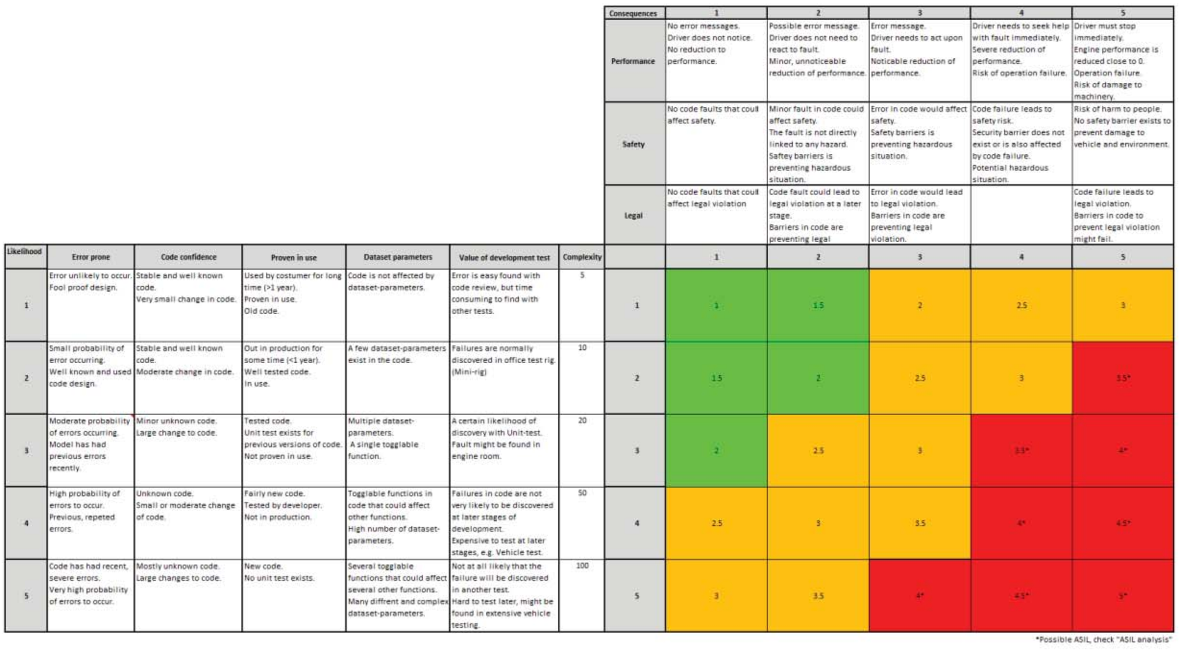


Consequences	1	2	3	4	5
Performance	No error messages. Driver does not notice. No reduction to performance.	Possible error message. Driver does not need to react to fault. Minor, unnoticeable reduction of performance.	Error message. Driver needs to act upon fault. Noticable reduction of performance.	Driver needs to seek help with fault immediately. Severe reduction of performance. Risk of operation failure.	Driver must stop immediately. Engine performance is reduced close to 0. Operation failure. Risk of damage to machinery.
Safety	No code faults that coull affect safety.	Minor fault in code could affect safety. The fault is not directly linked to any hazard. Saftey barriers is preventing hazardous situation.	Error in code would affect safety. Safety barriers is preventing hazardous situation.	Code failure leads to safety risk. Security barrier does not exist or is also affected by code failure. Potential hazardous situation.	Risk of harm to people. No safety barrier exists to prevent damage to vehicle and environment.
Legal	No code faults that coull affect legal violation	Code fault could lead to legal violation at a later stage. Barriers in code are preventing legal	Error in code would lead to legal violation. Barriers in code are preventing legal violation.		Code failure leads to legal violation. Barriers in code to prevent legal violation might fail.

Likelihood	Error prone	Code confidence	Proven in use	Dataset parameters	Value of development test	Complexity
1	Error unlikely to occur. Fool proof design.	Stable and well known code. Very small change in code.	Used by costumer for long time (>1 year). Proven in use. Old code.	Code is not affected by dataset-parameters.	Error is easy found with code review, but time consuming to find with other tests.	5
2	Small probability of error occurring. Well known and used code design.	Stable and well known code. Moderate change in code.	Out in production for some time (<1 year). Well tested code. In use.	A few dataset-parameters exist in the code.	Failures are normally discovered in office test rig. (Mini-rig)	10
3	Moderate probability of errors occurring. Model has had previous errors recently.	Minor unknown code. Large change to code.	Tested code. Unit test exists for previous versions of code. Not proven in use.	Multiple dataset-parameters. A single togglable function.	A certain likelihood of discovery with Unit-test. Fault might be found in engine room.	20
4	High probability of errors to occur. Previous, repeted errors.	Unknown code. Small or moderate change of code.	Fairly new code. Tested by developer. Not in production.	Togglable functions in code that could affect other functions. High number of dataset-parameters.	Failures in code are not very likely to be discovered at later stages of development. Expensive to test at later stages, e.g. Vehicle test.	50
5	Code has had recent, severe errors. Very high probability of errors to occur.	Mostly unknown code. Large changes to code.	New code. No unit test exists.	Several togglable functions that could affect several other functions. Many diffrent and complex dataset-parameters.	Not at all likely that the failure will be discovered in another test. Hard to test later, might be found in extensive vehicle testing.	100

	1	2	3	4	5
1	🟩 1	🟩 1.5	🟨 2	🟨 2.5	🟨 3
2	🟩 1.5	🟩 2	🟨 2.5	🟨 3	🟥 3.5*
3	🟩 2	🟨 2.5	🟨 3	🟥 3.5*	🟥 4*
4	🟨 2.5	🟨 3	🟨 3.5	🟥 4*	🟥 4.5*
5	🟨 3	🟨 3.5	🟥 4*	🟥 4.5*	🟥 5*

*Possible ASIL, check "ASIL analysis"

Software Unit Test

Unit Risk Prioritization

Riskprioritering av units

Bachelor of Science Thesis

Erik Andersson

Simon Börjeson

Department of Signals and Systems S2

Division of control, automation and mechatronics

CHALMERS UNIVERSITY OF TECHNOLOGY

Gothenburg, Sweden, 2014

Software Unit Test

Unit Risk Prioritization

Erik Andersson
Simon Börjeson

Department of Signals and Systems S2

CHALMERS UNIVERSITY OF TECHNOLOGY

Gothenburg, Sweden, 2014

Abstract

The upcoming version of the ISO standard 26262 requires all software to be unit tested by the developers. The work is about improving Volvos current risk prioritization method for software at a unit level.

