

Institutionen för systemteknik Department of Electrical Engineering

Examensarbete

Application of a New Software Tool for the Automated Test of Automotive Electronic Control Unit Software

Examensarbete utfört i Reglerteknik
vid Tekniska högskolan i Linköping
av

Hanna Amlinger

LITH-ISY-EX--09/4204--SE

Linköping 2009



Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Application of a New Software Tool for the Automated Test of Automotive Electronic Control Unit Software

Examensarbete utfört i Reglerteknik
vid Tekniska högskolan i Linköping
av


Hanna Amlinger

LITH-ISY-EX--09/4204--SE

Handledare: **Christian Lundquist**
ISY, Linköpings universitet
Dr. Martin Neumann
ZF Friedrichshafen AG
Dipl. Ing. Carsten Paulus
ZF Friedrichshafen AG

Examinator: **Dr. Inger Klein**
ISY, Linköpings universitet

Linköping, 10 February, 2009

	Avdelning, Institution Division, Department Division of Automatic Control Department of Electrical Engineering Linköpings universitet SE-581 83 Linköping, Sweden		Datum Date 2009-02-10
	Språk Language <input type="checkbox"/> Svenska/Swedish <input checked="" type="checkbox"/> Engelska/English <input type="checkbox"/> _____	Rapporttyp Report category <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	ISBN _____ ISRN LITH-ISY-EX--09/4204--SE Serietitel och serienummer ISSN Title of series, numbering _____
URL för elektronisk version http://www.control.isy.liu.se http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-16785			
Titel Title Application of a New Software Tool for the Automated Test of Automotive Electronic Control Unit Software Författare Hanna Amlinger Author			
Sammanfattning Abstract <p>Testing plays a very important role for assuring the quality of developed software. In a modern vehicle, more and more of the functionality is controlled by software and the complexity of the software always increases. The expectations on automating the testing process are to save time and to reach an even higher quality.</p> <p>In this thesis, which was performed at ZF Friedrichshafen AG, a new tool for automated tests is studied. The tool is used for software in the loop simulation based system tests. The user specifies which outputs that shall be observed and which inputs that can be controlled. Based on these prerequisites, test cases are generated.</p> <p>It has been studied how to apply the tool, how the test case generation can be influenced, on which systems it successfully could be used and which results that could be reached with the tool. The tool has been evaluated on the hand of two real-life examples; the software of an automatic transmission and of a pressure controller, a module of this software. It was found that there are many interesting possibilities to apply the tool in order to support the present testing process.</p>			
Nyckelord Keywords Test automation, Software in the loop, System testing			

Abstract

Testing plays a very important role for assuring the quality of developed software. In a modern vehicle, more and more of the functionality is controlled by software and the complexity of the software always increases. The expectations on automating the testing process are to save time and to reach an even higher quality.

In this thesis, which was performed at ZF Friedrichshafen AG, a new tool for automated tests is studied. The tool is used for software in the loop simulation based system tests. The user specifies which outputs that shall be observed and which inputs that can be controlled. Based on these prerequisites, test cases are generated.

It has been studied how to apply the tool, how the test case generation can be influenced, on which systems it successfully could be used and which results that could be reached with the tool. The tool has been evaluated on the hand of two real-life examples; the software of an automatic transmission and of a pressure controller, a module of this software. It was found that there are many interesting possibilities to apply the tool in order to support the present testing process.

Acknowledgments

This master thesis was performed at ZF Friedrichshafen AG in the department of Hybrid Drives (TE-H) during the summer and autumn of 2008. Firstly, I would like to thank ZF Friedrichshafen AG for the opportunity to write this thesis. A special thanks to my supervisors at ZF Friedrichshafen AG, Dr. Martin Neumann and Carsten Paulus for all their support and encouragement, for many interesting discussions and for coming with suggestions to improve my work.

I would like to express my thanks to my supervisor Christian Lundquist and examiner Dr. Inger Klein at Linköping University, especially for all the help with improving my writing.

The department TE-NL was very kind to provide me with a real-life example to work with, which helped me very much. Also, a special thanks to QTronic GmbH, in particular Dr. Andreas Junghanns and Dr. Mugur Tatar, for answering all my questions and for supporting me in many ways. I am also very grateful to all my colleagues at TE-H for welcoming me, making me a part of the team and for making my time at ZF Friedrichshafen AG a great experience.

Thank you all!

Contents

1	Introduction	1
1.1	Background	1
1.2	Problem Description	1
1.3	Purpose	1
1.4	Goal	2
1.5	Thesis Outline	2
2	Software Testing	3
2.1	Software Development	3
2.2	Test Levels	5
2.2.1	Unit Testing	5
2.2.2	Integration Testing	5
2.2.3	System Testing	6
2.2.4	Acceptance Testing	6
2.3	Testing Methods	6
2.3.1	Static Testing	6
2.3.2	Dynamic Testing	7
2.4	Black-Box Testing Techniques	7
2.4.1	Equivalence Class Testing	7
2.4.2	Boundary Value Testing	8
2.4.3	State Transition Testing	9
2.5	Code Coverage	10
2.5.1	Statement Coverage	10
2.5.2	Decision Coverage	12
2.5.3	Condition Coverage	12
2.5.4	Decision/Condition Coverage	13
2.5.5	Multiple Condition Coverage	13
3	Test Environment	15
3.1	System Overview	15
3.2	Instruments	16
3.3	Method	18
3.4	Test Focus	21
3.4.1	Coverage Report	21

3.4.2	Constraints	21
3.4.3	Instrumentation	22
3.5	Classification of the Method	22
4	Application	23
4.1	Automatic Transmission - EcoLife	23
4.2	Pressure Controller	25
4.3	Instrumentation Set Up	26
5	Tests	29
5.1	General Test	29
5.1.1	Input Instruments - Chooser	29
5.1.2	Output Instruments - Reporter	30
5.1.3	Test Focus	31
5.1.4	Results	31
5.2	Fault Finding	32
5.2.1	Instrumentation and Test Focus	32
5.2.2	Results	32
5.3	Code Coverage Test	33
5.3.1	Input Instruments - Chooser	33
5.3.2	Output Instruments - Reporter	34
5.3.3	Test Set Up and Test Focus	34
5.3.4	Results	34
5.4	Time Consumption	35
6	Conclusions	37
	Bibliography	41

Chapter 1

Introduction

1.1 Background

Testing is one of the most important actions to ensure the quality within the development of software for vehicle controllers. As the number of controllers in a vehicle just increases and the software constantly is getting more complex, automatically testing is essential.

In a modern motor vehicle, more than 85% of the functionality is controlled by software. Moreover, project schedules are getting shorter and the pressure on software development increases [1]. This means, more software has to be tested in a shorter time. With these conditions, an effective and reliable testing process is of great importance.

Today a lot of tests are implemented and evaluated by hand. The expectation on automatic or partly automatic tests are that they could increase the quality of the software even more and make the testing process simpler and faster.

1.2 Problem Description

At ZF Friedrichshafen AG, in the following referred to as ZF, a new software testing tool which systematically simulates a co-simulation of an environment and an application-software model shall be evaluated. The aim is to gain experience with the tool environment, analyse for which system configurations the procedure is purposeful and what kind of faults can be found.

1.3 Purpose

ZF is interested in new testing methods, which can make the testing process at ZF more efficient, more reliable and ensure a higher quality of the software compared to the testing methods that are used today.

1.4 Goal

The goal with this thesis is to be able to make a conclusion whether this new software testing tool could be useful for the software testing process within ZF and in that case how the tool should be used.

1.5 Thesis Outline

After the introduction with the background of the problem and the problem formulation in Chapter 1, an introduction to software testing is given in Chapter 2. Chapter 3 gives an overview of the test environment and describes the considered testing method. Further, Chapter 4 describes two examples studied to evaluate the tool and in Chapter 5, the implemented tests and their results are presented. Finally, in Chapter 6, the conclusions of this master thesis are presented.

Chapter 2

Software Testing

In this chapter, an introduction to software testing is given. In Section 2.1, the software development process and the corresponding test process are described. An overview of different standards used during the development process is given as well. The different test phases in the test process are described more in detail in Section 2.2. In Section 2.3, an overview of different testing methods is given. A more detailed description of different black-box testing techniques follows in Section 2.4. Lastly, a white-box testing technique is presented in Section 2.5.

2.1 Software Development

The software development process can be described by the V-model shown in Figure 2.1.

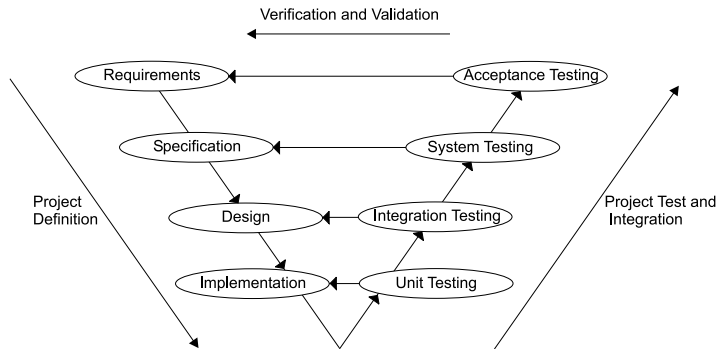


Figure 2.1. The V-model is a commonly used way to describe the software development process.

The descending left-hand arm of the V shows how the task is divided into smaller and smaller parts. On the top are the requirements from the customer and the lowest level is the implementation of the actual software. The ascending right-

hand of the V shows how, step by step, the project is tested. First single modules, then integrated groups of modules (components) and finally the whole system. The model also describes the connection between each level of the requirements and its corresponding test phase [19].

