Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Testing degradation in a complex vehicle electrical system using Hardware-In-the-Loop

Examensarbete utfört i Vehicular Systems
vid Tekniska högskolan i Linköping
av

Johannes Bergkvist

LiTH-ISY-EX--09/4193--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

Testing degradation in a complex vehicle electrical system using Hardware-In-the-Loop

Examensarbete utfört i Vehicular Systems
vid Tekniska högskolan i Linköping
av

Johannes Bergkvist

LiTH-ISY-EX--09/4193--SE

Handledare: **Anna Pernestål**
ISY, Linköpings universitet, Scania CV AB
Mikael Adenmark
REST, Scania CV AB

Examinator: **Erik Frisk**
ISY, Linköpings universitet

Linköping, 27 January, 2009

Avdelning, Institution
Division, Department

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

Datum
Date

2009-01-27

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ ______________

Rapporttyp
Report category

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

URL för elektronisk version

http://www.control.isy.liu.se
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-16549

ISBN
—

ISRN
LiTH-ISY-EX--09/4193--SE

Serietitel och serienummer
Title of series, numbering

ISSN
—

Titel Title	Test av degradering i helbil i ett Hardware-In-the-Loop labb Testing degradation in a complex vehicle electrical system using Hardware-In-the-Loop
Författare Author	Johannes Bergkvist

Sammanfattning
Abstract

Functionality in the automotive industry is becoming more complex, with complex communication networks between control systems. Information is shared among many control systems and extensive testing ensures high quality. Degradations testing, that has the objective to test functionality with some fault present, is performed on single control systems, but is not frequently performed on the entire electrical system. There is a wish for testing degradation automatically on the complete electrical system in a so called Hardware-In-the-Loop laboratory. A technique is needed to perform these tests on a regular basis.

Problems with testing degradation in complex communication systems will be described. Methods and solutions to tackle these problems are suggested, that finally end up with two independent test strategies. One strategy is suited to test degradation on new functionality. The other strategy is to investigate effects in the entire electrical system. Both strategies have been implemented in a Hardware-In-the-Loop laboratory and evaluated.

Nyckelord
Keywords Testing, degradation, HIL, Hardware-In-the-Loop, automotive industry

Abstract

Functionality in the automotive industry is becoming more complex, with complex communication networks between control systems. Information is shared among many control systems and extensive testing ensures high quality. Degradations testing, that has the objective to test functionality with some fault present, is performed on single control systems, but is not frequently performed on the entire electrical system. There is a wish for testing degradation automatically on the complete electrical system in a so called Hardware-In-the-Loop laboratory. A technique is needed to perform these tests on a regular basis.

Problems with testing degradation in complex communication systems will be described. Methods and solutions to tackle these problems are suggested, that finally end up with two independent test strategies. One strategy is suited to test degradation on new functionality. The other strategy is to investigate effects in the entire electrical system. Both strategies have been implemented in a Hardware-In-the-Loop laboratory and evaluated.

Acknowledgments

I would like to thank my supervisors Mikael Adenmark and Anna Pernestål for their help and support. Thanks to every person that has taken their precious time to answer my question, making this thesis possible. Also my examiner Erik Frisk at ISY in Linköpings Universitet deserves big thanks.

Contents

Chapter 1

Introduction

Murphy's law states: If something can go wrong, it will also go wrong! But how shall a system act if something goes wrong? The answer is quite simple: It shall act as we say it to act! The behaviour can be verified by choosing some input and compare the output with the expected output.

Degradation is a term that is correlated with the behaviour when something has gone wrong. It is as important to verify a correct behaviour during failure, as it is with no fault present. But the question is, how to perform these degradation tests the best?

1.1 Background

Functionality in the automotive industry is becoming more complex, with control systems replacing mechanics and hydraulics. Complex communication networks between control systems are emerging, with information shared among many control systems. With increasing complexity the occurrence of faults usually increases as well. It is becoming more important to validate functionality implemented in the control systems. Testing is performed on different levels during the development process to maintain high quality.

Not only validation of correct behaviour when the entire electrical system is fault free must be performed, but also validation during failure. How shall the electrical system behave with sensors broken or some other fault present? Fail-safe is a term used to describe that something is safe during failure. To maintain a fail-safe vehicle extensive testing has to be done to verify that functionality is never becoming dangerous.

Degradations testing, which has the objective to test functionality with some fault present in the vehicle, is not frequently performed. It must be verified that functionality is behaving correctly even in a degraded state. A method is needed to perform these tests on a regular basis.

1.2 Communication network

The Controller Area Network (CAN) is standard in the automotive industry. Communication is transmitted on a CAN bus, where electronic control units are connected. The protocol allows control units to communicate through a network.

1.2.1 Electronic Control Unit (ECU)

An Electronic Control Unit (ECU) is a physical unit, with a number of sensors and actuators attached to it. An ECU also receives external information from other ECUs via CAN. An ECU is responsible of certain functionality that uses sensors and external information as inputs. The outputs of the functionality are then used through actuators to control the vehicle.

Example 1.1

Some ECUs that can be found in a Scania truck

- **EMS:** Engine Mangagement System, controls the engine.
- **APS:** Air Pressure System, controls the flow and the pressure of the air.
- **AUS:** Audio System, controls the radio and CD-player.

1.3 Scania's electrical system

Scania's electrical system consists of three CAN-buses (Section 2.3.1), which are named Red, Yellow and Green. Each of these buses are connected to the Coordinator (COO) and can pass information between the buses. The Red bus consists of ECUs which can be paired with the driveline; engine, gearbox and braking system. The Yellow bus has systems with security related functions, but that are not critical for the vehicle to be driven. Other ECUs are to be found on the Green bus. These ECUs are related to driver comfort and are not critical to the vehicle or the security of the driver. See Figure 1.1 for the architecture of the electrical system.

The number of different combinations of the electrical system is immense. Each ECU has between 50 - 10000 parameters that can be changed to be compatible with every possible specification of vehicle, depending on hardware and functionality required. Despite all different configurations of ECUs the electrical system has to work for every valid combination of ECUs. To be able to ensure that there is no compatible issues extensive testing has to be performed on integrations level.

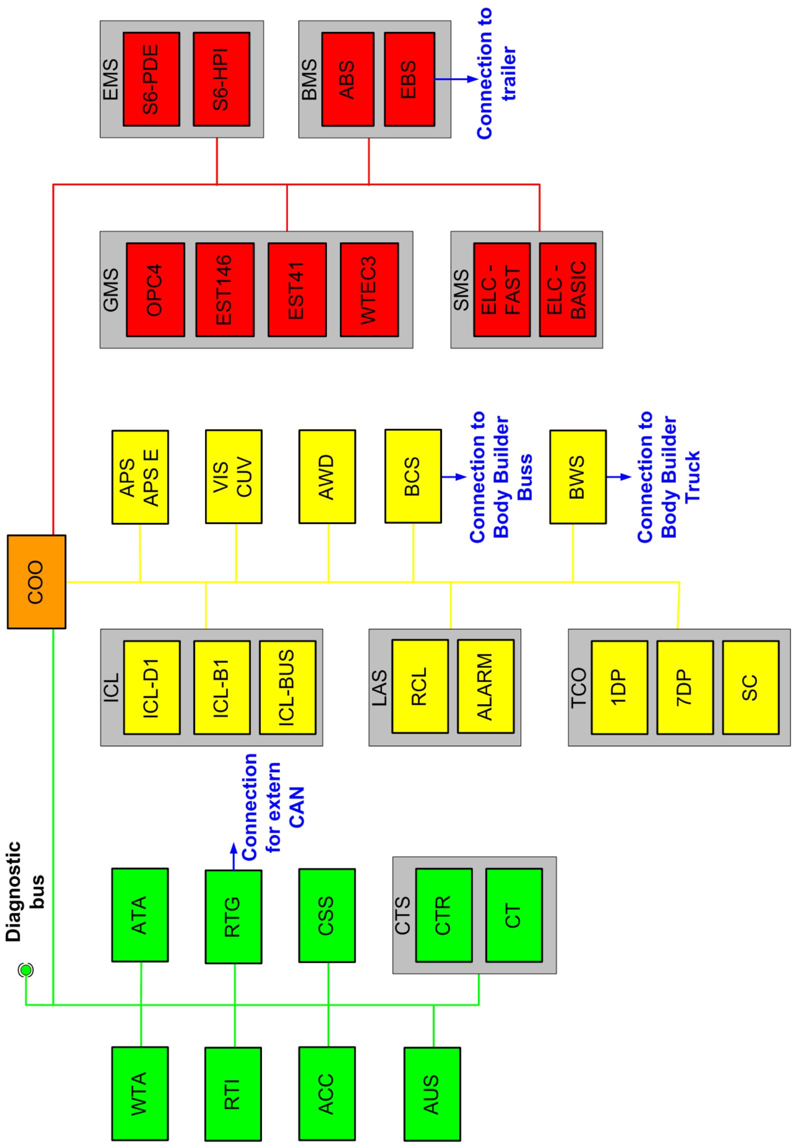


Figure 1.1. An illustration of the electrical system in a Scania vehicle. The three CAN-buses; Red, Yellow and Green; are connected to the COO. Only EMS, APS, ICL and VIS/CUV are mandotory. The other ECUs are arbitrary. Each ECU have 50 - 10000 parameters depending for example of different hardware configurations. The number of combinations is almost infinitive.

1.4 Test process

A commonly used model for the development process is the V-model. The name comes from that the process can be described as a V. The left 'leg' in the V symbolizes development and the right "'leg"' symbolizes the tests. See Figure 1.2. There are in general four basic levels, but depending on the project the number of levels can either be more or fewer. [16]. The test levels are:

- **Unit testing:** Verifies that separate units such as ECUs are programmed according to specification.
- **Integration testing:** Verifies that units are working together as expected; that the communication between the units are according to specification. Functionality is not tested in this step, just communication. [16]
- **System testing:** Verifies that functionality and communication fulfills the requirements set on the entire system. A common problem is that requirements may be incomplete or undocumented.
- **Acceptance testing:** Verifies that the complete system is ready for deployment. This is the last step of testing according to the four levels, but extending testing after acceptance testing are usually performed.

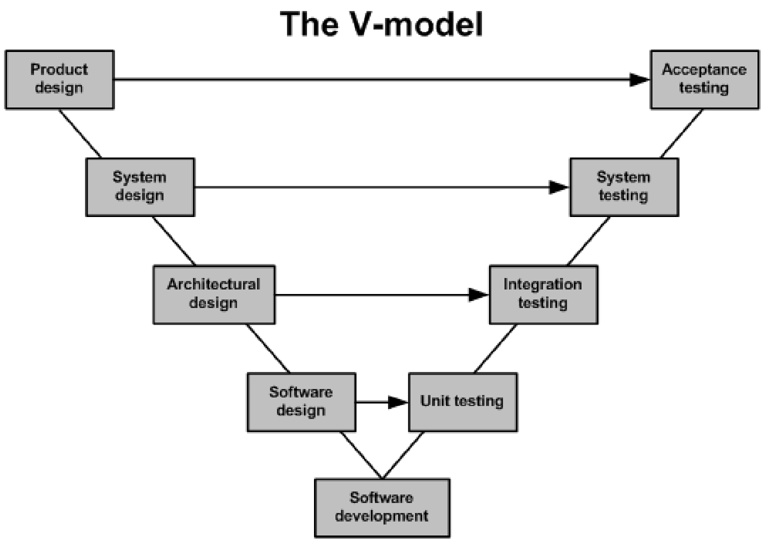


Figure 1.2. An illustration of the V-model. Different levels of specifications are to be found in the left leg, with resolves with Software Development at the bottom. Each level of specification has a corresponding level of test. Note that depending of the project, the V-model have an increased or decreased number of levels.

1.4.1 System and integrations test at Scania

At Scania both systems testing and integrations testing are performed by REST. System and integrations test is the last step in testing the entire electrical system. These kinds of tests shall verify that the electrical system is working properly. ECUs shall be compatible with each other for a number of different configurations. Communication on CAN and distributed functionality must be validated to meet the quality demands. Two commonly instruments to perform system and integrations test are field tests and Hardware-In-the-Loop.

Field tests are performed in an actual vehicle on the road and will not be considered in this thesis. Hardware-In-the-Loop is based on that the vehicle can be simulated using models, and that tests can automatically be performed using a computer.

1.4.2 Hardware-In-the-Loop (HIL)

Hardware-In-the-Loop (HIL) is used in the automotive industry when testing the electrical system. In a HIL-lab the real ECUs are connected to each other using the real communication network. As much as possible of the sensors and switches that are connected to the ECUs are simulated using models. The actuator signals from the ECUs are inputs to a model of the truck and the environment. The model then calculates sensor values which are then received by the ECUs. The ECUs are connected to a Fault Injector Unit (FIU) which simulates short circuits or open circuits on sensors and actuators. See Figure 1.3.

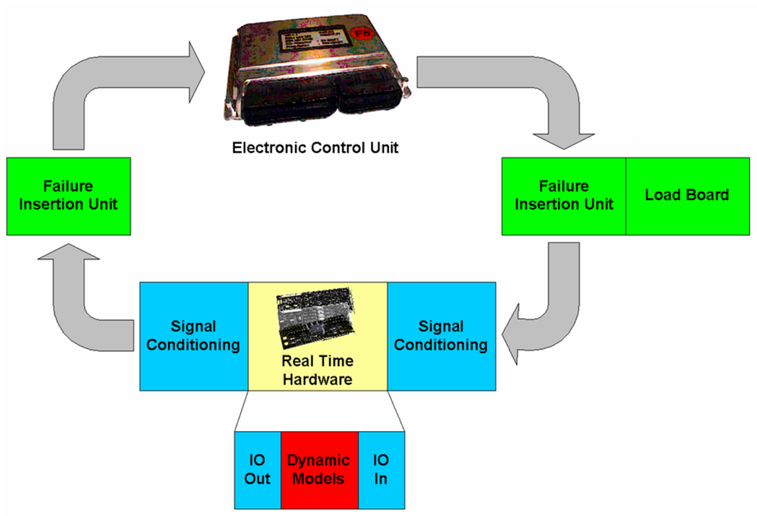


Figure 1.3. The process of HIL. Starting with the ECU at top of the figure. The outputs from the ECU are connected to a Fault Injector Unit (FIU) and then received by a Dynamic Model. The model then calculates the inputs to the ECU, which also is connected to a FIU.

One advantage is that everything can be run from a computer, meaning that the complete vehicle and functionality can be tested using a control panel in a computer. Since everything can be controlled automated test can be run using scripted tests in a programming language.

1.4.3 Testing in a HIL-lab today

The HIL-lab at Scania is called I-Lab2. The kind of tests today performed in I-Lab2 verifies that distributed functionality is working correctly with no fault present. The test case is written first and describes the test in words, with stimuli and expected response, and is then translated to a test script written in Python for automatically testing in I-Lab2. The tests verifiy Message Sequence Charts (MSC) (Section 2.2.1) that is a description of a distributed function. The tests consists of validating communication on CAN, that correct information is sent and then received and used in the right way.

The difference between tests performed in I-Lab2 and tests performed on unit level, so called ECU System Test Level, is that the entire chain is tested in I-Lab2. When performing ECU test it is only verified that the ECU sends the correct information, or that it treats received information correctly. It is of no interest what happens in the other end.

Degradations testing, which has the objective to test functionality with some fault present in the vehicle; i.e. test the vehicle being fail-safe; is performed sparsely at Scania. Some control systems test degradation regularly, but some are not. However degradations tests are not performed frequently on integration or system level. Are control systems communicating correct information during failure and is the received information interpreted in the right way? Is it even possible to perform degradations tests in a Hardware-In-the-Loop lab? This thesis shall investigate the possibilities for such kind of degradations tests.

1.5 Thesis objective

The objective is to find an applicable method usable to test degradation of distributed functionality in a Hardware-In-the-Loop lab on a regular basis. The objectives can be stated as four items:

1. Collect and compile information about distributed functionality
2. Investigate if there is any unspecified degradation effects in the electrical system
3. Suggest a test method to efficiently verify that the degradation of functionality behaves correctly
4. Write test cases and implement these for I-Lab2 to verify the efficiency of the suggested test method

1.6 Method

The method is first to compile information about degradation and degradation tests, by interviewing personnel at Scania and using internal documents. Using gathered information a test method how to best perform degradations test has to be suggested, that shall be usable with existing resources. The test method suggested shall be implemented and evaluated if suited for regular use.

1.7 Related work

There are many papers, mostly from the automotive industry, describing the process of HIL and the benefits for executing tests. However it is not well documented how testing is best executed using HIL. Especially methods of negative testing, using for example malfunctioning sensors, is sparsely found.

A paper written by Mauro Velardocchia and Aldo Sorniotti [11] uses HIL to test ABS and ESP functionality in a brake system. The tests are concentrated to faults on sensors used by the brake system. The correct behaviour, with fault free sensors, is then compared to the result given with faulty sensors. This problem is limited since they are only researching the brake system, and do not consider what happens outside their scope.

Another paper by Nabi and Balike [15] is discussing degradations tests in a more complete electrical system. They are discussing that one type of performed tests is to insert a fault in the system and then see the effect on performance, but this is not always sufficient. There is sometimes a need to test that the fault also has been detected, which require the use of diagnostic software. A small section also describes a process that can be run automatically.

A more general approach of fault modeling in complex distributed systems using CAN can be found in [4]. They say that the commonly used fault models, with faults on sensors and actuators, are insufficient. The effect on electromagnetic disturbances on CAN also needs to be considered when testing.