As a result, Volvo has a new method to use when they perform risk analysis on units from models in Simulink. The analysis consists of two major parts, the likelihood and consequence. Both parts are dependent on a number of subcategories which is visualized as a matrix. After the risk analysis, the unit receives a unit risk prioritization score, which indicates how important it is to perform a unit test for this part of the software. The score also determines how large coverage the test should have while testing.

Another part of the work was to find a way to split models into smaller parts for unit testing. When the software engineers split down the models to an understandable size of a unit, the group found the cyclomatic complexity of the units were at a similar value. This could be measured with MATLAB Verification and Validation toolbox.

The work does not consider what kind of unit test that should be preferred to different parts of the software.

Keywords: Unit test, risk assessment, prioritization, MATLAB, Simulink.

Sammanfattning

Den kommande versionen av ISO standard 26262 kräver att all mjukvara ska testas på modulnivå med ett så kallat unit-test. Arbetet handlar om att förbättra Volvos nuvarande riskprioriteringsmetod för mjukvara på unit-nivå.

Som resultat har Volvo nu en mer omfattande metod att använda när de utför riskanalyser på units från Simulinkmodeller. Analysen består av två stora delar, sannolikheten för fel och konekvensen ett fel skulle kunna medföra. Båda delarna har flertalet underkategorier och analysen visualiseras i en matris. Efter analysen fås ett riskprioriteringsvärde, vilket avgör hur viktigt det är att utföra unit-test på mjukvaran. Värdet avgör även hur stor del av koden som testet behöver täcka.

En annan del av arbetat handlar om att finna ett sätt för att dela modeller till mindre enheter för unit-test. När mjukvareutvecklarna delade modellerna till lämplig storlek upptäckte gruppen att enheterna hade liknande cyklomatisk komplexitet, vilket kan mätas med MATLAB Verification and Validation toolbox.

Arbetet behandlar inte vilken typ av unit-test som är att föredra till olika delar av mjukvaran.

Nyckelord: Unit-test, riskbedömning, prioritering, MATLAB, Simulink.

PREFACE

This thesis work was possible due to many great people taking their time to teach a couple of rookie engineers about their areas of expertise. This bachelor thesis is the final course of our education in mechatronics (180 credits) at Chalmers University of Technology. The project has been conducted during October 2013 to February 2014 and comprises 15 credits. It has been conducted at Volvo Group Trucks Technology (GTT) in Gothenburg.

First we would like to thank our supervisors Andrea Holladay at GTT and Bertil Thomas at Signals and Systems S2, and also our examiner Manne Stenberg at Signals and Systems S2.

We would like to give special thanks to Mikael Thorvaldsson, for sharing his extensive knowledge on testing and inspiring tips on how to learn more about testing. And also special thanks Stefan Eisenberg, for helping us with daily questions and Matlab support.

Finally, we want to thank everyone who contributed with their time, opinions and knowledge. Without input from those that does the real work, this thesis would not have been possible. These people are:

Martin Wilhelmsson - Volvo GTT

Mattias Johansson - Volvo GTT

Ingemar Eckerström - Volvo GTT

Henrik Nilsson - Volvo GTT

And rest of the EATS-group.

Gothenburg Mars 2014

Erik Andersson & Simon Börjeson

Table of content

Nomenclature

ASIL – Automotive Safety Integrity Level is a risk classification in the ISO 26262.

CC – Cyclomatic complexity, a way to calculate the logical difficulty of the software.

Controllability – Ability to avoid a specified harm or damage through the timely reactions of the persons involved, possibly with support from barriers.

EATS – Exhaust After Treatment System.

EATS Group – EATS Control, The division at Volvo - Powertrain Engineering - Control Systems where this thesis work is written.

ECU – Electronic Control Unit, an embedded system with hardware and software which control functional areas of a vehicle.

Error – Discrepancy between a computed, observed or measured value or condition, and the true, specified or theoretically correct value or condition. An error can arise as a result of unforeseen operating conditions or due to a fault.

Failure – The software cannot perform a function as required.

Exposure – State of being in an operational situation that can be hazardous if coincident with the failure mode under analysis.

Fault – Some defect code such as incorrect data definition, caused by the programmer or design weakness.

GTT – Group Trucks Technology, a department at Volvo Group there most of the technical developments take place.

Harm – Physical injury or damage to the health of persons.

Hazard – Potential source of harm caused by a failure.

MIL – Model in the loop, referees to test at the model-level in Simulink.

Risk – Combination of the probability of occurrence of harm and the severity of that harm.

Barriers – External or internal measure to avoid undesirable consequences.

Severity – Estimate of the extent of harm to one or more individuals that can occur in a potentially hazardous situation.

SIL – Software in the loop, referees to test the C-code compiled from a model in Simulink.

X-team – Cross functional team. Contains members with different areas of expertise.

1 INTRODUCTION

1.1 Background

Control Systems Technology at Volvo GTT Powertrain Engineering (hereinafter abbreviated as Volvo GTT-PE) has recently started working with agile methodology, where X-teams (Cross functional team) testing of their own code are an important part. This, together with the new version of ISO 26262 that will apply to heavy duty vehicle, will make it necessary to implement standardized unit test on all software development.

The main development tool for software functions at Volvo is MATLAB. At this moment, MATLAB is missing software to perform unit test at Simulink models inside the Simulink environment. Because of this Volvo developed their own framework to construct unit test of Simulink models. This framework is able to divide a big model into smaller pieces, “units”, by extracting user defined subsystems. Currently, there are no guidelines on how to choose an appropriate size of unit.

There are also a substantial number of models, and it’s not possible to implement unit test on all models at once since the work to create these tests would take a great deal of time from development.

1.2 Purpose

The work is about developing guidelines on how to prioritize the various Simulink models available at Volvo for unit test. Due to the time consuming work of creating a unit test, all units need to be prioritized. This is done with risk assessment methodology. These guidelines should be able to help a software engineer to decide how important it is that he creates a unit test for the unit he is currently working on. The group will also examine if it is possible to find guidelines about how to divide models into appropriate sized units for testing.

1.3 Restrictions

There are several different methods of creating unit tests, with different software. The group will not research which one of these is better or worse. The group will focus on the risk assessment of units. The group is going to work closely with the EATS group and their X-teams, so all Simulink models that the group will investigate are owned by the EATS group.

1.4 Clarification of the questions

• What characteristics of a model are important from a risk-based perspective?

• What types of changes in a model can lead to increased risk or devastating consequences?

