generation and coverage analysis in model-based software development

Master's thesis
performed in **Vehicular Systems**

by
Jonny Andersson

Reg nr: LiTH-ISY-EX-3765-2005

14th December 2005

Automatic test vector generation and coverage analysis in model-based software development

Master's thesis

performed in **Vehicular Systems**,
Dept. of Electrical Engineering
at **Linköpings universitet**

by **Jonny Andersson**

Reg nr: LiTH-ISY-EX-3765-2005

Supervisor: **Magnus Eriksson**
Scania
Anders Fröberg
Linköpings Universitet

Examiner: **Jan Åslund**
Linköpings Universitet

Linköping, 14th December 2005

Avdelning, Institution
Division, Department

Vehicular Systems,
Dept. of Electrical Engineering
581 83 Linköping

Datum
Date

14th December 2005

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ ______________

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ ______________

ISBN
—

ISRN
LITH-ISY-EX-3765-2005

Serietitel och serienummer
Title of series, numbering

ISSN
—

URL för elektronisk version

http://www.vehicular.isy.liu.se
http://www.ep.liu.se/exjobb/isy/2005/3765/

Titel Automatisk generering av testvektorer och täckningsanalys i modellbaserad programvaruutveckling

Title Automatic test vector generation and coverage analysis in model-based software development

Författare
Author

Jonny Andersson

Sammanfattning
Abstract

Thorough testing of software is necessary to assure the quality of a product before it is released. The testing process requires substantial resources in software development. Model-based software development provides new possibilities to automate parts of the testing process. By automating tests, valuable time can be saved. This thesis focuses on different ways to utilize models for automatic generation of test vectors and how test coverage analysis can be used to assure the quality of a test suite or to find ”dead code” in a model. Different test-automation techniques have been investigated and applied to a model of an adaptive cruise control system (ACC) used at Scania. Source code has been generated automatically from the model, model coverage and code coverage has therefore been compared. The work with this thesis resulted in a new method to create test vectors for models based on a combinatorial test technique.

Nyckelord
Keywords

Test vector generation, Test automation, Test coverage analysis, Model-based development

Abstract

Thorough testing of software is necessary to assure the quality of a product before it is released. The testing process requires substantial resources in software development. Model-based software development provides new possibilities to automate parts of the testing process. By automating tests, valuable time can be saved. This thesis focuses on different ways to utilize models for automatic generation of test vectors and how test coverage analysis can be used to assure the quality of a test suite or to find ”dead code” in a model. Different test-automation techniques have been investigated and applied to a model of an adaptive cruise control system (ACC) used at Scania. Source code has been generated automatically from the model, model coverage and code coverage has therefore been compared. The work with this thesis resulted in a new method to create test vectors for models based on a combinatorial test technique.

Keywords: Test vector generation, Test automation, Test coverage analysis, Model-based development

Preface

Thesis outline

Chapter 1	An introduction of the background and objectives with this thesis
Chapter 2	A brief introduction to testing and an introduction to test coverage
Chapter 3	A presentation of existing model-based testing tools
Chapter 4	A Description of the implementation of a combinatorial test
Chapter 5	A comparison between different methods to automate testing
Chapter 6	An analysis of automatically generated code and a comparison between code- and model coverage
Chapter 7	Contains a case study of what coverage analysis can lead to
Chapter 8	Results and conclusions

Acknowledgment

I would like to thank my supervisor at Scania, Magnus Eriksson for always inspiring me to work harder and for teaching me to appreciate British humor. I also thank the other members of the AiCC-group, Kristian Lindqvist and Håkan Andersson for always taking their time. Thanks also to my supervisor Anders Fröberg at the Division of Vehicular Systems at Linköping University. I thank Dan Duvarney at Reactive Systems inc. for his support and for arranging a trial version of Reactis® for me to use during my work with the thesis. Another thank you goes to Olavi Poutanen at Testwell for his efforts to help me use Testwell's CTC++ on automatically generated code and for the discussions we had about coverage criteria.

Contents

Abstract i

Preface and Acknowledgment iii

1 Introduction 1
- 1.1 Background 1
- 1.2 Objectives 2

2 Testing theory 3
- 2.1 Types of tests 3
  - 2.1.1 Unit testing 3
  - 2.1.2 Integration testing 3
  - 2.1.3 System testing 4
  - 2.1.4 Acceptance testing 4
- 2.2 Test techniques 4
  - 2.2.1 Static testing 4
  - 2.2.2 Dynamic testing 5
- 2.3 Test coverage 5
  - 2.3.1 Decisions and conditions 5
  - 2.3.2 Notation, boolean expressions 6
  - 2.3.3 Decision coverage 7
  - 2.3.4 Condition coverage 7
  - 2.3.5 Modified condition/decision coverage (MC/DC) 7
  - 2.3.6 Other coverage metrics 8
- 2.4 Coverage goals 9
- 2.5 Automating tests 10
- 2.6 Different approaches to automating testing 11
  - 2.6.1 Capture & replay 11
  - 2.6.2 Random test generation 11
  - 2.6.3 Combinatorial test generation 11
  - 2.6.4 Model-based test generation 11
  - 2.6.5 Limitations of automating testing 11

3 Test tools 13
- 3.1 Commercial model testing tools 13
  - 3.1.1 Simulink Verification and Validation (V&V) 13
  - 3.1.2 Reactis® 13
  - 3.1.3 Other model-based testing tools 14

3.2 Building own test tools . . . 14

4 Implementation of a combinatorial test **15**
4.1 Advanced Effective Test Generation (AETG) . . . 15
4.1.1 The AETG system . . . 15
4.1.2 The original AETG-algorithm . . . 15
4.1.3 Limitations . . . 16
4.2 Implementation . . . 16
4.2.1 The model under test . . . 16
4.2.2 Application of a modified AETG-algorithm . . . 16
4.3 Sequences . . . 20
4.3.1 Problems with using standard combinatorial testing . . . 20
4.3.2 Problems with different send period . . . 20
4.3.3 Solutions . . . 21
4.3.4 Using manual tests to complement the automatically generated tests . . . 23
4.3.5 Ways to improve model coverage . . . 23
4.4 Test design . . . 25
4.4.1 The test generation process . . . 25
4.4.2 Choice of test sequences . . . 25
4.4.3 Choice of data ranges . . . 25
4.4.4 Precision and input sample time . . . 26
4.4.5 Modification of the model . . . 27
4.4.6 User interface . . . 27
4.4.7 Overview . . . 28

5 Comparison between different test generating methods **29**
5.1 Manual testing . . . 29
5.2 Random testing . . . 30
5.3 Model based testing in Reactis . . . 31
5.4 Sequence based combinatorial testing (SBCT) . . . 32

6 Analyzing automatically generated code **35**
6.1 The relation between code coverage and model coverage
- a theoretical approach . . . 35
6.1.1 MC/DC on shortcircuited evaluation of logical expressions . . . 35
6.1.2 Extensions on MC/DC for coupled conditions . . . 35
6.1.3 Differences between model- and code coverage of a logical expression? . . . 37
6.1.4 Multicondition coverage . . . 38
6.1.5 Coverage criteria . . . 38
6.1.6 Code coverage tools . . . 38
6.2 Comparing model coverage and code coverage
- an experimental approach . . . 39
6.2.1 Method . . . 39
6.2.2 Problems with comparing code and model coverage . . . 39
6.2.3 Linear approximation . . . 41
6.2.4 Verifying normal distribution . . . 43
6.2.5 Verifying linear model . . . 43
6.2.6 Generalization . . . 43
6.2.7 Classification of uncovered code . . . 46

7 Case study: results of coverage analysis **47**
7.1 Finding bugs 47
7.2 Identifying dead code 47
7.2.1 Unreachable states 48
7.2.2 Guard for division with zero 48
7.2.3 Unreachable saturation limit 48
7.2.4 Condition redundancy 49
7.2.5 AND with constant 49
7.2.6 Substate never active at superstate exit 49

8 Results **51**
8.1 Conclusions 51
8.1.1 Testing approaches 51
8.1.2 Coverage analysis 52
8.1.3 Code coverage vs. Model coverage 52
8.2 Future work 52

References **53**

Chapter 1

Introduction

1.1 Background

A traditional cruise control is designed to keep the vehicle at the same speed. On heavy traffic roads a normal cruise control will be useless.

The adaptive cruise control maintains the distance to the vehicle ahead. It detects the presence of a preceding vehicle and measures the distance as well as relative speed using a forward-looking sensor such as a radar, and automatically adjusts the vehicle speed to keep a proper distance to the forward vehicle and to avoid a collision, see figure 1.1.

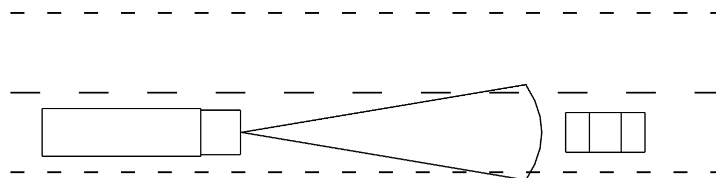


Figure 1.1: The adaptive cruise control uses a sensor to measure distance and relative speed to a preceding vehicle

At Scania, a model has been developed for an adaptive cruise control system in the model-based environments Simulink®/Stateflow®. The use of model-based software development offers some benefits:

- A focus on requirements lets engineers focus less on raw programming.
- Improved testability at an earlier stage in the development process.

Because of the improved testability, many software developers have started to employ automatically generated tests. With automatic test vector generation, it is possible to run more tests more often and earlier in the development process which probably results in improved quality of the software. This thesis will focus on how automatic test vector generation combined with test coverage analysis can be used to improve the quality of the software and save valuable time.

From the Simulink model, C-code can be generated using Mathworks' Real-Rime Workshop®(See figure 1.2). This code is then compiled and run on the target system.

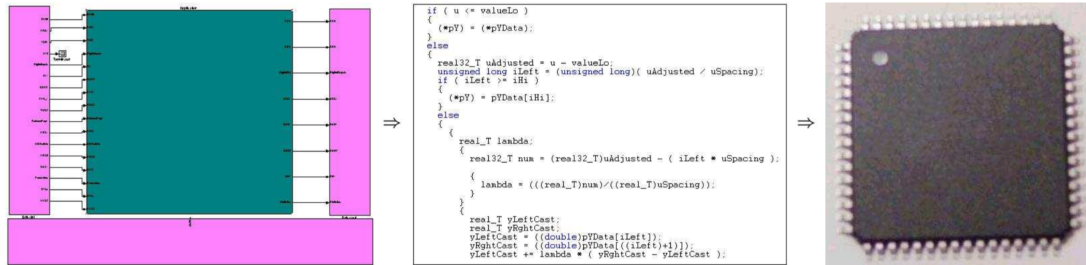


Figure 1.2: C-code is generated from the model and compiled to the target system

Since the automatically generated code is run on the target system, it is also interesting to see if the quality that is assured in the model by analyzing model coverage also guarantees good quality of the code.

1.2 Objectives

The goal of this thesis is to:

- Investigate different methods to automate testing of software implemented in Matlab®/Simulink®.
- Implement a method to generate test vectors and compare the results with existing test-vector generating tools.
- Examine model coverage and code coverage on automatically generated code from a model.

Chapter 2

Testing theory

Testing is a systematic way to check the correctness of a system by means of experimenting with it [13]. Tests are applied to a system in a controlled environment, and a verdict about the correctness of the system is given, based on the observation during the execution of the test [7]. Testing is an important and time-consuming part of a software developer's daily work. Thorough testing is necessary to be confident that the system works as it was intended to in its intended environment. When testing software, there are often a massive amount of possible test-cases even in quite simple systems. Running all the possible test-cases is almost never an option, so designing test-cases becomes an important part of the testing process.

2.1 Types of tests

Tests are normally performed in several different stages in the software development. The V-model in figure 2.1 on the next page illustrates that each implementation activity in the software development process has a corresponding testing activity. The acceptance tests are usually written first but run last and the unit tests are usually written last but run first. The V-model also illustrates the breaking down of the requirements into more detailed requirements and then the integration of the system into a ready product. The definitions of unit-, integration-, system- and acceptance testing in section 2.1.1-2.1.4 are those defined by Fewster & Graham [7]. These definitions are generally different depending on what type of product that is being developed.

2.1.1 Unit testing

A unit test is a test performed on a single component in the system, or in the case of software a single program or function. In the unit tests it is interesting to look at different coverage criteria to verify that most of the system has been covered by the test.

2.1.2 Integration testing

In the integration test, several units that are supposed to interact in the system are integrated and tested to verify the correctness of and to debug the system.

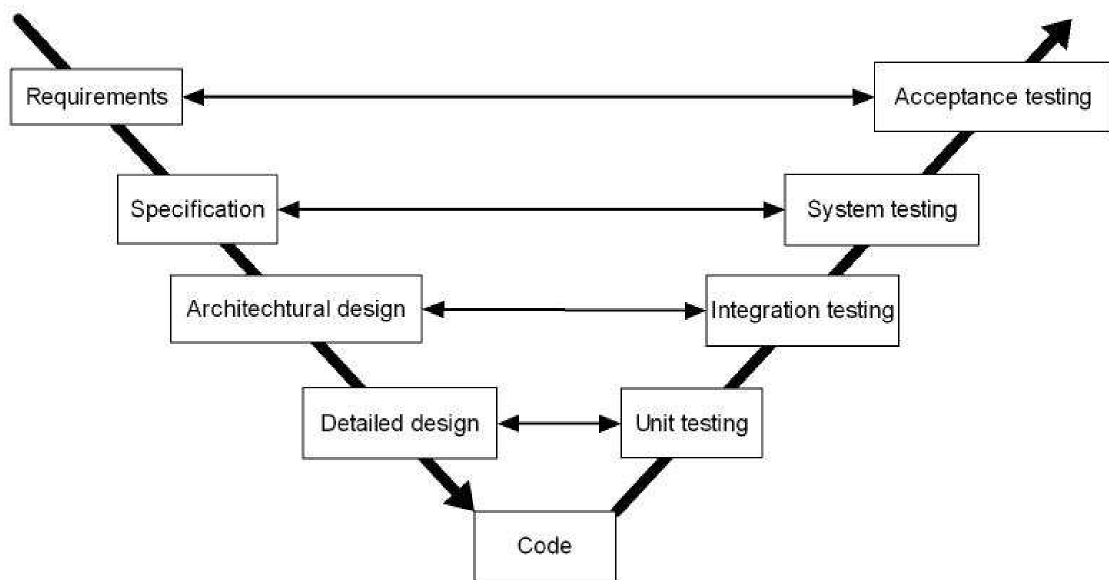


Figure 2.1: The V-model: Each stage in the software development process corresponds to a testing activity [7].

2.1.3 System testing

System testing is sometimes divided into functional system testing and non-functional system testing. In the functional system tests it is possible to test the whole system against the specification. The requirements can then be verified and the functionality of the system can be evaluated. Non-functional system testing is a way to test properties such as performance, economy, flexibility, etc.

2.1.4 Acceptance testing

Acceptance testing is the last level of the validation process. It should be performed in as realistic environment as possible. The acceptance test is often done by or with the customer. In this test the focus is on getting a confidence for the system, not to try and find defects.

2.2 Test techniques

There are mainly two types of test techniques; static and dynamic testing.

2.2.1 Static testing

Static software testing means analyzing programs without running them. The purpose of static testing is to find potential bugs or deficiencies in the program or to check the program against some coding standard, analogous to using a spelling checker. Static testing can be done manually by analyzing the code or it can be done automatically by a compiler or by a separate program. One of the tools used for static testing of C source code is *Lint* but static testing is usually done with the help of a compiler.