A paper on the requirements on failsafe [3] describes different methods of analyze the failsafe of a vehicle. Methods as Failure Mode Effect Analysis, Fault Tree Analysis and Fault Coverage Matrix are discussed.

A paper about degradation [14] uses a mathematical model to calculate the degradation of a suspension element in a vehicle, to measure change in performance. The method described can be compared with model based diagnosis.

1.8 Contributions

There are three major contributions with this thesis

- Compiling information about degradation of distributed functionality and describing problems about the subject (Chapter 4)
- Determining what behavioural modes on signals to provoke and test, where to presume effects of error propagation and finally suggesting an alternative method to test without exact expected outputs (Chapter 5)
- Suggesting two test techniques that can be performed in I-Lab2 on a regular basis using current hardware and software (Chapter 6)

Chapter 2

Theory: General concepts

The following chapter will give some general concepts needed further ahead. First the idea of fail-safe will be discussed in Section 2.1. Fail-safe is important in the automotive industry because of the high demands set on the safety of the driver and the surroundings. Definitions of fault and failure will be described in Section 2.1.1, including a discussion about degradation in Section 2.1.3.

After the basic concepts of fail-safe, a brief description of the electrical system at Scania will be followed in Section 2.3. Functional architecture in Section 2.2 and the rules set on communication on CAN in Section 2.3.3 will be described. Terms as function, Message Sequence Chart (MSC), Diagnostic Trouble Codes (DTC) and gateway will also be defined.

2.1 Fail-safe

According to Peters [6] the concept of fail-safe is often overlooked when designing a system. Fail-safe deals with fault management, to prevent that a failure or a malfunction is not causing severe damage. Any unwanted behavior of the vehicle must be avoided even with faulty sensors or broken cables. Especially safety critical functionality has to be tested to ensure that unwanted behavior does not emerge. The concept of fail-safe will lead to a discussion about degradation, but first fault and failure has to be defined.

2.1.1 Fault and failure

The definitions of fault and failure are taken from Nyberg and Frisk [10].

Definition 2.1 *A **fault** is an illicit deviation of at least one characteristic property or variable of the system from acceptable/usual/standard/nominal behaviour.*

Definition 2.2 *A **failure** is a fault that implies permanent interruption of a systems ability to perform a required function under specified operation conditions.*

According to the definitions, a fault is the underlying cause of a failure to a system. A failure occurs when there is hindrance of a system to perform according to specification. A deeper discussion about the relation between fault and failure can be found in [1]. A fault is said to be dormant until it is activated, which will then produce an error, meaning that there is a deviation from correct behavior. When the system no longer can perform according to specification the error will propagate and produce a failure. There can therefore be a fault in a system that has not yet created a failure since the fault is not active. See Figure 2.1.

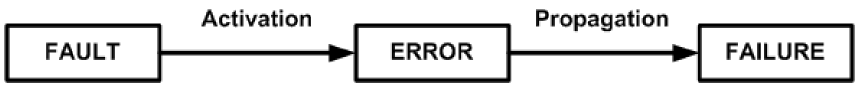


Figure 2.1. The relation between fault, error and failure

Because of error propagation, failures can in turn produce active faults which in the end produce more failures.

2.1.2 Failure Mode and Effect Analysis (FMEA)

There are several techniques to map the potential risks in a system, to provide help to develop a fail-safe system. One such method is Failure Mode and Effect Analysis (FMEA) which is an inductive or a bottom-up analysis [3]. All potential faults that can affect a system are first considered. Then all related failures are identified with possible measures to counteract the expected hazards.

There are different kinds of FMEA, depending of the area of application. The kind of FMEA that is used during development is called Design-FMEA (D-FMEA), which considers all faults relevant to the design of the product. For example all faults and failure effects regarding sensors and communication is analyzed.

Each failure is graded in three parameters: Severity, Occurrence and Detectability. The grades are multiplied to get an overall grade, a so called Risk Priority Number (RPN). The higher RPN, the higher risk and prioritize the fail-safe measures needed according to the FMEA.

Since the FMEA specifies how the system should react in the present of a failure, it is now time to discuss degradation.

2.1.3 Degradation

When a system is fault free it is said to be in normal operation mode. If there is a failure to the system, meaning that some functionality cannot execute properly, the system can enter a degraded mode. Degradation will ensure that some parts of the failing functions still will execute [1]. When a system is not in its normal operation mode the system is said to be degraded, which gives the following definitions:

Definition 2.3 ***Degradation*** *of a system is a controlled behavior when a failure is present, with the consequence that the system is leaving normal operation mode, often realized as decreased performance or loss of functionality.*

Definition 2.4 *A* ***degradation mode*** *is a specific degradation of a system during failure, depending of the present fault.*

The definitions above talks about degradation of a system, but degradation can also be applied on signals and functions. An important note is that degradation should not be misunderstood with intended reduced functionality, which is referred to as downgrading. Also note that according to Section 2.1.1 degradation is entered due to failure in the system. A fault will not cause any degradation until activated and a failure occurs. The underlying cause of a failure is always an activated fault and the degradation will depend on the occurring fault. It will from here on be assumed that all faults will be active faults, unless other stated.

Example 2.1

Opticruise which is Scania's own gearbox, has four well defined modes, normal operation mode and three degradation modes:

- **Normal operation mode**
  Fault free mode where opticruise operates according to specifications
- **Clutch mode**
  The driver has to use the clutch to change gear, which is not necessary in normal operation mode
- **Limp hold**
  The driver cannot change gear, and must drive with the current gear
- **Limp home**
  There is only functionality required to drive the vehicle to a mechanic

When an (activated) fault has occurred, normal operation mode will be left. The system is said to be degraded. Depending of the fault the system will enter one of the three degraded modes.

2.2 Functional architecture

The functionality at Scania is described by user functions which in turn is described by use cases and scenarios. Each scenario has a related Message Sequence Chart (Section 2.2.1) which is a diagram describing how the scenario is executed. But before discussing the above the terms function and distributed function must be defined.

Definition 2.5 *A **function** is describing a subset of a system, with the purpose to control a set of outputs in a specific way. **Functionality** is defined as the purpose of one or many functions in general.*

Definition 2.6 *A **distributed function** is a function where subsets of the distributed function is localized in more than one ECU, requiring communication between the ECUs to function properly.*

Definition 2.7 *A **user function** (UF) is an electrical-related service as perceived by a user [9].*

A user function can involve none, one or more ECUs, meaning that a user function does not have to be distributed. But most user functions are in fact distributed functions. User functionality does not cover all distributed functionality in a vehicle, meaning that there is functionality not described with user functions.

Example 2.2

Example of user functions, which all are distributed:

- **UF 18: Fuel Level Display**
  Function to display the fuel level in the Instrument Cluster Panel (ICL)
  The fuel level sensor is attached to the Coordinator (COO) which sends the measured fuel level to the Instrument Cluster Panel that displays the information to the driver.
- **UF 85: Seat Heating**
  Function to activate or deactivate the bottom heating of the driver seat.
- **UF 121: ABS Control**
  Function to prevent the wheels from locking while braking
  The logic of the ABS Control is placed in the Brake Management System (BMS) that gathers information from up to five control systems and then decides if ABS Control is needed. On top of that the ABS has to inform other ECUs that ABS is active.

To give structure to the complex functionality in a truck, there are user functions describing smaller parts of the complete system. Each user function is associated with one or more use cases, different ways to use the function. Activating and deactivating some functionality are in general two different use cases. Each use case consists of at least one scenario, specifying how the driver can interact within the use case. Scenarios are common dependent of the architecture in the vehicle, how the signal flow is within the electrical system. [9]

As seen in Example 2.3 every user function can have several different use-cases and each use-case can have several different scenarios. The same example also displays the relation between user functions, use-cases and scenarios.

Example 2.3

UF 456 Engine Oil Level Display
Function to display the oil level in the engine.

- Use-case: Display information
  - Scenario: Display engine oil level indication in instrument cluster
  - Scenario: Display last measured engine oil level in instrument cluster
  - Scenario: Display high/low engine oil level in instrument cluster
- Use-case: Malfunction handling
  - Scenario: Engine oil level malfunction

2.2.1 Message Sequence Chart (MSC)

Each scenario has a corresponding Message Sequence Chart or MSC, a diagram describing how the scenario is executed. The diagram shows which components (sensors and actuators), ECUs and communication are involved. The MSC also states in which order everything is executed. As seen in the Figure 2.2, each component and ECU is realized as a vertical line. Between these lines there are connectors, arrows, that symbolize the signal routing. The leftmost vertical line, called Environment, is the interface to the physical world, i.e. displaying information to the driver or gathering information used within in the MSC.

Today, test cases at REST are based on MSCs.

2.3 Communication

2.3.1 Controller Area Network (CAN)

CAN stand for Controller Area Network and is frequently used in the automotive industry. The protocol allows ECUs to communicate through a network. Com-

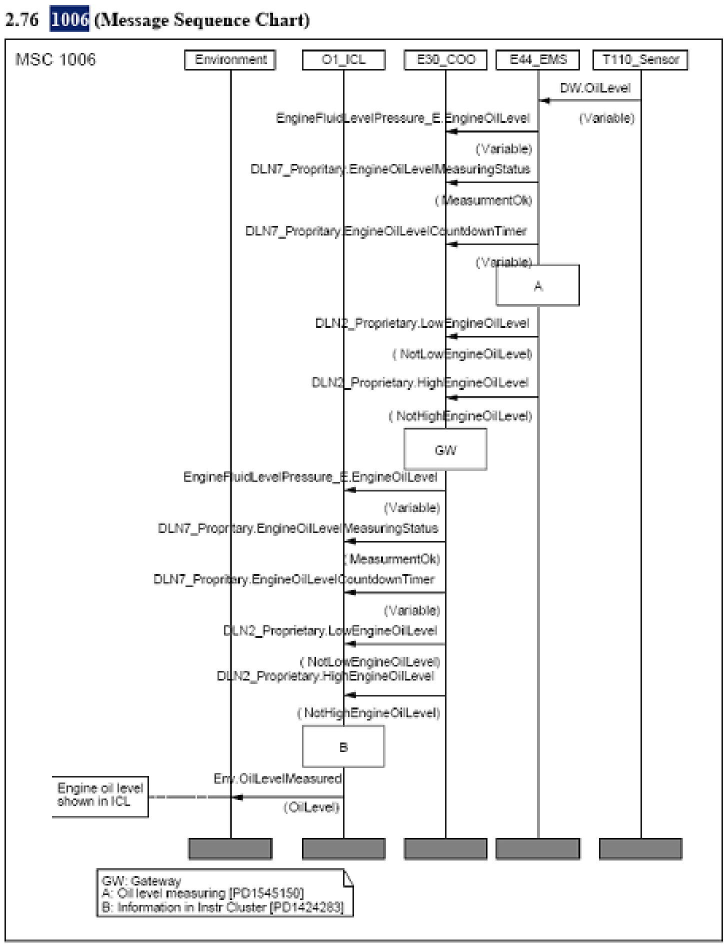

2.76 1006 (Message Sequence Chart)
MSC 1006
Environment
O1_ICL
E30_COO
E44_EMS
T110_Sensor
DW.OilLevel
(Variable)
EngineFluidLevelPressure_E.EngineOilLevel
(Variable)
DLN7_Propritary.EngineOilLevelMeasuringStatus
(MeasurmentOk)
DLN7_Propritary.EngineOilLevelCountdownTimer
(Variable)
A
DLN2_Proprietary.LowEngineOilLevel
(NotLowEngineOilLevel)
DLN2_Proprietary.HighEngineOilLevel
(NotHighEngineOilLevel)
GW
EngineFluidLevelPressure_E.EngineOilLevel
(Variable)
DLN7_Propritary.EngineOilLevelMeasuringStatus
(MeasurmentOk)
DLN7_Propritary.EngineOilLevelCountdownTimer
(Variable)
DLN2_Proprietary.LowEngineOilLevel
(NotLowEngineOilLevel)
DLN2_Proprietary.HighEngineOilLevel
(NotHighEngineOilLevel)
B
Env.OilLevelMeasured
(OilLevel)
Engine oil level shown in ICL
GW: Gateway
A: Oil level measuring [PD1545150]
B: Information in Instr Cluster [PD1424283]

Figure 2.2. UF456: Engine oil display: Display engine oil level indication in instrument cluster. The oil level sensor (T110_sensor) sends the measured oil level to the Engine management system (EMS), which in turn sends that information to the Instrument cluster panel (ICL), via the Coordinator (COO). The ICL uses the information from EMS to display the oil level to the driver.

munication is transmitted using frames, which can vary in size depending on the data length. Excluding the real data that is sent in a frame, information such as identifier and control bits are also sent. [2]

Definition 2.8 *A* ***message*** *is defined as the frame sent on CAN. A message always has a transmitter and one or more receivers.*

Since CAN is network based, messages that are distributed on CAN are available to every node (ECU) connected to the network. A message identifier consists of a unique Parameter Group Number (PGN), which is combined with the transmitting ECUs source address. The same message can therefore be sent by more than one ECU, but with different source address [7].

Definition 2.9 *A* ***signal*** *is a subset of a message and always sent with a message. A signal consists of one value that is generated in an ECU and describes only one parameter.*

Each message consists of several signals, which can be measured sensor values or requests that are sent to other ECUs.

2.3.2 A signal's behavioural modes

Before discussing the behavioural modes of signals, the definitions of behavioural mode and fault mode has to be made. Again with the help of Nyberg and Frisk [10]:

Definition 2.10 *A* ***behavioural mode*** *describes the state of a component or a system, and is related to faults that will cause a failure in that component or system*

Definition 2.11 *A* ***fault mode*** *is all behavioural modes that describes a component or system to be faulty.*

A behavioural mode is a description if something is whole or faulty, but can also specify what is faulty. Only one behavioural mode can be occupied in the same time.

To clarify the definitions two examples are given:

Example 2.4

A circuit consisting of a bulb and a switch can have the behavioural modes:

$$\{\text{switch not broken and lamp not broken,}\\ \text{switch broken and lamp not broken,}\\ \text{switch not broken and lamp broken,}\\ \text{switch broken and lamp broken}\}$$

The fault modes of system are the three later ones. The first mode describes a fault free state.

Example 2.5

The bulb in the previous example can itself have different behavioural modes, which can explain the behavioural modes above. The behavioural modes of the bulb can be:

$$\{\text{not broken, the thread broken, other fault}\}$$

The two later behavioural modes are fault modes.

When talking about behavioural modes (plural) it will mean all behavioural modes, all states a component or system can be in.

A signal on CAN has five well defined behavioural modes:

Behavioural mode	Description
Defined (Def)	The value is within valid limits
Undefined (Undef)	The value is not within valid limits
Error	An Error means that the ECU originally sending the signal has detected an error. See definition of Error below.
Not Available (NA)	Not Available is generally sent when the sensor generating the signal is missing. See definition of Not Available below
Time-out (TO)	Time-out means that the signal is not received by the receiver. This could happen if the transmitted ECU is short circuited

A split of {Defined} can be made to introduce a fifth fault mode {Unrealistic}, which is a value within valid limits, but not realistic. An unrealistic value could be if the vehicle speed has the value 2000 km/h, with the parking brake applied and reverse gear. The behavioural mode {Unrealistic} will not be considered in this thesis, which will be motivated later in Section 5.1. After this discussion the behavioural modes for signals will be defined as:

Definition 2.12 *A **signal's behavioural** modes are {Defined, Undefined, Error, NA, TO}.*

When talking about specific signal, u, the behavioural modes for that signal will be denoted

$$u \in \{u_{def}, u_{undef}, u_{error}, u_{na}, u_{to}\}$$

.

2.3.3 Fault convention

There are a number of rules that the communication on CAN has to follow, which concerns the behavioural modes of {NA} and {Error}. There are two rules when to send {NA}.

Error

The behavioural mode {Error} can only be sent when a fault in a sensor or switch has been detected, which means that the system itself has to know that something is wrong, and must also store a corresponding Diagnostic Trouble Code (DTC) [7]. See also Section 2.4

Not Available (NA)

If data from a sensor or a switch has not been received or validated, {NA} must be sent. But if the system can conclude that the sensor is really missing or broken, {Error} shall be sent [7].

If a vehicle is by specification missing a sensor, or if a signal in a message should not be sent by an ECU, {NA} should also be sent.

Coordinator as gateway

Since the Coordinator (COO) is connected to all three CAN buses it acts as a gateway, passing information between the buses. There are two ways to gateway information; gatewaying of a complete message or gatewaying of just one signal (also called data gatewaying).

Definition 2.13 ***Gateway***: *When an ECU acts as a gateway and receives a message or signal, it will forward the same message or signal unprocessed on the other CAN bus.*

Definition 2.14 ***Data gateway*** *is used when only one particular signal is gatewayed. That means that one signal is taken from one message and put in another message.*

The definition says that if a message or a signal is being gatewayed the content of that message or signal is not allowed to change. If a complete message is being gatewayed and is missing nothing is sent on the other CAN buses. The message will be missing on the other CAN buses as well. When data gateway and the message with the signal is missing, the signal enters the behavioural mode Not Available on the other CAN buses.

A note about gateway and message identifier regarding Section 2.3. A message that is completely being gatewayed should keep the original identifier. If a message in some way is being manipulated, the source address should be changed to the gatewaying ECU's source address.

2.4 Diagnostic Trouble Codes (DTC)

When a fault has been detected a DTC shall be set within the ECU, and can then be gathered by a user [5]. It is recommended that for every possible fault there should exist a corresponding DTC, with enough information so that a mechanic at the workshop can find the fault. For every ECU there is DTC-specification, a complete description of every DTC that can be stored within the ECU.