Standards

Various standards have a very important role in the software development. There exist many different types of standards, on the one hand there are various process standards and on the other hand there are different product standards. Standards conveys the uniformity of processes and products and are an important tool for the successful planning of processes, raised quality, lower costs and shorter development time (time to market). Process standards involve evaluation, improvement and determination of the capability of the process. Product standards are for example industry-specific standards regarding safety issues [18].

The comprehensive process standard used within ZF for the software development is Automotive SPICE (**S**oftware **P**rocess **I**mprovement and **C**apability **D**etermination) which is a variant of the standard SPICE adjusted for the automotive industry. SPICE is also known as the ISO/IEC 15504 [6] standard where ISO is the International Organization for Standardization and IEC the International Electrotechnical Commission. Automotive SPICE consists of two components: the process reference model (PRM) and the process assessment model (PAM). PRM mainly consists of general explanations of the model and descriptions of single processes and their purposes and outcomes [9, 8]. PAM is an elaboration of PRM and consists of categorisations of the processes from the PRM but also assessment of the processes. The processes are divided into groups depending on when in the life cycle they are performed and what type of activity they address. For example there are processes used when a supplier delivers products to a customer and organisational processes that help the organisation to reach its business goals. Further, each process is assigned an assessment level describing the capability to perform that certain process. There are six different levels from incomplete processes that are not implemented over established processes to optimizing processes [14, 7].

Another widespread process model is CMMI (**C**apability **M**aturity **M**odel **I**ntegration) [10]. Automotive SPICE focuses much on the software whereas the practices in CMMI are formulated more generally and therefore applicable on many application domains.

An important safety standard which is used in the software development process at ZF is IEC 61508: Functional safety of electrical/electronic/ programmable electronic safety-related systems. This standard describes how to classify systems in different “Safety Integrity Levels” (SIL) based on the probability that a potential hazardous state occurs. It also contains detailed directions which methods and documents that should be used for the different SIL [14].

There exist several other standards within ZF as well. These are for instance MISRA (Motor Industry Software Reliability Association) which is guidelines for the use of the C language in critical systems [9], ZF intern standards and standards of the customers.

2.2 Test Levels

As described in the V-model, see Section 2.1, it is common to divide the test process into the following four test phases: unit testing, integration testing, system testing and acceptance testing.

To illustrate the different levels, the software for an automatic transmission control unit will be studied as an example. The main functionality of this control unit is to control the gear changes. Thereby, the software consists of different units for different tasks, see Figure 2.2. Amongst others, there are pressure controllers for controlling the pressure for different actuators which control the clutches to shift gears. Further, there also exist units for the CAN (Controller Area Network) communication. The communication with other control units takes place by sending messages over CAN.

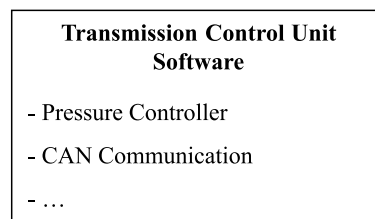


Figure 2.2. The software of a transmission control unit consists of several units, for example pressure controller and CAN communication.

2.2.1 Unit Testing

Unit testing is also called module testing and refers to the testing of single units or modules [17]. The units should be isolated as far as possible and the aim with the tests is to test the basic functionality defined in, for example, the requirement specification [19]. Due to the isolation, found errors can be directly connected to the current unit. It is also desirable to test the robustness, for example the error handling, since the module will be integrated with several other modules. Typical faults that are found at this level are for instance calculational errors or missing or incorrect program paths [17].

For the example in Figure 2.2, unit tests would be tests of the pressure controller separately.

2.2.2 Integration Testing

The next level in the test process is the integration testing where several modules are tested together as a component [19]. The objectives of the tests are the interfacing, integration and interaction of the modules in the component. Typical faults found at this level of testing are for instance errors in the data exchange or in the communication between the different modules in the component [17].

For the example in Figure 2.2, integration tests would be tests of the pressure controller and the CAN communication together. For example the communication between the two units would be tested.

2.2.3 System Testing

The third phase is the system test where the whole system is tested. The test environment should be as similar to the real system environment as possible. The tests can be divided into functional and non-functional tests. The functional tests verify the functionality against the specifications of the system at a high level. The goal of the non-functional tests is to test properties like performance, reliability, flexibility and robustness [17].

For the example in Figure 2.2, system tests would be tests of the complete software for the transmission control software.

2.2.4 Acceptance Testing

The fourth and last level in the test process is the acceptance testing. The main aim with these tests is not to find errors but to provide confidence that the system works correctly from the customer's point of view. Thereby, tests are carried out to validate that the system meets the prescribed requirements. Further, the user acceptance is tested, i.e. if the final user finds the system convenient to use. The tests are often carried out by or together with the customer in the real environment of the customer [17].

For the example in Figure 2.2, acceptance tests would be tests performed together with the customer of the automatic transmission. It would be tested if the customer thinks that the transmission changes the gear at the right moments and if the gears are changed in a comfortable way.

2.3 Testing Methods

There are many methods to test the developed software. Two comprehensive groups of testing are static and dynamic testing respectively.

2.3.1 Static Testing

Static testing is an analysis done without actually executing the software under test. The goal is to find faults and violations of the existing specifications, standards that should be complied or the project planning. This can for example be done by carrying out reviews of the existing documents where these are carefully studied to find such violations. The test can also be to compile the code or letting a person carefully study the code in order to check, for example, the data flow and try to find errors as uninitialized variables, out-of-bounds array access etc [17].

2.3.2 Dynamic Testing

In dynamic testing the software is executed and the outputs are analysed. Dynamic testing can be divided into black-box, white-box and grey-box testing.

Black-box testing: the actual code is not really considered or not accessible for the tester. Instead, only the inputs and the outputs of the system under test are studied. Black-box testing is most appropriate for higher levels of testing like system testing but can also be successfully applied on lower levels like unit testing.

White-box testing: the code is considered as well and the internal flow is also studied during the execution of the test cases. White-box testing methods are most appropriate for the lower levels of testing like for example unit testing and integration testing [17].

Grey-box testing: a combination of black-box and white-box testing, for instance a module structure can be known but not each module in detail [13].

2.4 Black-Box Testing Techniques

In black-box testing where the system under test is considered as a black box with unknown content it would be optimally to create a test for every possible combination of every possible value of the inputs. Of course this is impossible since the number of test cases would be vast, therefore there are several techniques for trying to reduce the number of test cases while maintaining appropriate test coverage.

There exist a great number of different black-box testing techniques and all of them cannot be presented here. In this section, three of the most common methods will be described in more detail [17].

2.4.1 Equivalence Class Testing

Instead of testing all possible input values, the inputs can be divided into equivalence classes. All values which are treated the same way of the system under test belong to the same equivalence class. It is assumed that all test cases in an equivalence class produce the same result. That is, if a test case of an equivalence class results in a fault then every other test case in the same equivalence class also will result in a fault. On the other hand, if the test case is faultless then all other test cases in the equivalence class are assumed to be faultless as well [4]. Therefore it is sufficient to test only one test case of every equivalence class. Not only the valid but also the invalid values should be divided into equivalence classes. An example to illustrate equivalence classes is given in Example 2.1.

The outputs can be divided into equivalence classes as well. However, it is a bit more complicated to determine the test cases since in addition it also has to be considered which input values cause the different outputs [17].

Example 2.1: Example to illustrate equivalence classes.

```

if (x>0 && x<=3) {
    a = a+1; }
elseif (x>3 && x<=5) {
    a = a-1; }

```

Here, `&&` means logical AND. For this example, the values of `x` can be divided into four equivalence classes:

```

x <= 0
0 < x <= 3
3 < x <= 5
x > 5

```

For this small example, it would be enough to create the test cases `x=-1`, `x=1`, `x=4` and `x=6`.

This technique with equivalence classes is most suitable for systems where the variables can be divided into ranges or sets. The advantage of this method is that the test cases can be reduced in an efficient way [4]. A disadvantage is that only single input or output conditions are considered. It is very difficult to take dependencies or interactions between variables into consideration. It would be possible by a further classification of equivalence classes where possible dependencies are considered but it would need a great deal of effort [17].

2.4.2 Boundary Value Testing

Boundary value testing is a very good complement to the equivalence class testing. This technique focuses on the boundaries of the equivalence classes simply because experience has shown that problems in most cases arise at the boundaries. For example conditions might be inaccurate programmed. In the example in Example 2.1, the first condition possibly should be `x < 0` [4].

Boundary value testing is implemented as follows. First of all, the equivalence classes and their boundaries have to be determined. Secondly test cases are created for the value on the boundary, just below the boundary and just above the boundary. In this manner the smallest possible increment should be used in both directions. Exactly what this is depends on the circumstances, for example the data type. If the value just above the boundary coincides with the value just below another boundary, this value naturally results in only one test case. For the example in Example 2.1, the interesting values, assuming `x` is an integer, would be `x = MIN_INT`, `x = MIN_INT+1`, `x = -1`, `x = 0`, `x = 1`, `x = 2`, `x = 3`, `x = 4`, `x = 5`, `x = 6`, `x = MAX_INT-1` and `x = MAX_INT`, where `MIN_INT` and `MAX_INT` are the smallest and largest integer value available on the computer.

A common assumption is that test cases with values in the middle of an equivalence class do not lead to any other results than the test cases with the boundary values of the equivalence class and therefore they can be omitted.

At the first glance, the technique with equivalence classes and boundary value analysis might seem simple. However, in reality the determination of equivalence classes and the proper test values at the boundaries can be rather complex. If the determination is carried out successfully, the combination of equivalence class and boundary value testing is a very effective testing method [17].

2.4.3 State Transition Testing

A disadvantage of the equivalence class and boundary value testing is that variable dependencies are disregarded. A method which takes this in consideration is state transition testing. Often, not only the current inputs but also the previous events in the system are decisive for the current output and the systems behaviour. Consider for example a stack of elements with a specified maximal number of elements. If a new element is put onto the stack, the state can change from empty to partly filled, the state can remain partly filled or the state can change from partly filled to full. Which transition that is taken depends on the previous states, i.e. how many elements that previously have been added to or removed from the stack.