2.2.2 Dynamic testing

Dynamic software testing means running programs and analyzing the outputs. The dynamic testing can be divided into *black-box*, *white-box* and *grey-box* testing.

Black-box test design treats the system so that it does not explicitly use knowledge of the internal structure of the system. The tests could be based on requirements and specifications or on possible use of the system. Random testing is also a black-box technique.

White-box techniques are based on having full knowledge of the system. With this technique it is possible to test every branch and decision in the program. When the internal structure is known it is interesting to look at different coverage criteria such as *decision coverage*. The test is accurate only if the tester knows what the program is supposed to do. The tester can then see if the program diverges from its intended goal.

Grey-box techniques: In between the black-box and the white box-testing there are also many degrees of grey-box testing, e.g. when the module structure of a system is known but not each module specifically.

Manual testing can be classified as any of the above techniques depending on how the testing is done. For example structured manual testing would be classified as white-box testing while ”ad hoc testing” would be classified as black-box.

2.3 Test coverage

Test coverage is a way to determine the quality of a test. Historically, when test coverage was first applied it meant a significant change for many testers in their approaches to testing. With test coverage, it was possible to combine black-box testing with measuring coverage instead of designing manual white-box tests which could take very long time [2].

To measure coverage is to quantify in some way how much of a certain program that has been exercised during an execution [14]. It is also important to mention that coverage analysis is only used to assure the quality of a test, not the quality of the actual product [4].

Test coverage can be divided into *code coverage* and *model coverage*. Code coverage is based on code, for example C-code or Java and model coverage is based on graphical models such as Simulink-models or UML[1]- models.

In some cases, it may be impossible to cover certain parts of the system. For example, a program may contain code to detect and handle error conditions that are very hard to simulate in a test.

Decision coverage, condition coverage and modified condition/decision coverage (MC/DC) are some of the most common coverage criteria in modeling environments. All these metrics originate from their original definitions for code coverage.

2.3.1 Decisions and conditions

The definitions of decisions and conditions below are general in testing terminology but the formulations of the definitions are those by Chilenski and Miller 1994 [2].

[1]Unified Modeling Language

Decisions

A decision is treated as a single node in a program's structure, regardless of the complexity of boolean structures constituting the decision. Consider the program as a flow chart, a decision is then a point where the flow can be divided into two or more branches. See figure 2.2

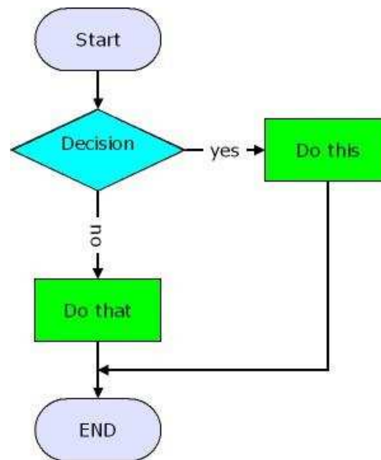


Figure 2.2: A decision is a point in the program's flow chart where the flow can be divided into one or several branches

Conditions

A condition is a boolean valued expression that cannot be broken down into simpler boolean expressions [2]. A decision is often composed of several boolean conditions. Consider the logical tree in figure 2.3, where *A,B and C* are conditions that together constitute a decision.

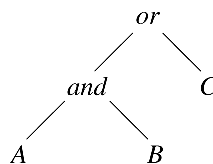


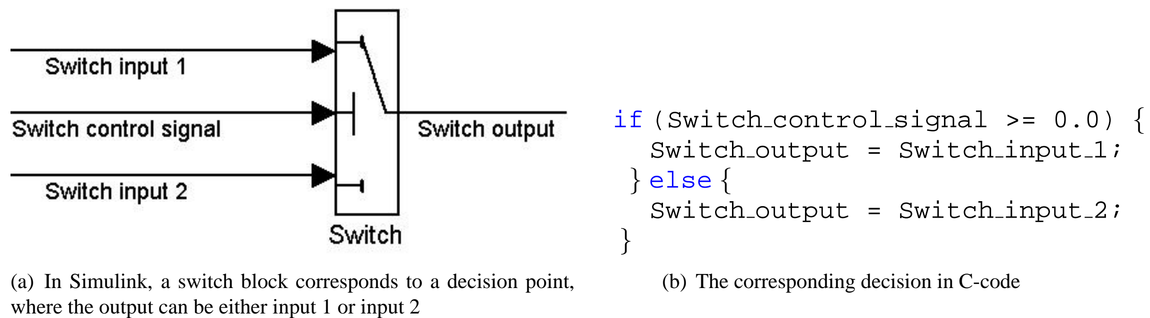
Figure 2.3: This logical tree represents a decision, the "leaves" of the tree are called conditions

2.3.2 Notation, boolean expressions

In this report, boolean variables will be used to represent the actual conditions. For example, the expression `IsValidTarget and (DistanceToTarget < 200)` can be written as *A and B* for simplicity. The *and*-operator will sometimes be written & and the *or*-operator will be sometimes be written |. Shortcircuited operators (explained in section 6.1.1) will be written && and ||. Boolean values will be written as *true* and *false* or `T` and `F`. The evaluation order of boolean expressions will always be from left to right.

2.3.3 Decision coverage

For each decision, decision coverage measures the percentage of the total number of paths traversed through the decision point in the test. If each possible path has been traversed in a decision point, it achieves full coverage. See figure 2.4.



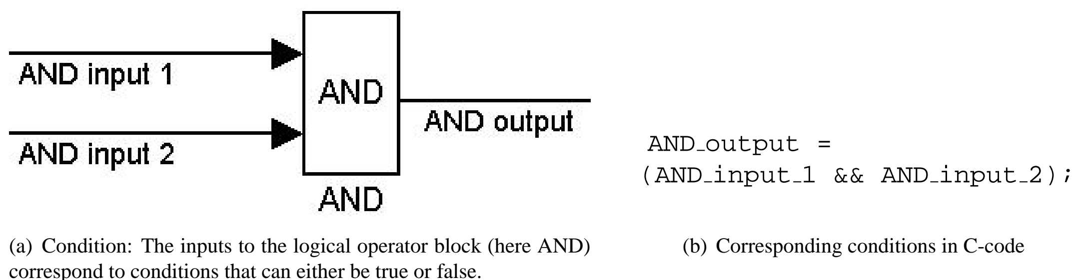
(a) In Simulink, a switch block corresponds to a decision point, where the output can be either input 1 or input 2

(b) The corresponding decision in C-code

Figure 2.4: Decision coverage

2.3.4 Condition coverage

Condition coverage in Simulink examines blocks that output the logical combination of their inputs (AND, OR, . . .) and state transitions. A single condition achieves full coverage if it has been evaluated both true and false at least once. See figure 2.5.



(a) Condition: The inputs to the logical operator block (here AND) correspond to conditions that can either be true or false.

(b) Corresponding conditions in C-code

Figure 2.5: Condition: The inputs to the logical operator block (here AND) correspond to conditions that can either be true or false.

2.3.5 Modified condition/decision coverage (MC/DC)

This measure requires enough test cases to verify that every condition alone can affect the result of its comprehending decision [2]. A condition is shown to independently affect the decision's outcome by varying just that condition while holding all other possible conditions fixed [2].

Example: For a condition to have full MC/DC, there has to be a pair of test cases where a change in the condition alone (all other conditions are fixed) changes the output. Consider the decision $d = a\&b$. The possible combinations of conditions are shown in table 2.1.

test case	ab	d	a	b
1	TT	T	3	2
2	TF	F		1
3	FT	F	1	
4	FF	F		

Table 2.1: All combinations of conditions for the decision $d = a\&b$. Test case 1 can be combined with test case 3 for condition a to satisfy MC/DC and test case 1 can be combined with test case 2 for b to satisfy MC/DC. Source [2]

In the two rightmost columns of table 2.1, test case combinations that make the condition satisfy MC/DC is presented. Condition a needs test case 1 and test case 3 to satisfy MC/DC and condition b needs test case 1 and test case 2 to satisfy MC/DC. Hence, the test cases 1, 2 and 3 are needed to achieve full MC/DC for the decision.

In Simulink, MC/DC can be measured on blocks that output the logical combination of their inputs or on state transitions in Stateflow that contains several conditions. The modified condition/decision coverage criterion was designed for programming languages that do not use shortcircuited evaluation of logical expressions. In section 6.1.1 a ”redefinition” of MC/DC is introduced for programming languages that use shortcircuited evaluation of expressions, examples of such programming languages are *Ada* and *C*.

MC/DC is a good compromise between decision/condition coverage and testing *all* combinations of condition outcomes, which is required by the *multicondition coverage* criterion explained in section 2.3.6. Instead of requiring 2^N test cases (where N is the number of conditions) which is required for multicondition coverage, a minimum of $N + 1$ test cases is required for MC/DC [2].

2.3.6 Other coverage metrics

Model coverage:

Lookup table coverage examines lookup tables in Simulink. A lookup table achieves full coverage if all interpolation intervals have been executed at least once.

Signal range coverage reports the ranges of each input or signal.

Code coverage:

Line coverage examines how many lines of code that has been executed.

Condition/decision coverage is a hybrid measure composed by the union of condition coverage and decision coverage. Condition/decision coverage requires all decisions and their comprehending conditions to have been both true and false [4].

Function coverage reports whether each function or procedure has been invoked.

Table coverage indicates whether each entry in a particular array has been referenced.

Relational operator coverage reports whether boundary situations occur with relational operators ($<$, $\leq$, $>$, $\geq$). Relational operator coverage reports whether the situation $a = b$ occurs. Consider ($a < b$). If $a = b$ occurs and the program behaves correctly, you can assume the expression is not supposed to be ($a \leq b$) [4].

Multicondition coverage: Instead of just requiring the whole decision to be true or false, the "multicondition coverage" criterion requires *all* combinations of conditions in a decision to be tested [8].

Statement coverage reports whether terminal statements such as *"BREAK", "RETURN", "GOTO"*, ... has been executed during the test.

2.4 Coverage goals

Having 100% coverage is of course desirable, but not always practically possible. Every test should have a coverage goal before release depending on testing resources and the importance to prevent post-release failures [9]. Safety critical systems such as aviation software or software in medical equipment should naturally have a high coverage goal. For example, the RTCA/DO-178B is a standard that provides guidelines for the production of airborne systems equipment software. In DO-178B, software is classified based on how safety critical the system is. The different classes and their requirements are shown in table 2.2.

Criticality level	Consequence	Coverage requirements
Level A	Catastrophic	MC/DC, Decision coverage and Statement coverage required
Level B	Hazardous / severe	Decision coverage and Statement coverage required
Level C	Major	Statement coverage required
Level D	Minor	None required
Level E	None	None required

Table 2.2: Coverage goals for aviation software using the DO-178B standard

Generally, when considering condition and decision coverage it is desirable to attain at least 80-90% coverage or more. When using the MC/DC criterion, it is often harder to attain higher levels coverage than when just using condition and decision coverage. This is understandable, because both the decision and all belonging conditions are covered if full MC/DC is achieved in a specific decision. This relation is easily derived directly from the definitions of MC/DC, condition coverage and decision coverage (see section 2.3).

Note: All decisions doesn't have two or more conditions, hence MC/DC is not measured in all decisions. This means that full MC/DC in a model does *NOT* imply full decision coverage for the same model.

In figure 2.6 on the next page the relative time/effort to achieve different levels of coverage with MC/DC, decision and condition coverage is shown. The time scale in figure 2.6 refers to the relative time to create tests with sufficient levels of coverage with the SBCT-method described in chapter 4. The graph is based on experience during this project only, the relation may very well be dependent on the structure of the system.

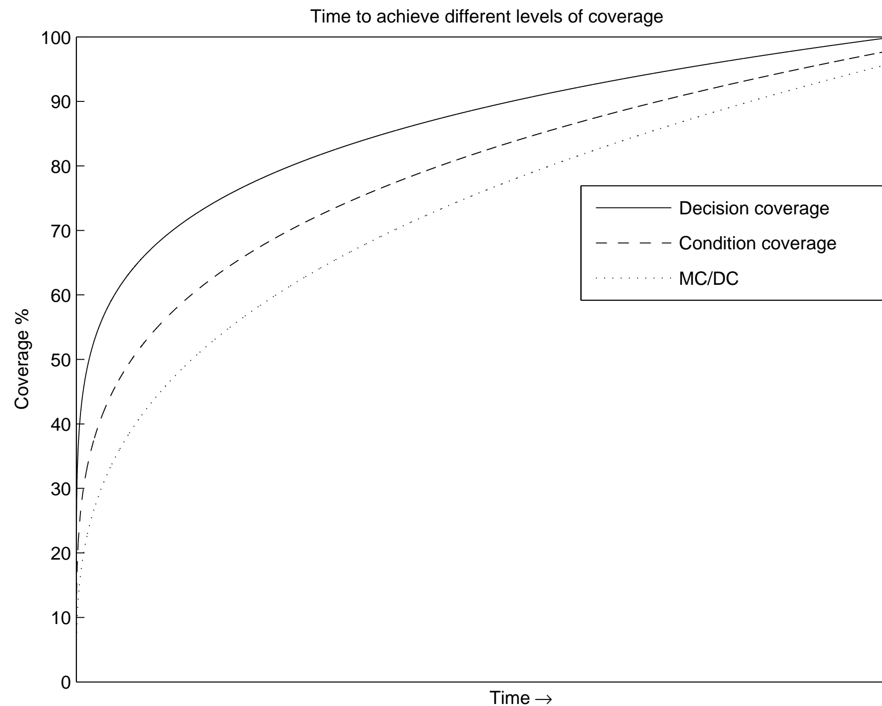


Figure 2.6: Time to achieve different levels of coverage

2.5 Automating tests

Test Automation can enable testing to be performed more efficiently than it can ever be done by testing manually. Some of the benefits of test automation are:

- Each time a system is updated, the new changes has to be tested in a regression test (a test that is run every time the system is updated to check that functionality that was not supposed to be changed works as it was intended). These tests are time-consuming and sometimes tiresome for the tester. If the tests can be automated, it should only take a few minutes to initiate the execution of the regression test.

- Ability to run more tests in less time and therefore it is possible to run tests more often. This will probably lead to better confidence in the system.

- Better use of resources. Many forms of manual tests are fairly menial, such as repeatedly entering the same test-inputs. If these tests can be automated, it gives better accuracy as well as more time for the testers design better test-cases.

- More thorough testing can be achieved with less effort than manual testing which probably gives a boost in both quality and productivity. It is also possible to run tests at an earlier stage of the software development process, thereby reducing the costs of removing defects.

2.6 Different approaches to automating testing

There are many approaches to automate testing. Some of these approaches that were considered particulary important for this thesis are presented in sections 2.6.1 to 2.6.4

2.6.1 Capture & replay

Capture & replay is simply a method to record the inputs to a system when a tester sits down to test, and replay the same test scenario later. This is the simplest form of test automation.

2.6.2 Random test generation

Random test generation is a black-box method. The basic idea is that the user specifies the data-types and data-range of the inputs and the test-program generates random inputs within the limits of the user's specification. Random tests must often be extremely large to achieve good coverage on complex systems.

2.6.3 Combinatorial test generation

Combinatorial test design is a way to keep the size of the test-vectors down to a reasonable level compared to random test-vectors. There are many systems where troublesome errors occur because of the interaction of a few inputs [3]. Another motivation is that the minimum number of tests required to cover all pairwise or n-way input combinations grows logarithmically in the number of inputs and linear in the number of possible values for each input , instead of exponential growth when testing all input combinations [3]. By combining the pairwise or n-way combinations it is possible to reduce the size of the tests significantly and still get better coverage than when using random testing.