There are two kinds of DTCs; primary and secondary:

- **Primary:** Is related to a fault detected within the own ECU. A primary DTC has to be set when communicating Error for any signal sent on CAN.

- **Secondary:** A secondary DTC has to be set for each signal the ECU receives Error and with the consequence of degraded functionality.

A time stamp is connected to the DTC telling when the fault was confirmed the last time. A counter keeps track of the number of times the fault has been confirmed. Both the time stamp and the counter is only updated when the fault has been unconfirmed and then confirmed again [5].

2.5 Summary

Constructing a system that is fail-safe is of great importance in the automotive industry. FMEA is a method of finding potential risks that have insufficient fail-safe measures. Degradation of a system or function is a technique preventing unwanted behavior (for example found during the construction of FMEA) to control an active fault.

A complete system can be split into many functions, which require systems to communicate with each other on CAN. Messages consist of signals and are sent on CAN. Signals are the lowest entity of communication and can have different behavioural modes. There are a set of rules that communication has to follow regarding these behavioural modes.

Chapter 3

Theory: Testing

The following chapter will discuss testing in general, why testing is necessary and some basic concepts. Two main test strategies, black box and white box, will be discussed which will finally lead to resolute test methods. Also a method of determine the level of testing currently used by REST at Scania is also described.

3.1 Testing

The following definition of testing can be found in IEEE Standard Glossary of Software Engineering Terminology, which is referenced to in Copland [8]:

Definition 3.1 ***Testing*** *is the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.*

In other words, to verify that the system or component under test is behaving as expected with a set of suitable inputs. The concept is not at all especially difficult, but several challenges emerge when performing tests. One challenge is the time aspect, meaning that there is not enough time to test everything. Large systems often have many inputs, which can lead to an uncontrollable number of input combinations. Bad specifications makes it difficult to determine expected result from the test is also a concern.

3.2 Test strategies

There exists in general two different test strategies, black box and white box. Using black box the tester does not in detail need to know how the system works. The strategy is based on which outputs are given for a specific set of inputs. White box is the direct opposite, which requires detailed knowledge about the system, its architecture and how it is programmed. There is also a combination of the two above test strategies called gray box.

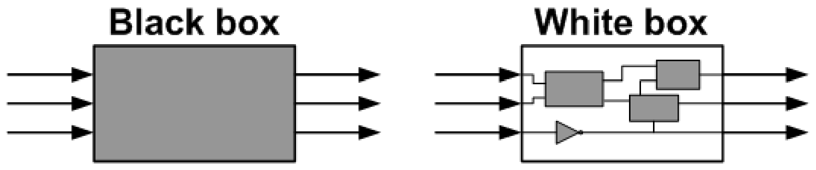


Figure 3.1. Difference between Black box and White box. Using Black box the logic within the system is unknown. White box is the opposite, that the system's logic is known

3.3 Black box testing

Black box testing does in general follow a certain process. At first the tester needs to analyze the specification of the system, how it works and which inputs are valid. Then a set of inputs are chosen and what outputs are expected. After the tests are run, a comparison between the actual and expected outputs is made.

The main advantage using black box is that testing in principle can be applicable on systems on all level, from testing a small unit to an entire system. The strategy is more efficient the larger the system, due to that no detailed knowledge is required of how the system is implemented. The behaviour of the system has to be known, meaning that detailed specifications are required to be able to determine expected response due to some choice of inputs.

It is difficult to achieve 100 percent detectability, since the tester does not know how the systems is coded and can miss a certain input that would led to an error. To maximize detectability both black box and white box can be used but on different levels.

3.4 White box testing

Instead of analyzing specifications of a system, the tester is analyzing the implementation. All internal paths in the code are identified. Inputs are then chosen so that a specific path is executed. The actual outputs are then compared to expected outputs.

The main advantages is that a tester can be sure to get 100% coverage of paths executing, making sure that every piece of code and every possible path is tested at least once. The White Box strategy can also be applied to larger systems to test different pahts between systems.

The number of paths can be very large, and it can be impossible or at least very time consuming, to test every possible path. Testing all paths can be compared with testing all possible inputs using Black box. White box can only test existing paths that can be identified in the code. New paths cannot be discovered with White box.

3.5 Test methods

There exist a number of test methods based on either Black box and White box. The two testing methods that are going to be discussed here and are later used are Equivalence Class Testing and Pairwise Testing. These are based on the Black box approach, meaning that no internal structure of the system under test has to be known. More test methods can be found in Copland [8], but requires good knowledge about the system under test, which is not always the case when testing degradation on the entire electrical system.

No methods based on the White box will be discussed, because the internal structure for every case of degradation requires extensive resources. The internal structure for a scenario in a user function can be found in the MSC, where signals can be traced within the MSC. However the MSCs does not describe alternative signal paths in case of degradation, required when performing White box testing.

3.5.1 Equivalence Class Testing

Instead of testing each possible input, one can instead try to classify inputs into groups, equivalence classes, and then say that every value in a selected group represents the entire group. Using these groups means that only one test case per equivalence class is required.

Example 3.1

CAN communication specifications at Scania consists of every message and signal transmitted and received by an ECU. In the specifications every signal has intervals defined with respective behavioral modes. A signal consisting of two bytes of data is in Scania's electrical system usually defined according to Table 3.1.

Binary value	Behavioural mode
0x0000 - 0xFBFF	Defined
0xFC00 - 0xFDFF	Undefined
0xFE00 - 0xFEFF	Error
0xFF00 - 0xFFFF	Not Available
No value	Time-out

Each of these behavioural modes can be used as an equivalence class, meaning that it is equivalent to test 0x0C5F and 0xF01A. Every value in the above behavioural modes represents the entire mode.

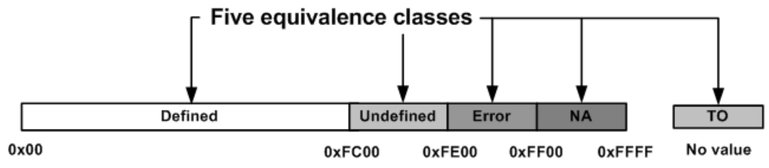


Figure 3.2. A graphical representation of behavioral modes for a signal consisting of two bytes

An assumption has to be made, that every ECU treats each value in each class the same; i.e. the classes according to Table 3.1. The logic in the ECU should be based on these classes, if Equivalence Class Testing can be used correctly. The next example will illustrate this.

Example 3.2

Consider the following equation:

$$\begin{cases} y = x + 1 & \text{if } x < \text{0xFC00} \\ y = 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Let say that a programmer has implemented the equation in code as:

```
if x == 0 then y = 1;
if x == 1 then y = 2;
.
.
if x == 655 then y = 656;
.
.
if x == 0xFFFF then y = 0;
```

Using code above could mean that somewhere in the code a mistake could have been made, for example:

```
if x == 700 then y = 1054;
```

which is completely wrong according to equation (3.1). Typically Equivalence Class Testing could not be applied here since there are no distinct groups or classes. Every input is here considered as a stand alone input.

Instead a more experienced programmer would have implemented the equation more easily like this:

```
if x >= 0 && x < 0xFC00 then y = x + 1;
if x >= 0xFC00 && x < 0xFE00 then y = 0;
.
.
if x >= 0xFF00 && and x <= 0xFFFF then y = 0;
```

Now there are distinct groups, meaning that there is no difference between `x = 12` and `x = 0xD201`. From now on, if Equivalence Class Testing is applied it is assumed that the code looks like this.

The advantage using Equivalence Class Testing is that instead of one test case per valid input, one test case per valid equivalence class can now be scripted. That means that the number of test cases will be decimated, with exactly the same outcome. As mentioned in the example above, the method requires that the code looks like the later of the two pieces of code in the previous Example 3.2, or at least it is assumed.

A variant to Equivalence Class Testing is Boundary Value Testing described in Copeland [8]. Boundary Value Testing means that the boundaries are chosen as inputs instead of a random chosen input within the equivalence class. Consider the following example:

Example 3.3

Consider Example 3.1 with the signal consisting of two bytes and is defined as:

Binary value	Behavioural mode
0x0000 - 0xFBFF	Defined
0xFC00 - 0xFDFF	Undefined
0xFE00 - 0xFEFF	Error
0xFF00 - 0xFFFF	Not Available

Using Boundary Value Testing means that inputs at the boundary values of the classes are chosen: 0x000, 0xFBFF, 0xFC00, ... , 0xFF00 and 0xFFFF.

Boundary Value Testing will not be applied because that there is a wish not to manipulate inputs directly on CAN, but provoke the ECUs to send the correct behavioral mode. It is therefore difficult to force an ECU to send a value on CAN chosen by the user. The value sent on CAN is dependent on how the ECU is programmed. See also Section 4.4.

3.5.2 Pairwise Testing

Pairwise Testing method is based on that the tester only test all pairs of behavioural modes, instead of all possible combinations. Consider 13 different inputs, each one having three behavioural modes, which means that there are totally $3^{13} = 1594323$ combinations and equally many tests. Only 15 tests are required to test all pairs [8].

Pairwise Testing has been proved (not theoretically, but by experience) to be efficient, both in reducing the number of test cases and detecting errors [8].

Example 3.4

Consider three signals, u_1, u_2 and u_3, with just two behavioral modes. Testing all possible number of behavioral modes would require $2^3 = 8$ test cases, which will give 100 % coverage. Using Pairwise Testing only all pairs of behavioral modes are tested, which will require only four test cases; $T_1 \dots T_4$. See Figure 3.3.

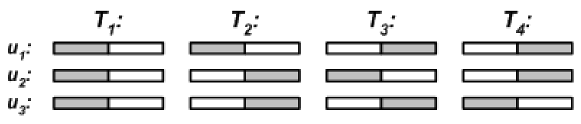


Figure 3.3. The gray box means that this particular behavioural mode for the signal u_i is under test. All pairs of behavioural modes are tested at least once in the four test cases T_j

There is no direct proof that Pairwise Testing is more efficient, it just turns out to be that way [8]. The Pairwise Testing includes all errors caused by one or two fault modes, but misses if there are three or more. The probability of more than two signals going wrong versus the cost of executing every combination has to be evaluated.

3.6 Risk Based Testing (RBT)

To conclude the chapter a method called Risk Based Testing (RBT) will be described, which is used to plan what to test and how. It is impossible to test everything according to what was written in the introduction to this chapter. With RBT the objects under test are classified depending on the risk the objects being erroneous. Risk is correlated with coverage, meaning the higher the risk the higher coverage and deeper level of testing.

To classify into risks two parameters can be used, consequence and probability. Each parameter has two categories, High or Low, meaning that there are four distinct classes. See Table 3.1:

Table 3.1. A table with the four distinct classes

Class	Probability	Consequence
A	High	High
B	Low	High
C	High	Low
D	Low	Low

Probability is a measure of the rate of occurrence of fault and can be described by for example chance of failure, usage frequency and complexity. Consequence is a measure how severe the effect is during failure and can be described by Vehicle Of Road, safety problems, sales volume, etc.

Using the classes in Table 3.1, Figure 3.4 can be drawn to illustrate the RTB.

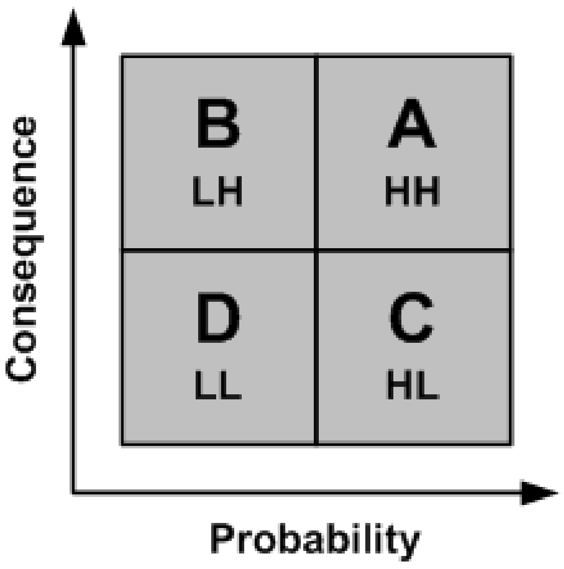


Figure 3.4. The four classes as squares in a coordinate system with probability and consequence as axles.

Depending on the class the test object is classed into, different test techniques are used to get the appropriate depth of testing. Class A is the most important class to test, consisting of objects with high severity and high probability of failure, and must be tested extensively. Class D in the other end are not needed to be tested as deep.

Suggestion of how to perform tests can be found in [12] and is summarized in

Table 3.2.

Table 3.2. A table with the four distinct classes

Class	Level of testing
D	Positive testing based on specifications
C	Level of testing for D + Testing edges of positive testing
B	Level of testing for C + Negative testing with inputs outside valid areas and not in specifications
A	Level of testing for B + Provoking the system to fail

The level of testing according to Table 3.2 is now described:

A: Positive testing
Testing according specifications. Only valid inputs within the MSC are used. Positive testing is currently performed in I-Lab2.

B: Edges of positive testing
Tests inputs according to Boundary Value Testing and is briefly described in Example 3.3.

C: Negative testing
Meaning that for example {Unrealistic} values are tested, that are outside valid areas.

D: Provoking the system to fail
Any means necessary to provoke the system to fail meaning multiple inputs are outside valid areas.

Using RBT it is easy to determine the level of testing, and what to test.

3.7 Summary

Some basic concepts of testing and the methods required to perform testing has now been described. Testing is performed to measure the quality of the product, to ensure the product being fail-safe. There exists in general two different test strategies, Black box (Section 3.3) and White box (Section 3.4), both with their advantages and disadvantages, making them suitable for different systems and on different levels in the development process.

Two of the test methods mentioned, Equivalence Class Testing (Section 3.5.1) and Pairwise Testing (Section 3.5.2) is both based on Black box. Both methods reduce the number of test cases, and one is especially suitable for multiple inputs. Risk Based Testing (Section 3.6) is not a test method but at way to plan what to test.

Chapter 4

Testing degradation of distributed functionality

The following chapter will describe some of the problems when testing degradation of distributed functionality. The information acquired is based on internal documents, interviewing personnel at Scania and an own analysis. The arisen problems will be discussed, which will form the base for the next chapter where suggested solutions to these problems will be discussed.

But first some background to the problem.

4.1 Background

The entire electrical system consists of a large number of ECUs. Each ECU sends and receives a large number of signals through three independent CAN-buses, to be able to execute up to 400 user functions.

The number of ECUs, signals and user functions depends on an almost infinite combination of vehicles, where each ECU has between 50 - 10000 parameters. The configuration of vehicle has not been considered, and should not have any impact on the problem in this thesis. A suggested test method should not depend on the specification of the vehicle to test.

It is also impossible to test every fault mode and distributed function. Any kind of classification system determining what function or fault mode to test will not either be considered in this thesis. A short review about the possibilities about automatically generating such classification system has been made without any result worth mentioning. Some thoughts will be mentioned in Section 4.5 using Risk Based Testing (Section 3.6).

Another question is if the 400 user functions could be classified into groups and if these groups could be tested in a certain way. That could lead to distinct test methods and test scripts that would focus on risks within the function class. A more concentrated and efficient method of testing user functions could be developed. Some thoughts about the subject can be found briefly in Section 4.5, but will otherwise not be regarded.

Two examples will now follow, to display some of the problems testing distributed degradation that will later be discussed.

Example 4.1

UF98: Fan Control Figure 4.1 displays the entire user function UF98. In reality UF98 is divided into several MSCs, but is here illustrated as one function.

Several ECUs need cooling from the fan connected to the EMS. The COO receives information from a number of ECUs and then calculates a rotation speed for the fan for every ECU that needs cooling. The COO then chooses the highest calculated rotation speed and requests it to the EMS, which then tries to control the fan to requested speed. The EMS also has an internal fan request.

The big question is: How to best test degradation of this particular function? Or in other words: How to best verify that this function is behaving correctly when there is an active fault mode?
There are many fault modes that could affect Fan Control, every sensor used by the function could for example be short circuited, open circuited or measure a completely wrong sensor value. These fault modes affect the corresponding CAN-signals by entering any of the five behavioural modes {Defined, Undefined, Error, NA, TO}. What relation is there between fault modes and behavioural modes? See Section 4.4.

It is the electrical system that shall be tested, or in other words the communication on CAN, meaning that every possible combination of behavioural mode should be tested for 100% coverage. It is not certain that every behavioural mode even is possible to generate in a HIL-lab or that there is enough time and resources to test them all. Is there any realistic way of decreasing the number of test cases? See Section 4.4.

When testing, an expected output should always be stated. With a fault mode activated within the function the COO will receive some behavioural mode for some signal on CAN. It has to be determined how the COO behaves, and what affect that causes the EMS. The document describing the UF98 does not include information how the COO treats {Undefined, Error, NA, TO} for any signal, which requires reading other documents and finally requesting documents from the developers of the COO. It is difficult to determine what happens. Is there any easier method determining if a test has passed or failed? See section 4.2.