• Is it possible to find clear patterns in large models that provide suggestions for boundaries for splitting the model into smaller "units" for the test?

2 Technical Background

2.1 Software design

The software for the after-treatment system at Volvo GTT-PE is constructed with Simulink. Simulink is a MATLAB based block diagram environment for multidomain simulation and model-based design (MathWorks, 2014). The models are then converted to C-code for use inside the vehicles embedded system. The conversion is made with TargetLink, a Simulink plugin/blockset that generates C-code straight from the Simulink model (dSPACE, 2014).

2.2 ISO 26262:2011

According to ISO 26262 (functional safety for road vehicles standard) some requirements must be achieved for an acceptable level of code safety (International Organization for Standardization, 2011). Today it only affect passenger cars with a maximum weight of 3 500 kg, but the upcoming ISO standard is also going to include trucks and other vehicles. By following 26262, the vehicles software quality will be assured. At software level, the main points are standardised variable names, modular code and early testing.

Every part of the code that affects safety has to be risk analysed and placed in an ASIL class (Appendix D). The safety requirements are then defined by the four ASIL classes, A-D, where A is the least critical and D is the most critical. Parts of the code that are not classified as being in the A – D range are put in the QM category. The QM category has no safety requirements. The class depends on the severity of the hazard, how often it will be exposed and how well the driver can control the vehicle to avoid any harm during a hazard.

2.3 Unit Test

A unit is the smallest testable part of the software. In regular software code this would be a function or class (Xie, et al., 2007). At model-level it is harder to define what a unit is. When doing unit testing, the tester creates test cases for each part of the unit. The goal of each test case is to show that each part is working as intended. A collection of test cases for a unit is called a test suite, a test suite shows that the unit is working as intended.

The tester needs to define the appropriate size of a unit, the only criteria is the functionality should be very clear to understand (International Organization for Standardization, 2011). The tester creates test cases by providing specific input and expected output of the unit. When a whole test suite is finished, it’s possible to have it run automatically every time the code is built. Common practice is to have each case marked as green if it passes, red if it did not get the expected result and yellow if the test was not run.

This allows a developer to check if anything has broken while doing changes to a unit that already has an existing test suite. Unit testing also provides a living documentation of the software, since each test case has a small description of the functionality it is testing, a new developer could easily understand how the code works by running through some old test cases. Other benefits of unit testing are the ability to discover faults early in the development cycle and enhanced understanding of the code for the developer. The one big drawback with unit testing is the time it takes to create unit tests for all parts of the code if the software is really big and no previous unit tests exists.

2.4 Risk Assessment

Risk assessment is the combination of risk analysis and risk evaluation (Rausand, 2011). Risk analysis is a proactive investigation to identify hazards and estimate risk to individuals, property and the environment. This is done by systematic use of the available information of the process that is being analyzed. Risk evaluation is the judgment on the tolerability of risk with the basis in the risk analysis.

The risk assessment used in this thesis work uses a simple comparison between the possible consequences of failing code and the corresponding likelihood that the consequences will occur. These factors are added together in a diagram to give each possible part of the software a risk value. This value is later used to evaluate the need of unit testing for each part.

2.5 Unit test framework

Volvo is currently developing a framework to unit test models in MATLAB Simulink. The goal is to have a framework that works with both Model-in-the-Loop (MIL) and Software-in-the-Loop (SIL). This framework will work as a complement or replacement to the existing test methods. It is able to handle extraction of subsystems, create stubs and create a harness around the unit to control the input to the unit with the help of a signalbuilder-block. The user will define test cases by creating blocks that compares the outputs of the unit to predefined signals. It will then measure the coverage of the test by using the MATLAB toolbox; Verification and Validation. The finished test cases should be able to run automatically.

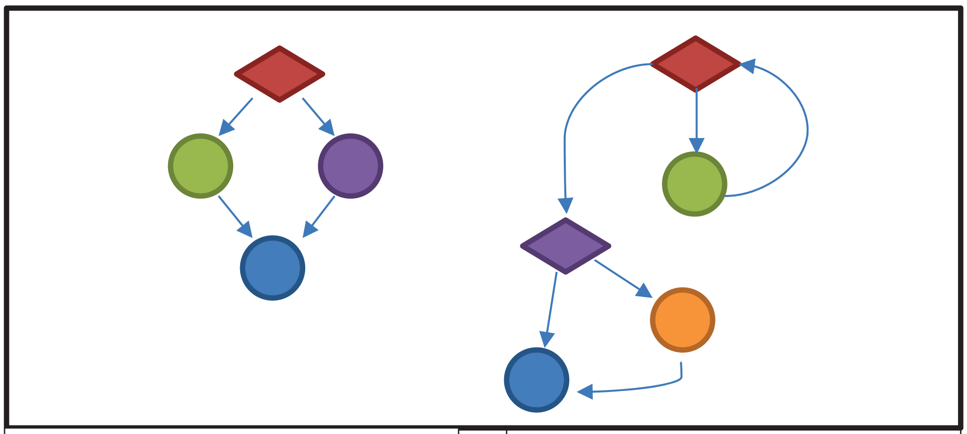
2.6 Cyclomatic Complexity

One way to estimate the probability of failure in a part of the software, like a function or class, is to measure the cyclomatic complexity (CC). Cyclomatic complexity is a value of how complex the function is (McCabe, 1976). A CC value of 1 means the function is very simple and just contains a few, if any, branches. A value of 100 means that there is a lot of different paths through the function.

This measurement was developed by Tomas J McCabe in 1976. MATLAB has a built in tool to measure cyclomatic complexity in the Verification and Validation toolbox for Simulink. CC is measured for a complete model, including the subsystems within the model. The version of McCabes formula that MATLAB uses is (MathWorks, 2013):

$$c = \sum_{\mathbf{1}}^{\boldsymbol{N}} (o_n - 1)$$

MATLAB Cyclomatic Complexity.
N = Number of Decision points.
O_n= Numbers of outcomes for n:th decision point.
c = cyclomatic complexity.
Also adds 1 for each atomic subsystem and stateflow chart.



IF – ELSE

$$c = \sum_{1}^{3} (o_n - 1)$$

$$= (2_1 - 1) + (1_2 - 1) + (1_3 - 1) = 1$$

FOR – IF

$$c = \sum_{1}^{4} (o_n - 1)$$

$$= (2_1 - 1) + (1_2 - 1) + (2_3 - 1) + (1_4 - 1) = 2$$

Figure 1 Cyclomatic Complexity examples, MathWorks formula

2.7 Coverage

According to ISO 26262, unit test should cover different parts of the software depending on the ASIL classification (International Organization for Standardization, 2011). The following coverage is recommended, but can be replaced with equivalent coverage (Hayhurst, et al., 2001).