2.6.4 Model-based test generation

The model-based approach is an advanced method to generate tests without manual effort. This method is employed by several modern test-tools. These test tools can convert models into a form where it is possible to generate test vectors [1]. The algorithms that are used to create test vectors from a model strive to increase model-coverage dynamically during the test generation process. These algorithms are test-tool specific and often quite sophisticated, hence they are not published officially.

2.6.5 Limitations of automating testing

There will always be some testing that is much easier to do manually than automatically, or it is not economic to automate. Manual testing requires all test cases to be written manually, which requires lots of time and effort but in some cases it is more economic than to automate a testing process. For example, tests that involve lots of physical interaction, such as turning power on and off or disconnecting some equipment will probably be easier done manually. Another example is that tests that are run very seldom would probably not be worth automating.

Chapter 3

Test tools

Test tools are programs that can assist the tester to generate test suites, create test harnesses, measure test coverage, validate outputs etc.

3.1 Commercial model testing tools

There are a few tools available for testing models made in Simulink/Stateflow on the commercial market.

3.1.1 Simulink Verification and Validation (V&V)

One of the products, Simulink Verification and Validation (V&V), is a toolbox for Matlab®/Simulink®. It is not capable of generating test-vectors directly from a model, but it is excellent to use together with manual scripts or home-made automated tests to measure the model coverage and verify requirements. The main features are that the model coverage information is displayed directly in the model and that requirements can be associated directly with the model. Another benefit is that the test environment is the same as the development environment.

3.1.2 Reactis®

Reactis is one of the leading tools for model-based test generation and validation of Simulink®/ Stateflow® models. Reactis automatically generates test suites directly from models. The tests are generated to achieve good model coverage which is measured dynamically during the test generation process. Reactis is a separate program, where Simulink models can be loaded and simulated with the automatically created test inputs. In Reactis it is possible to use *targets* and *assertions* for verification.

A *target* is defined by a separate Simulink model. Reactis will then try to cover the target as well as the model under test. Targets are customized test scenarios and are useful when it is interesting to check model behavior in certain circumstances. For example, a target can specify a variable that is held constant for a certain period of time. Reactis then tries to cover the target model, thereby testing the model under test at least once in the specified scenario.

Assertions are also defined by a separate Simulink model. Assertions specify properties that should always be true for a model and can be used for model verification [11].

The test-generation component of Reactis relies on model coverage criteria to construct tests. The tool tracks several coverage criteria as it computes test data, and it uses uncovered elements of these criteria to influence subsequent tests that it creates. Reactis uses information about uncovered parts of the model to select input data that will advance coverage of the model [11].

3.1.3 Other model-based testing tools

T-VEC® The T-VEC Tester uses advanced algorithms to analyze the model and select test cases that most effectively reveal errors. T-VEC also measures all the common coverage criteria. Just as Reactis the T-VEC tester can create tests directly from models which is useful for verifying that a system behaves correctly [12].

BEACON is a tool for generation of code from Simulink models and automatic generation of test vectors. The Automatic Unit Test Tool (AUTT)-part of BEACON creates test vectors. These test vectors target several coverage criteria and other common error sources such as numerical overflows. [6].

3.2 Building own test tools

When attempting to build an own test tool, there are some things that should be considered [7]:

+ The tool may be able to use knowledge of the system under test, thereby reducing the work necessary to implement automatic tests

+ It will be most suitable for the specific needs

– The user interface may leave something to be desired (the ease of use will probably be considered unimportant)

– It will probably be cheaper to achieve a given level of features and quality to buy a commercial test tool

Chapter 4

Implementation of a combinatorial test

An existing algorithm for generating tests based on a combinatorial technique was modified and applied to the ACC-model. This process is described in the following sections.

4.1 Advanced Effective Test Generation (AETG)

The algorithm that was used is called *Advanced Effective Test Generation* and is described in a paper by David M. Cohen [3].

4.1.1 The AETG system

In many applications, a significant number of faults are caused by parameter interactions that occur in atypical, yet realistic situations [3]. The AETG (Advanced Effective Test Generation) system uses an algorithm to cover all pairwise or n-way combinations of input values.

The idea behind AETG is that each input is treated as a *parameter*, with different *values* for each parameter. For example, a system has 20 parameters (inputs) with 3 possible values for each parameter. The design will ensure that every value of every parameter is tested at least once together with every other value of every other parameter, which ensures *pairwise coverage*. Pairwise coverage provides a significant reduction in the number of test cases when compared to testing *all combinations* [5]. If the example with 20 parameters with 3 values each would be tested with all combinations covered, 3^{20}; more than 3 billion test cases would be required. With combinatorial testing, the number of test cases were reduced to 44 with full pairwise coverage, using a *modified* AETG algorithm that was implemented during the work with this thesis, this modified algorithm is described in section 4.2.2. The AETG algorithm does not guarantee that the minimal amount of test cases required for pairwise coverage is used, but experience during this project has shown that the algorithm is quite effective.

4.1.2 The original AETG-algorithm

This is the original AETG algorithm published in a technical paper by David M. Cohen [3]:

Assume that we have a system with k test parameters and that the i:th parameter has l_i different values. Assume that we have already selected r test cases. We select the $r + 1$ by first generating M different candidate test cases and then choosing one that covers the most new pairs. Each candidate test case is selected by the following greedy algorithm:

1. *Choose a parameter f and a value l for f such that that parameter value appears in the greatest number of uncovered pairs.*

2. *Let $f_1 = f$. Then choose a random order for the remaining parameters. Then, we have an order for all k parameters $f_1, \ldots, f_k$.*

3. *Assume that values have been selected for parameters $f_1, \ldots, f_j$. For $1 \leq i \leq j$, let the selected value for f_i be called v_i. Then, choose a value v_{j+1} for f_{j+1} as follows. For each possible value v for f_j, find the number of new pairs in the set of pairs $\{f_{j+1} = v$ and $f_i = v_i$ for $1 \leq i \leq j\}$. Then, let v_{j+1} be one of the values that appeared in the greatest number of new pairs. Note that, in this step, each parameter value is considered only once for inclusion in a candidate test case. Also, that when choosing a value for parameter f_{j+1}, the possible values are compared with only the j values already chosen for parameters $f_1,, f_j$.*

4.1.3 Limitations

Applying the AETG system in its original version is only practical when testing systems with few values in each test parameter. When the number of possible values for each input grows large, the number of possible pairs grows as the square of the number of possible values for each input. This will cause an increase in the number of test-cases required to cover all pairs; Cohen et al. proves that the number of test cases grows linearly in number of possible values [3]. Altogether this will make AETG unsuitable to use on systems with inputs that have floating-point data types or other data types with many possible values. Systems with many test-parameters do not restrain effective test generation with the AETG system as long as there are not too many values in each parameter.

4.2 Implementation

With the help of a slightly modified AETG-algorithm, a new method was developed to create tests for the ACC-model. This method will be called *Sequence based combinatorial testing* (SBCT) in the rest of this report.

4.2.1 The model under test

The model that was used as a reference model is a Simulink / Stateflow model of an adaptive cruise control system (ACC). This model has 33 inputs and more than 30 outputs. Out of the 33 used inputs, 9 had floating-point values. The remaining inputs were integers or boolean.

4.2.2 Application of a modified AETG-algorithm

The AETG-algorithm was slightly modified before it was applied to generating tests for the ACC-model. Firstly, only pairwise coverage was considered. Secondly, the idea with candidate tests was discarded because it takes much less time to create just one test. Experience during this project showed that creating several candidate tests were not equivalent to shorter or better tests so instead of creating candidate tests, it was made possible to run several tests, covering each pair at least once. In this project the number of test cases was not a priority to keep down, just to keep the number of test cases on a controlled level. The feature of running several tests was implemented because there was a level of uncertainty that was caused by randomized values (explained in the following sections). Therefore, ten tests would most probably give better results than just one. A good result in this case means high model coverage.

An overview of the modified algorithm is shown in figure 4.1

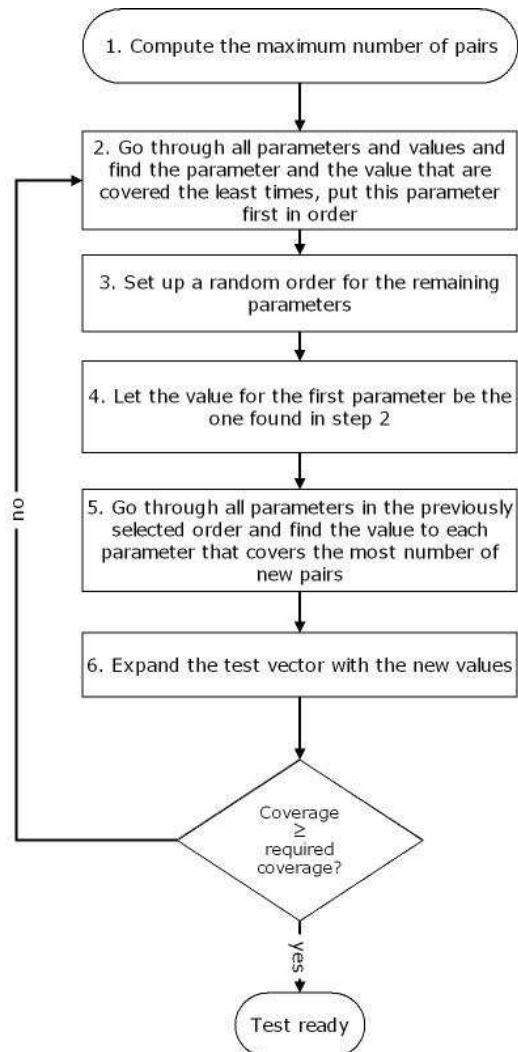


Figure 4.1: Overview of the modified AETG algorithm.

Floating-point inputs

It would have been impossible to test all the possible values in an input of a floating-point data type. Therefore, the inputs with floating-point values had to be replaced using one of the following solutions:

1. A fixed number of different constant values. For example, a floating-point value between 0 and 100 would be represented by the set $[0, 12, 19, 39, 52, 78, 99, 100]$. The value set has to be carefully selected with respect to the model properties to be able to function well.

2. A number of intervals can be selected and in each interval, the values can be randomized. For example, a floating-point value between -1 and 1 where the negative and positive values mean significant differences can be divided into two intervals. $[-1, 0]$ and $[0, 1]$. In each interval, the values can be random between $[-1, 0]$ and $[0, 1]$ respectively. See figure 4.2a on the next page. The indexes to each interval can be combined with the modified AETG-algorithm.

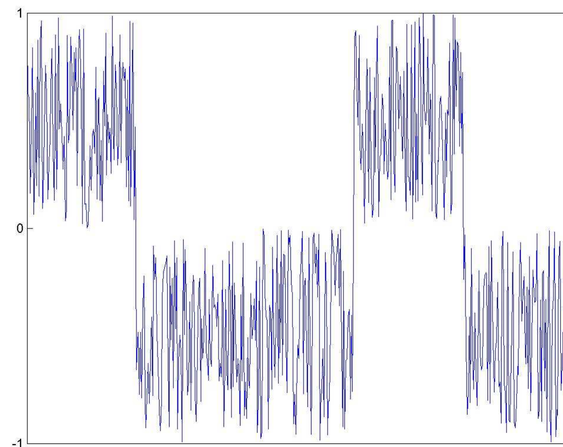
3. In some systems it is also important if the input is increasing or decreasing. It is then possible to randomize an initial value and increase or decrease the value with a small random number each sample. For example, consider a floating point value input between 0 and 100. Every 40 samples, randomize a new initial value and increase or decrease the value with a random value between 0 and 0.5 each sample.

4. Sometimes it is also interesting when an input crosses a fixed value. By combining increasing/decreasing inputs and a sort of ”logic” that makes sure that the input crosses the fixed value as many times as possible but still in a random way. See figure 4.2b on the facing page.

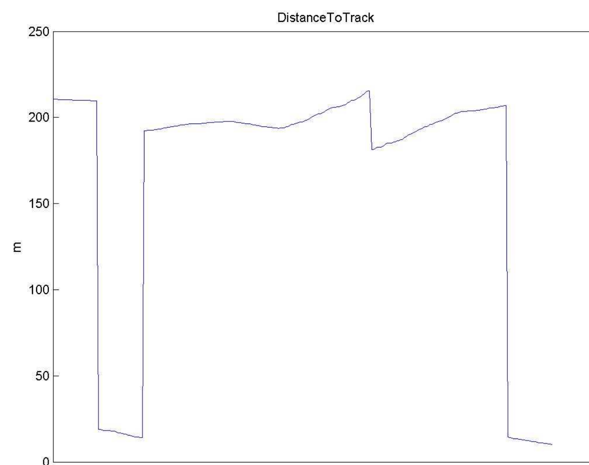
Some of the methods above can of course be modified to better fit the demands. For example, a modification of alternative 2 can have different probabilities to end up in the different intervals, or the intervals can overlap each other. Alternatives 2, 3 and 4 or modifications of them were finally used as solutions for the floating-point inputs in the test for the ACC-model. Each method had some parameters (e.g. increase/decrease) that were combined using the modified AETG-algorithm.

Reset signals

Another problem is signals that reset the system or a part of the system. If these signals reset the system too often, there will not be much space for thorough testing because the system will reset all the time. The solution to that problem will be to let the signals that reset the system be in the passive mode (not resetting) most of the time, easier said than done perhaps, but the problem will be better explained in the next section concerning sequences.



(a) Random between intervals



(b) Crossing a fixed value (here 200m)

Figure 4.2: Floating-point handling

Computing the maximum number of pairs in a system

All pairwise combinations of a set of test-parameters can easily be calculated with some matrix operations. Consider a system with n parameters with $v_i, i = 1, ..., n$ values in each parameter i.

Let

$$V = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \Rightarrow VV^T = \begin{pmatrix} v_1 v_1 & \cdots & v_n v_1 \\ \vdots & \ddots & \vdots \\ v_1 v_n & \dots & v_n v_n \end{pmatrix} \tag{4.1}$$

The maximum number of pairs is then the sum of all elements in the lower triangular part of the matrix

VV^T.

For example, consider a system with four parameters, with $v_1 = 1, v_2 = 2, v_3 = 3, v_4 = 4$. Hence

$$VV^T = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \\ 4 & 8 & 12 & 16 \end{pmatrix} \tag{4.2}$$

The lower triangular part (all the elements below the main diagonal) of VV^T is then:

$$LTP(VV^T) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 3 & 6 & 0 & 0 \\ 4 & 8 & 12 & 0 \end{pmatrix} \tag{4.3}$$

The maximum number of pairs is then the sum of all elements in $LTP(VV^T)$: $2+3+4+6+8+12 = 35$ pairs.

4.3 Sequences

To solve some problems of directly applying combinatorial tests on a complex model, a new method was developed to be able to design tests that are better suited to achieve good model coverage on a model with several state-machines and complex logical expressions. Generally, complex state-machines can be very hard to cover independently of what test method that is being used.

4.3.1 Problems with using standard combinatorial testing

Some of the disadvantages of directly applying combinatorial testing on a complex Simulink / Stateflow model are:

- To be able to avoid the resetting of parts of the system caused by different inputs when directly applying test vectors from the AETG algorithm, many unnecessary values of those signals must be defined. For example, consider a reset signal that can be 0 or 1. If the signal is 1, the entire system is reset. The most obvious solution to that problem would be to extend with more values for that parameter. More zeros than ones, as for example $[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]$, which means that it is only approximately 10% chance to reset the system each sample. This method is very ineffective and it contributes to large test cases.