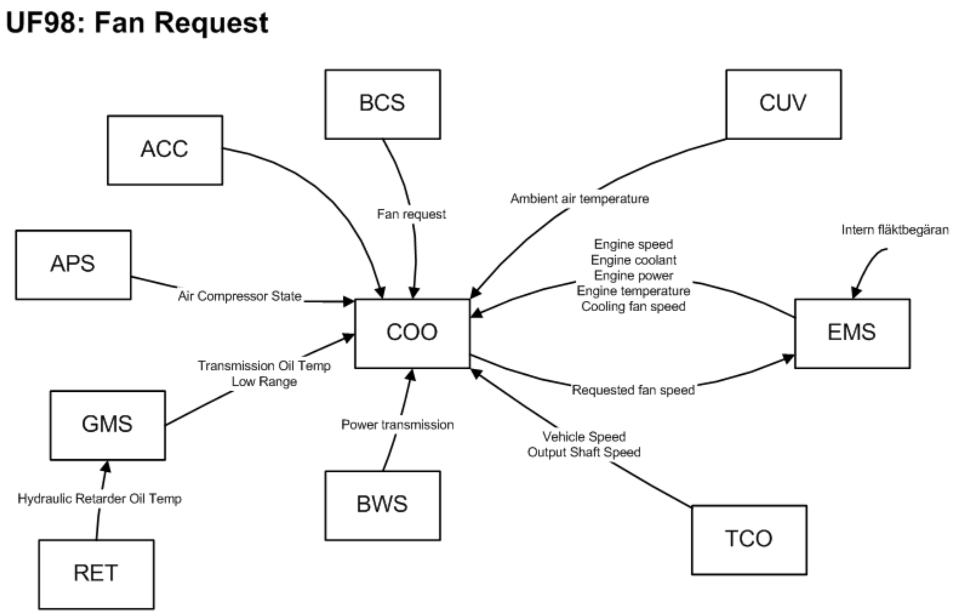


Figure 4.1. The entire UF98: Fan Control. The squares represent ECUs and the arrows represent signal paths. The names on top of the arrows are signals sent between the ECUs.

It cannot however be overruled that just fault modes outside the function cause degradation within the function, that is not specified in any documentation. There could be an 'invisible' ECU, for example AUS mentioned in Example 1.1, not drawn in Figure 4.1 sending some faulty signal to the APS, which in turn affects the signal 'Air Compressor State'. The opposite could also be stated, if UF98 could affect any other part of the system not drawn in Figure 4.1, for example that the radio in AUS stops working. Is there any unspecified error propagation in the electrical system and how could it be detected and tested? See Section 4.3. Error propagation will only be considered in one of the later suggested test techniques (Fault Mode Test) in Chapter 6.

Instead of choosing one User Function, just one sensor can be considered:

Example 4.2

Engine Speed Sensors
There are two sensors attached to EMS that measures the engine speed, which then generates the CAN-signal EngineSpeed. The EngineSpeed is transmitted on CAN and then received by over fifteen ECUs which is then used by some functionality.

Consider that some fault mode involving the engine speed sensors are present. It has to be determined how CAN is affected, what behavioural mode EngineSpeed signal is transmitted. Can every behavioural mode, {Defined, Undefined, Error, NA, TO}, be generated and degradation tested for all of these modes? See Section 4.4

Since the EngineSpeed signal is received by over fifteen ECUs many documents has to be read since there is no compiled document about the effect of a faulty EngineSpeed signal. It is not certain that all of these documents contain satisfied information how to test the behaviour of the receiving ECUs. See section 4.2.

Can any of fifteen earlier mention ECUs causes failures elsewhere in the electrical system, by transmitting degraded signals? See Section 4.3.

The examples above discuss problems about testing distributed degradation in general. A workable method is then needed to be developed to be able to test distributed degradation on a regular basis. That problem will be discussed later in Chapter 6.

The problems found that will be discussed will now be described in more detail.

4.2 Expected outputs

There are some existing documentation with a description about signals and what they affect. The CAN communication specifications contains detailed information about which ECUs transmits and receives which messages and signals. With each signal the intervals for the behavioural modes {Defined, Undefined, Error, NA} are defined. The document does not describe what affects a signal or what it is used for.

The FMEAs (Section 2.1.2) grades the failure effect for different behavioural modes for signals, describing the usage of the signal. What specific functionality a signal affects is not described. There is not enough information in the FMEAs to determine expected outputs when performing degradations tests.

The MSCs contain all signals that are used within a scenario. These are probably the best source to determine the usage of a signal. The MSCs are only valid in the fault free case. No alternative signals paths can be found within the MSC during failure, meaning that the MSCs cannot be used to determine expected outputs when testing degradation. To determine expected response the tester has to tend to the MSC related document; a specification of the user function. The problem is that these specifications in general do not include information about the behaviour when receiving signals being {Undefined, Error, NA, TO}. Also, MSCs and user functions only describe around 80 percent of the all distributed functionality [13].

None of the above mentioned documentations are especially useful when determining expected outputs for some function when testing degradation.

Exact behaviour of some function can be acquired, but requires substantial work talking to personnel all around Scania and requesting internal documentation that is only accessed by people developing the ECUs. That method proved inefficient and another approach is needed. Instead of determining exact behaviour of a function, acceptable and unacceptable behaviour can be stated. With acceptable criteria general rules can be stated and tested against. More about acceptance criteria in Section 5.3.

4.3 Error propagation

As mentioned in Section 2.1.1 failures in turn create other failures, meaning that error propagation can exist. There is a need to investigate how error propagation and unspecified degradation effects exists in the electrical system.

Only error propagation within the electrical system will be considered. Another possibility is malfunctioning control loops, where the failure is propagation via the environment into the electrical system. This will not be considered in this thesis but could be covered with methods later presented (see Section 6.2).

Within the electrical systems it is CAN-signals that generate error propagation. Signals could simply be classed into two signal types, sensor signals and computed signals. Sensor signals are signals that are generated from sensor values, meaning that it is a direct map from what a sensor measures. These sensor signals are the sources so to speak and the origin can be determined. Computed signals are signals that are generated from other signals, meaning that there are more than one source of information. How computed signals are generated are not easily determined (compare with Section 4.2), and could be a source of error propagation. How does a computed signal behave if one of the input signals has the behavioural modes {Undefined, Error, NA, TO}?

A compilation of the CAN communication specifications could be made to gather these eventual error propagation effects. A matrix, called ECU vs. ECU matrix, usable when determining error propagation will be described in Section 5.2.1.

4.4 The number of behavioural modes

As stated in Section 2.3.2 there are totally five different behavioural modes that can describe the state of a signal:

$$u \in \{u_{def}, u_{undef}, u_{error}, u_{na}, u_{to}\}$$

The number of behaviour modes leads to a drastically increase of test cases with the number of inputs to achieve high coverage. Consider n inputs to some function. Using Equivalence Class Testing (Section 3.5.1) the total number of combinations of behavioural modes are 5^n. Even for $n = 3$ the number of test cases is uncontrollable. Instead using Pairwise Testing (Section 3.5.2) will drastically reduce the number of test cases, but will still produce more test cases than desirable. A method of decreasing the number of combinations could be to decrease the number of behaviour modes. What is a reasonable loss of coverage with testing fewer behaviour modes?

There also is a wish to provoke the ECUs to send different behavioural modes on CAN by introducing fault modes on components. The reason is that during system and integration test the complete chain is under test, from ECU sending the correct signal to another ECU receiving it and process it in a correct way. Direct manipulation of CAN is of no interest which would just test that the ECU receiving the signal behaves correctly.

Fault modes on components will now be introduced to force different behavioural modes on signals. Assume that the only fault modes (components) considered are these that affect sensors, ECUs and CAN-buses. The behavioural modes on components, f, can be described as:

$$f \in \{\text{NF, Sensors, ECUs, CAN}\} \tag{4.1}$$

Example 4.3

Consider a sensor signal, u, that is a direct map from a sensor value, $s(t)$. A sensor value is the value a sensor measures. The relation between the sensor signal and the sensor value can be stated as $u = s(t)$. Introducing behaviour modes on components, f, will according to Section 2.3.3 and Section 2.3.2 create different behaviour modes on the sensor signal as $u = u(s(t), f)$, meaning that the signal depends on both the sensor value and the status of the sensor.

The following scenario could be possible:

$$u = u_{def} = s(t), \qquad f \in \{\text{NF}\} \tag{4.2}$$
$$u = u_{error}, \qquad f \in \{\text{Sensor faulty}\} \tag{4.3}$$
$$u = u_{to}, \qquad f \in \{\text{ECU disconnected, CAN disconnected}\} \tag{4.4}$$

Note that equation (4.2) is just an example. It is part of the thesis to investigate the relation. This will further be discussed during the implementation in Section 7.1.2 where it will be described that it is difficult to generate all possible behavioural modes of signals in a HIL-lab.

4.5 Complexity of user functions

There are over 400 active user functions, each with a number of use-cases consisting of a number of MSCs making it difficult to test them all at once. There has to be some way of determine which MSCs that are going to be tested. There is already a solution for that; Risk Based Testing in Section 3.6; which is used today to classify the MSCs.

A way to graphically represent which user functions are affected by some fault mode can be found in Section 5.2.2.

The complexity of the user functions is a problem. Depending on the number of inputs the number of test cases will increase to get good coverage. Compare with behavioural modes of signals in Section 4.4. What shall be tested?

Another matter briefly investigated is if MSCs can be collected into different classes and if different techniques of testing could be performed depending on the class. How different classes of a MSC reflect the level of testing will just be mentioned here but will not later be tackled. Time issues is the reason for the exclusion of that particular problem. The two examples below displays some thoughts about the problem, showing which fault modes must be activated to test the MSC.

A large number of MSCs can be described with just serialized communication, with one input. The MSCs gives a limited number of test cases as the following example will show:

Example 4.4

UF109/SCN52: Display engine oil pressure has only the engine oil pressure as input.

UF109 means that it is user function 109, and SCN52 means that is it scenario 52.

The scenario has only one fault free state, which displays the engine oil pressure in the ICL. A number of degraded states exist depending on the different fault modes. The fault modes of interest are: {Engine oil pressure sensor broken, EMS broken, COO broken}, meaning that there could be three different degraded states. A broken ICL is of no interest since the outcome is obvious; no engine oil pressure display in ICL. Fault mode on the actuator ECU is of no interest.

Multiple faults are of no interest either. Consider the fault mode {Engine oil pressure sensor broken & EMS broken} is equivalent to just {EMS broken}. If the EMS is broken, no CAN traffic is transmitted from the EMS, meaning that the state of the engine oil pressure does not matter.

Another large group are MSCs that describes control loops.

Example 4.5

UF98/SCN1039: Request engine fan from APS is a AFC 3.

The Air Pressure System (APS) sends a request to the Engine Management System (EMS) to start the engine fan. The APS is dependent that the EMS does respond correctly, since otherwise there is a risk of overheating the air pressure system.

The problem is more complex since there is an invisible correlation between output and input that is not seen in the MSC. Testing how the function behaves if the EMS is disconnected, or the fan stops working must be performed. In other words fault modes regarding actuators is now also of interest. Could there be instability in the control loop due to some fault mode? As mentioned earlier, these questions will not be tackled in this thesis.

4.6 Summary

There are five different difficulties when testing degradation: Difficulties determine expected output, error propagation, number of behaviour modes, number of MSCs and the complexity of an MSC. The first four will be tackled in the next chapter. The fifth, complexity of an MSC has not been considered further in this thesis because of time issues.

Chapter 5

Suggestions testing degradation of distributed functionality

Solutions for four of the five problems found according to the previous chapter are suggested.

A brief motivation which behavioural modes needs to be considered is discussed in Section 5.1. The communication matrices in Section 5.2 is a tool to make the electrical system easier to overview. Finally the chapter ends with the acceptance criteria in Section 5.3 describing what to test and how to verify the outcome of the test.

Solutions presented will be used in the two test strategies in chapter 6.

5.1 Behavioural modes

As described in Section 4.4 the high number of defined behavioural modes generates large number of test cases. A way of decreasing the number of test cases is to decrease the number of behavioural modes under test. A suggestion which behavioural modes that shall be tested will now follow.

As said earlier there is a wish not to manipulate CAN directly, but instead force the ECUs themselves to send different behavioural modes on signals. In this thesis it is assumed that all sensors values are correct. Detecting errors in the sensor values is a subject on diagnosis, which is not discussed here. With that motivation, unrealistic values will not be considered, and will not be regarded as a behavioural mode (Section 2.3.2). The ECUs could be faulty and send unrealistic values on CAN, despite reading a correct sensor value. Such behaviour should have been validated during ECU System Level Tests (unit tests); not system and integrations test. It is also illegal to send {Undefined} on CAN and is also a subject of ECU System Level Test. With the motivation above neither {Unrealistic} or {Undefined} will be provoked on CAN. Validation that these modes are not present on CAN should be performed for any signal on CAN while executing a test.

According to Section 4.4, the fault modes on components that will be considered is $f \in$ {Sensors, ECUs, CAN}. Almost certainly {Error, NA, TO} will be provoked on CAN. Studying FMEAs, many ECUs do not consider {Error} and {NA} as two independent modes, and usually are programmed to behave in the same way. Still there are many signals where the ECUs treat {Error} and {NA} in different ways. There is a balance between cost and coverage. Combining {Error} and {NA} into {Error/NA} will generate $4^n - 3^n$ less combinations with the consequence of less coverage. With the difficulties of generating some behavioural modes in a HIL-lab described in Section 7.1.2 the assumption that {Error/NA} can be treated together will now be used. Collecting {Error} and {NA} into one behavioural mode means that if {Error} is provoked for one signal , {NA} will not be provoked for the same signal, or vice verse. A valid set of behavioural modes that will be tested will be: {Def, Error/NA, TO}. See Table 5.1.

Table 5.1. Table with motivations for decreasing the number of behavioural modes

Behavioural mode	**Provoke**	**Motivation**
Unrealistic	No	Faulty sensor values are not considered. Should be validated during ECU test that {Unrealistic} signals never are sent.
Undefined	No	Should be validated during ECU test that {Undef} signals never are sent
Error/NA	Yes	According to FMEA, many ECUs treat Error and NA the same and will be considered the same equivalence class. Error and NA will not be tested separately.
Time out	Yes	Is easy to provoke, and should be tested

Note that both {Error} and {NA} usually are defined as intervals (see example 3.1). But applying theory from Equivalence Classes (Section 3.5.1) there is no difference between different values of {Error/NA}.

Using the behavioural modes {Defined, Error/NA, TO} instead of {Defined, Undefined, Error, NA, TO} decreases the number of combinations for n signals with $5^n - 3^n$. It will also later discussed in Section 7.1.2 that the behavioural modes chosen will simplify which fault modes to test in a HIL-lab.

5.2 Communication matrix

Two kinds of matrices have been developed to give an overview of the failure effect using a graphical representation. The two matrices described are 'ECU vs. ECU' and 'Component vs. UF'. Each matrix is described with fault modes as the columns and the failure effect as the rows.

The 'ECU vs. ECU' matrix could be used to trace error propagation described in Section 4.3. The 'Component vs. UF' matrix shows graphically which UFs are affected by some fault mode and could simplify the choice of MSCs to test according to Section 4.5 about the complexity of user functions.

None of these two matrices includes any new information but is just a simplification of existing documents. The 'ECU vs. ECU' matrix is based on CAN communication specifications and FMEAs (Section 4.2). The 'Component vs. UF' matrix collects information from the MSCs.

5.2.1 ECU vs. ECU matrix

The 'ECU vs. ECU' matrix uses the fault mode {ECU disconnected} versus the failure effect in other ECUs. The fault mode {ECU disconnected} also corresponds to if all signals sent from the ECU has one of the following behavioural modes: {Error, NA, TO}. The matrix can be automatically generated using the internal CAN database containing which ECU communicating with which. The failure effect of loosing one ECU has been graded with the scale according to Table 5.2.

The FMEA could be used to help grading the failure effect of {ECU disconnected} according to Table 5.2. No new information is acquired by using the grade according to Table 5.2, but is adapted to degradation and can be more easily be used for general acceptance criteria. The grade does not consider the effects outside the ECU, to what degree failure will cause harm to the driver or environment. But broken communication with one ECU is not always considered a fault mode in FMEA. Instead interviewing people responsible for the ECUs were performed, to grade the failure effect.

The main advantage with the 'ECU vs. ECU' matrix is that it can help which ECU and what kind of functionality to test. If one ECU is lost it can easily be seen in the row of the disconnected ECU, which other ECUs will be affected and how severe. Because of the overview of the complete electrical system effects from error propagation can be hinted (see also Section 4.3).

One example showing the application of the 'ECU vs. ECU' matrix:

Example 5.1

Using Figure 5.1, consider that BMS is disconnected. Looking at the row with BMS it can be seen that those ECUs that are affected the most are EMS and CUV, that both are graded 5. Functionality contained in these ECUs should be

Table 5.2. A sixth graded scale classifying the effects a particular component/signal has to an ECU

	Description	**Expected failure effect**
1.	Data from the disconnected ECU is not used for anything	No effect
2.	Data from the disconnected ECU is only used when storing DTCs.	No degraded functionality
3.	Data from the disconnected ECU is redundant, or is used as a parameter for a control loop.	Reduced efficiency
4.	Data from the disconnected ECU is used as a reference for a control loop, or to request functionality	Loss of functionality, that is not safety critical.
5.	Data from the disconnected ECU is used for safety critical functionality	Loss of safety critical functionality or main functionality within the ECU
6.	The ECU is completely dependent on the disconnected ECU	Total collapse of functionality (including safety critical functionality) within the ECU. Vehicle Off Road (VOR) could also be expected.