The states and the changes between the states can be described by a so called state diagram. For example, a state diagram for the stack example could look as follows in Figure 2.3.

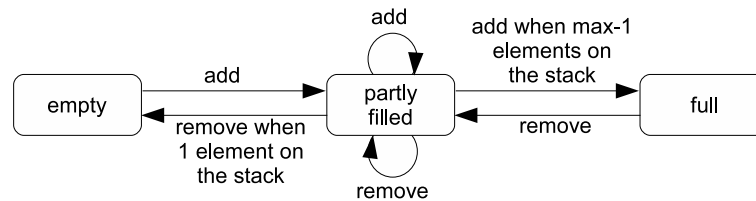


Figure 2.3. A state diagram describing the possible state changes for a stack where elements are added and removed.

Based on the state diagram, test cases can be derived using different strategies. One can be to reach every state at least once. Another can be that every function shall be called at least once. Calling every function does not result in each state being reached. Therefore a third possible strategy is that every possible function in a state shall be called at least once. If the robustness also shall be tested, the functions not defined for the current state can also be called to test how the system is reacting and if infeasible state changes take place. Independent of which strategy that is used, the actual behaviour of the system shall be compared to the specified behaviour for the different test cases [17].

For large and complex systems, the state diagrams fastly get very complicated. An alternative to the state diagram is a state transition table. A transition table basically consists of the same information as the state diagram but in table form. There are several different types of state transition tables. A state transition table can for example be a table which describes the current state, possible events, the actions that follow upon different events and the next state. Unlike state diagrams, a state transition table also allows the listing of invalid state transitions [4].

This method is, of course, most suitable for systems where different states play an important role and a state diagram is an adequate way to describe the system [17].

2.5 Code Coverage

Code coverage is a white-box testing technique where the goal is that every part of the code shall be executed at least once. It is common to give the code coverage as a percentage and this number can be used as a type of quality measure of the implemented tests. At the same time as test cases are designed to cover as much as possible of the code, it is also studied if the outputs of the tests agree with the expected outputs and thereby faults can be detected [17].

With code coverage tests it is possible to discover if some parts of the code have not been executed at all. It can be difficult to achieve 100% code coverage since some parts of the code might be error handling for instance. It can be very hard and expensive, or even impossible, to reach every line of code during the test phase [4].

Code coverage is measured by so called instrumentation of the code where measure statements are inserted at proper places in the code. These statements can for example be counters which are incremented every time the certain code is executed. It would be very time-consuming to carry out this instrumentation by hand and therefore there are many different software tools on the market doing this automatically when the code is compiled [17].

There are several different well known levels of coverage measurements. Statement coverage, decision coverage, condition coverage, decision/condition coverage and multiple condition coverage are some of the most common coverage criteria.

The code segment in Example 2.2 will be used to illustrate the different coverage criteria [4].

— Example 2.2: Example to illustrate different coverage criteria. —

```
if (a>0 && c==1) {  
    x = x+1; }  
if (b==3 || d<0) {  
    y = 0; }
```

Here, `&&` means logical AND and `||` means logical OR. This code segment can also be illustrated as a flowchart. The statements are represented by nodes and the control flow between the statements is represented by edges, see Figure 2.4. The two diamond shaped nodes represent a decision consisting of two conditions.

2.5.1 Statement Coverage

Statement coverage is a coverage criterion which only requires that each statement is executed at least once. In terms of the flowchart this means that every node

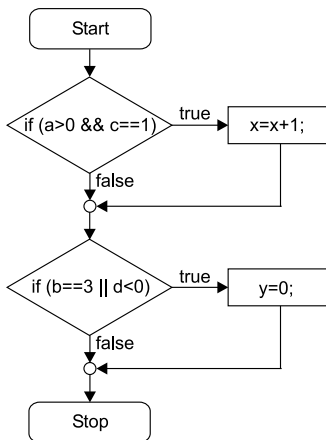


Figure 2.4. A flowchart of the four line code segment. The nodes represent the statements and the edges the control flow between the statements.

has to be passed by at least once.

For the example in Example 2.2, 100% statement coverage can be reached with the single test case

$a=1, c=1, b=3, d=0$.

The path in the flowchart for this test case is drawn in Figure 2.5.

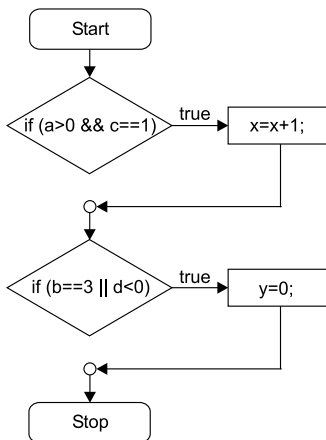


Figure 2.5. For the test case which gives 100% statement coverage, both decisions are evaluated to true.

This is a rather weak criterion where a lot of errors can be missed since every node can be passed leaving a lot of edges unexecuted. However, it is helpful for easily determining if some parts of the software are not being executed at all [17].

2.5.2 Decision Coverage

Decision coverage is also called branch coverage and the criterion is that every decision is evaluated both to true and false at least once. For **case**-statements it is required that all exits should be taken. In terms of the flowchart this means that every edge has to be passed by at least once [17].

For the example in Example 2.2, 100% decision coverage can be reached with the two test cases

$a=1, c=1, b=3, d=0$ and

$a=1, c=0, b=0, d=0$.

In the first test case, both decisions are evaluated to true and in the second test case, both are evaluated to false. The paths in the flowchart for the both test cases are drawn in Figure 2.6.

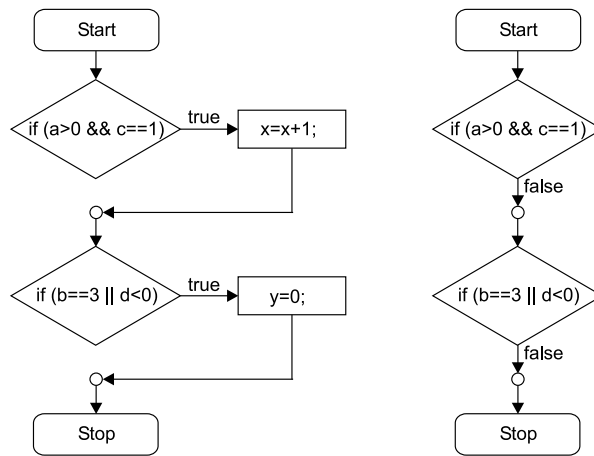


Figure 2.6. In the first test case, both decisions are evaluated to true and in the second test case, both are evaluated to false. These two test cases lead to 100% decision coverage.

2.5.3 Condition Coverage

For condition coverage the criterion is that each condition with a true or false result in a decision is evaluated both to true and false at least once.

For the example in Example 2.2, 100% condition coverage can be reached with the two test cases

$a=1, c=0, b=0, d=-1$ and

$a=0, c=1, b=3, d=0$.

The results of the logical expressions for the both test cases are presented in Table 2.1.

Notice that decision coverage does not follow from condition coverage. For the two test cases above, the first decision is never evaluated to true and the second one is never evaluated to false.

Table 2.1. The results of the logical expressions for the four test cases leading to 100% condition coverage.

a>0	c==1	a>0 && c==1	b==3	d<0	b==3 d<0
true	false	false	false	true	true
false	true	false	true	false	true

Condition coverage is usually a stronger criterion than decision coverage since every condition is evaluated at least once whereas 100% decision coverage can be reached without evaluating each condition to true and false [4].

2.5.4 Decision/Condition Coverage

An even more complete coverage level is the decision/condition coverage where decision coverage and condition coverage are combined. That is, test cases are created so that every decision in a condition as well as every condition is evaluated to both true and false.

For the example in Example 2.2, 100% decision/condition coverage can be reached with the two test cases

a=1, c=1, b=3, d=-1 and
a=0, c=0, b=0, d=0.

The results of the logical expressions for the both test cases are presented in Table 2.2.

Obviously, both decision coverage and condition coverage follows from decision/condition coverage [4].

Table 2.2. The results of the logical expressions for the four test cases leading to 100% decision/condition coverage.

a>0	c==1	a>0 && c==1	b==3	d<0	b==3 d<0
true	true	true	true	true	true
false	false	false	false	false	false

2.5.5 Multiple Condition Coverage

Multiple condition coverage resembles condition coverage with the difference that every combination of outcomes from decisions in a condition is tested as well.

For the example in Example 2.2, 100% multiple condition coverage can be reached with the four test cases

a=1, c=1, b=3, d=-1,
a=0, c=0, b=0, d=0,
a=0, c=1, b=3, d=0 and
a=1, c=0, b=0, d=-1.

The results of the logical expressions for the four test cases are presented in Table 2.3.

Table 2.3. The results of the logical expressions for the four test cases leading to 100% multiple condition coverage.

a>0	c==1	a>0 && c==1	b==3	d<0	b==3 d<0
true	true	true	true	true	true
false	false	false	false	false	false
false	true	false	true	false	true
true	false	false	false	true	true

Achieving 100% multiple condition coverage, also means achieving decision/condition coverage, condition coverage and decision coverage. This is an extensive criterion but it is also very intricate since the possible combinations of n conditions are 2^n in total and hence the number of test cases increases fast with the number of conditions in a decision [17].

Chapter 3

Test Environment

In this Chapter, an overview of the test environment and a description of the considered testing tool are given. In Section 3.1, the structure of the system under test is described. The instruments that are needed for the communication between the test tool and the system under test are described in Section 3.2. In Section 3.3, the testing method used by the considered testing tool is described. How the test focus can be influenced is discussed in Section 3.4. Lastly, in Section 3.5, the testing method is classified in terms of the different testing methods described in Chapter 2.