- State-machines that are used in the model under test sometimes require that several signals are constant over a longer period of time. The only effective solution to the problem would be to have a much longer sample time for the inputs than the sample time of the system itself. This can be a problem if some part of the system requires fast changes of some signal, and at the same time constant values of another signal. It also contributes to long test times or simulation times.

4.3.2 Problems with different send period

The inputs to the ACC-software during execution in the target system are received from the vehicle's CAN[1]-bus. The inputs arrive with different messages that are normally being sent with a specific interval. These intervals normally vary between 20ms and 100ms. To be able to compare the target system's

[1]Controller Area Network

outputs with the ones from the model, these update-intervals for the inputs must be resembled when the tests are created. The CAN-messages that contain the inputs to the ACC-software are being sent periodically over the CAN-bus with a period of 20, 40, 50 or 100ms.

4.3.3 Solutions

To solve the problems, the 33 inputs were divided into several ”classes”. Because the binary and integer inputs to the system used only 50 or 100 milliseconds update interval, only sequences with 50 and 100 milliseconds update interval was needed.

Binary 50ms Update-interval 50ms. The input is 0 or 1 with 50% chance of having the value 1, see table 4.1 on the next page.

Binary 50ms- mostly false Update-interval 50ms. The input is 0 or 1 with approximately 6% chance of having the value 1, see table 4.2 on the following page.

Binary 50ms- mostly true Update-interval 50ms. The input is 0 or 1 with approximately 94% chance of having the value 1, see table 4.3 on the next page.

Ternary 50ms Update-interval 50ms. The input is 0, 1 or 2, with approximately 5% chance of having the value 1 and 5% chance of having the value 2. This class is mostly used for error code simulation, thereby the low probability of having a different value than zero. See table 4.4 on the following page.

Binary 100ms Update-interval 100ms. The input is 0 or 1 with 50% chance of having the value 1, see table 4.5 on the next page.

Binary 100ms- mostly false Update-interval 100ms. The input is 0 or 1 with approximately 6% chance of having the value 1, see table 4.6 on the following page.

Binary 100ms- mostly true Update-interval 100ms. The input is 0 or 1 with approximately 94% chance of having the value 1, see table 4.7 on page 23.

Ternary 100ms Update-interval 100ms. The input is 0, 1 or 2, with approximately 5% chance of having the value 1 and 5% chance of having the value 2. This class is mostly used for error code simulation, thereby the low probability of having a different value than zero. See table 4.8 on page 23.

Floating-point values These values are treated separately because every signal has different properties.

Some of the inputs had values that differed from the classes but were still very much alike. It was then possible to use one of the classes and apply some aftertreatment afterwards. For example, a signal that can be either 0 or 100. This signal can then be treated as a binary signal and then multiplied with a constant value (100).

For each of the classes, a set of sequences were created. The sequences have proven to be a good complement to combinatorial test generation because they provide some random behavior and at the same time it is possible to have sequences that are almost constant at some times. The different sequences are selected through indexes that are the *values* from the combinatorial test-generating algorithm. That way, the sequences are combined which hopefully improves the effectiveness of the tests. When creating several tests that are executed in a row, it is also possible to have different input sample time for the inputs.

The sequences are randomly generated but are constant through each test. Each sequence is 40 samples long and each sample is defined by the design parameter *input sample time*. Consequently, the length of each sequence can be a factor of 400 milliseconds.

Index	0	1	50ms 100ms 150ms Example
1	30%	70%	
2	70%	30%	
3	20%	80%	
4	80%	20%	

Table 4.1: Binary 50ms sequence

Index	0	1	50ms 100ms 150ms Example
1	98%	2%	
2	90%	10%	

Table 4.2: Mostly false 50ms sequence

Index	0	1	50ms 100ms 150ms Example
1	2%	98%	
2	10%	90%	

Table 4.3: Mostly true 50ms sequence

Index	0	1	2	50ms 100ms 150ms Example
1	98%	1%	1%	00000-00000-00000-00000-00000-00000-00000-00000
2	85%	10.5%	4.5	00000-00000-00000-00000-00000-11111-00000-00000
3	85%	4.5%	10.5%	00000-22222-11111-00000-00000-00000-00000-00000

Table 4.4: Ternary 50ms sequence

Index	0	1	100ms Example 300ms
1	30%	70%	
2	70%	30%	
3	80%	20%	
4	20%	80%	

Table 4.5: Binary 100ms sequence

Index	0	1	100ms Example 300ms
1	98%	2%	
2	90%	10%	

Table 4.6: Mostly false 100ms sequence

Index	0	1	100ms Example 300ms
1	2%	98%	
2	10%	90%	

Table 4.7: Mostly true 100ms sequence

Index	0	1	2	100ms Example 300ms
1	98%	1%	1%	0000000000-0000000000-0000000000-0000000000
2	85%	10.5%	4.5	0000000000-0000000000-1111111111-0000000000
3	85%	4.5%	10.5%	0000000000-2222222222-0000000000-0000000000

Table 4.8: Ternary 100ms sequence

4.3.4 Using manual tests to complement the automatically generated tests

Sometimes it is better to complement the automatically generated tests with some manual test cases instead of attempting to automate all testing. Some parts of the ACC-model are extremely hard to cover with automated tests. These parts are theoretically possible to cover using combinatorial testing, but attempting to do so would require very long tests because of the low probability of covering these parts. It is therefore more effective to design manual test cases to cover such parts.

The manual tests are preferably run before the automated tests since it is easier to design a manual test that starts when the system is in its initial condition. Having manual tests run before the automated tests guarantees that the *hard-to-cover* parts of the systems are exercised.

An example of a *hard-to-cover* condition is a state transition in the ACC-model that can only be active when the state-machine has been in the same state for more than *thirty* seconds. To be able to test this transition automatically, a very long input sample time would be needed. Since this is very ineffective it is better to create a manual test case that is executed before the automated tests. Finding these *hard-to-cover* parts is quite easy when performing a coverage analysis with Simulink Verification and Validation.

A method to develop and run manual tests before the automatically generated tests was implemented together with the test generation program.

4.3.5 Ways to improve model coverage

There are some simple ways to improve the condition coverage (see 2.3.4) and the MC/DC (see 2.3.5 and 6.1.1) without having to run more test cases. The reason is that transitions in Stateflow use shortcircuited evaluation of decisions. Shortcircuited evaluation of decisions means that when a decision consists of several conditions and the decision can be computed by just evaluating some of the conditions, the rest of the conditions are never evaluated. For example, consider a decision d, consisting of two conditions, a and b as $d = a\&\&b$. Let $a =$ *"false"* and $b =$ *"true"*, since a is evaluated *"false"* the decision is automatically *"false"* independently of what b is. Hence b is never evaluated.
The idea is to increase the chance that conditions that are rarely true or rarely false get full condition coverage and MC/DC by letting Stateflow first evaluate the condition that has the greatest possibility to be evaluated *"true"* if the decision is an *and*-operator between two or several conditions. If the decision is an *or*-operator between two or several conditions, the condition with the least chance of being true should be evaluated first for maximum testability. This method naturally requires good knowledge of the model. A disadvantage is that the execution time might increase. The increased execution time may lead to a processor load that comes closer to the worst execution time of the software. This may be negative

in some aspects but when the processor load increases it is easier to estimate the worst execution time, thereby increasing the reliability of the system.

The most intuitive way to write decisions is probably to write conditions in an order that leads to a minimal evaluation of conditions, thereby also minimal coverage. The programmer should then reconsider the evaluation order of the conditions if software testability is of higher priority than execution time.

and-operators

Example: Consider a decision $d = a \&\& b$ where a and b are boolean. Let the decision be evaluated with a first and then b. With shortcircuited evaluation, the decision b is only evaluated if a is true. If $p(a) = 0.1$ and $p(b) = 0.4$ the probabilities of getting the different condition combinations are shown in table 4.9.

	a	b	probability		
1	F	-	$(1 - p(a))$	=	0.9
2	T	F	$p(a)(1 - p(b))$	=	0.06
3	T	T	$p(a)p(b)$	=	0.04

	b	a	probability		
1	F	-	$(1 - p(b))$	=	0.6
2	T	F	$p(b)(1 - p(a))$	=	0.36
3	T	T	$p(b)p(a)$	=	0.04

Table 4.9: Probabilities of different results of $a \&\& b$ and $b \&\& a$

For condition coverage, all the test cases in table 4.9 must have been run. Hence, the probability of getting full condition coverage for both a and b with just 3 test cases is:
$6 * 0.9 * 0.06 * 0.04 = \underline{0.01296}$

If the decision is evaluated as $d = b \&\& a$ (b is evaluated first) the probability of getting full condition coverage with just 3 test cases is:
$6 * 0.6 * 0.36 * 0.04 = \underline{0.05184}$

With this manoeuvre, we have improved the chance of getting full coverage by a factor of $\frac{0.05184}{0.01296} = 4$ compared to the previous method. This does not improve the decision coverage, since the outcome of the decision is independent of the order of evaluation. It is important to mention that applying this method may increase the processor load during execution of the software. Hence it should only be applied on systems where testability is of higher importance than execution time, because the shortcircuited evaluation normally saves computation time.

or-operators

Example: Consider a decision $d = a||b$ ($||$ is shortcircuited logical *or*) where a and b are boolean. Let a be evaluated first and then b with shortcircuited evaluation. If $p(a) = 0.9$ and $p(b) = 0.3$ the probabilities of getting the different condition combinations are shown in table 4.10

	a	b	probability		
1	F	F	$(1 - p(a))(1 - p(b))$	=	0.07
2	F	T	$(1 - p(a))p(b)$	=	0.03
3	T	-	$p(a)$	=	0.9

	b	a	probability		
1	F	F	$(1 - p(b))(1 - p(a))$	=	0.07
2	F	T	$(1 - p(b))p(a)$	=	0.63
3	T	-	$p(b)$	=	0.3

Table 4.10: Probabilities of having different condition combination of $a||b$ and $b||a$

The probability of getting full condition coverage with just 3 test cases is:
$6 * 0.07 * 0.03 * 0.9 = \underline{0.01134}$

If the decision is evaluated in the opposite order ($d = b||a$) the probability of getting full condition coverage with 3 test cases is:
$6 * 0.07 * 0.63 * 0.3 = \underline{0.07938}$

This means an increase in the probability by a factor $\frac{0.07938}{0.01134} = 7$. This method can mean a significant chance of increasing coverage especially for decisions with many conditions.

4.4 Test design

It is still a skill to design a good test with the program that uses *sequence based combinatorial testing*. Without proper specifications the program will not be able to create a test with acceptable coverage. There are some design parameters that must be specified before the program is set to generate a test. These design parameters are:

- Test sequences
- Ways to handle floating-point inputs
- Choice of data ranges and classification of inputs
- *Precision* and *input sample times* in each test as well as the number of tests (with precision means how many of the pairs that at least must be covered in each test)

A tester that is familiar with the model under test probably has some knowledge about data ranges of the signals. He also probably knows in which intervals an input is particulary interesting and which inputs that reset the system with certain values.
With this knowledge it is possible to create a good test with SBCT with much less effort than manual testing. With the help of a model-coverage tool such as Simulink Verification and Validation it is possible to run tests and analyze the coverage results to see what can be done to improve the coverage.

4.4.1 The test generation process

The test suites are created in a Matlab script from where the modified AETG-algorithm is called. The sequences and methods for floating-point handling are then applied to the vectors of indexes that are generated by the AETG-algorithm. An overview of the test generation process can be seen in figure 4.3 on the following page

4.4.2 Choice of test sequences

Sequences are suitable for binary or integer inputs. If many tests are executed in a row it is a good idea to generate random sequences in each test. This will ensure that many different variations of the inputs are combined. Yet, the distributions of the test sequences have to be carefully selected.

4.4.3 Choice of data ranges

The binary or integer inputs must be classified for a certain sequence type. There are often several inputs that have similar properties and they can therefore use the same sequences.

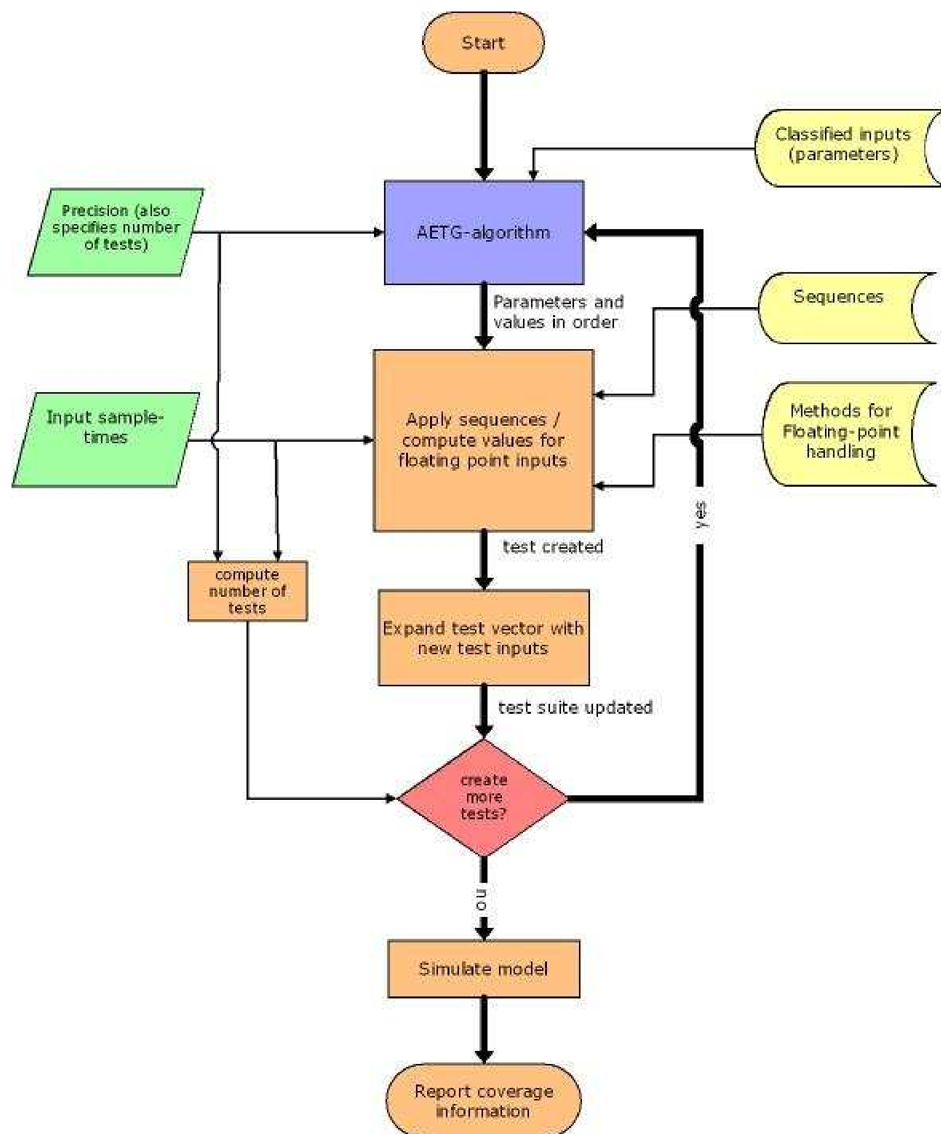


Figure 4.3: Overview of the test generation process with SBCT-method

4.4.4 Precision and input sample time

Precision

Precision is a design parameter that specifies how thorough a test is. Precision = 1 means that all pairs must be covered before a test is complete. Creating tests with full precision tend to demand much more test cases than a test with a precision just below 1. For example, see table 4.11 on the next page.