- Statement coverage – Percentage of statements within the software that have been executed
- Branch coverage – Every outcome should been tested, both true and false in an if-statement
- MC/DC coverage – Requires every condition to independently show it affects the outcome, and should have taken all possible outcomes

With Simulink verification and validation toolbox, the following coverage is possible to measure in Simulink models (MathWorks, 2013):

- Condition coverage – Requires all possible in-port conditions
  Example, both Signal 1 and Signal 2 have to be both true and false
- Decision coverage – The test has to cover all possible outcomes
  Example, the switch should have been both true and false
- MC/DC coverage – See above
  Example, Signal 1 and Signal 2 have to be both true and false but not at the same time, the switch should have given both Constant A and Constant B

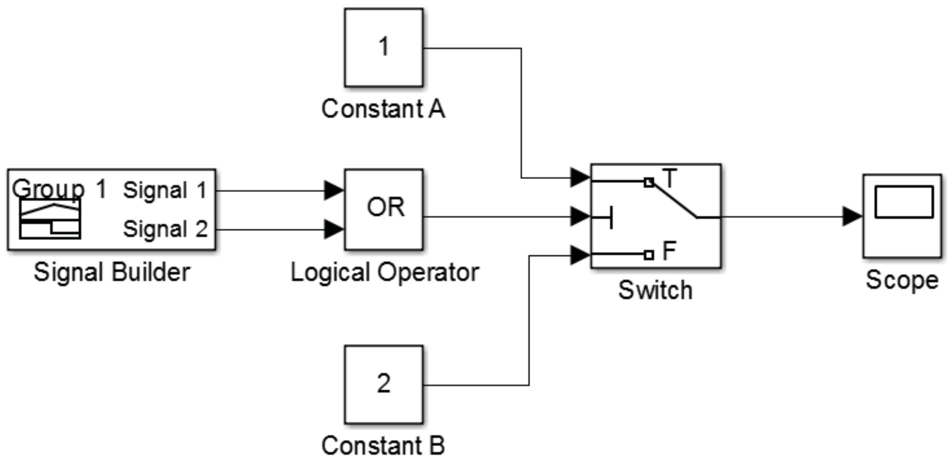


Figure 2 An if-statement built into Simulink

When running tests in MIL there is no way to analyse statement coverage, since Simulink does not support that type of coverage measurement. It should be replaced with a condition or decision coverage which requires more test cases to receive full coverage, but is supported in Simulink. The replacement is based on the assumption that all statement is executed with full condition or decision coverage (Cornett, 1996). The reason for not having statement coverage in a model environment is because it's impossible to measure statement coverage on a model, it only works on lines of code.

Branch coverage is very similar to decision coverage but not completely. According to ISTQB Foundation it is possible to use decision coverage instead of branch coverage (Graham, et al., 2007). They do not give exactly the same result, but they are closely related to each other.

The conclusion is that all recommended coverage methods in ASIL, except MC/DC, need to be replaced with equal or better methods in Matlab in order to work.

2.8 Old risk prioritization method

There is an existing risk prioritization method, which was developed as a test. In this method most criticality consequence is given a criticality rating between 1 (low) to 4 (catastrophic). The other decisive part of the function is the likelihood. A selection of likelihood attributes is listed and the likelihood rating is an approximately decided average value between 1 (low) to 3 (high). After these parameters are determined, the criticality is multiplied with the likelihood and becomes the unit risk priority value, CxL. The units will have a final risk rating between 1 to 12, indicating the priority to unit test.

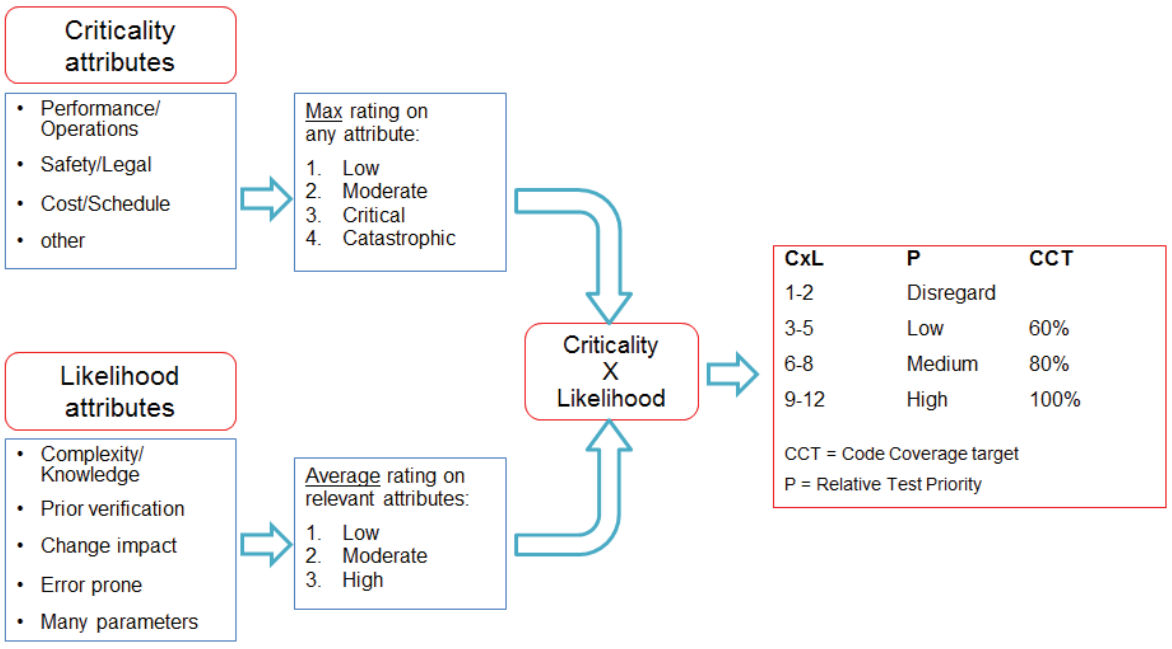


Figure 3 The old risk prioritization method

All results is stored in an Excel-file, the file contains information about the unit; names and description. Information from the risk analyse is stored as well, like; criticality rating, likelihood rating, CxL and comments about the analyse from the tester. A code coverage target (CCT) will be calculated from the CxL value and this decides how thoroughly the unit should be tested.