Unless anything else is said, the information in this chapter is taken from the user guide of the considered testing tool TestWeaver¹.

3.1 System Overview

TestWeaver is the testing tool to be considered in this thesis. It systematically simulates a co-simulation and thereby automatically generates test cases. A co-simulation is a system consisting of several different modules which are simulated together. These modules can consist of C++-code, C-code, Matlab/Simulink models and Dymola models, see Figure 3.1. For an automatic transmission the modules can for example be the software of the automatic transmission, a model describing the vehicle and the environment respectively. For the co-simulation, the tool Softcar is used. This is a tool developed and used within ZF. A simulation of this type is called software in the loop since the software is run on a computer and not on the real control unit and the vehicle and the rest of the environment is also simulated on the computer. The communication between TestWeaver and the co-simulation is realised with so called instruments specifying the inputs and outputs for TestWeaver.

¹TestWeaver 1.0 User Guide, QTronic GmbH.

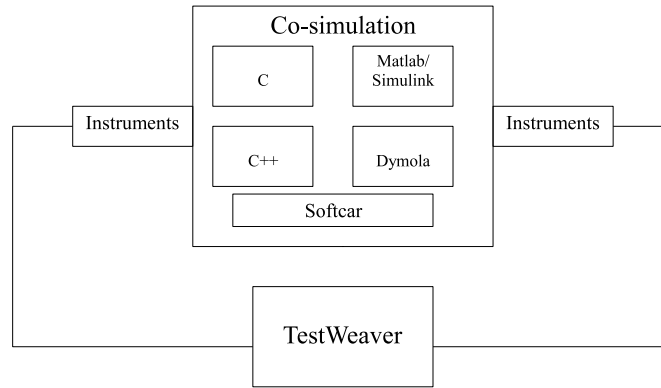


Figure 3.1. The test environment consists of a co-simulation of several modules. The communication between the co-simulation and the testing tool TestWeaver is realised with so called instruments.

3.2 Instruments

There are instruments for controlling the inputs and observing outputs. The instruments for the inputs are called choosers in TestWeaver and those for the outputs are called reporters. The different instruments can either be programmed by hand in C or C++ but there are also libraries available for inserting them directly in Matlab/Simulink or Modelica, see Figure 3.2.

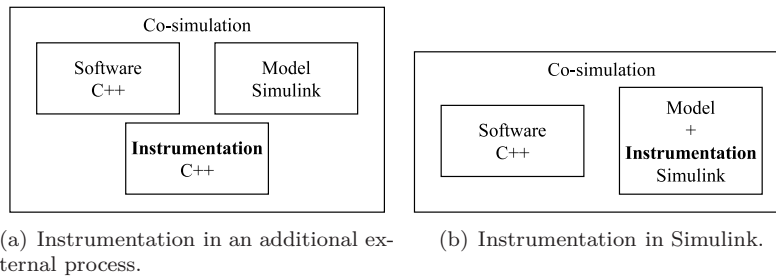


Figure 3.2. There are two different possibilities to instrument the system under test.

There are two different types of input variables. They can be regular input variables of the system under test but there is also a possibility to define faults. This makes it possible to simulate the appearance of a fault during the execution. Both the inputs and the faults are defined as discrete values. The insertion of faults requires that there are components in the system under test with fault modes. The classification of the inputs takes place by specifying an occurrence from one to ten for each value of the variable. Occurrence one means a fault occurring very rarely, nine a fault occurring very often and ten means no fault. A variable with all values classified to occurrence ten is seen as a regular input and every variable with at

least one value with an occurrence smaller than ten is seen as a fault. For both types of inputs, one of the discrete values is defined to be the default value of that variable. Every input is also associated with a trigger determining when an input or a fault can be changed. The trigger can for example be a periodically signal enabling a change of the variable twice every second.

There are also two different types of output variables. They can be state variables or alarm variables. The values of the output variables are divided into intervals and every interval is classified with a severity. The possible severities are the values zero to eleven and twenty where zero means no fault. A severity larger than zero represents different degrees of errors and how serious they are. An output variable with all intervals classified to severity zero is a state variable and the variables with at least one interval classified with a severity greater than zero is seen as an alarm variable. Exactly as the input variables is every output instrument associated with a trigger determining when an output value should be noted [12].

The different types of variables can be illustrated with the following simple example. Consider a tank which is filled with water from a tap, see Figure 3.3.

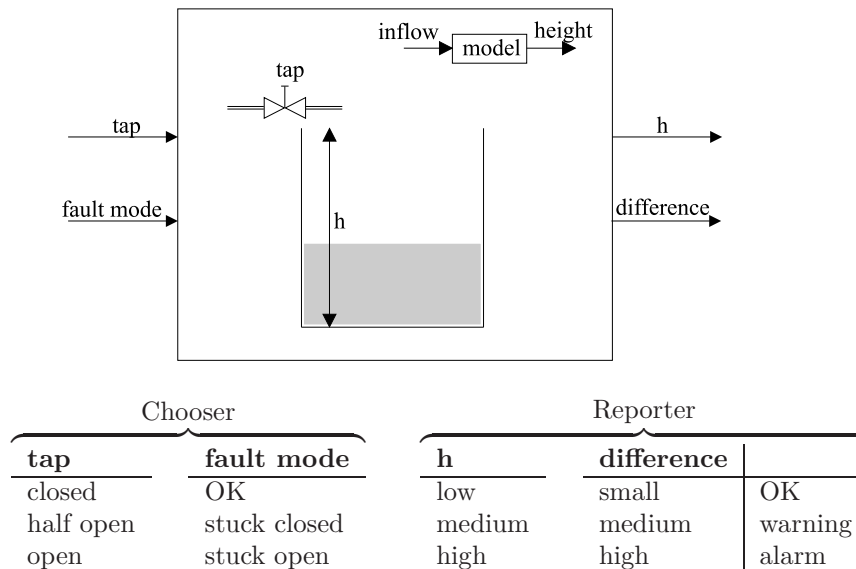


Figure 3.3. A simple tank example illustrating the different instruments.

An input variable is how open the tap should be, the possible values are closed, half open and open. The tap also has the fault modes ok, stuck closed and stuck open. Stuck closed means that despite of the input, the tap stays closed and stuck open means it is always open. These different fault modes can be inserted with a fault variable. In the tank a sensor measuring the current water height h is placed. The output from this sensor is a state variable where the possible values of the height are divided into the three ranges low, medium and high. There

is also a model calculating the height out of the required inflow into the tank. The difference between the measured value and the calculated value is an error variable. A difference means that the tap is stuck open or stuck closed or the sensor is defective. The possible values of the difference are also divided into three ranges. A very small difference is no problem and has the severity zero; a bit larger difference is set to a warning and large differences report an error.

3.3 Method

The goal of TestWeaver is to reach a faulty state, that is a state where a variable with severity greater than zero is reported. This is done by a strategic variation of the variables. The inputs and outputs and their discrete values span an n -dimensional discrete state space. In two dimensions the state space can be visualized as a grid where each square represents a state. In the discrete state space, some states have been reached and some have not and out of the reached states, some have caused an alarm, see Figure 3.4. TestWeaver tries to maximize the coverage of this discrete state space, i.e. to maximize the number of reached squares.

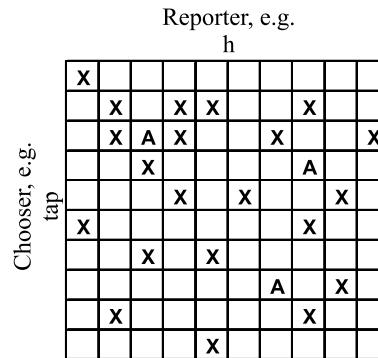


Figure 3.4. A discrete state space in two dimensions. Reached states are marked with an X and alarm states with an A.

The different generated test cases are called scenarios. All test cases have the same length (in time), predefined by the user. There are two strategies for generating scenarios: a default strategy and a random strategy. In the random strategy, the inputs are set to the different possible values randomly and the values are changed very often. The default strategy on the other hand is a strategically and systematically way of generating scenarios. In the first scenario, all inputs are set to their default values and kept to these values the whole scenario. State changes, i.e. when the outputs changes are noted as interesting points. The strategy is to change the inputs at these points where a state change takes place. First, only one input is changed to another value and one by one are all inputs changed to all their possible values. The scenario generation can now continue with either changing one input at another state change or generating scenarios

where an additional input is changed at another state change. Every change of an input means a branching point for the simulation and step by step a tree arises, or more precisely a directed graph, see Figure 3.5. A directed graph is a graph where the edges between the nodes have a direction which is indicated with an arrow [11].

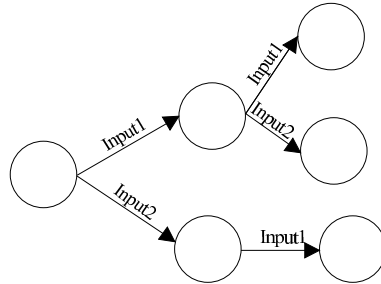


Figure 3.5. A directed graph has directed edges between the nodes.

Returning to the tank example, the default values might be tap half open and fault mode ok. In the first scenario, with the default values, the height changes from low to medium at the time t_1 and from medium to high at the time t_2 . In some later scenario, the tap changes from half open to open at the time t_1 , then the height changes from medium to high at the time t_3 ($<t_2$). In another scenario, the change from half open to open takes place at the time t_2 instead. In some later scenario, the tap is changed from half open to open at the time t_1 followed by the insertion of the fault mode tap stuck open at the time t_3 . The different scenarios are illustrated as timelines in Figure 3.6. There are also scenarios created with all the other possible input values at the different points in time. The scenario generation can also be illustrated as a directed graph, see Figure 3.7. The scenarios are generated in a deterministic way. If the scenarios are deleted and generated again without any changes in the settings, exactly the same scenarios will be generated and in the same order as previous.