Precision	n.o. sequences
1	81
0.99	49
0.95	27
0.90	20
0.5	5

Table 4.11: An example of the number of sequences needed for the ACC model with 33 inputs. Consider that each sequence consists of 40 test cases, thus each sequence is at least 40 simulation steps.

Input sample time

Input sample time is the least time that each input is held constant. When creating several tests that are executed in a row, it is possible to vary the input sample time in each test. Using long and short input sample times have their advantages:

- When parts of the system requires that some inputs are constant a certain period of time it is favorable to use a longer input sample time for the inputs.
- Short input sample times result in shorter simulation time. When creating many tests, it is important to keep the simulation time down to a reasonable level.

To combine the advantages it is possible to create a few tests with long input sample times and the rest of the tests with a short input sample time. This will keep the simulation time down and at the same time exercise as many parts of the model as possible.

4.4.5 Modification of the model

To be able to run the tests on the model, all top level inputs and outputs were connected to from/to workspace blocks. There were also some parts of the model that was ”dead”; states that are not reachable independently of the inputs. These parts were removed. In chapter 7, a case study is presented where more information about dead code that was discovered is listed.

4.4.6 User interface

In the Matlab script that calls the modified AETG-algorithm, two design parameters need to be specified to create a test suite. Those variables are *precision* and *input sample time*. They are specified as vectors, and the length of those vectors also specifies how many tests that are to be created. In figure 4.4 a screenshot from the simple user interface in the Matlab script is presented.

```
%Design parameters

precision = [0.75 0.90 0.90 0.90 0.95 0.95 0.90 0.99 0.99 1.00];
t_sample  = [0.01 0.01 0.01 0.10 0.01 0.04 0.04 0.15 0.01 0.02];
```

Figure 4.4: The user specifies the precision and input sample time for each test and thereby also the number of tests

4.4.7 Overview

Figure 4.5 shows the sequence of work to create a test with the SBCT-method.

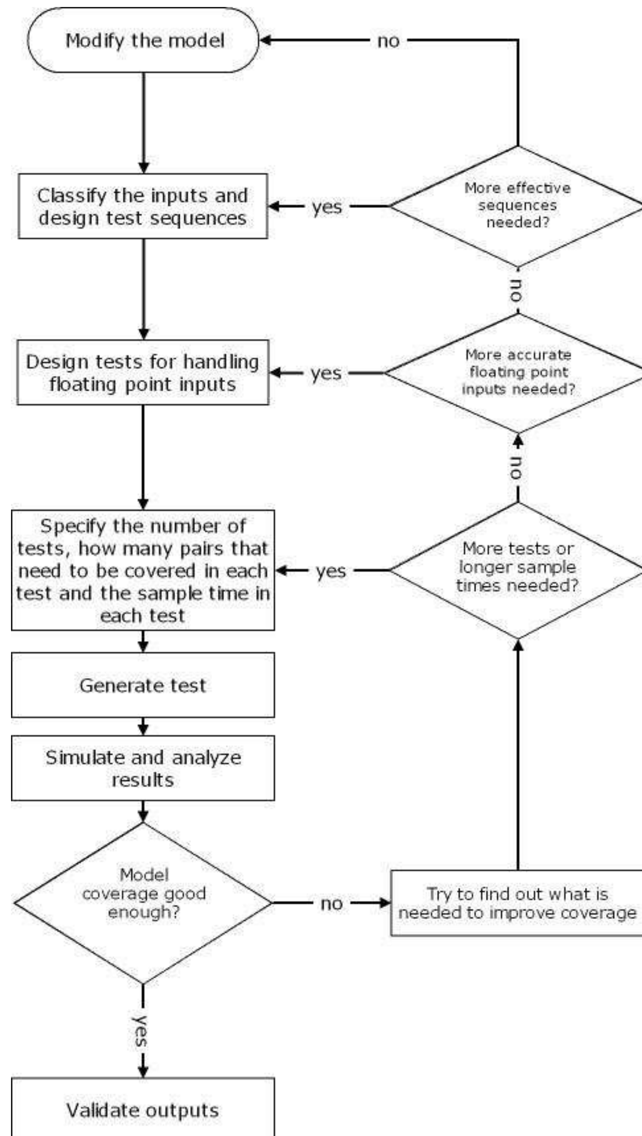


Figure 4.5: The steps to create a test suite with the SBCT-method

Chapter 5

Comparison between different test generating methods

For the comparison between the different methods, the ACC-model that was described in section 4.2.1 was used as a reference model.

5.1 Manual testing

Manual unit tests have been created and run on the adaptive cruise control system to verify the functionality of the system. These manual tests were not run on the model, only the target system was tested against its requirements. The test cases has been developed in close relation with the hardware, using a program that has a CAN[1]-interface to the target system. These tests are partly *event-triggered*, meaning that the response from the system affects the inputs.

A method for parsing log-files and decoding CAN-message data was developed during the work of the thesis. This method was implemented as a matlab function. With this method, it was possible to run the manual unit tests on the model. Thereby it was possible to see what levels of model coverage that is achieved with the manual tests. The cumulative coverage after running the manual unit tests is found in table 5.1

Decision coverage	85.4%
Condition coverage	73.8%
MC/DC	53.9%

Table 5.1: Model coverage for the cumulative results of the manual unit tests

From the coverage of the manual tests, we can see that most of the decisions have been covered. This means that most of the basic functionality probably have been tested. It can be seen that MC/DC is low in these tests and this could possibly be explained by the fact that no effort has been put to explicitly test all combinations of conditions. The typical relation between decision coverage, MC/DC and condition coverage appears, compare the results with the graph in figure 2.6 on page 10.

With the newly introduced help of coverage analysis the manual tests can be improved in the future.

[1] Controller Area Network

5.2 Random testing

A simple random test generation program was created. The program requires input specifications (data types and data ranges). The tester then specifies the number of test cases and the input sample time of the inputs. The program then generates a test with randomly generated inputs, given the input specifications. Random testing is a black-box test technique, so no knowledge about the system has been used. This means the values of the inputs are distributed evenly. For example a binary signal has the same probability of having either 1 or 0 and floating point signals are evenly distributed within the specified interval. This causes a problem when signals that resets the system or a part of the system is generated (see section 4.3.1).

Another problem with random testing is that the tests tend to become very large in order to achieve acceptable levels of coverage. One of the few advantages is that the time and effort to generate the tests is close to none. No real effort is required to generate random tests, though on more complex systems random testing is ineffective and insufficient to exercise the system enough.

When the ACC-model was tested with the random testing method using as much as *one million* steps and an input sample time of *10ms* the coverage shown in table 5.2 was achieved.

Decision coverage	35.7% (137/384 decisions)
Condition coverage	51.9% (178/343 conditions)
MC/DC	40.9% (61/149)

Table 5.2: Model coverage for random testing with *1 million* test steps and *10ms* input sample time

The same model was then tested with another random test, using *10 000* steps and an input sample time of *200ms*. The coverage results is presented in table 5.3.

Decision coverage	70.8% (272/384 decisions)
Condition coverage	70.6% (242/343 conditions)
MC/DC	53.7% (80/149)

Table 5.3: Coverage for random testing with *10 000* test steps and *200ms* input sample time

Apparently the coverage is much better in the test with longer sample-time. This depends on the fact that several parts of the model requires signals to be constant a while for validation. However, the coverage results are not yet satisfactory.

It is particulary interesting to see that the test with short sample time gives better condition coverage and MC/DC percentage than its decision coverage percentage. Compare this relation to the graph of typical relation between the coverage metrics in figure 2.6 on page 10. An explanation to this may be that many state-machines in the model requires validation times for a state-transition to be active. With the short sample time, the signals change so fast that it does not allow these validation counters to be true. Therefore, transitions with validation counters never become evaluated *true* but many combinations of the other conditions in the transition are combined because so many test steps were used.

5.3 Model based testing in Reactis

Reactis®(see 3.1.2 on page 13) is a tool that is not integrated with Simulink/Stateflow as Simulink Verification and Validation is. The model is loaded directly in Reactis where test suites can be generated and run to check predefined requirements. Using a separate testing tool such as Reactis has its benefits but also it has disadvantages.

Some of the positive aspects with Reactis are:

+ Assuming that the effort of modifying the model is minimal, it is really fast to create a test with good and not rarely full coverage. This can be done without any knowledge at all about the internal structure of the model. The only design parameters in the tests are the data ranges of the inputs.

+ The target and assertion features of Reactis are really handy when evaluating model requirements. It is clever that the test generation algorithms are trying to cover the assertions, meaning that Reactis is trying to produce a ”failure” if possible.

+ Reactis is excellent when it is important to keep the simulation time down and still get acceptable coverage. Reactis removes all test cases that are redundant which results in very short and efficient tests.

+ Reactis is an excellent tool for easily creating tests for a model that is changed frequently. Again assuming that not much effort is required to modify the model.

+ Reactis identifies unreachable states in the model, this is a good way to identify dead code.

There are also some negative aspects with Reactis, some of them are:

– One of the main disadvantages is that Reactis does not support the entire Simulink library. It also has different modeling rules than Simulink. Together this means that the model must be developed from the beginning together with Reactis to be able to run properly in Reactis. It also implies some limitations. In control systems, for example it is practical to use integrators, derivatives and state-space representation of models. However, using these Simulink blocks are prohibited by Reactis. It is also prohibited to use embedded Matlab functions. In many situations it is possible to go around the limitations but building the model after Reactis' rules requires additional time and effort and could lead to ineffective solutions. Even when the model is developed from the beginning using Reactis, it is annoying not to be able to use the most obvious or effective solution to a problem just because the test-tool prohibits the usage of certain blocks.

– A disadvantage of having a test tool that is separate from the modeling tool is that, when the modeling tool is updated, it is not possible to use new features in the modeling tool until the test tool has been updated to support the new features.

– An experience from trying to test the ACC-model in Reactis is that the interaction between Matlab and Reactis seems to be quite unstable.

Overall, Reactis is great when it is important to create tests on a model that is changed often. Though it is strongly recommended that the model is developed together with Reactis from the beginning. The ideal test tool would be integrated with the modeling environment and still capable of generating automated tests of good quality such as Reactis. The test generation process in Reactis is divided into two phases. The first phase, the random phase, generates inputs at random and keeps only those that are relevant. The second phase, the targeted phase, consists of the sophisticated algorithms that attempt to exercise all parts of the system.

For some types of models, Reactis is a superb test-generating tool but the ACC-model described in section 4.2.1 was not possible to simulate or create a test for in Reactis because of the strict rules.

A part of the model was extracted and tested in Reactis. The subsystem was a quite large state-machine with 15 inputs. After a thorough test-generation a test suite with a total of 13 557 test cases was created. Reactis required ten minutes to create a test suite that fully covered the subsystem. The results were excellent, see table 5.4.

Decision coverage	100% (46/46 decisions)
Condition coverage	100% (94/94 conditions)
MC/DC	100% (47/47)

Table 5.4: Coverage for an automatic test created by Reactis on a part of the ACC-model

5.4 Sequence based combinatorial testing (SBCT)

The *sequence based combinatorial testing*-method that was described in chapter 4 was applied to the ACC-model. With the help of Simulink Verification and Validation it was possible to design a test with good coverage. A few parts of the model required that a small manual test was executed before the automatically created test to achieve full coverage. These parts are possible to cover even with the automated tests but the probability of getting these covered are so small that the tests must be huge to have a reasonable chance of covering them. Therefore it is more effective to have a simple manual test run before the automated tests (see 4.3.4 for more details).

After the modification of the model, the manual and automatic tests consisting of totally *30 000* test cases and a total simulation time of *18* minutes were able to cover the model completely, when considering decision, condition and modified condition/decision coverage, see table 5.5. The test took *14* minutes to create.

Decision coverage	100% (384/384 decisions)
Condition coverage	100% (343/343 conditions)
MC/DC	100% (149/149)

Table 5.5: Coverage for an automatic test created by the SBCT-method on the full ACC-model

The combinatorial approach has proven itself to consequently achieve full coverage when creating a thorough test. The algorithm requires between 10 and 20 minutes to create a test for the ACC-model with full coverage which is considered acceptable. The simulation time is also acceptable (15-25 minutes will mostly be enough for the ACC model), which makes it possible to run the tests on the target systems in realtime.

Using the SBCT-method to test complex systems has its advantages.

+ It is possible to use knowledge of the system to design effective tests that exercise as many parts of the system as possible.

+ Sequences can be specified to resemble actual system inputs.

+ The tests can be run in the modeling environment, meaning that no problems with interaction between different environments occur.

+ The size of the tests are kept down to a reasonable level compared to random testing.

+ It produces excellent results!

There are also drawbacks of using the modified AETG algorithm.

− To be able to develop a good test, it is necessary to have a tool that can measure the coverage of the system. These tools will most likely cost money fully comparable to the cost of buying a model-based testing tool such as Reactis.

− The test design is more time consuming than when using a commercial test tool or applying a random test. Especially when the automated tests are complemented by manual tests.

− The test design requires that the tester has a little more knowledge than data types and data ranges. It is therefore recommended that the tester is somehow involved in the development.

− If the model is drastically changed, it is possible that the test design also must be changed to meet the new model requirements. That is to say, the tests must be maintained to keep up with a changing model.

Chapter 6

Analyzing automatically generated code

6.1 The relation between code coverage and model coverage - a theoretical approach

In an environment where code is automatically generated from the model, it is particulary important to compare the differences between model coverage and code coverage.

6.1.1 MC/DC on shortcircuited evaluation of logical expressions

Shortcircuit logical operators only evaluate conditions when their result can affect the encompassing decision [4]. The MC/DC criterion needs to be redefined for programming languages that use shortcircuited logic since the condition outcomes are never measured for conditions that are not evaluated. It is therefore necessary to leave the requirement that *all other conditions should be held fixed* [2].

In table 6.1 on the following page the possible test cases of a non-shortcircuited evaluation of the expression $d = (a\ or\ b)\ and\ c$ is compared to the possible test cases in shortcircuited evaluation of the same expression. The combinations of test cases needed for a condition to satisfy MC/DC are grouped together with the possible test cases.

6.1.2 Extensions on MC/DC for coupled conditions

Conditions that cannot be varied independently are said to be *coupled* [2].

Weakly coupled conditions

Two or more conditions are weakly coupled if varying one condition *sometimes* varies the others. For example, the two conditions $a = (x > 10)$ and $b = (x < 12)$ are weakly coupled because when x changes from 9 to 14, both conditions change. Weakly coupled conditions normally do not mean a problem when measuring MC/DC. It is still possible to achieve MC/DC in a decision with two or more weakly coupled conditions.

Non-shortcircuited versus shortcircuited evaluation

Non-shortcircuited: $d = (a|b)\&c$

test case	abc	d	a	b	c
1_n	TTT	T			3
2_n	TFT	T	7		4
3_n	TTF	F			1
4_n	TFF	F			2
5_n	FTT	T		7	6
6_n	FTF	F			5
7_n	FFT	F	2	5	
8_n	FFF	F			

Shortcircuited: $d = (a||b)\&\&c$

test case	abc	d	a	b	c
1_s	T-T	T	5		2
2_s	T-F	F			1
3_s	FTT	T		5	4
4_s	FTF	F			3
5_s	FF-	F	1	3	

Table 6.1: With shortcircuited evaluation of logical expressions, the test cases 1_n and 2_n can be grouped together as 1_s. Similarly test cases 3_n and 4_n can be grouped together as 2_s and test cases 7_n and 8_n can be grouped together as 5_s.