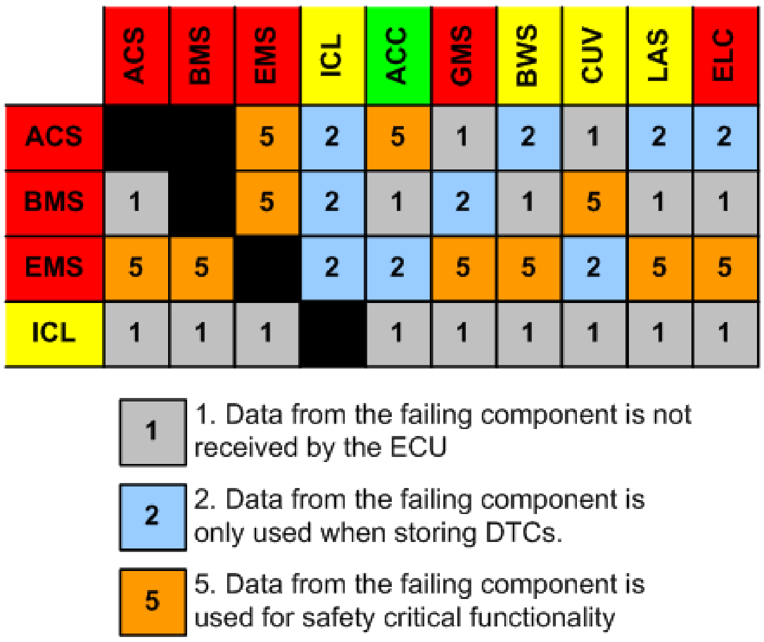
prioritized during testing since information received by the BMS is used for safety critical functionality.

The next step is to see that EMS in turn affects several other systems graded with a 5. Because of error propagation these systems could be prioritized during testing as well.

5.2.2 Component vs. UF matrix

Another matrix is the 'Component vs. UF' matrix describing failure effects on user functions using fault modes on individual component. The matrix gives an overview which user functions are affected by a particular component. See figure 5.2. The failure effects can be graded, using for example severity, but have not been studied in this thesis.

The 'Component vs. UF' graphically presents possible failure effects on user functions and may choose the most important user functions to test regarding some fault mode. Since MSCs contains information such as components used within the corresponding scenario, the Component vs. UF matrix could be automatically generated using MSCs.



	ACS	BMS	EMS	ICL	ACC	GMS	BWS	CUV	LAS	ELC
ACS	■	■	5	2	5	1	2	1	2	2
BMS	1	■	5	2	1	2	1	5	1	1
EMS	5	5	■	2	2	5	5	2	5	5
ICL	1	1	1	■	1	1	1	1	1	1

1	1. Data from the failing component is not received by the ECU
2	2. Data from the failing component is only used when storing DTCs.
5	5. Data from the failing component is used for safety critical functionality

Figure 5.1. An example showing the concept of the ECU vs. ECU matrix

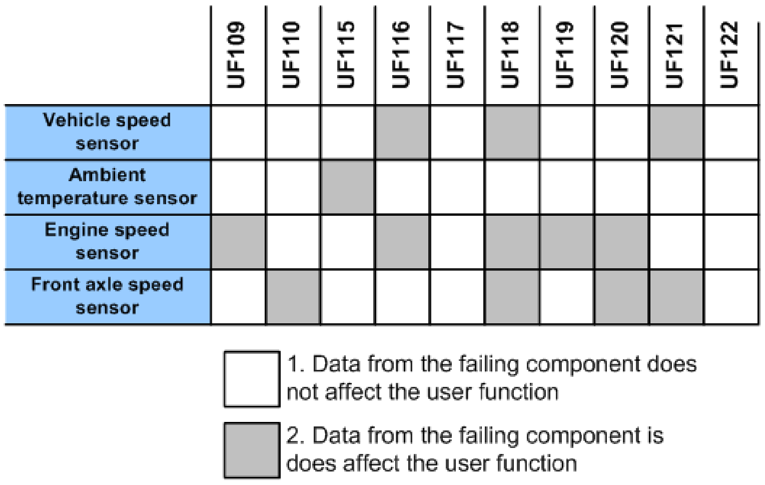
Example 5.2

Consider that the engine speed sensor is disconnected. Using Figure 5.2 it can be seen that five User Functions are affected by the engine speed sensor. All these user functions should be tested to verify that they do not behave incorrectly according to Section 5.3. Consider that all User Functions are implemented as test scripts, some program could automatically gather the relevant user functions and test them based on the fault mode.

If there is not time to test them all, the Risk Based Testing in Section 3.6 can help prioritize what to test.

5.3 Acceptance criteria

According to both the Black box test strategy in Section 3.3 and the White box test strategy in Section 3.4 actual outputs are compared to expected outputs. According to Section 4.2 it requires extensive time compiling information about



	UF109	UF110	UF115	UF116	UF117	UF118	UF119	UF120	UF121	UF122
Vehicle speed sensor	□	□	□	▨	□	▨	□	□	▨	□
Ambient temperature sensor	□	□	▨	□	□	□	□	□	□	□
Engine speed sensor	▨	□	□	▨	□	▨	▨	▨	□	□
Front axle speed sensor	□	▨	□	□	□	▨	□	▨	▨	□

□ 1. Data from the failing component does not affect the user function

▨ 2. Data from the failing component is does affect the user function

Figure 5.2. An example showing what the Component vs. UF matrix could look like

correct output during degradation. Acceptance criteria have to be introduced to specify what an acceptable behaviour, or what an unacceptable behaviour is. After a test case is executed it is determined to what degree these criteria have been fulfilled. The rules creating the acceptance critera must be so general that they should hold as good as always. However it turns out that it is difficult to set up these general rules, but must depend for example on the state of the vehicle, i.e. if the engine is on or not. From now on because of simplicity it will be assumed that the engine is on, and the vehicle is in a driveable state.

The acceptance criteria will be based upon some rules that have to be followed. These rules will be denoted with **R** as in rule or requirement. A way to test these rules will also be suggested and will be denoted **T** as in test, and will be used in the two suggested test techniques in Chapter 6.

Information about proper acceptance criteria and what an unacceptable behaviour is has been gathered when talking to people knowing the systems in the vehicle. Four categories of acceptance criteria will be used trying to categorize the rules later set:

- Fault codes (DTCs)/Warnings in ICL (displayed to the driver)
- Functionality/Driveability
- Security
- Communication

Sixteen rules, **R1-R16**, have been suggested. These rules could hold as good as always under the prerequisite that the engine is on and the vehicle is in a

driveable state. The rules **R1-R4** about DTCs/Warnings are based on material in Section 2.4 and Requirements on diagnostic functionality [5]. The rules **R12-R16** about Communication are based on Section 2.3.3 and the SESAMM-concept [9]. Rules **R5-R11** about Functionality/Driveability and Security are suggested by personnel at Scania and are just examples of acceptance criteria in these categories (see Section 5.3.2 and Section 5.3.3 for further explanation).
A more detailed description of the four categories will now be discussed, with reference to Chapter 2.

5.3.1 DTC/Warnings

According to Section 2.4, a primary DTC has to be set when sending {Error} on CAN and a secondary DTC has to be set when receiving an {Error} on CAN that degrades functionality within the own system. It has to be validated for every test that these rules are followed and that DTCs not allowed to be set are never set. Because of error propagation (Section 2.1.1) one fault mode can lead to many different failures, which in turn can lead to other failures in other systems. It cannot be overruled that an unexpected DTC, to the tester unrelated to the present fault mode, is not set correctly. Every DTC set during a failure should be investigated, and be traced back to its source.

Depending on the severity of the failure an appropriate warning should be sent to inform the driver. Notices in the Instrument Cluster Panel should be read during test and evaluated.

Summarized as distinct rules:

R1.	A primary DTC has to be stored within the ECU connected to the faulty component
R2.	A secondary DTC has to be stored for every ECU receiving information about the faulty component, that will lead to degraded functionality
R3.	No DTC is allowed to be stored in ECUs that are not using information from the faulty component
R4.	Appropriate warnings and notices has to be set in the Instrument Cluster Panel to communicate failures to the driver

To be able to evaluate these four rules two actions should be executed during the test:

T1.	All active DTCs should be read continuously during the test case
T2.	All warnings in the ICL should be read continuously during the test case

5.3.2 Functionality/Driveability

Basic driveability functionality such as acceleration, braking and steering should be as robust as possible. These functions should be tested accordingly.

Components not directly connected to the MSC should not have an effect on that particular MSC, or otherwise the MSC is wrong. Using the Component vs. UF matrix (Section 5.2.2) gives an overview which fault modes should (or should not) have any effect on particular user functions. It cannot be overruled that a degraded user function will not affect another user function that is not specified anywhere. MSCs not affected by some fault mode according to the 'Component vs. UF' matrix could be tested as well to verify that no failure effect is present in that MSC.

ECU specific functionality is also of interest, in what way an ECU is affected by a fault mode. The category is directly correlated with the ECU vs. ECU matrix in Section 5.2.1. Tests should confirm if main functionality within an ECU is working properly or not. An example could be the Audio System; to verify that the radio is still working and that the volume can be raised or decreased from the audio buttons attached to the steering wheel. That particular functionality is not safety critical, but are of great importance to the Audio System itself.

Error propagation is a reason why to test acceptance criteria stated on functions. Is there any way the radio can stop working, especially if a fault mode occurs in a system that does not communicate with the Audio System? With the use of the ECU vs. ECU matrix these error propagation effects can be seen, by choosing ECU1 and see that it communicates with ECU2 but not with ECU3. But ECU2 communicates with ECU3. The question is: Can a fault mode in ECU1 cause any failure in ECU3? See also Example 5.1 for an illustration about error propagation and why to test functionality in ECU3.

The following rules has been suggested and discussed when talking to personnel at Scania:

R5.	The vehicle should always be able to brake to stand still in a controlled manner
R6.	The vehicle should as far as possible be able to accelerate in a controlled manner
R7.	The vehicle should always be able to maneuver
R8.	Functionality with no connection to the fault mode should be executed as usual
R9.	The vehicle should as far as possible not enter the fault mode Vehicle Off Road (VOR)

T3.	Error propagation should be investigated
T4.	Test acceleration, braking and maneuvering
T5.	Test appropriate functionality according to the Component vs. UF matrix
T6.	Test appropriate functionality according to the ECU vs. ECU matrix

The rules above are just suggestions of acceptance criteria set on functionality and driveability, and could be expanded to cover more of the functionality within the vehicle. The reason for choosing as specific rules as **R5-R7** are because they are easy to test in a HIL-lab, and because that this basic main functionality should be as robust as possible. **R8** and **R9** are of the more general kind and states what an unacceptable behaviour is. **R8** is based on that functionality shall behave as usual if the activated fault mode is not in the specification. Vehicle Off Road (VOR) means that the vehicle will not longer be in a driveable state and no longer fulfill the prerequisite 'engine is on and the vehicle is in a driveable state' and must be prevented as far as possible.

5.3.3 Security

The vehicle must not be dangerous to the driver or its environment (see Section 2.1 about fail-safe). The acceptance criteria for security was more difficult to determine, especially when these criteria are performed in a lab. Some examples of acceptance criteria related to security are:

R10.	The vehicle should always be able to brake to stand still in a controlled manner
R11.	The engine shall not be able to rev unprovoked and uncontrollably
T7.	The engine speed should be logged for every test case and evaluated
T8.	The vehicle speed should be logged for every test case and evaluated

Exactly as the acceptance criteria set on Driveability/Functionality, the criteria set on Security are just suggestions and could be expanded. The two rules suggested are easy to verify in a HIL-lab.

5.3.4 Communication

There are distinct rules set on communication, when different behavioural modes on signals are allowed to be set. According to Section 2.3.3 the following rules can be set:

R12.	The SESAMM-concept [9] always has to be followed
R13.	Every ECU should transmit every signal on CAN, except when impossible (for example if a ECU is disconnected)
R14.	No {Undef} are allowed to be sent
R15.	Every gatewayed signal must be consistent on every CAN bus
R16.	No signal may contain information that in any way does not reflect the truth
T9.	Log every signal on CAN during the test case
T10.	Compare all relevant signals directly in the test script according to the rules above

5.4 Summary

The numbers of behavioural modes have been discussed, and with motivation that the three behavioural modes needed to be tested are: {Defined, Error/NA, TO}.

Communication matrices are a more perspicuous way of describing the failure effect from ECU perspective, than an FMEA. There are two kinds of matrices ECU vs. ECU, and Component vs. UF. The former matrix is used to grade the severity of failure effect and can be used to track down possible error propagation paths. The Component vs. UF matrix can be used to determine which UFs to test depending on the fault mode.

Last but not least acceptance criteria have set up as general rules as possible of what is allowed and not allowed. Tests are suggested to verify the outcome of these criteria and will be discussed in the next chapter. There are four categories that will be investigated: DTC/Warnings, Driveability/Functionality, Security and Communication.

It is now time to discuss test strategies that can be used to test degradation.

Chapter 6

Two types of test techniques

One object in this thesis is: 'Suggest a test method to efficiently verify that the degradation of functionality behaves correctly'. The problems discussed in Chapter 4 and the suggested solutions in Chapter 5 does not fulfill the above objective, but are just discussing generally about testing degradation. A new question arises: How shall degradation test be performed in a HIL-lab?

The suggested solutions in Chapter 5 must now be applied and implemented to form a structured test technique usable when testing degradation on a regular basis. Two types of test techniques for testing degradation will now be suggested; Degraded User Function Test (DFT) and Fault Mode Test (FMT). These test techniques have different area of application. Methods and theory described in previous chapters will be used to explain both techniques.

Concepts used in the chapter must first be explained:

Test technique is the process from first choosing what to test to the test result.

Test case is the actual test written as text, how to force some behaviour and what response to expect. The test case is always written before the test is executed.

Test script is the test case transformed to code, being able to run the test in a HIL-lab.

User Function Test (UFT) are tests that tests an MSC positively, verifying correct behaviour in a fault free environment. Compare with class D used in Section RBT 3.6.

6.1 Degraded User Function Test (DFT)

Degraded User Function Test (DFT) has much in common with today's ordinary User Function Test (UFT) with the exception that the vehicle will no longer be in normal operation mode.

The suggested approach for a Degraded User Function Test is:

1. Choose an MSC and study it carefully. The choice of MSC could be based on Risk Based Testing (Section 3.6).
2. Identify all possible fault modes within the MSC according to equation (4.1).
3. Using Risk Based Testing decide the depth of testing. Use Pairwise Testing (Section 3.5.2) if considering multiple faults
4. Write the test case according to Section 6.1.1.
5. Compare the responses with the acceptance criteria (Section 5.3) and expected behaviour according to the MSC.

The goal with the DFT is to test degradation of one MSC. Only fault modes within the MSC should be identified and injected. Multiple faults can accord to Pairwise Testing in Section 3.5.2 be executed with a controllable number of test cases. The acceptance criteria that should be verified should be within the scope of the test; DTC/Warnings and communication within the MSC and the behaviour of the MSC under test.

Discussing item 3 in the list above; RBT can be used to decide the depth of testing. As written in Section 3.6 different test levels can be used depending on classification. A suggestion could be that for MSCs classed as 'A' shall be tested with Pairwise Testing for all behavioural modes. MSC classed as 'B' shall be tested with only single faults.

6.1.1 DFT: Test case

The same test cases will be iterated with different fault modes present. A reset of the system is required between the test cases and fault modes, since failure effects from previous fault modes are unwanted. Functions with many inputs with many possible fault modes will create several resets and iterations. See Figure 6.1.

The process of the test case for DFT is:

1. Prerequisites
2. Inject fault mode
3. Run test of MSC
4. Postrequisites
5. Restart from beginning, choosing another fault mode

6.1.2 DFT: Area of application

Degraded User Function Tests are suited to test new MSCs and should be written in parallel with ordinary User Function Tests (UFT). The reason is that the tester writing a UFT, has already acquired information about the MSC under test and will therefore write the DFT more efficiently if done directly. If the test scripts are written properly they can be used for both Degraded User Function Tests and ordinary User Function Tests. The only difference is the fault mode and the expected response.

Depending on which class the MSC is classed into according to RBT in Section 3.6, the depth of the Degraded User Function Test can vary. The number of fault modes that shall be tested is the choice of the tester writing the test case. Only fault modes within the test should be considered, not loosing the scope of the test. Already written test cases can be used, only needing to include fault modes and determine expected responses.

6.2 Fault Mode Test (FMT)

The Fault Mode Test (FMT) is based on one fault mode which then determines which functions to test. The process can be described as:

1. Use the Component vs. UF matrix in Section 5.2.2 to choose a fault mode and then identify all possible MSCs that can be affected by the fault. Use the ECU vs. ECU matrix in Section 5.2.1 to identify possible ECUs that are affected by the fault.
2. Use RTB in Section 3.6 to decide which functions to test from the previous item. Functions in class A and B should be prioritized.
3. Implement the test case and test script of the functions to test in a way that they later can be reused
4. Write the test case and test script, with prerequisites and with the function test modules, driveability test modules and security related test modules
5. Compare the responses to the acceptance criteria (5.3) stated on function test modules
6. Use the acceptance criteria on DTC/Warnings, communication, driveability and security related functionality.

The main difference between DFT and FMT is that FMT is based on one fault mode instead of one MSC. Since functionality is tested in many parts of the electrical system, FMT is suited to investigate the failure effect and error propagation in the entire vehicle. More general acceptance criteria have to be used to investigate the entire electrical system, not just within one function. The criteria for Driveability/Functionality, Security and ECU specific functionality must also be satisfied, aside from DTCs/Warnings and communication.

6.2.1 FMT: Test cases

Compared with the test cases in DFT, there is no iteration of the entire test case. Each test case is implemented as its own. See Figure 6.1.