Increasing the number of input instruments, or having input instruments with a lot of possible values, rapidly increases the possibilities for varying the inputs, i.e. the possible branches in the directed graph. It would take very long time to examine all these branches and therefore it is preferable to keep the number of instruments and the number of values for each instrument rather low. Usually a system under test has four to six input instruments and twenty to thirty output instruments.

If the default or the random strategy should be used for generating the test cases depends of the system under test and how the variables vary in the real environment. Often the default strategy is more appropriate, for example a car is normally not driven in a random way where inputs like accelerating pedal and brake pedal are changed every second. However, there are system where changing the inputs randomly is a proper strategy.

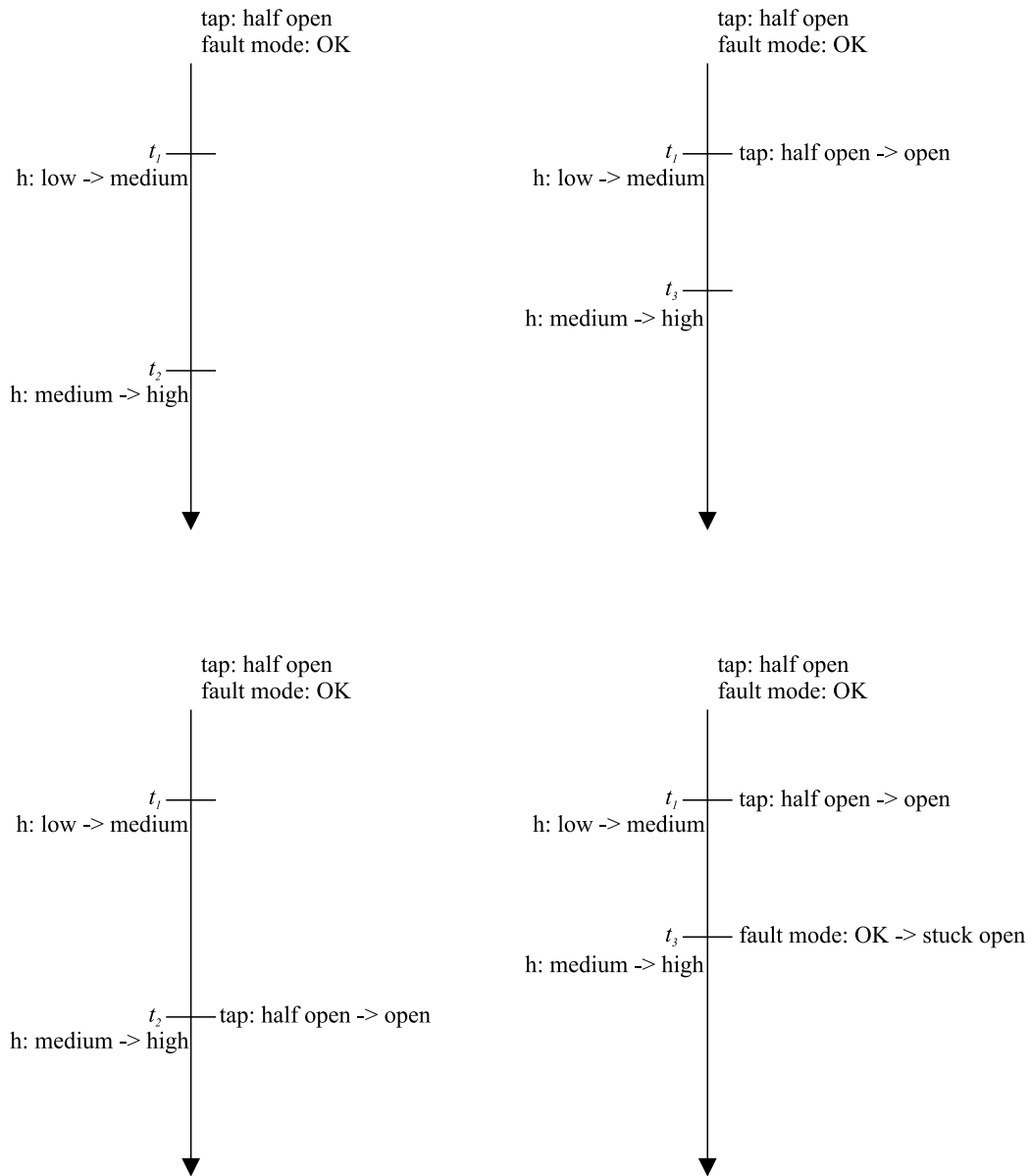


Figure 3.6. Each timeline represents one scenario. The inputs at the beginning of the scenario can be seen at the top of the timeline. Changes in a reported variable stands to the left of the line and changes in the inputs to the right. It can be seen that the inputs are changed when a state change, i.e. a change in a reported variable, takes place.

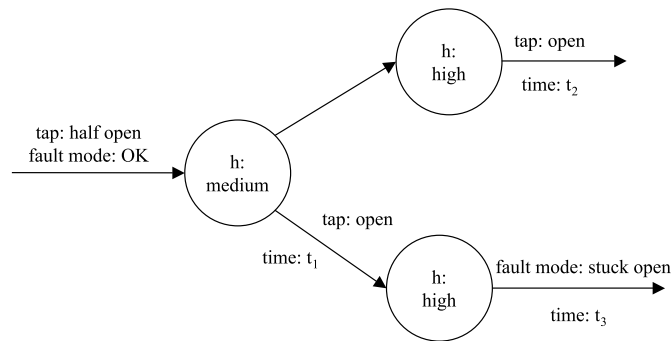


Figure 3.7. The scenario generation for the tank example as a directed graph. Each node represents a state and the changes in the inputs can be seen at the edges.

3.4 Test Focus

The test focus can be influenced in several ways. Which these are and what impact they have on the generated scenarios will be described in this section.

3.4.1 Coverage Report

The so called coverage reports are state tables defined by the tester. Each column in the table represents a state variable and the table itself describes which states have been reached in which test cases. This state table defines which state changes that shall be considered as important. That is, at which state changes the inputs shall be varied. TestWeaver changes the inputs at the state changes that are represented in the coverage report. Consider the tank example in Figure 3.3 once more. If the coverage report for this example only consist of one column; the height, the inputs will be changed when the height switches range, e.g. from low to medium. If, instead, the coverage report would have the column “difference”, the inputs will be changed when the difference switches range, e.g. from small to medium. Lastly, if the coverage report consists of two columns; height and difference, the inputs will be changed when a state change takes place either in the height or in the difference.

TestWeaver also considers these state tables as a sort of priority list and tries to drive the system under test into states that are reported in the coverage report. Thereby the first column has the highest priority and TestWeaver foremost tries to maximize the number of entries in this column, the second column has the next highest priority and so on.

3.4.2 Constraints

There is the possibility to add constraints in the form of logical expressions to the system under test in order to limit the permissible state space. For example

futile or unrealistic test cases or certain states which cannot be handled by the model can be omitted. An unrealistic scenario which is desirable to avoid could be accelerating and braking simultaneously in a car. Scenarios that violate the constraints are either terminated or not generated at all, depending on how good TestWeaver can handle the constraints. However, sometimes it can also be of interest to study what happens in these futile or unrealistic test cases.

3.4.3 Instrumentation

When the instruments are programmed in C/C++, also the instrument sequence in the file can have an impact on the scenario generation. If the function for changing the inflow in the tank is programmed before the function for changing the fault mode, scenarios where the inflow is varied are generated before scenarios where the fault mode is varied. However, the order of the instruments does not matter if the test runs long enough because in the long run, the coverage reports and constraints are more important.

3.5 Classification of the Method

Since the system under test is executed when generating the test cases, dynamic testing is performed with TestWeaver. The instrumented variables are the only information TestWeaver has about the system, i.e. the actual code is unknown, and therefore the method is a black-box testing method, see Section 2.3.2.

Both input and output variables are divided into equivalence classes. A number of representatives (discrete values) are chosen for each input variable. The output variables are divided into different ranges but it is not really necessary to consider which input values lead to which output values when the partitioning is performed. It is definitely possible to choose the discrete values for the inputs and the partition of the output variables without considering equivalence classes. There is also the possibility to do the instrumentation in a more formal manner where equivalence classes are determined and based on the outcome; the discrete inputs and the output partitions are chosen. Thereby, the boundary values can be considered as well and boundary value testing can be carried out by choosing the discrete inputs to the boundary values.

The method is also a type of testing based on state transitions since state changes are considered and the progress in the state space is recorded and remembered by TestWeaver. However, it should be noticed that the states considered by TestWeaver are a little different than the states in a state diagram or a state transition table. The state changes considered by TestWeaver can be discrete state changes (for example from partly filled to full in Figure 2.3) but it can also be changes of physical values; that the number of elements on the stack changes from three to four (but the stack is still partly filled). The goal of the method is to cover as much of the state space defined by the instrumented variables as possible, especially the parts of the state space specified in the so called coverage reports.

Chapter 4

Application

To evaluate the testing tool TestWeaver, mainly two different examples were studied: the software of an automatic transmission control unit and a module of this automatic transmission. This module is responsible for the pressure control of the actuators inside the transmission. In this chapter, an introduction to these two examples is given. Firstly, in Section 4.1, the example of the automatic transmission control unit is described. A short introduction to gear changes in an automatic transmission is given as well. In Section 4.2, a description of the pressure controller and its functionality follows. Lastly, in Section 4.3, the different possibilities to instrument these both examples are discussed and the chosen alternative is justified.

Unless anything else is said, the information in this chapter is taken from [5].

4.1 Automatic Transmission - EcoLife

The example studied to evaluate TestWeaver is a real life example of a ZF six gear automatic transmission for buses called EcoLife, see Figure 4.1. The test object is the software of the transmission control unit (TCU) which has the main task to decide how and when to change the gear.