Strongly coupled conditions

Two or more conditions are strongly coupled if varying one condition *always* varies the others. For example, the conditions a and $\overline{a}$ are strongly coupled as they can never be true or false at the same time ($\overline{a}$ is equivalent to $NOT(a)$). With strongly coupled conditions it is not possible to vary one condition while holding all others fixed. The independence of single conditions that are required by the general definition of MC/DC are then impossible to prove. Therefore, the original definition of MC/DC needs to be modified for strongly coupled conditions if full coverage should be reachable. In [2], Chilenski and Miller suggest two alternatives, *weak* and *strong* MC/DC.

Weak **modified condition/decision coverage**

In this alternative, the strongly coupled conditions are treated as a single condition. For example, the decision $d = (a\&b)|(\overline{a}\&c)$ contains two strongly coupled conditions (a and $\overline{a}$). Instead of requiring all of a, b, $\overline{a}$ and c to independently affect the decision outcome, only a, b and c must have shown independence to affect the decision outcome. Unfortunately, weak MC/DC does not guarantee decision coverage [2].

Strong **modified condition/decision coverage**

In *strong* MC/DC each instance of a coupled condition is viewed as a separate entity. Each instance of that variable must satisfy the general definition of MC/DC while the effects of all other instances of that variable is *masked*. An instance of a variable is *masked* if it cannot affect the overall outcome. For example, consider the decision $d = (a\&b)|(\overline{a}\&c)$. The first instance of a is *masked* if $b = false$ and the second instance of a is *masked* if $c = false$.

Finding coupled conditions

Measuring weak or strong MC/DC requires that the coverage tool can identify coupled conditions which can be far from trivial in some cases, sometimes it may be nearly impossible. Having decisions with strongly coupled conditions in a model or program may sometimes be unnecessary. If they can be changed, it will probably lead to a simpler logical expression. A way to find such decisions is to use the standard definition of MC/DC, run a thorough test and find that the coupled conditions can never

test case	abc	d	a_1	b	a_2	c
1	FFF	F	masked		masked	2
2	FFT	T	masked	4	6	1
3	FTF	F	7		masked	4
4	FTT	T		2		3
5	TFF	F	masked	7	masked	
6	TFT	F	masked	8	2	
7	TTF	T	3	5	masked	
8	TTT	T		6		

Table 6.2: *Strong* MC/DC

satisfy MC/DC. Hopefully the decisions can then be rewritten so that no coupled conditions are used. For example, $(a\&b)|(a\&c)$ can be rewritten as $a\&(b|c)$. Simulink V&V and Reactis®do not use weak or strong MC/DC for coupled conditions. It is therefore impossible to achieve full MC/DC on decisions with coupled conditions in Simulink Verification and Validation or in Reactis®.

6.1.3 Differences between model- and code coverage of a logical expression?

Model representation of a logical expression is usually done in Simulink with logical operator blocks. If these blocks are connected together, they represent a more complex logical expression. Coverage tools like Simulink V&V or Reactis®treat each logical block separate when they compute the model coverage. The same logical expression represented in code and analyzed by a code coverage tool would in some cases not be covered with the exact same test cases as in the Simulink model, assuming that the expression is written as just *one* expression and not divided into several steps. For example: Consider the expression $(a\&b)|(\overline{a}\&c)$ that includes the coupled conditions a and $\overline{a}$. This expression represented in a Simulink model would include four logical blocks; two AND, one NOT and one OR. See figure 6.1

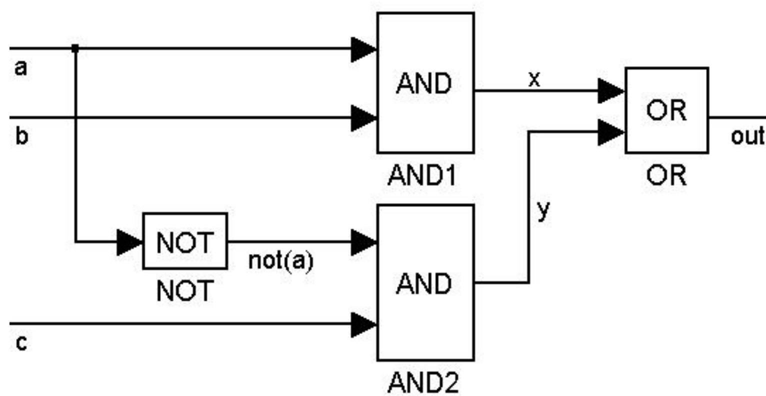


Figure 6.1: The logical expression $(a\&b)|(\overline{a}\&c)$ represented in a Simulink model

A model coverage tool would treat each logical block separate. Decision coverage and condition coverage are basically equivalent in model and code coverage. For example, full decision or condition coverage in one environment would imply full decision or condition coverage in the other environment with the same test cases. Only where full decision- or condition coverage is not achieved, model coverage will provide more information about which part of the expression that is uncovered.

The real difference is when measuring MC/DC. Here, an expression can be covered in model environment but not in the equivalent code expression. For example, consider expression $(a\&b)|(\overline{a}\&c)$, that is represented as a model in figure 6.1 on the preceding page. Full model MC/DC for all blocks in the expression can be achieved with the test cases in table 6.3 while full code MC/DC on the expression is impossible.

a	b	c	out
F	F	T	T
F	T	F	F
T	F	T	F
T	T	F	T

Table 6.3: Four test cases provide full MC/DC for all blocks in the model representation of $(a\&b)|(\overline{a}\&c)$ seen in figure 6.1 on the previous page

If an expression is divided into several steps, the same effect as when measuring model coverage can be obtained. Doing this will reduce the effectiveness of both MC/DC and similar coverage metrics such as multicondition coverage. For example, the expression $z = (a\&b)|(\overline{a}\&c)$ can be divided into three simpler expressions:

$$x = (a\&b)$$
$$y = (\overline{a}\&c)$$
$$z = x|y$$

When the expression is written like this, the full complexity of the expression can not be investigated by measuring MC/DC or multicondition coverage. The fact that the expression contains coupled conditions will also never be discovered.

6.1.4 Multicondition coverage

In section 2.3.6 on page 8, a brief summary of multicondition coverage is introduced. Multicondition coverage does not explicitly require all conditions to decide the outcome of its decision but since all possible combinations of condition outcomes are required it can be said that: *If MC/DC can be achieved for an expression, multicondition coverage guarantees MC/DC*. This would mean that multicondition coverage is a stronger metric than MC/DC but in many cases, multicondition coverage is unsuitable as it requires at least 2^n test cases for a decision with n conditions.

6.1.5 Coverage criteria

Different test tools have their own definitions of coverage criteria. When it comes to code coverage, the definitions are generally a little different than in model coverage. Although, it is interesting to draw some parallels between model coverage and code coverage, especially when the code is automatically generated from a model.

6.1.6 Code coverage tools

There are many code coverage tools available for analyzing coverage on C/C++ code, but most of these are not made to be able to use on automatically generated code from Matlab. To be able to measure the

code coverage on the automatically generated code, a tool that is either built for use on this type of code or something very adaptable is needed.

The choice of coverage tool for this project was Testwell's CTC++. This is a coverage tool that is run from the command prompt and generates HTML-coverage reports. An advantage of CTC++'s command prompt user interface is that it is very adaptable. By editing some configuration files in the Matlab directory and adding a few lines in the automatically generated makefile and main C-file, the automatically generated code could be instrumented and coverage reports could be generated.

6.2 Comparing model coverage and code coverage - an experimental approach

This section is a case study where the tests have been applied to a model in the model environment and the model coverage has been compared to the code coverage that is achieved when running the same tests on the the automatically generated code.

6.2.1 Method

For comparison between model coverage and code coverage, fifteen test suites were created with the SBCT-method described in section 4.2.2. These test suites had different levels of model coverage. By applying these test suites on the model and also on the automatically generated code, a relation between model coverage and code coverage was investigated.

6.2.2 Problems with comparing code and model coverage

One problem with directly comparing model coverage and code coverage is that Real-Time Workshop creates lots of code that is just there to prevent unexpected errors that never occur during normal execution of the program. Real-Time Workshop also creates functions that are run just once per simulation or execution. This code can not be covered by using better tests, hence the code coverage is generally much lower than the model coverage.

Another problem is that the code coverage tool, CTC++ does not measure MC/DC as Simulink Verification and Validation does. Instead of MC/DC, CTC++ measures multicondition coverage. The multicondition coverage is quite similar to MC/DC, but generally harder to satisfy (see section 6.1).

A good example of how many measure points that can be independent of the inputs is this simple Simulink block representing an ”absolute value” function.

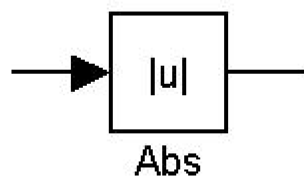


Figure 6.2: Absolute value block

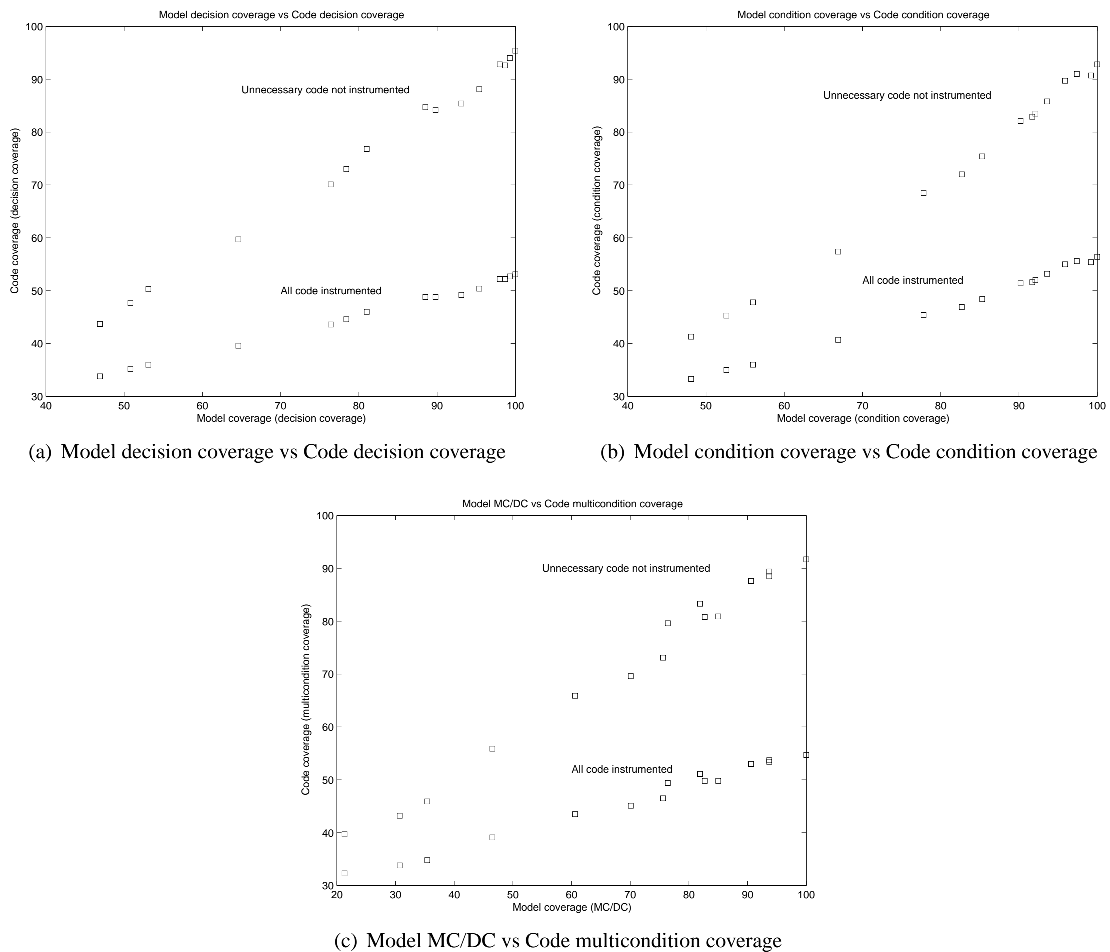
In Simulink Verification and Validation, this block corresponds to one decision. The input to the block must be both ≥ 0 and < 0 for the block to achieve full decision coverage. When generating an s-function from this block, the following code represents the function of the block:

```
((real_T *)ssGetOutputPortSignal(S, 0))[0] = (fabs((*(((const
  real_T**)ssGetInputPortSignalPtrs(S, 0))[0]))));
```

The CTC++ coverage reports contained a total of 423 decision points but the piece of code that really represents the ”absolute value” was not seen as a decision point. So, apparently we have 423 decisions that can not be affected by changing the input values. This might be the worst case scenario but it is also illustrative and suggests that *Model coverage* $\nRightarrow$ *Code coverage*

To make the code coverage percentage come closer to the model coverage, some functions could be decided to not be instrumented by CTC++. By choosing not to instrument functions that do not represent any model functionality, the code coverage was drastically improved.

In figure 6.3 the relation between the code coverage and the model coverage is shown.



(a) Model decision coverage vs Code decision coverage

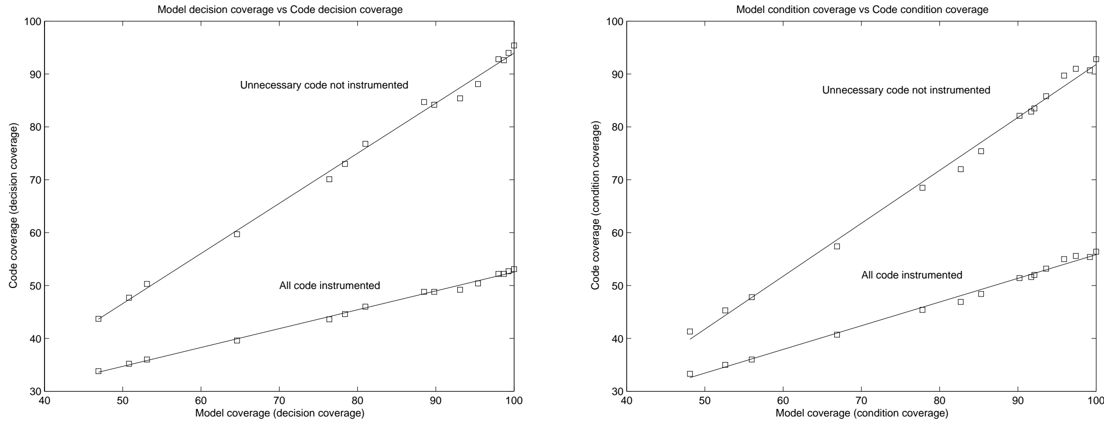
(b) Model condition coverage vs Code condition coverage

(c) Model MC/DC vs Code multicondition coverage

Figure 6.3: Model coverage vs Code coverage

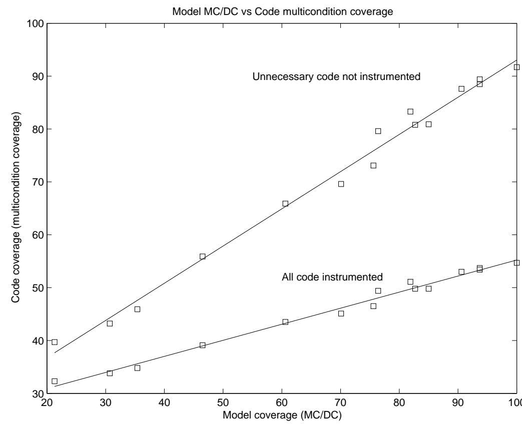
6.2.3 Linear approximation

It is clear that, at least in the figure 6.3a and 6.3b the relation is linear. Approximate linear relations was computed using the least squares method. (See figure 6.4)



(a) Model decision coverage vs Code decision coverage

(b) Model condition coverage vs Code condition coverage



(c) Model MC/DC vs Code multicondition coverage, linear approximations

Figure 6.4: Model coverage vs Code coverage, linear approximations using least squares method

The deviation from the linear model seems to be relative, increased deviation with increased model coverage. To be able to compute the standard deviation from the linear approximation, the code coverage percentage was weighted with the approximate value for the model coverage. Let $c_i, i = 1, \ldots, n$ be the code coverage percentage; $m_i, i = 1, \ldots, n$ the model coverage percentage and $c_{approx}(m_i)$ the linear approximation of the code coverage. The weighted coverage, w_i is then

$$w_i = \frac{c_i}{c_{approx}(m_i)} \tag{6.1}$$

An approximation of the standard deviation σ can then be computed as

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(w_i - \hat{w})^2} \qquad (6.2)$$

where $n = 15$ and $\hat{w} = 1$.