3 Methodology

3.1 Target

The target of the thesis work was to answer the questions stated in chapter 1.4. To be able to fully understand the challenges of creating a good risk prioritization method at Volvo GTT-PE, deeper knowledge was needed. Both in the usage of the test framework Volvo GTT-PE has developed and in basic risk assessment theories.

To find a pattern that decided where to split large model into smaller units, the group researched if there was a way to get a characteristic of a model similar to size or complexity. The most used method to measure complexity of software is cyclomatic complexity (McCabe, 1976), something that the Verification and Validation toolbox from Matlab supports. Since that toolbox is used in Volvo GTT-PEs test framework, it was ideal for the purpose of deciding how to split large models.

3.2 Collection of information

The main part of information for the thesis work came from interviews (Appendix E) and meetings with the software engineers in the EATS Group. The topics of the interviews were about different testing methods, what type of Simulink models they worked with and what type of faults that were common. The meetings were about unit extraction and testing the early risk assessment method, meeting notes can be found in appendix G. The opinions from these meetings were valuable information to continue work of developing the risk matrix.

The same engineers that attended the meetings were the early risk assessment method was tested, were later interviewed to get feedback on the new risk matrix. This lead to further changes in the descriptions of different levels of likelihood and consequence.

Another source of good information was the ISO standard 26262. This allowed to group to adapt the new method to the standards recommendations.

3.3 Tests

Tests with CC measurement tool were done on the units that the software engineers chosen during the unit extraction meetings. These were simple tests with the goal to see if a common denominator could be found in CC value. These were done with small string of code that toggled several parameters that were necessary for the CC tool to work properly.

3.4 Development of risk matrix

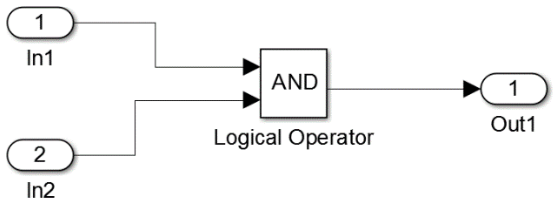
The risk matrix was based on the consequence – likelihood comparison that was already used by the old method. There is several other methods for conducting risk assessment (Rausand, 2011), these were not researched since this method was already in place.

The matrix has been in constant development during the thesis work with input from different stakeholders. The goal has always been to both have a working matrix that is easy to understand and still captures most of the different situations that affects the risk assessment. It has been adapted to be used together with the ASIL-classification from ISO standard 26262. This was done since the ASIL-standard was found to better handle the risk regarding the safety aspect.

4 Results

4.1 Stubs

The group wrote a program in MATLAB with m-code to manipulate Simulink models. The stub function was created to enable control of the output-signals from any block in Simulink. The program looks in an excel-file for all blocks that will be stubbed and search the model for these blocks. The previous blocks is deleted and all out-ports is replaced with blocks who is controlled by a signal builder in the harness. All previous in-ports can be analysed in the harness outside the model.



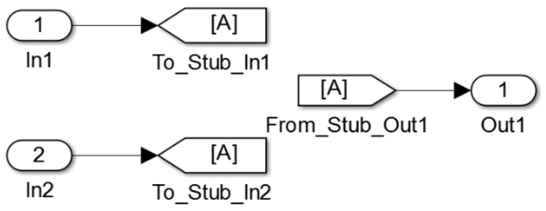


Figure 4 A block in Simulink before the stub

Figure 5 A stubbed block in Simulink

Since multiple blocks can be stubbed at the same time, some logical problems are involved. It is unnecessary to stub a block that exists within a subfunction that also will be stubbed. If two blocks who will be stubbed is connected between out-port and another stub in-port, their connection must be excluded during the creation of the stub. It is unnecessary to control a signal that not will do anything.

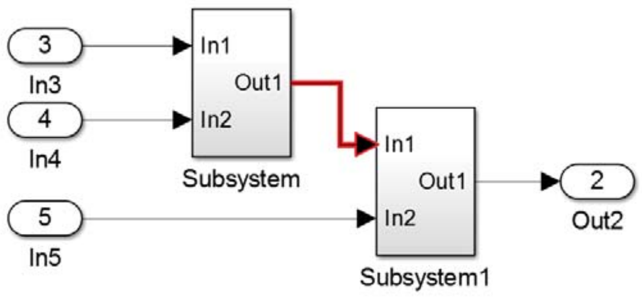


Figure 6 If both subsystems will be stubbed, it is unnecessary to control the out-port from the first stub.

A problem with the program is to set the correct data type to new in and out-ports. This makes it difficult to run SIL-tests. The unit test development team found a solution by tracing the source and destination of the in- and out-ports, and use the same data type.

4.2 Selecting models / Interviews

Short interviews were held with some of the software engineers in the EATS group. The goal with the interviews was to learn how the engineers approached testing today, what they think is common problems and errors, which models they think is safe and which is complex and problematic. A complete list of all questions and answers can found in Appendix G. The answers were later used as a basis for model selection toward complexity tests and to get better understanding of the characteristics of models from the developer's view.

4.3 Testing the method

A small test group of six persons were convened to try out the old test method and find what could be improved. The group split the models into units, in this method a unit was described as a subfunction where the tester thought the functions and risks were very clear. Later, these units were used to test cyclomatic complexity (4.5). Afterwards the risk prioritization method was applied to the units. The results from this could be found in Appendix B. During these meetings it was discussed if a later test could make unit test less important. While the prioritization a few different decision factors were discovered that impacts the likelihood of a fault:

- Proven in use, old functional code is considered very stable and safe as long as no changes are made
- Knowledge about the code, how well the engineer knows the code he is working with
- Confidence in the code, have it been several old errors in the code
- How many Dataset parameters there is, they can, in some cases, have big impact on different functionalities in the code

4.4 New risk prioritization method

With a risk prioritization it will be possible to decide which is the most critical unit and has the highest priority to unit test. The new prioritization method is based on the former one, but with some improvement. Just as the earlier method there are two major parts, likelihood and consequences.