The system under test consists of three external processes running under the ZF co-simulation tool Softcar: the software for the automatic transmission, the model of the vehicle and the shift lever. The software is the real software which is used in the TCU. The model describes the rest of the vehicle, for example the motor and the hydraulic. The shift lever is a separate process since there are various shift levers for different buses. There is also a graphical user interface in Softcar belonging to the example in order to visualize the variables and simplify the understanding of the progress of the test.

The simulation is software in the loop simulation and the preformed tests are system tests since the automatic transmission is considered as whole and not only single components of the system.



Figure 4.1. The ZF six gear automatic transmission EcoLife, [3].

Gear Changes in an Automatic Transmission

In an automatic transmission, the different gears are realised with a planetary gear set which consists of a sun gear (1), a planet carrier (2) with planet gears (3) and a ring gear (4), see Figure 4.2. By holding one of these three parts fixed, different gear ratios can be realised. Thereby, one of the remaining two parts is driven by an input torque and the third part of the planetary gear provides the output torque. For example, a large reduction in torque is obtained by holding the ring gear fix and using the sun gear as input and the planet carrier as output. To provide four, five or six forward gears and one reverse gear, one planetary gear set is not enough. Instead, two, three or four planetary gear sets are connected to each other to enable several different gear ratios.

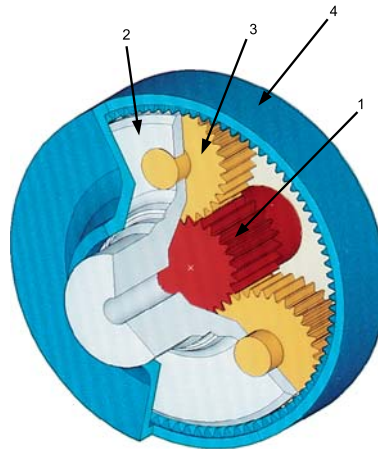


Figure 4.2. A planetary gear with a sun gear (1), a planet carrier (2) with planet gears (3) and a ring gear (4) [5].

One main difference between manual and automatic transmissions is that the gears are changed without interruption of the force flow in automatic transmissions. This is achieved by electronic-hydraulic actuated clutches and bands which connect different transmission elements (different parts in the planetary gear sets). Thereby, the torque is transferred between the elements and the engine speed is automatically adjusted to the new gear. Hence, a gear change can take place without disconnecting the motor from the transmission and without that the driver has to release the accelerating pedal.

The gear changes are controlled by the TCU which also makes sure that the gear changes take place in a comfortable way. Input signals to the control unit are, among others, the vehicle speed, engine speed, engine torque, throttle position, wheel speeds and shift lever position. Output signals from the TCU are signals for the electronic-hydraulic actuators mentioned above to control the gear changes, signals for the instrument panel (current gear, shift lever position etc.) and signals with information for the diagnostic system which saves possible errors in a memory, see Figure 4.3. More information about automatic transmissions and their functionality can be found in [5].

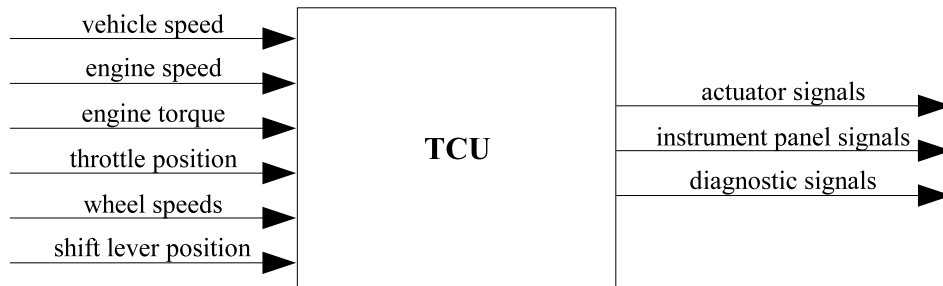


Figure 4.3. The TCU takes several inputs and depending on these decides whether a gear change should take place or not.

4.2 Pressure Controller

Since the automatic transmission EcoLife is a very large and complex example, it is not appropriate for all type of tests. Therefore, a module of the software responsible for handling the pressure control, was taken as another example. The clutches and bands that control the transmission elements are actuated with a hydraulic pressure. The pressure in turn is changed by an electronic actuated valve. For the comfortable gear change it is very important that the pressure is adjusted in an accurate way. Therefore, the pressure for the clutches and bands is controlled. The input to the pressure controller module of the software is a target pressure from the rest of the controller software. The pressure controller module transforms this target pressure into a current which is forwarded to the actuators by the control unit. The actuators are the valves that adjusts the pressure of the clutches and bands.

The software of the pressure controller is the implementation of a finite state machine. This state machine describes the different states of the pressure controller and which events that are required for a state change, the principle is shown in Figure 4.4. This figure only shows some of the states of the pressure controller. The controller can be switched on and off and it can be changed between open and closed loop control. There is also a state where the pressure is limited to an upper or lower limit [5, 16].

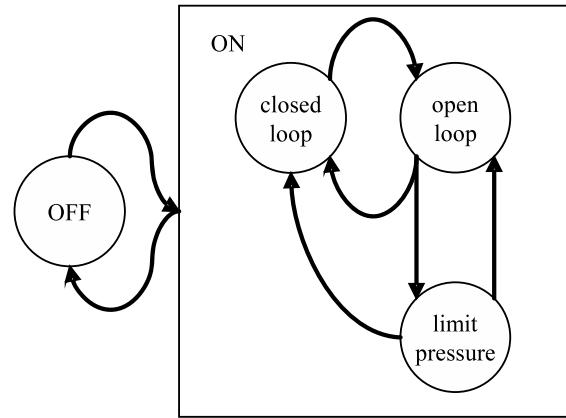


Figure 4.4. The software of the pressure controller is the implementation of a state machine. Notice that this figure does not show all states of the controller.

This simulation is software in the loop simulation as well but the preformed tests are unit tests since the pressure controller does not represent a whole system but a module of the automatic transmission software.

4.3 Instrumentation Set Up

The instrumentation defines the communication between the co-simulation environment and TestWeaver. It can take place either in an additional external process programmed in C or C++ or the instruments can be added directly in the Simulink model, see Section 3.2. For the two considered examples, the instruments were programmed in C++ in an additional external process.

One advantage with the chosen method is that no changes have to be done to the actual project since an additional process is created independently. The original files do not have to be changed or rebuild, only the new C++ project with the instruments and the communication between TestWeaver and the co-simulation tool Softcar has to be rebuild. This project is rather small and easy and fast to rebuild. Another advantage is that the same instrumentation can be used on different models and software versions (as long as the variable names are the same) without any changes to the file. One disadvantage is that the tester has to program the instruments by hand while in Simulink, assistance is given by a

graphical user interface. However, there are not particularly many functions that have to be added and when the structure is known it is not that complicated.

An additional argument to choose the first variant with the instruments in C/C++ is that since the software is the object to test, the interesting variables to report are not model variables but software variables, see Figure 4.5. These are not available in the model and the reporters therefore have to be designed directly in C/C++.

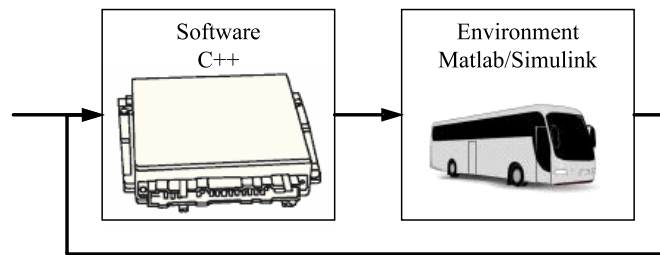


Figure 4.5. It is an advantage to program the instruments in an additional external process since this allows the instrumentation of variables of the software as well.

The variables that are desirable to report can mainly be of two kinds. They can be the values of a signal directly, but a common case is when not the variables directly are of interest but logical conditions based on the values of the variables. For example, a difference of two variables can be of interest (as in the tank example, see Section 3.2). Another example is that an output variable can be an error or not, depending on the current input variables. Such conditions can easily be programmed in the C/C++ project and it is also possible to program them in Matlab/Simulink. A variable depending on the result of these conditions can then be reported. However, the alternative with Matlab/Simulink is only possible if all needed variables are available in the model and no variables that are only accessible in the software are required for the logical condition. If a C/C++ project is needed for the reporters anyhow, the choosers might be designed there as well.

For the two considered examples, variables of the software were of interest and it was desirable not to have to change the present environment model. Therefore, the alternative with the additional external process was chosen.

Chapter 5

Tests

In this chapter, the performed tests on the both examples are described. Mainly, three different tests were carried out. In Section 5.1 a general test performed on the EcoLife example is described. The aim with this test is to see if any known or unknown errors could be found. In Section 5.2 a test on a faulty version of the software for the EcoLife transmission is presented. The goal with this test is to examine if and how fast TestWeaver could find this fault. The last test is described in Section 5.3. This test is performed on the pressure controller with the aim to look into the reached code coverage when testing with TestWeaver. For each test, the instrumentation, test focus and results of the test are presented. Finally, in Section 5.4 the time needed for performing the tests with TestWeaver is discussed.

5.1 General Test

A general test was performed on the EcoLife example, see Section 4.1, with the goal to see if any errors (known or unknown) could be found.