The computed approximation of the standard deviation is shown in the diagram in figure 6.5.

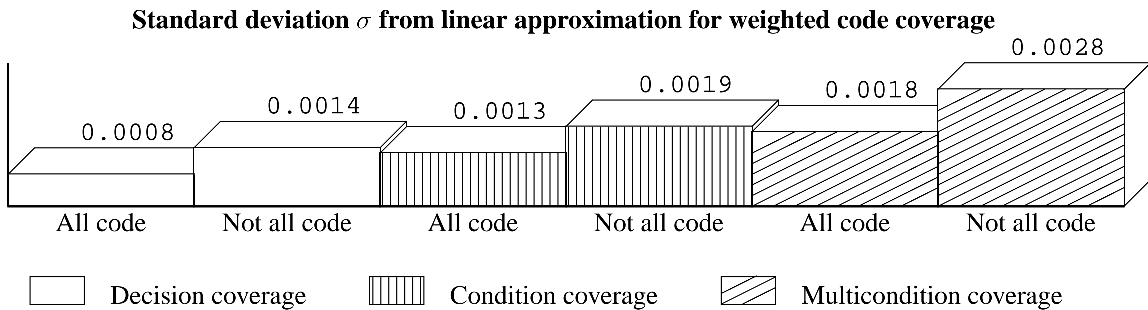


Figure 6.5: Standard deviation σ from linear approximation for weighted code coverage

From the diagram in figure 6.5 it appears that when some code was left uninstrumented by CTC++, the standard deviation of the weighted code coverage is significantly higher than when all code is instrumented. One possible explanation to this, might be that only around 40% of all decisions was instrumented, whereof about 90% of the decisions varied between different tests. Decisions that are unaffected by the test inputs correspond to the constant value in the linear approximation.

The overview in figure 6.6 shows how much of the code that is possible to cover by using different test inputs. Since the range of code coverage that can be affected by test inputs is smaller when all code

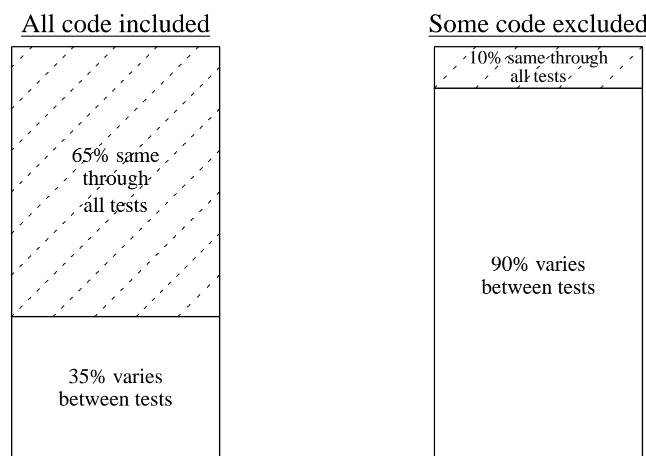


Figure 6.6: Percentage of decisions where the coverage varies between tests

is instrumented, the deviation from the linear model will be smaller. It is worth noticing that the number of measure points that can be affected by test inputs are the same in the case of instrumenting all code and the case of excluding certain parts of the code from instrumentation. See the sketches in figure 6.7.

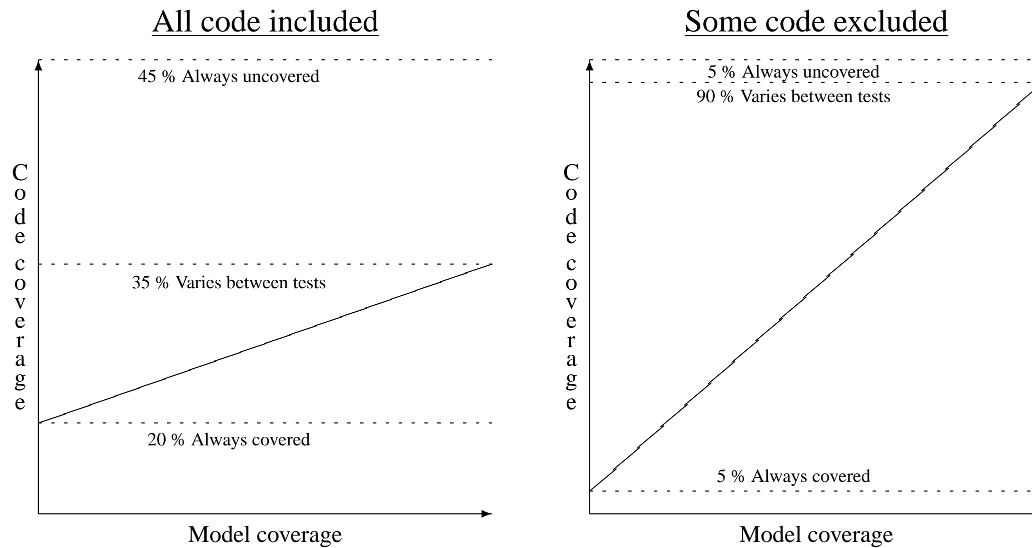


Figure 6.7: Portions of code that is always covered or always uncovered in a test

6.2.4 Verifying normal distribution

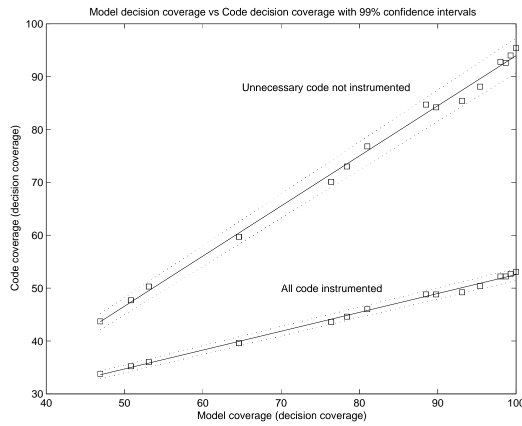
To verify that a normal distribution of the weighted coverage could be used, the matlab function `normplot` was used. This function makes a normal probability plot. The purpose of a normal probability plot is to graphically assess whether the data could come from a normal distribution. If the data is normally distributed, the plot will be linear. Other distribution types will introduce curvature in the plot [10]. The normal probability plots are shown in figure 6.9 on page 45.

6.2.5 Verifying linear model

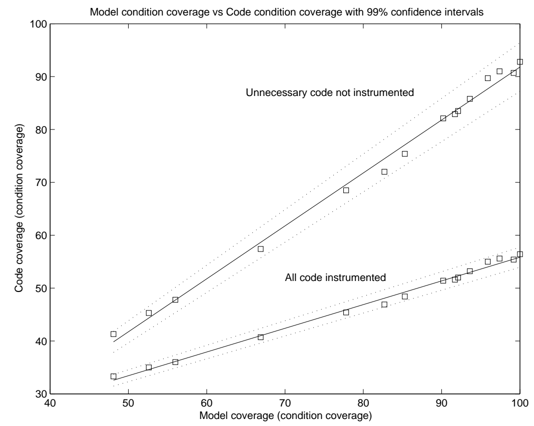
The normal probability plots in figure 6.9 on page 45 show that the weighted code coverage could be normally distributed. With this information it is possible to create confidence intervals. The purpose of the confidence intervals is to show that the relations really are linear. If a 99% confidence interval leaves no space for another relation than a linear, the relation is most probably linear. In figure 6.8 on the following page the relations are shown with 99% confidence intervals, with these confidence intervals it is reasonable to assume a linear relation.

6.2.6 Generalization

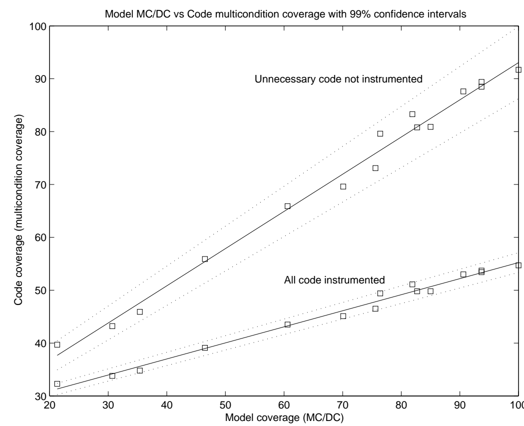
The results of the previous section, that the relation between code coverage and model coverage is linear is based on tests on the ACC-model only. There may very well be models where a linear relation does not exist, for example a model full of absolute value blocks (see section 6.2.2). Normally, large models contain many types of blocks and state-machines. It is then likely that the relation between code and model coverage is relatively linear as in the case of the ACC-model.



(a) Model decision coverage vs Code decision coverage

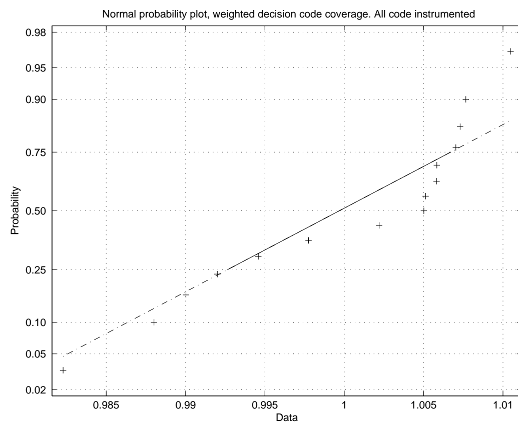


(b) Model condition coverage vs Code condition coverage

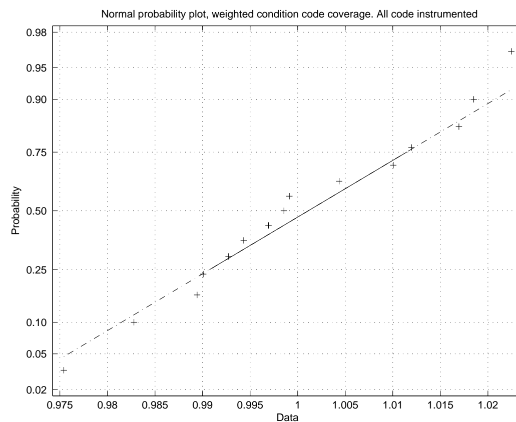


(c) Model MC/DC vs Code multicondition coverage

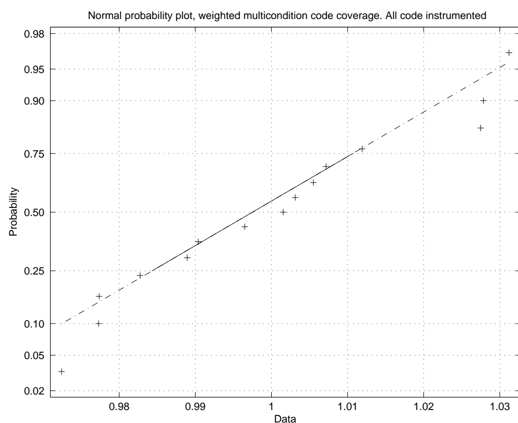
Figure 6.8: Model coverage vs Code coverage



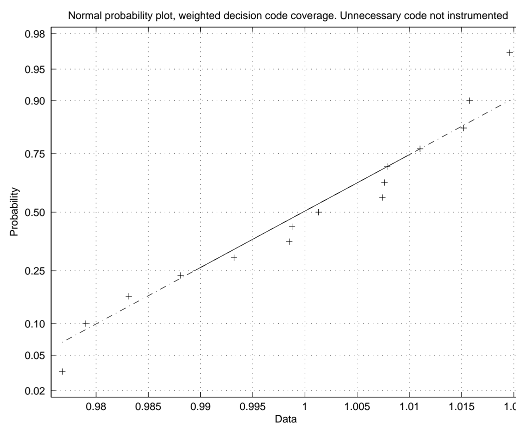
(a) Normal probability plot, weighted decision code coverage. All code instrumented



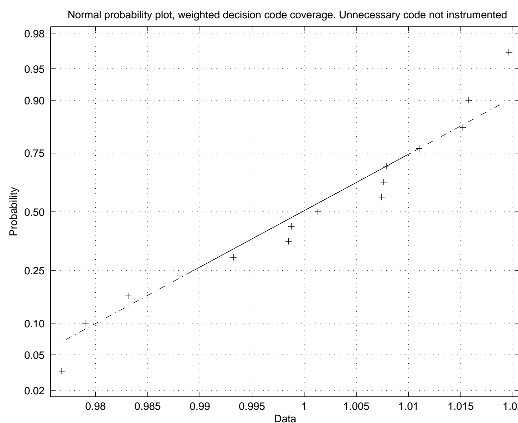
(b) Normal probability plot, weighted condition code coverage. All code instrumented



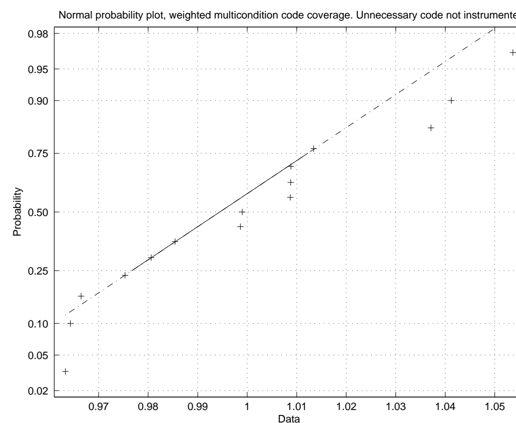
(c) Normal probability plot, weighted multicondition code coverage. All code instrumented



(d) Normal probability plot, weighted decision code coverage. Unnecessary code not instrumented



(e) Normal probability plot, weighted condition code coverage. Unnecessary code not instrumented



(f) Normal probability plot, weighted multicondition code coverage. Unnecessary code not instrumented

Figure 6.9: Normal probability plots for weighted code coverage

6.2.7 Classification of uncovered code

When full model coverage is achieved, the 5% code coverage that remain uncovered in the case where irrelevant code was excluded is particulary interesting, especially to see if there are any special structures of code that are never covered. The diagram in figure 6.10 classifies all uncovered decisions encountered in the code generated from the ACC-model. The 5% uncovered decisions correspond to the total of 25

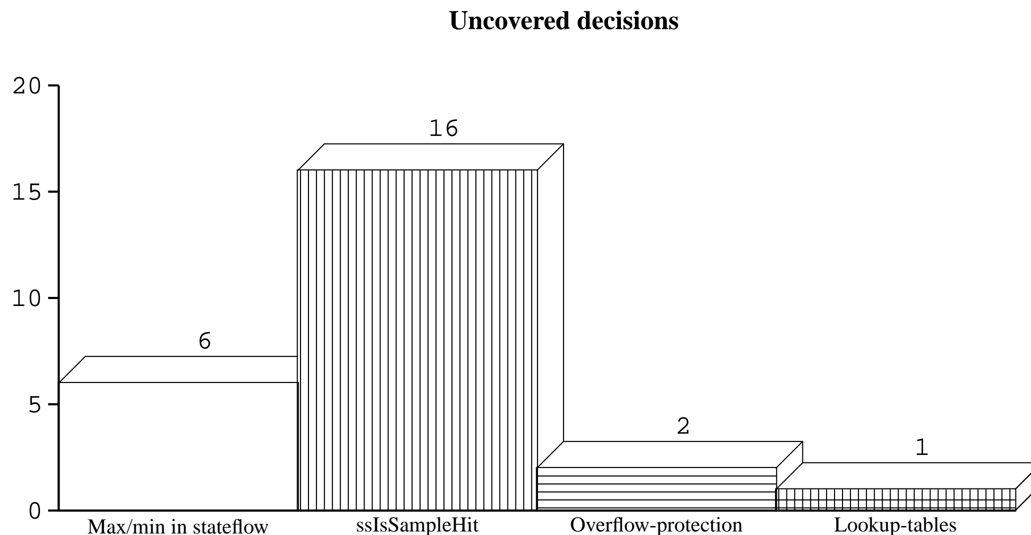


Figure 6.10: Uncovered decisions measured on code generated from the ACC-model and from a test with 100% model coverage

decisions above. A description of the classes of code follows:

Max/min in stateflow: Coverage for Max- and min-functions written in the *entry:, during: or exit:*-phase in a stateflow state is not measured by Simulink Verification and Validation. The fact that these max- and min-functions are uncovered is very interesting, since it is possible that the code is dead.

ssIsSampleHit is a function that:

- is run as: `if ssIsSampleHit(...) { ... }`
  to ensure that the following lines of code is supposed to be executed at the current simulation step.
- returns *true*, if the local sample time matches the current simulation time.
- When a part of the system has the same sample time as the global sample time, the expression above will never be evaluated *false*

Overflow-protection In the code generated from some *data type conversion*-blocks, code to protect from data type overflow is included. If an overflow does not occur, a decision is left uncovered.