The process of the test case for FMT is:

1. Prerequisites
2. Inject fault mode
3. Run tests of functionality, driveability and safety
4. Postrequisites

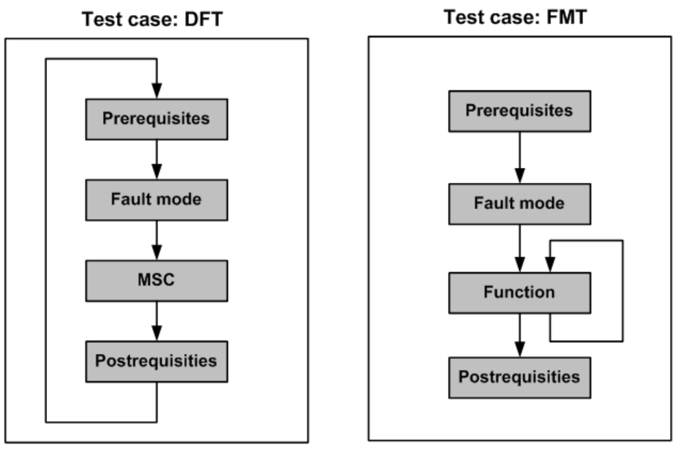


Figure 6.1. Difference between the cases for Degraded User Function Test and Fault Mode Test. The test case for Degraded User Function Test is iterated through every fault mode, each iteration executing prerequisites and postrequisites. In Fault Mode Test only all functions are iterated through, only requiring execution of pre- and postrequisites once.

6.2.2 FMT: Area of application

The Fault Mode Test problably suits ordinary regression testing, verifying that updates have not changed the software with any erroneous behaviour.

FMT also suits investigation of error propagation because of the broad nature of the test, but the test is probably not deep enough testing and verifying new functionality.

One advantage with FMT is that the test scripts could be more easily automatically generated than DFT. A program can choose a specific set of fault modes, and depending on the fault modes decide which test modules to execute. Both communications matrices described in Section 5.2 can help making that decision. The test modules to be run can also depend on configuration of vehicle, fault mode or some other parameter, using some algorithm.

The disadvantage is that the tests are not as deep as DFT regarding to one MSC. The test modules also should be scripted so general that they can be used in any test case. To get an efficient test case the test modules has to be implemented that they easily can be run after another, without any unnecessary starts and stops. They must also be independent of previous actions within the test cases. The general approach of small test modules can cause a problem and can be difficult to script.

6.3 Comparison

A short comparison between both methods is described in Table 6.1. Since none of these methods has been analyzed from many test cases implemented and written, much of what to follow is just an own analysis.

Table 6.1. Comparison between Degraded User Function Test and Fault Mode Test

	DFT	FMT
Fault modes	Several fault modes relevant to an MSC	One fault mode only
Functions under test	One MSC	Many, even functions not described by an MSC
Coverage, locally	Medium	Low
Coverage, globally	None; nothing outside the function is tested	Low-Medium
Test cases	Much work compiling specifications and writing the test case	Little work if everything is already written. More work if a new function test module has to be written
Test scripts	Many or long test scripts	One test script
Acceptance criteria	Communication, DTCs/Warning within the function. Comparison with expected output	All acceptance criteria stated in Section 5.3. No research for exact behavior needed

6.4 Summary

Two test techniques are suggested: Degraded User Function Test and Fault Mode Test. Both of them have there own area of application.

Degraded User Function Test suits testing new functionality in parallel with ordinary User Function Tests. The DFT tests one MSC or user function with regards to fault modes specified within the MSC. The test gets substantial coverage within the MSC, but none outside the MSC. The test cases also tends to be many or lengthy.

Fault Mode Test suits to test the behaviour of the complete vehicle during failure and is suited for regressions testing and investigation of error propagation. The FMT tests functionality all over the vehicle, but the tests performed are not as deep as a DFT or ordinary User Function Test. One test case is written per fault mode.

Next chapter will shortly describe how both methods have been implemented in a HIL-lab.

Chapter 7

Test scripts in HIL-lab

To verify what previously has been discussed a number of test scripts were written and executed in a HIL-lab. Test scripts based on both test techniques, Degraded User Function Test and Fault Mode Test, were implemented. Examples of some of the test scripts written will be found in Appendix B.

There where not much time available in the I-Lab2, with the result that tests were sparsely executed. Still test scripts using both techniques were implemented. But first some problems performing degradation tests have to be discussed.

7.1 Problems

Writing and executing the test scripts led to a couple of problems described here.

7.1.1 Present DTCs and warnings

I-Lab2 consists of ECUs and models for different configurations of vehicles (Section 1.4.2). As mentioned earlier each ECU has between 50-10 000 parameters to reflect each possible configuration of vehicle. For each configuration the ECUs must be programmed with a correct set of parameters which should be consistent with the HIL-models. There are some difficulties setting every ECU parameter correctly, meaning that the vehicle configuration often is somewhat inconsistent with the models. When the ECUs, from its perspective, do not read a consistent sensor value, the ECU will interpretate it as an active fault mode and produce a DTC.

To execute a degradation test, the HIL-lab should preferable be DTC free, but the opposite generally holds, with DTCs in ECUs and warnings in the ICL. If any unwanted DTCs are present in the HIL-lab when performing tests, there is an active fault mode somewhere. That fault mode can affect the tests performed (see Example 7.1).

Example 7.1

UF 415 Hill Hold
Hill Hold is a function maintaining the brake pressure when releasing the brake pedal when standing still until the accelerator pedal is pressed and the vehicle is accelerating. The function is usable for example when starting the vehicle in a hill, preventing the vehicle from rolling downwards when switching the foot from the brake pedal to the accelerator pedal.

The function uses the wheel speed sensors which measures the speed the wheels have and the brake pedal sensor. If any of these to correspondent CAN-signal is faulty Hill Hold will not activate. Because of some inconsistency between the ECU parameters and the models, the ECU receiving the wheel speed sensors overrules the sensor signal and sends {Error} on the corresponding signal on CAN, and stores a DTC. Despite any other fault mode, Hill Hold will never activate.

7.1.2 Fault modes

The tool used to simulate different fault modes is called Fault Injector Unit (FIU) which can be used for simulating short circuits and open circuits on inputs and outputs on ECUs. The FIU is produced and bought in from dSpace that connects it to the hardware. Most of the ECU ports representing actuators and some supply voltages to sensors are connected to the FIU. But almost no ports representing sensor values are attached to the FIU. To force different behavioural modes on an arbitrary signal, direct manipulation of the sensor values in the software are needed. The easiest way is to set the sensor values to unrealistic values. The effect on CAN with an unrealistic value is unpredictable, meaning that the method of manipulating sensor values is unwanted. Since {Error} and {NA} are together considered one behavioral mode meaning that if one of these modes are provoked the other should not.

Disconnecting entire ECUs is implemented and works fine, and generates the behavioural mode {TO}.

7.2 Degraded User Function Test: Test script

Already existing test scripts were mostly used for the DFT, with the modification of injecting fault modes. A few test scripts were written from scratch. The MSC under test were researched for the behaviour during different fault modes. All relevant fault modes were implemented as its own class (in the programming language Python), making it easy to call a fault mode (Section 7.2.1) using one line of code.

The test script is executed according to the following process:

Prerequisites

1. Start engine
2. Realize the prerequisites that the test of the MSC can be executed
3. Start CAN-logger and wait 10 seconds (optional)

Fault mode

1. Store DTCs and active warnings in ICL
2. Inject the fault mode
3. Wait 10 seconds
4. Store DTCs and active warnings in ICL

Test of MSC

1. Run the test script testing the MSC

Postrequisites

1. Stop CAN-logger (optional)
2. Restore everything back to normal without the fault mode
3. Read DTCs and active warnings in ICL
4. Stop vehicle
5. Shut down engine

Restart the process from the beginning choosing another fault mode

Using the process above it is assumed that the fault mode is already active when performing the actual test of the UF. Another possibility is to activate the fault mode within in the test at an arbitrary moment. Also deactivation of a fault mode within the test is possible. Neither of these two variants has been considered. It can be discussed how probable an active fault mode is deactivated or that a fault mode is activated a given instant. It is more likely that a fault mode is instead activated somewhere in the past, which reflects the process suggested above.

7.2.1 Fault mode class

In order to simplify calling a fault mode a separate class was written. If a FIU existed, it was used to activate the fault mode. Otherwise manipulation of sensor values had to be used. Setting an unrealistic sensor value was tested online in the HIL-lab, to determine the effect on the corresponding signal on CAN. After a confirmation that the unrealistic value generated {NA/Error} on CAN, the fault modes were implemented in the fault mode class. See Figure 7.1. Short circuiting entire ECUs were easy to provoke.

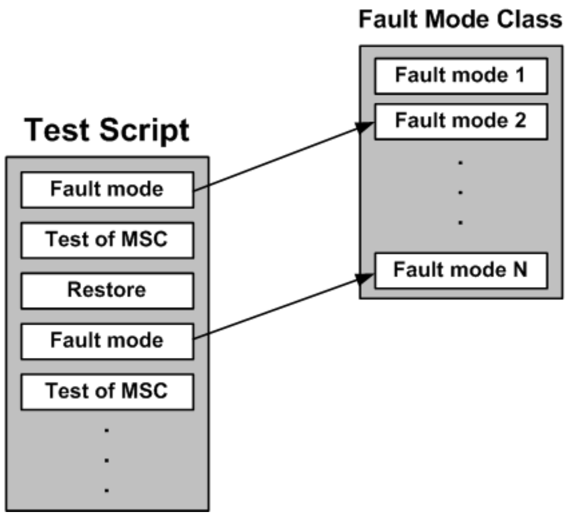


Figure 7.1. The test script calling for Fault Mode 2 and then Fault Mode N. As seen both fault modes are callable from one separate class.

7.2.2 Evaluation of test scripts

Documentation was clearly a problem. Extensive time was consumed with detective work, compiling information from many different documents to map how the function is supposed to act in case of some fault mode. When expected response were determined, it was easy to modify existing test scripts, just adding the fault mode class to the code and call for the fault mode in the beginning of the script. The easiest way of iterating through the test script with different fault modes, was to first run the test script as usual and then modifying the test script with another fault mode and rerunning it.

Except the documentation test scripts written worked fine, but were time consuming to execute. The CAN-logger, which stores all communication on CAN, radically decreased performance. One suggestion is that the test script is first run without the CAN-logger, and then rerun with the logger active if strange behaviour has occurred, to be able to analyze all communication on CAN afterwards.

7.2.3 A test case according to DFT

Hill Hold (UF415) where mentioned in Example 7.1. The test case shall test degradation when activating Hill Hold with an active fault mode. When Hill Hold is active the CAN-signal HillHolderMode will be set active and the vehicle speed should be constant 0 km/h corresponding to maintaining brake pressure. The script can be found in Appendix B.1
The prerequisites will not be described, but consists of that all signals used by

the function must be in within a certain limit, for example that the Vehicle Speed must be 0 km/h. Also the slope of the road is set, meaning that the vehicle will accelerate downwards if Hill Hold does not activate.

Prerequisites

1. Start engine
2. Check that all relevant signals are within valid limits
3. Press brake pedal
4. Set road slope to X %
5. Enable Hill Hold
6. Start CAN-logger and wait 10 seconds (optional)

Fault mode

1. Store DTCs and active warnings in ICL
2. Short circuit the wheel speed sensor
3. Wait 10 seconds
4. Store DTCs and active warnings in ICL

Test of MSC

1. Release brake pedal*
2. Check CAN-signal corresponding to the wheel speed sensor on all buses*
3. Check HillHolderMode on all buses*
4. Check Vehicle Speed on all buses*

Postrequisites

1. Stop CAN-logger (optional)
2. Connect the wheel speed sensor
3. Restore everything back to normal
4. Read DTCs and active warnings in ICL
5. Stop vehicle
6. Shut down engine

Restart the process from the beginning choosing another fault mode

(Items marked with '*' has a test attached to it, verifying the statement.

7.3 Fault Mode Test: Test script

Completely new material for test scripts was written for Fault Mode Test. First a test script shell was written with two variables; active fault mode and function test modules to test. Fault modes of components (Section 7.2.1) and function test modules (Section 7.3.1) were implemented as two separate classes. Executing the test, only these two variables had to be set to generate a complete test script. See also Figure 7.2. Appendix B.2 contains the test script written in Python.

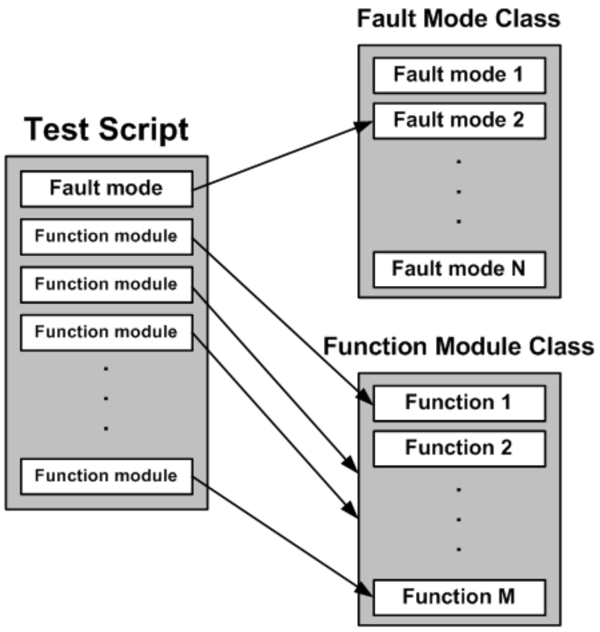


Figure 7.2. The test script calling for Fault Mode 2 and a number of function test modules. As seen both fault modes and function test modules are callable from two different classes.

The test script is executed according to the following process:

Prerequisites

1. Start engine
2. Increase vehicle speed to X km/h

Fault mode

1. Read DTCs and active warnings in ICL
2. Start CAN-logger and wait 10 seconds
3. Inject the fault mode defined by a programming variable

4. Wait 10 seconds
5. Read DTCs and active warnings in ICL
6. Stop CAN-logger

Function test modules

1. Run all relevant function test modules (Section 7.3.1) defined by a programming variable

Postrequisites

1. Restore everything back to normal
2. Read DTCs and active warnings in ICL
3. Stop vehicle
4. Shut down engine

7.3.1 Function test modules

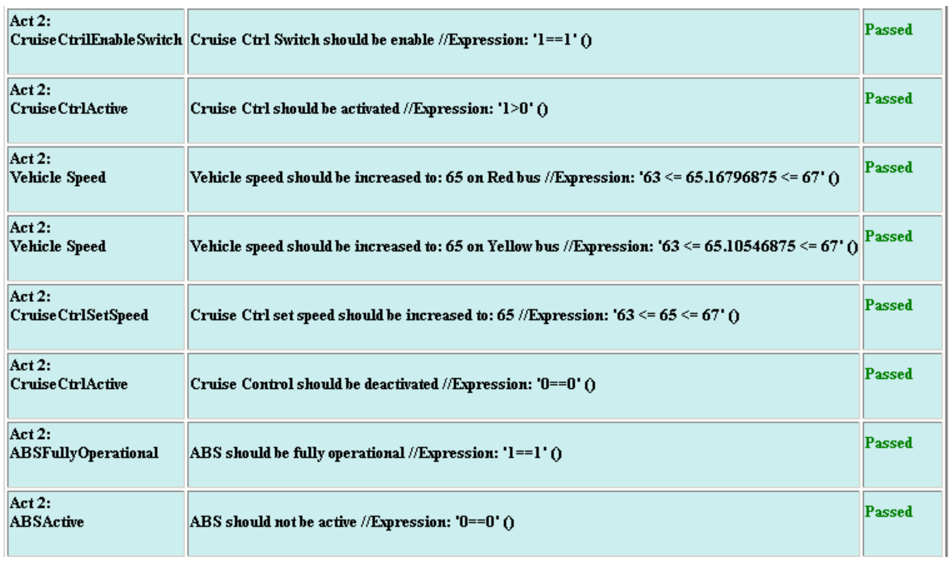
Since tests of the same functionality often are executed in different test scripts using Fault Mode Test, function test modules were introduced. Rather than writing a completely new test script for every function to test, function test modules are callable from every test script using only one line of code. The function test modules were collected together in an independent class. The test modules should be implemented that they can be executed in an independent sequence, but should be executed so that they can benefit from previous actions.

The function test modules shall be implemented as small tests of some functionality. Each test module in turn has a couple of sub tests verifying some particular functionality within the function test module. The sub tests compare a number of CAN-signals with expected outputs. All tests are written out to a test report used for the entire test script. The tester can easily see the outcome of the test script. See Figure 7.3.

Example of some of the test modules implemented:

Cruise Control (CC):

1. Start CAN-logger (optional)
2. Enable CC*
3. If vehicle speed < X km/h increase speed to Y km/h
4. Engage CC*
5. Store DTC/Warnings
6. Increase Set Speed with 5 km/h*
7. Disengage CC by pressing the brake pedal*
8. Stop CAN-logger (optional)



Act 2: CruiseCtrilEnableSwitch	Cruise Ctrl Switch should be enable //Expression: '1==1' ()	Passed
Act 2: CruiseCtrlActive	Cruise Ctrl should be activated //Expression: '1>0' ()	Passed
Act 2: Vehicle Speed	Vehicle speed should be increased to: 65 on Red bus //Expression: '63 <= 65.16796875 <= 67' ()	Passed
Act 2: Vehicle Speed	Vehicle speed should be increased to: 65 on Yellow bus //Expression: '63 <= 65.10546875 <= 67' ()	Passed
Act 2: CruiseCtrlSetSpeed	Cruise Ctrl set speed should be increased to: 65 //Expression: '63 <= 65 <= 67' ()	Passed
Act 2: CruiseCtrlActive	Cruise Control should be deactivated //Expression: '0==0' ()	Passed
Act 2: ABSFullyOperational	ABS should be fully operational //Expression: '1==1' ()	Passed
Act 2: ABSActive	ABS should not be active //Expression: '0==0' ()	Passed

Figure 7.3. A test report from a Fault Mode Test, with no fault mode present. The vehicle is in normal operation mode during the test. Every row consists of one test and the result of the test is found in the rightmost column. For this particular test case all tests have passed.