5.1.1 Input Instruments - Chooser

The natural inputs of this system are the variables that the driver can change such as the accelerator pedal, the brake pedal and the shift lever. Besides these, it is also possible to specify the road inclination during the simulation. The input variables chosen for this system are the following variables which all are inputs to the model:

shift lever with the values R, N and D, where R means reverse gear, N is neutral with no gear selected and D is drive which is the mode where the vehicle can move forward and the gear can be changed. The shift lever also has the possible modes D1, D2 and D3 where the gear is restricted to gear 1, 2 and 3 respectively. These modes are omitted to limit the state space, see Section 3.3.

accelerator pedal with the values 0%, 50% and 100%. With 0% the pedal is not actuated at all and 100% means that the pedal is depressed maximally.

brake pedal with the values 0 N, 10000 N and 20000 N. These forces can be illustrated on the hand of Example 5.1.

Example 5.1

Consider a bus with the weight 21000 kg (Mercedes CapaCity with the fore-runner transmission of EcoLife has an empty weight of about 18500 kg [2]) and the velocity 50 km/h. With the formulas $F = ma$, $v = at$ and $s = 0.5at^2$, it can be calculated that a braking force of 10000 N ideally and very simplified (neglecting rolling resistance and air resistance) corresponds to a braking distance of about 200 metres for this bus. In the same way it can be calculated that a braking force of 20000 N corresponds to a braking distance of about 100 metres.

In the formulas above, the notation stands for a : acceleration [m/s^2], F : force [N], m : mass [kg], s : distance [m], t : time [s] and v : velocity [m/s].

road inclination with the values -10%, 0% and 10%. The road inclination a in percent is the ratio of the height x to the distance y , see Figure 5.1. That is, an inclination of 10% is for example reached with the height 10 meter and the distance 100 meter.

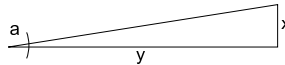


Figure 5.1. The road inclination a in percent is the ratio of the height x to the distance y .

The reason for only defining three discrete values for each input variable is, again, for limiting the state space. The values were chosen to be one extreme low value, one extreme high value and one value in between.

5.1.2 Output Instruments - Reporter

Since the system under test is the software for an automatic transmission, natural important courses of events are the gear changes. The gear changes can be observed by studying the current and the target gear. If these are different, a gear change is about to take place. It can also be of interest to consider the gear change type and the current velocity. Exactly these four variables are chosen to be state reporters:

current gear which is a discrete variable that reports the current gear, for instance neutral, gear one, gear two etc.

target gear which is a discrete variable that reports the target gear, for instance neutral, gear one, gear two etc.

gear change type which for example can be increasing or decreasing the gear. The gear can also be changed in different ways depending on how the pressures actuating the clutches and bands proceeds. Either the pressure can be changed as a step or as a ramp. The gear change type is described as a discrete variable as well.

vehicle velocity which is divided into intervals of 20 km/h, i.e. the velocity is reported to be 0-20 km/h, 20-40 km/h etc.

As alarm reporters, two diagnosis variables are chosen:

error number which is an integer value set in every cycle to describe the current error, the value 0 means no error.

error type which is an integer value as well and contains additional information about the error and again the value 0 means no error.

5.1.3 Test Focus

Since the gear changes are of interest, this has to be told TestWeaver and it is done by a coverage report, see Section 3.4.1. If the report contains one column for the current gear and one for the target gear, TestWeaver tries to maximize the number of entries in the table, i.e. tries to reach as many combinations of current and target gears as possible which means as many gear changes as possible. Another consequence of defining the coverage report as just described is that the inputs will be changed just as gear changes take place.

5.1.4 Results

The default strategy was chosen for generating the test cases since this resembles the normal operation of a bus. It is not normal to drive a bus in a randomly way and constantly changing the chosen inputs.

It was early found that a too inclined road brought an error since the bus gradually went arbitrary fast. This was a known error and depends on instability in the model; the model cannot handle very steep roads. As a result of this, the discrete values defined for the road inclination were changed to -5%, 0% and 5%.

With the new values for the road inclination, TestWeaver was run for a long time and generated numerous test cases. An error was reported and it turned out that it was a parameterisation error. The software contains several parameters that enable the adaptation of the software to many different vehicles, engines etc. The tests are carried out with a standard set of parameters and one of these was set to an incorrect value. Parameterisation errors are not as serious as errors in the actual code of the software. However, the parameterisation is important as well and if an error of this type is found, the parameter is normally adjusted. This problem had been noticed and was already adjusted.

The tested software was known to be of very good quality and no serious errors were expected. This test confirms the quality of the software since no unknown problems were found and the test also shows that TestWeaver is able to detect small problems, like parameterisation errors, in a software of good quality.

5.2 Fault Finding

To test the fault finding properties of TestWeaver, the EcoLife example but with a software version known to be faulty was instrumented and tested.

5.2.1 Instrumentation and Test Focus

The fault was inserted by software developers and was not known when testing the example. However, it was given which variables had to be changed to enter the faulty state. Exactly these variables were set to be the input variables:

shift lever with the values R, N and D.

accelerator pedal with the values 0%, 50% and 100%.

brake pedal with the values 0 N, 10000 N and 20000 N.

ABS (antilock braking system) with the values 0 and 1 where 0 means deactivated and 1 activated.

The output instruments, both those reporting states and those reporting alarms, were chosen exactly in the same way as in the test described in Section 5.1. Also the coverage report, i.e. the test focus, was exactly the same and again, the default generation strategy was used.

5.2.2 Results

The faulty state was reached by accelerating to the fourth gear with ABS deactivated. When in the fourth gear, the accelerator pedal was completely released (accelerator pedal = 0%). Finally, during the gearshift from the fourth to the third gear, the ABS was activated. The error was found already after about one hour of testing and about 100 test cases.

There are two main reasons for finding this error at all. The first reason is that it was known that the ABS should be one of the input variables that are varied by TestWeaver. It is not trivial to conclude that the ABS should be activated and deactivated to trigger the fault and without this information, it would probably have been very difficult to find the inserted fault. The second reason is that the test focus was chosen to be the gearshifts. Since the coverage report consists of the current and the target gear, the input variables were changed when the current or the target gear changed. This fault only occurs when the variables are changed exactly as described above and the ABS has to be activated during the gear change and this is achieved with the chosen test focus. Since the system under test is an automatic transmission, it is natural to include the gearshifts in the test focus.

5.3 Code Coverage Test

With the aim to be able to make a statement according to how much of the software is actually tested when TestWeaver is used, a code coverage test was carried out. Since the EcoLife example is rather big and complex the decision to only consider a smaller part of the example was made. The chosen part is the pressure controller described in Section 4.2. One advantage with choosing this example is that code coverage tests for different testing methods have been implemented on this controller. Therefore, the pressure controller example provides the possibility to compare TestWeaver to other testing methods.

The code coverage test was implemented through the tool Testwell CTC++, in the following referred to as CTC. The tool instruments the code by adding some additional statements to the source files and thereby enables CTC to recognize if and how often different parts of the code have been executed. The test coverage can be chosen to be measured according to different criteria like for example function coverage (tests if all functions have been called at least once), decision coverage and condition coverage. For this code coverage test, the decision coverage option was chosen, see Section 2.5.2. In addition to measure if all decisions have been evaluated to both true and false, CTC also considers if all functions have been entered at least once, if all branches in switch-statements and each explicit control transfer (goto, break, continue, return etc.) has been taken [15]. The decision coverage option was chosen because this criterion had been used for measuring the code coverage when using other testing methods.

CTC provides the user with a report describing the coverage of each function expressed as a percentage and there is also the possibility to study exactly which parts of the functions have been executed and which have not.

The difficulty with applying TestWeaver to a system like the pressure controller is that there is no simple external stimulation of just a few variables that makes the controller to enter almost every possible state which would give large code coverage. It is not enough to switch the controller on and vary the target pressure. This only leads to that the states ON and OFF are reached, see Figure 4.4, but it does not mean that the other states within ON are automatically reached. Consequently, only a code coverage of 35% is achieved and as mentioned, several states are not reached at all. This in turn means that many parts of the code are not executed. In order to increase the percentage of executed code the user has to give TestWeaver the possibility to vary several variables.

5.3.1 Input Instruments - Chooser

As mentioned above, the possibility to change several variables has to be provided. Therefore, the following variables were chosen as input instruments:

active with the values 0 and 1 where 1 means that the controller is activated and 0 that it is deactivated.

mode with the values 0 and 1 where 0 means open loop control and 1 closed loop control.

reference pressure with the values 10000 mbar, 20000 mbar and 30000 mbar.

This is the input to the controller and is the desired pressure for the clutch or the band belonging to the current pressure controller.

main pressure which is the available pressure and consequently this pressure also defines an upper limit for the pressure controlled by the pressure controller. The values 18000 mbar and 25000 mbar are defined for the main pressure.

As in the example with the automatic transmission, the number of values for each variable is kept low to limit the state space.

5.3.2 Output Instruments - Reporter

Two state reporters have been chosen for this example:

state which is a discrete variable describing the current state of the pressure controller. Different states are, among others, deactivated, open loop control, closed loop control and pressure limited by main pressure.

actual pressure which should have the same value as the reference pressure. However, the pressure can be limited by a defined minimal value, a defined maximal value or the main pressure and in these cases; the actual value takes the limited value as current value. The values of the actual pressure are divided into three ranges and these ranges are reported.

5.3.3 Test Set Up and Test Focus

Tests with two different set ups were carried out. In both cases, the pressure controller was initialized with a defined minimal pressure and with a limitation on the gradient of the pressure change activated. In one set up, a maximal pressure was defined as well. In the other set up this upper limit was omitted.

The two available state reporters were chosen for the coverage report.

5.3.4 Results

After just about 200 generated test cases for each test set up, a coverage that is 4 percentage points under the coverage reached with a testing method developed within ZF, has been reached. This method in turn reaches a coverage that is 7 percentage points under the coverage obtained when the pressure controller was tested manually. One reason that the coverage obtained when testing with TestWeaver is lower than the coverage reached with the other two methods is that with the instrumentation described above, not even all states are reached. For example there is a state where the pressure controller is deactivated automatically because the pressure has had a very low value for a long time. However, this state is never reached when a minimal pressure is defined for the controller. There is also a possibility to insert an error in the controller which leads to other states.