Lookup-tables The code generated from a lookup-table contains a decision for each interpolation or extrapolation interval. Although Lookup-table coverage is possible to measure in Simulink Verification and Validation, the tests were not designed to achieve full Lookup-table coverage. Only the MC/DC, decision- and condition coverage were considered.

Chapter 7

Case study: results of coverage analysis

Automated test-vector generation and coverage analysis of model-based software provide excellent possibilities to find programming bugs and dead code at an early stage in the development process. This chapter is a case study of what the results of automatic test-vector generation and coverage analysis could lead to.

7.1 Finding bugs

Even though the focus of the work during this thesis was to just develop a method to generate tests, several program deficiencies were quickly found as a result of a fairly simple verification process. Not much effort has yet been put to verifying the correctness of the ACC-model, but probably the automated tests will continue to find program deficiencies once a more extensive method for verification is implemented.

7.2 Identifying dead code

During the calibration of the test, some situations where full coverage is not possible was discovered in the reference model.

Once a good test-suite has been run on a model, coverage analysis is perfect to identify dead code. The method to identify dead code during this project was:

- When a part of the model consequently does not achieve coverage, that part was reviewed.
- During the review, dead code was discovered.
- The model was changed to get rid of dead code but still keep the same functionality.

As a result of dead code the following amount of measure points was left uncovered:

- 10 Decisions
- 6 Conditions
- 4 MC/DC

The classes of dead code discovered in the ACC-model are listed in figure 7.1 on the following page

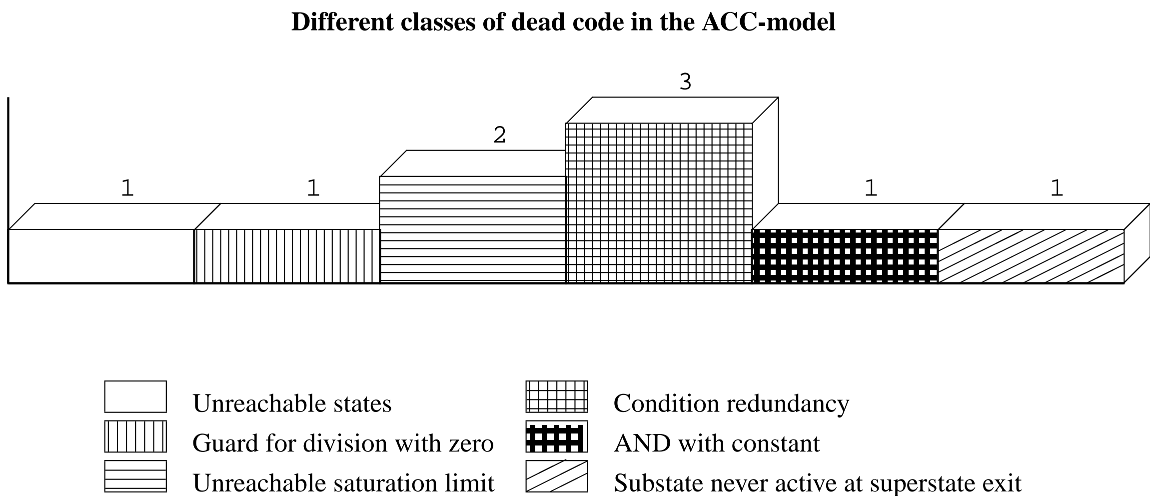


Figure 7.1: Different classes of dead code in the ACC-model

7.2.1 Unreachable states

In one of the state-machines of the model, one of the states was separated from the others and therefore unreachable. This resulted in that transitions from the state was never evaluated and that the state was never active when its superstate was exited, which is required by Simulink V&V.

Solution: The unreachable state and all transitions from it can be removed without having a guilty conscience.

7.2.2 Guard for division with zero

The model contained a guard to prevent division with zero. The guard was a maximum function between the divisor and a constant value. Because of other functions in the model, the divisor would never be less than the constant value. Therefore, the maximum function was always left uncovered.

Solution: A division with zero can lead to serious problems during execution of software. If there is complete confidence that the divisor is never less than a certain value, perhaps the max-function can be removed. Another alternative is to remove the *first* safety check and let the guard at the divisor remain.

7.2.3 Unreachable saturation limit

Two cases of unreachable saturation limits was discovered in the model. In one case, the upper limit of the saturation block was *infinity*. In the other case, the upper limit was unreachable depending on preceding actions in the model. In both these cases, the saturation block is uncovered.

Solution: Replace with max- or min-function

7.2.4 Condition redundancy

The most common situation that led to dead code in the model was when there was (at least) two transitions from a state and one of the conditions are redundant (i.e. always true or always false), see figure 7.2. Such situations makes it impossible to achieve full model coverage and it is therefore recommended to change the model so that full model coverage *can* be achieved.

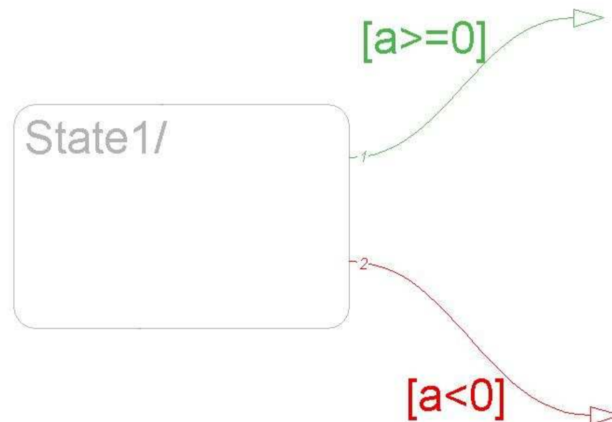


Figure 7.2: *Condition redundancy*: The decision $[a < 0]$ will never be evaluated false because it is evaluated last (The numbers on the state transitions indicate the evaluation order).

Solution: The solution to the example in figure 7.2 is to simply remove the condition $[a < 0]$. If the evaluation order is correct, the second transition will be true when $[a \geq 0]$ is false which obviously means that $a < 0$. That way, the functionality is the same and the redundancy is removed.

7.2.5 AND with constant

In one logical operator block, one of the inputs was a constant value. This of course led to uncovered conditions and MC/DC.

Solution: Just remove the AND-block, perhaps replace with a *data type conversion*-block so that the output becomes boolean.

7.2.6 Substate never active at superstate exit

Another situation where full coverage can not be achieved is when a substate can never be active while its superstate is exited, which makes full decision coverage impossible. This is illustrated in figure 7.3 on the following page.

Solution: Remove the superstate and if the superstate has its own functionality, include this in the substates.

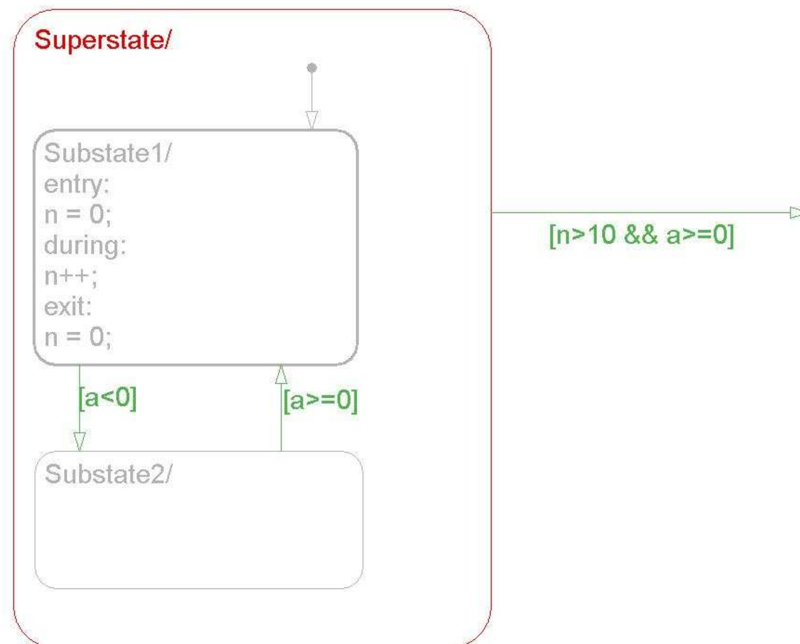


Figure 7.3: In order for a superstate to get full decision coverage, the superstate must have been exited at a time when each of the substates have been active. In this case, it is easy to see that *substate2* can not be active when the superstate is exited.

Chapter 8

Results

8.1 Conclusions

Model-based development and coverage analysis open new doors to automate testing. In this thesis, automated testing of software developed in a model-based environment such as Simulink has proven to be both more time-efficient and more accurate than previously used manual tests. The use of a model-coverage tool such as Simulink Verification & Validation provides possibilities to verify the accuracy of manual tests as well as new possibilities to automate a substantial part of the testing process.

8.1.1 Testing approaches

Different approaches to automate the generation of test vectors have been applied to a model of an adaptive cruise control system (ACC):

Random test generation is a very simple way to automate testing and may sometimes be enough for small models. Unfortunately it leads to very large tests and insufficient coverage on relatively complex models.

Model-based test generation with the test-tool Reactis®is a very fast way to generate test vectors for a model. Reactis has shown to be able to handle relatively complex models and to provide tests with good model-coverage. Unfortunately, there are some limitations to the models that Reactis can use to generate tests from. This made it impossible to generate tests for the full ACC-model.

Sequence based combinatorial testing (SBCT) is a method developed during the work of this thesis. The SBCT-method is a hybrid between combinatorial testing, random testing and manual testing and is mainly designed for systems where many binary or integer inputs exist. The tester must specify sequences where the probability of full coverage is maximized. This requires some knowledge about the model structure but this is generally not a problem when testing is performed by the same people that develops the software. Other limitations of the method is that maintenance of the test-script may take some time when the model is updated. It also takes some time to design a test for a new model.

Manual testing may sometimes be the most effective way to test certain functionality. Automated tests of different kinds are ineffective when it comes to testing certain parts of the ACC-model. Therefore, manual tests will not be totally replaced be automatic tests.

8.1.2 Coverage analysis

The structure of the model determines the maximum coverage that can be achieved. Modifying the model by removing dead code or by rearranging the evaluation order of conditions may improve the coverage without having to run more or longer tests. In chapter 7 a case study where the results of coverage analysis of the ACC-model is presented.

8.1.3 Code coverage vs. Model coverage

In chapter 6, code- and model coverage has been compared with both experimental and theoretical approaches.

The results of the experimental approach suggest that code coverage measured on automatically generated code follows a linear relation with the model coverage, although the automatically generated code normally contains many decisions and conditions that have no model counterparts.

The theoretical approach showed that logical expressions represented as several logical blocks connected together in Simulink do not have the same prerequisites for achieving MC/DC as a code representation of the same expression would have.

8.2 Future work

The SBCT-method could be even more automated and provided with a graphical user interface that would facilitate the generation of test vectors, the design of sequences and the handling of floating-point inputs. One idea of further automation is that user still classifies the binary or integer inputs with the probability of the input being true and the program automatically generates the structure of the sequences. The floating-point inputs could be classified as one of many different types and for each type (e.g. cross a fixed value), a number of parameters is specified. The parameters could be data type, signal range, the speed of increasing the value, conditions for continuing from last sequence etc. The SBCT-method is now optimized for the ACC-model. In order for the method to be more generalized, more methods for floating-point handling could be added and more options of sequences (e.g. integer sequences with an optional amount of possible values) could be added.

The method could also be complemented with a verification tool integrated with the test generating program where the model can be tested against its requirements.

References

[1] Mark Blackburn, Aaron Nauman, and Bob Busser. Defect identification with model-based test automation. SAE 2003 World congress, Detroit, Michigan, USA, March 3–6 2003. Society of Automotive Engineers, Inc. Document number 2003-01-1031.

[2] J. Chilenski and S. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.

[3] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Trans. Softw. Eng.*, 23(7):437–444, 1997.

[4] Steve Cornett. Code coverage analysis, July 29 2004. http://www.bullseye.com/coverage.html.

[5] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 285–294, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[6] Applied Dynamics International. Beacon for Simulink and Stateflow product description. http://www.adi.com.

[7] Mark Fewster and Dorothy Graham. *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[8] David A. Gustafson. Software testing and inspection. AccessScience@McGraw-Hill, July 26 2002. .

[9] B. Marick. Experience with the cost of different coverage goals for testing, 1991. http://citeseer.ifi.unizh.ch/marick91experience.html.

[10] The Mathworks. Matlab documentation. http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html.

[11] Reactive systems inc. Reactis User's guide V2005, 2005. http://www.reactive-systems.com.

[12] T-VEC. T-vec Tester for Simulink product description. http://www.t-vec.com.

[13] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.

[14] Robin Westberg. Using full system simulation to perform non-intrusive measuring of code coverage. Master's thesis, Swedish Royal Institute of Technology, 2005.

Notation

Symbols used in the report.

Variables and parameters

σ	Standard deviation
`T`	boolean value *TRUE*
`F`	boolean value *FALSE*
`-`	boolean value *don't care*

operators

$\vert$	logical *OR*-operator
$\&$	logical *AND*-operator
$\vert\vert$	shortcircuited *OR*-operator
$\&\&$	shortcircuited *AND*-operator
$\overline{a}$	logical *NOT*-operator ($\overline{a} = NOT(a)$)
$p(a)$	probability of a being true

LINKÖPING UNIVERSITY
ELECTRONIC PRESS

LINKÖPINGS UNIVERSITET
TEKNISKA HÖGSKOLAN

Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida:
`http://www.ep.liu.se/`

English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page:
`http://www.ep.liu.se/`

© Jonny Andersson
Södertälje, 14th December 2005