ABS control

1. Start CAN-logger (optional)
2. If ABS is not fully operational or active abort test*
3. If vehicle speed < X km/h increase speed to Y km/h
4. Lower front left wheel speed*
5. Test ABS active*
6. Restore front left wheel speed*
7. Stop CAN-logger (optional)

Driveability

1. Start CAN-logger (optional)
2. Brake to stand still*
3. Accelerate to X km/h*
4. Steer vehicle to the left*
5. Steer vehicle to the right*

(Items marked with '*' has a test attached to it, verifying the statement. See also Figure 7.3). The test script for the function test module Cruise Control can be found in Appendix B.3.

All function test modules have their own CAN-logger, only logging while the function test module is active. One suggestion is to first run the test script without the CAN-logger, since logging decreases performance. When detecting a strange behaviour in one of the test modules, the test case could be rerun with the CAN-logger active, to be able to analyze the entire communication on CAN afterwards.

Another example of a function test module with the corresponding test script is found in Appendix B.4.

7.3.2 Acceptance criteria

As already mentioned in Section 7.3.1, test script generates a test report with every test within the test script. A more extended discussing of the tests performed and the use of acceptance criteria are now followed. References are made to the rules defined in Section 5.3.

Tests within the function test modules are as mentioned earlier a comparison of a CAN-signal with an expected response, verifying some functionality within the function test module. These tests also checks if the signal is in the behavioural modes {Unrealistic, Undefined, TO}, which are forbidden according to **R12-R14, R16**. For the signals under test, a consistency check is also made between the three CAN-buses, testing gateway (**R15**).

DTCs and warnings in the ICL are stored within the function test modules. No corresponding tests are attached to the DTCs or warnings, mostly because error propagation making it difficult to overrule a stored DTC or warning. DTCs and warnings must be analyzed afterwards and manually. Looking for changes in DTCs or Warnings can be performed automatically, but have not been implemented and must now be done manually (**R1-R4**). To summarize the function test modules shall always verify DTCs/Warnings and Communication.

Depending on which function test modules are executed the acceptance criteria on Functionality/Driveability and Security are verified (**R5 - R9, R10-R11**).

The CAN-logger before and after fault mode should always be active. An analysis of the static behaviour of the electrical system can then be made, which in turn could point out suggestions of error propagation (**R10**). A program has been written analyzing every signal saved with a CAN-logger, detecting changes in mean-value before and after the fault mode. The prerequisite for such an analysis is during the logging of CAN everything has to be static.

7.3.3 Evaluation of test script

Only one test shell was written, and eight function test modules. Since the function test modules were so few to the number, the function test modules executed were only limited by the configuration of the vehicle. Different configurations have different functionality. The same test script shell was executed on several different configurations with different function test modules executed and worked fine. It was also easy to troubleshoot the function test modules implemented, since they were written as short tests.

Despite tests within the function test modules manual work was needed to verify a correct behaviour (for DTCs and warnings). Tests within the function test modules helped checking communication and gateway.

7.4 Difficulties and lessons

There were a number of difficulties writing test cases and test scripts. One problem already mentioned was the difficulties determining expected responses, requiring time compiling specifications.

Much time were required to debug the test scripts written. Especially test scripts written for DFT which tended to be long with much code. Because the function test modules consists of less code they were also easier to debug. The function test modules could easily be reused but with a different fault mode, without any debugging.

At last, there were some problems with the models when executing some tests scripts, where the vehicle had difficulties to accelerate to a certain speed. Several tries were sometimes required to fulfil the prerequisites.

7.5 Summary

Both methods were implemented in the I-Lab2, and required much work but in different areas. The investigation of expected response when writing a Degraded User Function Test took as much time as implementing the function test modules in Fault Mode Test. The function test modules in FMT could be reused for many test scripts later.

The method of using classes instead of writing stand alone test scripts for FMT proved to be beneficial. Also the fault mode class, used by both test techniques, proved to be efficient once implemented.

Chapter 8

Conclusions

REST could perform degradations tests. The requirements mentioned in Section 5.3 has to be validated to verify that the electrical system behaves properly even in a degraded state. There is only one way for the electrical system being in normal operation mode ({NF}), but infinite many combinations of fault modes and failures. Therefore testing degradation requires more work than testing a correct behaviour. The current documentation is not suited for writing distributed degradations test. There are newer documents that are quite specific about degradation (collected in one document), but is not always informative enough regarding DTCs, warnings or degradations modes depending on different fault modes on components or behavioural modes on signals. Other documents has to be tended to.

Two test strategies have been suggested and can be used today with existing material. However problems arise because of inconsistency between models and the set ECU parameter, and not enough tools generating appropriate fault modes. A recommendation is to look into if the FIU needs to be extended, to cover all pins on an ECU. The kind of test suited mostly for REST is Fault Mode Test, which is a test strategy for testing integration and error propagation, because of the capabilities in I-Lab2. The two major advantages with I-Lab2 are that a large number of possible combinations of vehicles can be tested with the same hardware and that the entire electrical system can be tested. No other tool has that capability at Scania. However Fault Mode Test requires more work to implement from today's testing, and therefore Degraded User Function Test is recommended to start with. Degraded User Function Tests can be implemented and executed today without any changes, but is recommended only to be performed when writing new test cases. Only the fault mode class mentioned in Section 7.2.1 has to be implemented.

Testing degradation will probably take some time before performed on a regular basis. Degradations testing has to be delegated to smaller units, verifying DTCs, Warnings and requirements set on communication in Section 5.3. The Degradation User Function Test could later be delegated to groups performing functions tests, just validating a certain function.

8.1 Future work

In general the tests strategies mentioned in this thesis has to be evaluated and tested more in I-Lab2, to see what improvements have to be made before using the two strategies on a regular basis. Despite some suggestions mentioned, a method or time plan how to best integrate degradations tests into today's testing must be developed. The introduction of degradations tests must be divided into smaller steps to start making it routine. Also an investigation if any of the two test suggested techniques are more suited in field test in an actual vehicle, or just in I-Lab2. Scripted tests, as in I-Lab2, will never find unexpected behaviour, but will just find behaviour we tell the script to look for. That may be unsatisfied and inefficient.

Today the lowest entity in testing is the independent test script consisting of a number of tests. With the introduction of possible two new kinds of tests, maybe the lowest entity should be the actual test. A number of test modules could build a test script, and can be compared with methods presented about function modules in Section 7.3.1. It turned out that function modules were an efficient and variable way of constructing test scripts and easier to troubleshoot. However to restructure the existing test scripts into smaller test modules must be evaluated to compare the cost and future benefit.

Methods presented in this thesis have to be refined and evaluated more. Today every MSC is classed according to Risk Based Testing and can be used to decide the level of testing of MSCs. Fault modes are not classed according to Risk Based Testing, but could be. The classifications could then decide the test level for a Fault Mode Test. Some ideas during the thesis have been tried for a classification system for signals, based on the parameters distributivity and severity, but has been discarded because of time issues. A simple model were developed that could be used to automatically generate a grade for every signal, but turned out to be flawed.

If documentation should or could be improved is the last suggestion. To test degradation well defined requirements has to be specified, with expected outputs that can be validated during test. Today the specifications are not enough for degradation tests, but requires plenty of work to improve. Again it is cost against benefits.

Bibliography

[1] Algirdas Avizienis and Jean-Claude Laprie and Brian Randell and Carl Landwehr. *Basic Concepts and Taxonomy of Dependable and Secure Computing*. IEE Computer Society, 1 edition, 2004. 1545-5971/04.

[2] Christer Nordström and Kristian Sandström and Jukka Mäki-Turja and Hans Hansson and Henrik Thane and Jan Gustafsson. *Robusta realtidssytem*. Mälardalen Real-Time Research Centre, 1 edition, 2001. ns-bok00-11 00-08-18.

[3] Eldon Leaphart and Barbara Czerny and Joseph D'Ambrosio and Christopher Denlinger and Deron Littlejohn. *Survey of Software Failsafe Techniques for Safety-Critical Automotive Applications*. SAE International, 1 edition, 2005. 2005-01-0779.

[4] Fernando Henrique Ataide and Fabiano Costa Carvalho and Carlos Eduardo Pereira and Max Mauro Dias Santos. *An overview of Hardware-In-the-Loop Testing Systems at Visteon*. SAE International, 1 edition, 2005. 2005-01-4144.

[5] Anders Florén. Requirements on diagnostic functionality. TB4091, 2008.

[6] George A Peters and Barbara J Peters. *Automotive Vehicle Safety*. Taylor & Francis, London, 1 edition, 2002. ISBN 0-415-26333-6.

[7] Johnny Johansson. Requirements on CAN communication for Scania ECUs. PD1443613, 2003.

[8] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House Publishers, Boston and London, 1 edition, 2008. ISBN 1-58053-791-X.

[9] Hans Lind, Björn Westman, and Peter Madsen. The SESAMM Concept. PD1422538, 2003.

[10] Mattias Nyberg and Erik Frisk. *Model Based Diagnosis of Technical Processes*. Department of Vehicle Systems, Linköping, 1 edition, 2008. .

[11] Mauro Velardocchia and Aldo Sorniotti. *Hardware-In-the-Loop to Evaluate Active Braking System Performance*. SAE International, 1 edition, 2005. 2005-01-1580.

[12] Ingvar Nordström. Risk Based Testing, Embedded Systems. REST08021, 2008.

[13] Interview of various personell at Scania CV AB.

[14] S.A Hassan and A.M.A.A. Abou-El-Nour. *A Prognostic Technique for Vehicle Suspension Elements Using Degradation Mesaures*. SAE International, 1 edition, 2005. 2005-01-1445.

[15] Syed Nabi and Mahesh Balike and Jace Allen and Kevin Rzamien. *An overview of Hardware-In-the-Loop Testing Systems at Visteon*. SAE International, 1 edition, 2004. 2004-01-1240.

[16] Thomas Müller and Rex Black and Dorothy Graham and Debra Friedenberg and Erik van Veendendal. *Certified Tester - Foundation Level Syllabus*. International Softeare Testing Qualifications Board, 1 edition, 2007. http:.

Appendix A

Abbreviations

A list of abbreviations used with a short explanation and reference.

APS:	Air Pressure System. ECU controlling the air pressure
AUS:	Audio System. ECU controlling the audio system
CAN:	Controller Area Network. Network where communication between control units are transmitted 2.3.1
COO:	Coordinator. The ECU connected to all three CAN-buses
DTC:	Diagnostic Trouble Code. 2.4
DFT:	Degraded User Function Test. 6.1
EMS:	Engine Management System. The ECU controlling the engine
ECU:	Electrical Control Unit. 1.2.1
FMEA:	Failure Mode and Effect Analysis. 2.1.2
FMT:	Fault Mode Test. 6.2
HIL:	Hardware-In-the-Loop. 1.4.2
ICL:	Instrument Cluster Panel.
MSC:	Message Sequence Chart. 2.2.1
NA:	Not Available. Behavioural mode of a signal. 2.3.2
REST:	Group at Scania with the responible of conducting systems and integrations test.
TO:	Time Out. Behavioural mode of a signal. 2.3.2
UF:	User Function. 2.2
VOR:	Vehicle Of Road. Means that the vehicle cannot be driven anymore.

Appendix B

Test scripts

Some test scripts written will be found in this appendix. All test scripts have been written in the programming language Python. The test scripts have been stripped from code that has been considered of less importance with regards to this thesis. Also to make the code more perspicuous long lines of code has somewhat been replaced with three dots ...!

Some explanation of the classes and objects used:

`self._vtExec` is the main class in the test framework initiating all objects needed to perform scripted testing in I-Lab2

`self._vtExec.VtPrint.DebugPrint(string)` prints `string` into a debug file

`SignalUtil.ReadCANSignal(...)` reads a specified CAN-signal on one CAN-bus

`self._vtExec.VtXMLReport` is a class responsible for writing text to the test report shown in Figure 7.3.

`self._testdrv` is the class with all function test modules mentioned in Section 7.3.1.

`self._sensors` is the class activating and deactivating fault modes mentioned in Section 7.2.1.

B.1 Hill Hold test script

The following is the test script for the test case Hill Hold based on DFT, only testing degradation for one fault mode.

```
# Prerequisites ######################################################################
self._act_name = "Prerequisites"
self._vtExec.VtTracker.StartAction("Executing %s" %(self._act_name), self._act_name)
```

```
# Voltage level 24 Volt ###########################################################
self._vtExec.VtPrint.DebugPrint('Voltage level 24 Volt')
self._ba.SetVoltage(self._PREREQ_BATTERY_VOLTAGE)

# Ignition On #####################################################################
self._vtExec.VtPrint.DebugPrint("Ignition On")
self._drv.IgnitionOn()
self._timer.Sleep(1)

# Start engine ####################################################################
self._vtExec.VtPrint.DebugPrint("Starting the engine")
self._drv.EngineOn()
self._timer.Sleep(1)

# Enable Hill Hold ################################################################
self._vtExec.VtPrint.DebugPrint("Activate Hill Hold")
self._ebs.EnableHillHold()
self._timer.Sleep(1)

# Press brake pedal and release clutch ############################################
self._vtExec.VtPrint.DebugPrint("Pressing brakepedal and releasing clutch pedal")
self._bp.PressBrakePedal(80)
self._timer.Sleep(1)

# Set road slope ##################################################################
self._vtExec.VtPrint.DebugPrint("Set road slope")
self._road._SetSlope(-3)
self._timer.Sleep(1)

# Check all prerequisties #########################################################

# Check that pedal is released ####################################################
brakePedal = SignalUtil.ReadCANSignal('BrakePedalPosition', 'RED_1', 'EBC1', 'A')
self._vtExec.VtPrint.DebugPrint("Brake Pedal Position is = %s" %(brakePedal))

if(brakePedal < 75):
    self._vtExec.VtXMLReport.AddText("Brake pedal position should be at least 75 %")
    self._vtExec.VtError.TryTestCaseAgain("Brake pedal position should be at least 75 %")

# Check vehicle speed #############################################################
vehicleSpeed = SignalUtil.ReadCANSignal('TCOVehSpeed','YELLOW_1','TCO1','TCO')
self._vtExec.VtPrint.DebugPrint("Vehicle Speed is = %s" %(vehicleSpeed))

if(abs(vehicleSpeed) > 2):
   self._vtExec.VtXMLReport.AddText("Vehicle speed should be zero", self._act_name)
   self._vtExec.VtError.TryTestCaseAgain("Vehicle speed should be zero")

# Check ABS active ################################################################
absActive = SignalUtil.ReadCANSignal('ABSActive', 'RED_1', 'EBC1', 'A')
self._vtExec.VtPrint.DebugPrint("ABSActive is = %s" %(absActive))

if(absActive != 0):
   self._vtExec.VtXMLReport.AddText("ABS should not be active")
   self._vtExec.VtError.TryTestCaseAgain("ABS should not be active")

#Store ICL-notices ################################################################
ICL = self._icl2.Notices.ReadAll(True)
```

```
self._vtExec.VtXMLReport.AddText(str(ICL))
self._timer.Sleep(1)

#Store DTCs ###################################################################
DTC = self._vtExec.VtProgramEcu.ReadAllDtcs('ALL',True)
self._vtExec.VtXMLReport.AddText(DTC)
self._timer.Sleep(1)

#Start CAN-Logger #############################################################
self._vtExec.VtPrint.DebugPrint("Starting CAN-Logger")
self._canLogger.Start()
self._canLogger.WriteToLog("Act 0: Standing still")

# End Prerequisites ###########################################################
```

Act 1 short circuits a wheel speed sensor (also called front axle speed sensor in the test script) and then releases the brake pedal to test if hill hold activates.

```
self._act_name = "Act 1"
self._canLogger.WriteToLog("Act1: Speed sensor")
self._vtExec.VtTracker.StartTest(self._act_name)

# Stimuli #####################################################################

# Activate fault mode #########################################################
self._sensors.disconnectFrontAxleSpeedSensor()
self._timer.Sleep(3)

# Release brake pedal #########################################################
self._vtExec.VtPrint.DebugPrint("Release brake pedal")
self._bp.ReleaseBrakePedal()
self._timer.Sleep(1)

# Expected response ###########################################################

# Store DTCs ##################################################################
DTC = self._vtExec.VtProgramEcu.ReadAllDtcs('ALL',True)
self._vtExec.VtPrint.DebugPrint("Storing DTC")
self._vtExec.VtXMLReport.AddText(DTC)

# Store ICL-notices ###########################################################
ICL = self._icl2.Notices.ReadAll(True)
self._vtExec.VtPrint.DebugPrint("Storing ICL notices")
self._vtExec.VtXMLReport.AddText(str(ICL))

# Check FrontAxleSpeed signal #################################################

frontAxleSpeed_Red = SignalUtil.ReadCANSignal('FrontAxleSpeed','RED_1',...)
frontAxleSpeed_Yel = SignalUtil.ReadCANSignal('FrontAxleSpeed','YELLOW_1',...)
frontAxleSpeed_Gre = SignalUtil.ReadCANSignal('FrontAxleSpeed','GREEN_1',',...)

self._vtExec.VtPrint.DebugPrint("Front Axle speed is: %s" %(frontAxleSpeed_Red))

# Check HillHolderMode signal #################################################
hillHolderMode_Red = SignalUtil.ReadCANSignal('HillHolderMode','RED_1','EBC5','A')
hillHolderMode_Yel = SignalUtil.ReadCANSignal('HillHolderMode','YELLOW_1','EBC5','A')
```

```
self._vtExec.VtPrint.DebugPrint("Hill hold is: %s" %(hillHolderMode_Red))
self._vtExec.VtXMLReport.AddSingleLimitComparison(hillHolderMode_Red, 0 ,'==',...)

# Check VehicleSpeed signal ######################################################
vehicleSpeed_Red = abs(SignalUtil.ReadCANSignal('TCOVehSpeed','RED_1','TCO1','TCO'))
self._vtExec.VtPrint.DebugPrint("VehicleSpeed is: %s" %(vehicleSpeed_Red))
self._vtExec.VtXMLReport.AddSingleLimitComparison(vehicleSpeed_Red, 0, '>', ...)

# End Act 1 ######################################################################
```