In the consequence section all possible hazards is analysed. The worst possible consequence caused by the software is ranked 1 to 5. Some standard consequences are listed in the method, but if the tester could think of something more severe they should use that instead. Possible consequences:

- Performance - Any possible error who might affect the performance of the operation
- Legal - If a fault might lead to legal violation

Consequences	1	2	3	4	5
Performance	No error messages. Driver does not notice. No reduction to performance.	Possible error message. Driver does not need to react to fault. Minor, unnoticeable reduction of performance.	Error message. Driver needs to act upon fault. Noticeable reduction of performance.	Driver needs to seek help with fault immediately. Severe reduction of performance. Risk of operation failure.	Driver must stop immediately. Engine performance is reduced close to 0. Operation failure. Risk of damage to machinery.
Legal	No code faults that could affect legal violation	Code fault could lead to legal violation at a later stage. Barriers in code are preventing legal violation.	Error in code would lead to legal violation. Barriers in code are preventing legal violation.	Not applicable	Code failure leads to legal violation. Barriers in code to prevent legal violation might fail.

Figure 7 The matrix shows the guidelines for the consequence analysis, also shown in Appendix C

Several properties affect the probability of a fault to exist in the code. The more likely the code is to include faults the more likely is it for a devastating consequence to occur. In this method the tester should make an estimated average value from all likely parameters. Following parameters all affects the likelihood of a fault. In some software is it possible for more parameters to affect the likelihood. If the tester knows something which affects the likelihood, it should also be considered.

- Error prone - A unit with a lot of previous errors, is more likely to include an error again
- Code confidence - This category includes two parts. The tester specifies how much code knowledge he or she has and how large amount of the code is changed
- Proven in use - Based on analysis of field data from use of a unit indicates it is unlikely to include faults
- Dataset parameters - Dataset-parameters makes the software complex and likely to include rarely run software, with a large number of path it is likely not every combination have been tested
- Value of development test - Does not really affect the probability of a fault, is affected of the necessity to perform a unit test. The value of an early development test increases if the possible fault will be found at a late stage in the development
- Complexity - A complex unit may be hard to understand, therefore the unit´s cyclomatic complexity affects the likelihood of a fault

Likelihood	Error prone	Code confidence	Proven in use	Dataset parameters	Other Verification	Complexity
1	Error unlikely to occur. Fool proof design.	Stable and well known code. Very small change in code.	Used by costumer for long time (>1 year). Proven in use. Old code. Well tested with this configuration.	Code is not affected by dataset-parameters.	Fault is easy found with code review	5
2	Small probability of error occurring. Well known and used code design.	Stable and well known code. Moderate change in code.	Out in production for some time (<1 year). Well tested code with this configuration. In use.	A few dataset-parameters exist in the code.	Expected and unexpected behaviour is easily detected by other documented tests. Low cost (time and money) to correct.	10
3	Moderate probability of errors occurring. Model has had previous errors recently.	Minor unknown code. Large change to code.	Tested code. Unit test exists for previous versions of code. Not proven in use.	Multiple dataset-parameters. A single togglable function.	Expected and unexpected behaviour is most likely to be detected by other documented tests. Low cost (time and many) to correct.	20
4	High probability of errors to occur. Previous, repeated errors.	Unknown code. Small or moderate change of code.	Fairly new code. Tested by developer. Not in production.	Togglable functions in code that could affect other functions. High number of dataset-parameters.	Expected and unexpected behaviour are most likely to be detected by other verification activities or a detected fault is expensive (time and money) to correct.	50
5	Code has had recent, severe errors. Very high probability of errors to occur.	Mostly unknown code. Large changes to code.	New code. No unit test exists.	Several togglable functions that could affect several other functions. Many different and complex dataset-parameters.	Expected and unexpected behaviour are not likely to be detected by other verification activities or a detected fault is expensive (time and money) to correct.	100

Figure 8 The matrix shows guidelines for the likelihood analysis, also shown in Appendix C

The result from the risk analyses is stored in an Excel-file together with info about the analyses. The file contains information about the unit name and description. From the risk analyse is criticality rating, likelihood rating, risk and comments from the tester about the analyse stored. From the risk matrix a Code Coverage Target (CCT) will be calculated, according to the formula below, and set a target for how thoroughly the unit has to be tested. If there is an ASIL-class will the CCT automatically be set to 100% and the risk will not be below 4.0.

$$\boxed{CCT = 0.5 + 0.1 \cdot Risk}$$

4.5 Cyclomatic Complexity

To test the cylcomatic complexity measurement built in to the Verification and Validation tool, a number of different models in the ECU and AMS were chosen. Some of these were described in the interviews as “safe” or “unsafe”, models that the team felt were either very simple and fault free or complex and error prone. The group also picked some models at random from the library to get a bigger test group.

After creating a small program to configure the models so that the complexity measurement tool would work, the CC was measured across the whole model test group (Appendix E). The results were the two “safe” models had a CC value below 50. The seven models which the team described as “unsafe” all had a CC value above 230. The four random chosen had a CC value between 24 and 212.

To find what an appropriate size of a unit was, the results from the test of the old matrix was used (4.3). These units was picked as reasonable size of a unit by the test group and, in Appendix F, the Cyclomatic Complexity tool have measured all those units who was picked out during the test of the old matrix. All units except two have a CC value below 20 and these two could not be divided into smaller subfunctions.

4.6 ASIL

According to the ISO 26262 all models need to be ASIL-classed. The ASIL-level will determine what type of coverage measurements the unit tests should have for all units within the model. The analysis to decide the ASIL is done according to the ISO standard, there a matrix is used, Appendix D. Three different parameters affects the ASIL-class, all considering the vehicle’s safety, is given a value between 0-4.

- Severity - How severe any possible accident will be to any involved human
- Exposure - The time a hazard would be exposed during a fault
- Controllability - The ability of the driver to avoid a hazard

To assure the quality of an ASIL classed model the CCT (code coverage target) should be close to 100%. Models with ASIL QM are not safety concern and should have a CCT between 60% and 100% depending on the risk rating (1-5). If the wanted CCT is not achieved, the tester has to leave an explanation in the comment field in “Unit list”. The ASIL classed models have at least 4 as risk prioritization.

Different ASIL-levels requires different code coverage. Following requirements is according to the ISO 26262 with coverage technics who can be measured with MATLAB Simulink Verification and Validation toolbox. The coverage is a recommended coverage and should be achieved unless the tester has a good reason to not fulfil the target.

- ASIL QM - CCT% from "Unit list" Condition or Decision coverage.
- ASIL A - 100% Condition or Decision coverage
- ASIL B - 100% Decision coverage
- ASIL C - 100% Decision coverage
- ASIL D - 100% MC/DC coverage

5 Discussion

5.1 Stubs

Writing a function to stub blocks was a really good way to learn about the test framework. The group had a really thoroughgoing knowledge about the framework after the programming task. Unfortunately it was a bit more complex than expected, so it took a couple more weeks than planned to accomplish.