With no doubt, the code coverage when testing the pressure controller with TestWeaver could be increased by adding more choosers or by changing the basic conditions, for example by removing the minimum pressure limit. The only information that TestWeaver has about the system under test is the information given in the instruments and therefore the degree of coverage depends very much on the instrumentation. TestWeaver can only test the regions of the software that are made accessible, i.e. that are covered by the chosen instrumentation. To be able to increase the code coverage, very good knowledge of the system under test is required since only then it is possible to change the instrumentation in a successful way.

It can be seen that a very good coverage is reached for the parts of the system that is defined to TestWeaver, i.e. instrumented. Therefore, it can be assumed that tests with TestWeaver have a good coverage of the state space defined in the test focus, i.e. the coverage report.

This example also shows that TestWeaver is more suitable for system tests than for unit tests. One difficulty by unit tests is, like previous mentioned, that it is difficult to choose just a few input instruments that excite the whole system under test sufficiently. It is as shown, possible to carry out unit tests with TestWeaver as well but the recommended area of application is system testing.

5.4 Time Consumption

The time needed for the test mainly depends on how fast or slow the original setup without the instrumentation is. It does not make any particular difference if just a few or several instruments are added or if a number of constraints are defined or if the test runs completely without constraints. The only difference that the constraints could make is that if a scenario violates the constraints it can be terminated. This results in that it takes a longer time to generate the same number of complete scenarios since the start and stop of a scenario take some time.

Obviously, more instruments or more intervals for each instrument make the current state space, that shall be searched, larger. This might lead to that it takes a longer time for TestWeaver to reach a part of the state space where errors occur.

It is impossible to make a statement about how long it takes for TestWeaver to find an error. It depends on the current system under test, the instrumentation, the test focus and other settings of the test and of course, the error itself. However, if tests are executed on the same type of system several times, experience can be gained about which coverage that can be reached with a particular instrumentation in which time. This gives information about which type of faults that could be expected to be found in a certain time; the faults in the set of reached states.

Chapter 6

Conclusions

TestWeaver is a tool for automated software in the loop tests. The greatest advantage of the tool compared to other testing methods is that the test cases are generated automatically. However, to succeed with the tests, the instrumentation and test focus must be considered carefully. The tool can only find faults in regions where it is allowed to influence the system under test, i.e. in regions that are affected by changes made by the input instruments. This is the challenge when using the tool; to learn how to instrument the system under test and how to choose the test focus in order to achieve the desired results. It will probably take some time and experience needs to be gained in order to reach this point. For example, the inserted fault in the TCU software was only found because of the knowledge that the ABS had to be varied (activated and deactivated) to trigger the fault. In a vehicle, there are many variables that could be varied and since the number of input instruments ought to be kept low, the challenge is to choose the right selection and number of input variables to instrument.

When the tool is well-known, it can be used in a very time efficient way. The time needed until it is possible to start generating test cases is considerably shorter than when writing test cases by hand. The only actions that have to be performed are the instrumentation and defining the test focus. The test case generation is autonomous and can be performed over night or over a whole weekend. When the instrumentation has been carried out, many different test cases are created in a rather short time. Just by changing the test focus it is also possible to test different parts of the system still using the same instrumentation.

Another advantage of the tool is that it does test everything it is allowed to test. A human tester often omits test cases which are considered to be unrealistic or unimportant. TestWeaver does not have any such valuations but varies the inputs in order to increase the covered state space. Thereby, faults that normally would have been missed might be found.

The aim with the tool is not to replace the existing testing methods but to complement them in the testing process. Commonly performed tests are test scripts which are written by hand to test of fault reactions. The system under test is driven into a certain mode, in this mode a fault is inserted and afterwards

it is examined if the correct fault reaction has been triggered. Tests of this type are rather difficult to perform in a simple and direct way with TestWeaver. It is not especially complicated to insert the faults; this can be done by defining an input instrument for the fault of interest. However, the results of the test cannot really be evaluated in a convenient way. It is desirable to check if a fault reaction is triggered even if the system is fault free, if a fault reaction is triggered when a fault exists and if this fault reaction is the correct one. Since the input instruments and output instruments are not connected, i.e. there is no easy way to determine if an output is the expected one depending of the input; it is complicated to do these checks with TestWeaver. The checks could be programmed by hand in C/C++ but for a complex system with many possible errors and several fault reactions this would mean a great deal of effort and take rather long time. If the instrumentation, despite the extra effort, would be carried out, it would give the benefit that many more test cases easily could be carried out compared to the test cases programmed by hand where normally only a few test cases for each fault and corresponding fault reaction are programmed.

Area of Application

The tool is most appropriate for system testing. By unit testing of a single module, it is difficult to find just a few variables that excite the whole system under test sufficiently. The same problem occurs by integration testing where the system under test consists of several modules integrated into a component. The difficulty to choose just a few inputs that stimulate large parts of the considered component is still the same.

It is possible to use TestWeaver for testing at an early phase of the software development process. As soon as the software can be run in a co-simulation the testing can begin. The results at this point can make it possible to find many faults at an early stage in the development process which is always aimed for. Often, the costs for correcting an error increases the later in the development process the error is found. In addition, the results can support the generation of test cases that are written by hand. It may be desirable to further test some of the found problems in regular test scripts as well.

Another appropriate area of application for TestWeaver is testing at the end of the regular test phase. Tests could be carried out with the aim to find problems that have not been found up to that point due to, for example, omitted or insufficient test cases.

Since the tool generates many different test cases and it has been shown that the coverage of the instrumented and considered parts of the code is high. TestWeaver can also be used for confidence-building purposes: the more tests are run, the higher the confidence in the system under test can be seen.

When a found fault is believed to have been eliminated, the tool could be used for testing if it really has ceased to occur. It would also be possible to test the surroundings of the state where the fault has occurred to study if it arises in some other states as well.

It would be very interesting to be able to use TestWeaver for hardware in

the loop tests as well. By hardware in the loop, the software is run on the real electronic control unit while the rest of the environment still is simulated on a computer. Such tests are carried out in real time which leads to completely other requirements on the tool. A version of TestWeaver that can be used for hardware in the loop simulations is planned.

Bibliography

- [1] <http://www.automotivespice.com/web/introduction.html>. Visited on 2008-12-11.
- [2] http://www.mercedes-benz.de/content/germany/mpc/mpc_germany_website/de/home_mpc/bus/home/new_buses/models/regular_service_busses/capacity/technical_data.html. Visited on 2008-11-22.
- [3] http://www.zf.com/media/media/img/corporate/press/presse/fachpk_2008/064-1_zf-eocliffe_lg.jpg. Visited on 2008-09-18.
- [4] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artec House, Incorporated, 2003.
- [5] Hansjörg Dach, Wolf-Dieter Gruhle, and Peter Köpf. *Pkw-Automatgetriebe, Sicher, komfortabel und wirtschaftlich fahren*. Verlag moderne Industrie, 2nd edition, 2001.
- [6] International Organization for Standardization. *ISO/IEC 15504 Information Technology - Process Assessment*, 2004-2008.
- [7] Automotive Special Interest Group. *Automotive SPICE Process Assessment Model, release v2.3*, 2007.
- [8] Automotive Special Interest Group. *Automotive SPICE Process Reference Model, release v4.3*, 2007.
- [9] Klaus Hörmann, Lars Dittmann, Bernd Hindel, and Markus Müller. *SPICE in der Praxis, Interpretationshilfe für Anwender und Assessoren*. dpunkt.verlag GmbH, 1st edition, 2006.
- [10] Software Engineering Institute. *CMMI for Development, version 1.2*, 2006.
- [11] Andreas Junghanns, Jakob Mauss, and Mugur Tatar. *TestWeaver - Funktionstest nach dem Schachspielerprinzip*. 2. Fachkonferenz zum Thema Test von Hard- und Software in der Automobilindustrie, FKFS, Stuttgart, Germany, 2008.

-
- [12] Andreas Junghanns, Jakob Mauss, and Mugur Tatar. *TestWeaver - Testautomation based on Computer Chess Principles*. 7th International CTI Symposium Innovative Automotive Transmissions, Berlin, Germany, 2008.
- [13] Scott Loveland, Geoffery Miller, Richard Prewitt Jr, and Michael Shannon. *Software Testing Techniques : Finding the Defects That Matter*. Charles River Media, Inc, 2004.
- [14] Markus Müller, Klaus Hörmann, Lars Dittmann, and Jörg Zimmer. *Automotive SPICE in der Praxis, Interpretationshilfe für Anwender und Assessoren*. dpunkt.verlag GmbH, 1st edition, 2007.
- [15] Testwell Oy. *CTC++ Test Coverage Analyzer for C/C++, User's Guide, Version 6.1.1*, 2005.
- [16] Carsten Paulus, Michael Wolff, Peter Feulner, and Hans-Christian Reuss. *Ein neues Framwork zum Testen von Kfz-Steuergräte-Software*. 2. Fachkonferenz zum Thema Test von Hard- und Software in der Automobilindustrie, FKFS, Stuttgart, Germany, 2008.
- [17] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest, Aus- und Weiterbildung zum Certified Tester*. dpunkt.verlag GmbH, 2nd edition, 2004.
- [18] Andreas Spillner, Thomas Roßner, Mario Winter, and Tilo Linz. *Praxiswissen Softwaretest, Testmanagement*. dpunkt.verlag GmbH, 1st edition, 2006.
- [19] John Watkins. *Testing IT: An Off-the-Shelf Software Testing Handbook*. Cambridge University Press, 2001.



Linköpings universitet

Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära 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 icke-kommersiell 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/>

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years 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 his/her 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/>

© Hanna Amlinger