The Postrequisites restores everything back to normal

```
self._act_name = "Postrequisites"
self._vtExec.VtTracker.StartAction("Executing %s" %(self._act_name), self._act_name)

# Restore road slope #############################################################
self._vtExec.VtPrint.DebugPrint("Restore road slope")
self._road.EndSlope()
self._timer.Sleep(1)

# Deactivate fault mode ##########################################################
self._sensors.connectAll()

# Save logger ####################################################################
self._vtExec.VtPrint.DebugPrint("Start saving to log")
self._canLogger.Stop()
self._canLogger.SaveLogTextFile(fileName)
self._vtExec.VtPrint.DebugPrint("Done saving to log.")

# Store DTCs #####################################################################
DTC = self._vtExec.VtProgramEcu.ReadAllDtcs('ALL',True)
self._vtExec.VtXMLReport.AddText(DTC)
self._timer.Sleep(1)

# Store ICL-notices ##############################################################
ICL = self._icl2.Notices.ReadAll(True)
self._vtExec.VtXMLReport.AddText(str(ICL))
self._timer.Sleep(1)

# Stopping vehicle ###############################################################
self._vtExec.VtPrint.DebugPrint("Stopping vehicle")
self._drv.Stop()
self._timer.Sleep(1)

# Engine off #####################################################################
self._vtExec.VtPrint.DebugPrint("Turning engine OFF")
self._drv.TurnOff()
self._timer.Sleep(1)
self._vtExec.VtTracker.EndAction(True)

# End Postrequisites #############################################################
```

B.2 Test script shell for FMT

The following presents the test script shell for the Fault Mode Test. Some lines of code has been removed to clarify the test script shell. The test script shell consists of Prerequisites, Act 1 (Fault Mode), Act 2 (Function test modules) and Postrequisites.

```
# Start Prerequisites ############################################################

self._act_name = "Prerequisites"
self._vtExec.VtTracker.StartAction("Executing %s" %(self._act_name), self._act_name)

# Voltage level 24 Volt ##########################################################
self._vtExec.VtPrint.DebugPrint('Voltage level 24 Volt')
self._ba.SetVoltage(self._PREREQ_BATTERY_VOLTAGE)

# Ignition On ####################################################################
self._vtExec.VtPrint.DebugPrint("Ignition On")
self._driver.IgnitionOn()
self._timer.Sleep(1)

# Start engine ###################################################################
self._vtExec.VtPrint.DebugPrint("Starting the engine")
self._driver.EngineOn()
self._timer.Sleep(1)

# Accelerate to 50 km/h ##########################################################
self._vtExec.VtPrint.DebugPrint("Drive at 60km/h" )
self._driver.DriveAtSpeed(50)
self._vtExec.VtPrint.DebugPrint("Sleep for 5 seconds")
self._timer.Sleep(5)

# End Prerequisites ##############################################################
```

Act 1 activates the one fault mode. The fault mode is called from a separate class. This particular FMT disconnects the front axle speed sensor.

```
# Act 1 ##########################################################################

self._act_name = "Act 1"
self._vtExec.VtTracker.StartTest(self._act_name)

# Save DTCs and ICL notices ######################################################
self._vtExec.VtPrint.DebugPrint("Save DTCs anc ICL notices")
DTC = self._vtExec.VtProgramEcu.ReadAllDtcs('ALL',True)
self._vtExec.VtXMLReport.AddText(DTC, "Before fault mode")

ICL = self._icl2.Notices.ReadAll(True)
self._vtExec.VtXMLReport.AddText(str(ICL), "Before fault mode")
self._timer.Sleep(1)

# Start logger ###################################################################
self._vtExec.VtPrint.DebugPrint("Start CAN-logger")
self._canLogger.Start()
self._vtExec.VtPrint.DebugPrint("Sleep for 10 seconds, while logging CAN")
self._timer.Sleep(5)

# Activate a fault mode ###########################################################
self._canLogger.WriteToLog("#: Fault mode")
self._sensors.disconnectFrontAxleSpeedSensor()
self._vtExec.VtPrint.DebugPrint("Sleep for 10 seconds")
self._timer.Sleep(10)

# Save DTCs, ICL notices and CAN-logger ###########################################
DTC = self._vtExec.VtProgramEcu.ReadAllDtcs()
self._vtExec.VtXMLReport.AddText(DTC, "After fault Mode")

ICL = self._icl2.Notices.ReadAll(True)
self._vtExec.VtXMLReport.AddText(str(ICL), "After fault Mode")

# Stop CAN-logger #################################################################
self._canLogger.Stop()
self._vtExec.VtPrint.DebugPrint("Start saving to log")
self._canLogger.SaveLogTextFile(testFolder)
self._vtExec.VtPrint.DebugPrint("Done saving to log.")
self._timer.Sleep(1)

# End Act 1 #######################################################################
```

Act 2 calls the function test modules that are going to be executed. The variable `testList` determines which function test modules to call. The function test modules are then sorted and executed in a order that they could benefit from earlier executed function test modules. Only the function test modules `CC` will be found in later in the appendix. All of the modules found in `Act2` is implemented.

```
# Act 2 ###########################################################################

self._act_name = "Act 2"
self._vtExec.VtTracker.StartTest(self._act_name)

# Function test modules to execute ################################################
testList = ['CC', 'ABS', 'LowEngineOilPressure']

if "BrakeToStandStill" in testList:
   self._testdrv.TestBrakeToStandStill()
   self._timer.Sleep(1)

if "KickDown" in testList:
   self._testdrv.TestKickDown(50)
   self._timer.Sleep(1)

if "CC" in testList:
   self._testdrv.TestCC()
   self._timer.Sleep(1)

if "ABS" in testList:
   self._testdrv.TestAbs()
   self._timer.Sleep(1)

if "LowEngineOilPressure" in testList:
   self._testdrv.TestLowEngineOilPressure()
   self._timer.Sleep(1)

if "HillHold" in testList:
   self._testdrv.TestHillHold()
   self._timer.Sleep(1)

# End Act 2 ######################################################################
```

The Postrequisites restores everything back to normal, deactivating the fault mode, stopping the vehicle and turns of the engine.

```
# Start Postrequisites ###########################################################

self._act_name = "Postrequisites"
self._vtExec.VtTracker.StartAction("Executing %s" %(self._act_name), self._act_name)

# Connecting the component the component disconnected in Act 1 ###################
self._sensors.disconnectFrontAxleSpeedSensor()

# Stopping vehicle ###############################################################
self._vtExec.VtPrint.DebugPrint("Stopping vehicle")
self._driver.Stop()
self._timer.Sleep(1)

# Engine off #####################################################################
self._vtExec.VtPrint.DebugPrint("Turning engine OFF")
self._driver.TurnOff()
self._timer.Sleep(1)
self._vtExec.VtTracker.EndAction(True)

# End Postrequisites #############################################################
```

B.3 Function test module Cruise Control

The test script for the function test module that tests Cruise Control. Again some lines of code has been removed to clarify the function test module. The function test module Cruise Control is a callable function within the class `testdrv` according to act 2 in the Fault Mode Test shell above.

```
self._testName = "CC-test"
# Start CAN-Logger ################################################################
self._canLogger.Start()
self._canLogger.WriteToLog("#: Start CC")
self._timer.Sleep(1)

# Release brakepedal #############################################################
acceleratorPos = self._accPedal.GetPos()
self._brakePedal.ReleaseBrakePedal()
self._vtExec.VtPrint.DebugPrint("Brake pedal is Released.")

# Enable CC ######################################################################
self._canLogger.WriteToLog("#: Enable CC")
self._cc.EnableCc()
self._timer.Sleep(1)

# Check if CC is enable ###########################################################
cruiseCtrlEnableSwitch = SignalUtil.ReadCANSignal('CruiseCtrlEnableSwitch', ...)
self._vtExec.VtPrint.DebugPrint("CruiseCtrlEnableSwitch is: %s" %(cruiseCtrlEnableSwitch))
self._vtExec.VtXMLReport.AddSingleLimitComparison(cruiseCtrlEnableSwitch, 1, "==", ...)

# Drive at least in 50 km/h #######################################################
vehicleSpeed = SignalUtil.ReadCANSignal('TCOVehSpeed', 'YELLOW_1', 'TCO1', 'TCO')
self._vtExec.VtPrint.DebugPrint("Vehicle Speed is: %s km/h" %(vehicleSpeed))
if vehicleSpeed < 40:
   self._canLogger.WriteToLog("#: Increase speed")
   self._vtExec.VtXMLReport.AddText("Vehicle speed is %s km/h and below 50km/h)
   self._driver.DriveAtSpeed(50)

# Release accelerator position and engage CC ######################################
self._canLogger.WriteToLog("#: Activate CC")
self._accPedal.SetPos(0)
self._timer.Sleep(1)

self._cc.ClickAccButton()
self._timer.Sleep(1)

# Check that CC is active #########################################################
cruiseCtrlActive = SignalUtil.ReadCANSignal('CruiseCtrlActive', 'RED_1', ...)
self._vtExec.VtPrint.DebugPrint("Cruise Control Active is: %s" %(cruiseCtrlActive))
self._vtExec.VtXMLReport.AddSingleLimitComparison(cruiseCtrlActive, 0, ">", "CC-test")

# Increase speed with 5 km/h. If not within time out return False #################
self._canLogger.WriteToLog("#: Increase set speed")
self._timer.Start()
cruiseCtrlSetSpeed = SignalUtil.ReadCANSignal('CruiseCtrlSetSpeed', 'RED_1', ...)
self._vtExec.VtPrint.DebugPrint("Set speed is: %s" %(cruiseCtrlSetSpeed))

setSpe = cruiseCtrlSetSpeed + 5
self._vtExec.VtXMLReport.AddText("Set speed is: %s" %(cruiseCtrlSetSpeed), "CC-test")
while cruiseCtrlSetSpeed < setSpe:
   if self._timer.ReadTime() > 5:
      break

   self._cc.ClickAccButton()
   cruiseCtrlSetSpeed = SignalUtil.ReadCANSignal('CruiseCtrlSetSpeed', 'RED_1', ...)
   self._vtExec.VtPrint.DebugPrint("Set speed is: %s" %(cruiseCtrlSetSpeed))
   self._timer.Sleep(0.5)

# Check that set speed has increased with 5 km/h ##################################
vehicleSpeed = SignalUtil.ReadCANSignal('TCOVehSpeed', 'RED_1', 'TCO1', 'TCO')
self._vtExec.VtXMLReport.AddLimitPairComparison((setSpe - 2), vehicleSpeed, ...))

vehicleSpeed = SignalUtil.ReadCANSignal('TCOVehSpeed', 'YELLOW_1', 'TCO1', 'TCO')
self._vtExec.VtXMLReport.AddLimitPairComparison((setSpe - 2), vehicleSpeed, ...))

cruiseCtrlSetSpeed = SignalUtil.ReadCANSignal('CruiseCtrlSetSpeed', 'RED_1', ...))
self._vtExec.VtXMLReport.AddLimitPairComparison((setSpe - 2), cruiseCtrlSetSpeed, ...)
self._timer.Sleep(1)

# Disable CC by braking ############################################################
```

```
self._canLogger.WriteToLog("#: Brake")
self._vtExec.VtPrint.DebugPrint("Pressing brake pedal")
self._brakePedal.PressBrakePedal(40)
self._timer.Sleep(1)

self._vtExec.VtPrint.DebugPrint("Releasing brake pedal")
self._brakePedal.ReleaseBrakePedal()
self._timer.Sleep(1)

# Check that CC is not active ######################################################
cruiseCtrlActive = SignalUtil.ReadCANSignal('CruiseCtrlActive', 'RED_1', ...)
self._vtExec.VtPrint.DebugPrint("Cruise Control Active is: %s" %(cruiseCtrlActive))
self._vtExec.VtXMLReport.AddSingleLimitComparison(cruiseCtrlActive, 0, "==", "CC-test")

# Restore everything to normal #####################################################
self._cc.DisableCc()
self._accPedal.SetPos(acceleratorPos)

# Stop CAN-logger ##################################################################
self._canLogger.Stop()
self._vtExec.VtPrint.DebugPrint("Start saving to log")
self._canLogger.SaveLogTextFile(testFolder)
self._vtExec.VtPrint.DebugPrint("Done saving to log")

# END TEST #########################################################################
```

B.4 Function test module Low Engine Oil Pressure

The following section will give another example of an function test module written. The test module will test the display of the engine oil pressure in the Instrument Cluster Panel (ICL), as well as warning of low engine oil pressure.

1. Start CAN-logger (optional)
2. Lower engine oil pressure to X bar
3. Test that correct engine oil pressure is sent on CAN
4. Test that warning of low engine oil pressure is sent on CAN
5. Store DTC/Warnings
6. Increase the engine oil pressure to normal level
7. Test that warning of low engine oil pressure is not sent on CAN
8. Store DTC/Warnings
9. Stop CAN-logger (optional)

DTCs and warnings has to be manually checked after the test in the test report generated, to confirm that correct warnings in the ICL are displayed, and that appropriate DTCs are set.

The function test module is implemented as followed:

```
self._testName = "Oilpressure-test"

# Start CAN-logger ####################################################################
self._canLogger.Start()

# Lower Engine Oil Pressure ###########################################################
self._canLogger.WriteToLog("#: Decrease pressure")
self._oilsens.SetEngineOilPressure_Value(0.5)
self._timer.Sleep(15)

# Check that correct engine oil pressure is sent on CAN ##############################
engineOilPressure = SignalUtil.ReadCANSignal('EngineOilPressure','RED_1',...)
self._vtExec.VtPrint.DebugPrint("Oil Pressure on RED CAN is: %s" %(engineOilPressure))
self._vtExec.VtXMLReport.AddSingleLimitComparison(engineOilPressure, 52, "==", ...)

engineOilPressure = SignalUtil.ReadCANSignal('EngineOilPressure','YELLOW_1',...)
self._vtExec.VtPrint.DebugPrint("Oil Pressure on YELLOW CAN is: %s" %(engineOilPressure))
self._vtExec.VtXMLReport.AddSingleLimitComparison(engineOilPressure, 52, "==",...)

# Check that low engine oil pressure is sent on CAN #################################
oilPressureLow = SignalUtil.ReadCANSignal('LowEngineOilPressure','RED_1','DLN2','E')
self._vtExec.VtPrint.DebugPrint("Low Oil Pressure on RED CAN is: %s" %(oilPressureLow))
self._vtExec.VtXMLReport.AddSingleLimitComparison(...)

oilPressureLow = SignalUtil.ReadCANSignal('LowEngineOilPressure','YELLOW_1','DLN2','E')
self._vtExec.VtPrint.DebugPrint("Low Oil Pressure on YELLOW CAN is: %s" %(oilPressureLow))
self._vtExec.VtXMLReport.AddSingleLimitComparison(...)

# Save ICL notices and DTCs ###########################################################
self._vtExec.VtPrint.DebugPrint("Save DTCs anc ICL notices")
DTC = self._vtExec.VtProgramEcu.ReadAllDtcs('ALL',True)
self._vtExec.VtXMLReport.AddText(DTC, self._testName)

ICL = self._icl2.Notices.ReadAll(True)
self._vtExec.VtXMLReport.AddText(str(ICL), self._testName)
self._timer.Sleep(1)

# Restore Oil Pressure and end test ###################################################
self._canLogger.WriteToLog("Increase pressure")
self._oilsens.SetEngineOilPressure_Control_Model()
self._timer.Sleep(15)

# Check that no warning of low engine oil pressure is sent on CAN ###################
oilPressureLow = SignalUtil.ReadCANSignal('LowEngineOilPressure','RED_1','DLN2','E')
self._vtExec.VtPrint.DebugPrint("Low Oil Pressure on RED CAN is: %s" %(oilPressureLow))
self._vtExec.VtXMLReport.AddSingleLimitComparison(engineOilPressureLow, 0, "==", ...)

oilPressureLow = SignalUtil.ReadCANSignal('LowEngineOilPressure','YELLOW_1','DLN2','E')
self._vtExec.VtPrint.DebugPrint("Low Oil Pressure on YELLOW CAN is: %s" %(oilPressureLow))
self._vtExec.VtXMLReport.AddSingleLimitComparison(engineOilPressureLow, 0, "==", ...)
```

```
# Save ICL notices and DTCs #########################################################
self._vtExec.VtPrint.DebugPrint("Save DTCs anc ICL notices")
DTC = self._vtExec.VtProgramEcu.ReadAllDtcs('ALL',True)
self._vtExec.VtXMLReport.AddText(DTC, self._testName)

ICL = self._icl2.Notices.ReadAll(True)
self._vtExec.VtXMLReport.AddText(str(ICL), self._testName)
self._timer.Sleep(1)

# Save logger ########################################################################
self._canLogger.Stop()
self._vtExec.VtPrint.DebugPrint("Start saving to log")
self._canLogger.SaveLogTextFile(testFolder)
self._vtExec.VtPrint.DebugPrint("Done saving to log")

# END TEST ###########################################################################
```