5.2 Split models

In the results 4.5 Cyclomatic Complexity it was found out all the units has a CC of about 20. This can be used to semi-automatic divide models into units. The developer have to run a script in MATLAB to check the complexity in different submodels and thereafter manually check all submodels around 20 to determine if it is reasonable to make it a unit.

5.3 Existing risk prioritization method

In the old method consequence could be ranked between 1 to 4 and the probability between 1 to 3. This low range gives all the units quite similar risk rating. By using 1 to 5 instead would make the units more separated.

The consequence and likelihood was multiplied in the old method, to create a risk score (1-12). It is unnatural to have 12 as the highest value, it could create problems when a new person take a look at the unit prioritization list. A problem with multiplication is the incomplete scale. It is impossible to achieve prime numbers such as 5, 7 and 11. Or any other number there the invalid prime number is a factor, like 10.

The existing method is also missing clear guidelines, for example some clarification about the difference between moderate and critical, catastrophic. All testers need to have the same opinion about what is a catastrophic attribute for a vehicle.

5.4 New matrix

The new developed method is quite similar to the existing one, but with some improvement. One graphical difference is the matrix, which is a common tool and is easy for many people to understand. In the matrix the tester can find some guidelines about what specifies the different levels of consequence and likelihood.

The risk prioritization value is the average value between consequence and likelihood. There is a bit different in the prioritization between multiplication and average value, as could be seen in the two matrixes below. The risk is increasing exponentially with multiplication but with average value the increase is linear. An effect of this is units who have a either a high likelihood or consequence appear to be less risky than a unit with medium likelihood and consequence, if multiplication is used.

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

	1	2	3	4	5
1	1	1,5	2	2,5	3
2	1,5	2	2,5	3	3,5
3	2	2,5	3	3,5	4
4	2,5	3	3,5	4	4,5
5	3	3,5	4	4,5	5

Figure 9 Shows the risk achieved with multiplication and average value

The old method was missing clear guidelines, in the new matrix there are descriptions to all the different consequence and likelihood ratings. It will make the testers conclusions more equal, and give them the same acceptance about what is devastating and what is safe. The risk analysis will be independent on which tester who makes the analysis.

If a unit get a high risk prioritization score it is possible the whole model has an ASIL class. All units with an ASIL class get a risk prioritization score of at least 4, according to Appendix D. The risk score is the highest value from either the ASIL analysis or the regular risk prioritization. High ASIL classed units will get a higher priority than the QM classed units. If a unit´s regular risk rating is higher, the regular risk prioritization should be used. The ASIL classed models is more important to test because they control safety regarding software.

5.5 Interviews

The questions asked during the interviews were formed in a way to get some models considered "safe" and some "unsafe". The interviews were restricted to three persons in the EATS group, all persons in different teams to cover all models in the EATS group. After the specified models were compared it was clear that the large models were considered unsafe. After some research a tool to measure the complexity was found in MATLAB Verification and Validation toolbox. This function was implemented in a program and also showed all "unsafe" models were complex and all models with a low complexity were considering "safe". Two of the models have a complexity more than twice compared to the other unsafe models. Both these models are built as a stateflow and it is possible that a higher complexity could be acceptable for stateflow charts.

5.6 Previous errors

A set of models were checked for previous errors in Serena. The aim was to find out where the error would be found and what was typical for this type of error. Even if the error descriptions are very detailed, it was unfortunately not possible to decide with our experience within different tests.

5.7 Cyclomatic Complexity

The tool used to get the complexity values is complicated and very much "under the hood". If this is to be fully understood, so that one can be completely sure about getting correct data, it would be good for the user to get some education from MathWorks on the functionality of the tool.

There is also not possible to use this tool on extracted subsystems or stubbed systems, since this will affect the overall complexity value. Different number of in- and out-ports will make the result a bit different.

The most interesting part is that when asked to pick out units, the software engineers was very consistent in that almost all units had the same level of cyclomatic complexity. This indicates that cyclomatic complexity is a good way to decide appropriate size of a unit with a few exceptions. One problem is that when you chose a “depth” in the model to divide into units, you will get some units that are above or below a value that is meaningful to do testing on.

5.8 Using the method

To use the method there are three steps.

1. Conduct ASIL analysis
2. Conduct Risk Assessment
   2.1 Maximum Consequence
   2.2 Average Likelihood
3. Take Maximum value of Risk Assessment and ASIL Risk Value.

First an ASIL analysis must be conducted to the model according to Appendix D. As example can the code decrease the engine speed, but is unlikely it would give the engine an incorrect speed and if it would it could be avoid by releasing the gas. This would give us S3, E2 and C1. That is an ASIL QM with an ASIL risk value of 1.

Then the ASIL class is determined the risk assessment should take place, Appendix C. The maximum consequence could be severe reduction of performance (4). The tester should then decide however the code reliable, if there are a lot of previous errors, how large change is, if the code has been proven in use and if the code includes rare run software. And figure out in what else test the error would be found. And finally measure the complexity with the MATLAB tool. The average of these parameters would give us the unit’s likelihood score, as example 2. The Risk assessment value would then be (4+2)/2=3.

The maximum value between the ASIL risk value(1) and the risk assessment value(3) becomes the units risk prioritization value, as in this example would be 3.

6 Conclusion

Volvo is currently missing a good method to risk prioritize units. With the new developed risk prioritization method it is possible to rank the criticality to different Simulink units. The characteristics of the risk for a unit are the likelihood for a fault to exist and the worst possible consequences any fault could bring. There are many different properties affecting the likelihood of a fault to exist in the code. The average value of following parameters is controlling the likelihood:

- Earlier error prone code.
- The tester's knowledge about the code.
- Size of the change in the code.
- If the code have been proven in use.
- If a lot of dataset parameters is used.
- The value of a unit test, however a later test will find any possible faults and how expensive it will be to correct at a later stage.
- The complexity of the unit.

The other main part affecting the risk of a unit is the possible consequence if it is a fault in the code. The worst possible of following parameters is determining the consequence:

- A failure could affect the performance of the vehicle's efficiency.
- However a fault may lead to any safety related harm.
- If a fault may lead to any legal violation.

It would be possible to develop a program to risk prioritize. But to get a good prioritization as possible the developer expertise has to be considered. This gives the developer who has been worked with the model, controllability to the outcome of the risk prioritization.

Some types of changes in a model lead to increased risk, recent code change leads to a higher risk. A large change in the code increases the risk of a fault to exist in the code, if the change is done in software there the tester has a lack of knowledge it will lead to an improved risk. If new parameters are added in the code they might affect the worst consequence and give the unit a higher risk.

To find guidelines for dividing models into units the group recommends that CC is used. It was found that almost all units with good functions to test was subfunctions with a complexity close to 20 (Appendix F). Some units had a CC above 20 because they do not include any subfunctions, so they could not be split into smaller units, these could be both time consuming and difficult to test. And units with a CC value below 5 is sometimes so simple that a test will not give the tester any meaningful information. Therefore, if a tester find units with very low or very high CC values, the tester should either change the depth of were units is chosen in a model, or evaluate if a subsystem is to complex.

6.1 Further work

The next step to move forward is to learn the group how to split units and risk prioritize, so the prioritization can begin. To improve the method in further work, there are some parts there a closer look would be recommended. Check the size of units build as state flow, is the cyclomatic complexity value to high or is a state flow easier to understand and does the high complexity affects the understanding of a state flow.

Extremely complex models they should be investigated to increase the readability. It is possible to decrease the complexity by changing the programing logic. I is also possible there is dead code included. Another option is to divide the models into a couple smaller models to increase the understanding of the models.

The current program to measure cyclomatic complexity would need some improvement to make it easier to use. The current version was only developed to see if complexity was something to consider while risk prioritizing and when the program runs it shows a lot of warning texts which should be removed.

In Serena it is possible to check for old errors, a next step would be to analyse these errors and categorize them in different groups. For example if it possible to find the error with unit test, and what coverage type would be required to find it.

References

Bach, J., 2013. *Agile Software Testing.* [Online]
Available at: http://www.youtube.com/watch?v=SAhJf36_u5U
[Accessed 12 February 2014].

Cornett, S., 1996. *Code Coverage Analysis.* [Online]
Available at: http://www.bullseye.com/coverage.html
[Accessed 12 Februray 2014].

Dabney, J. & Harman, T. L., 2004. *Mastering Simulink.* Upper Saddle River, NJ: Prentice Hall.

dSPACE, 2014. *TargetLink Automatic production code generator.* [Online]
Available at: http://www.dspace.com/en/pub/home/products/sw/pcgs/targetli.cfm
[Accessed 18 Februari 2014].

Graham, D., Veenendaal, E. V., Evans, I. & Black, R., 2007. *Foundations of Software Testing: ISTQB Certification.* London: Thomson Learning.

Hayhurst, K., Veerhusen, D., Chilenski, J. & Rierson, L., 2001. *A Practical Tutorial on Modified Condition/Decision Coverage.* [Online]
Available at: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20010057789_2001090482.pdf
[Accessed 12 February 2014].

International Organization for Standardization, 2011. *ISO 26262:2011 Road vehicles – Functional safety,* s.l.: s.n.

Kaner, C., 2010. *BBST Foundations.* [Online]
Available at: http://www.youtube.com/watch?v=2g4EqP57l7I&list=PL1C98945CECC21E22

MathWorks, 2013. *Types of Model Coverage.* [Online]
Available at: http://www.mathworks.se/help/slvnv/ug/types-of-model-coverage.html
[Accessed 18 Februari 2014].

MathWorks, 2014. *Simulink Simulation and Model-Based Design.* [Online]
Available at: http://www.mathworks.se/products/matlab/
[Accessed 18 Februari 2014].

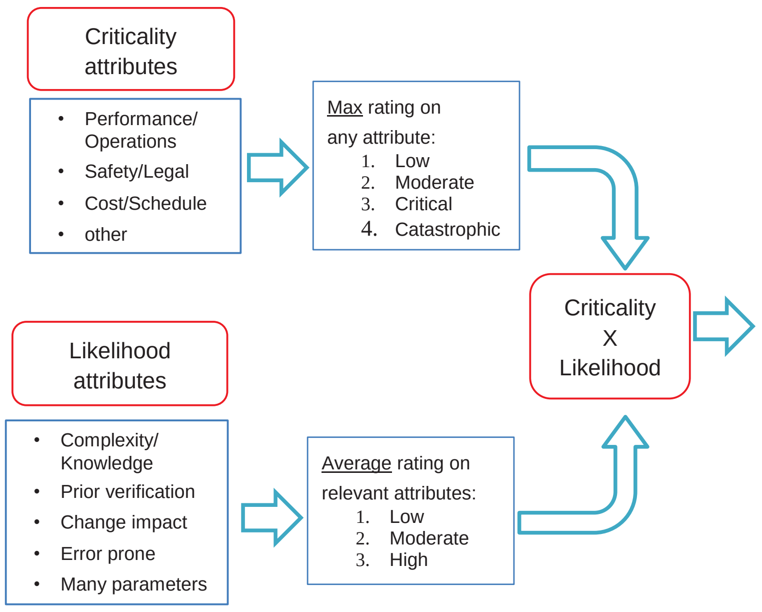
McCabe, T. J., 1976. A Complexity Measure. *IEEE Transactions on software engineering,* pp. 308-320.

Rausand, M., 2011. *Risk Assessment: Theory, Methods, and Applications.* Hoboken, New Jersey: John Wiley & Sons, Inc..

Stürmer, I., Stamatov, S. & Eisemann, U., 2009. Automated Checking of MISRA TargetLink and AUTOSAR Guidelines. *SAE International Journal of Passenger Cars - Electronic and Electrical Systems,* 10, p. 68.

Xie, T., Taneja, K., Kale, S. & Marinov, D., 2007. *Towards a Framework for Differential Unit Testing of Object-Oriented Programs,* North Carolina;Illinois : North Carolina State University; University of Illinois at Urbana-Champaign.

APPENDIX A – Old risk prioritization method



CxL	P	CCT
1-2	Disregard	
3-5	Low	60%
6-8	Medium	80%
9-12	High	100%

CCT = Code Coverage target

P = Relative Test Priority

Table used by the automatic formulas

0	-	0%
1	Disregard	0%
2	Disregard	0%
3	Low	60%
4	Low	60%
5	Low	60%
6	Meduim	80%
7	Meduim	80%
8	Meduim	80%
9	High	100%
10	High	100%
11	High	100%
12	High	100%

APPENDIX A (*continued*) – Old risk prioritization method

Unit ID	Baseline	Model	Subfunction	Unit functional description	C	L	Risk	Prioity	CCT	Consequence assumption	Likelyhood assumption	Comment
X.SF10	14A23	**X**	SF10		1	1	1	Disregard	0%			The code is not used
X.SF20	14A23	X	SF20		3	1	3	Low	60%		Simple. Not changed. Proven in use.	Can be covered by code review
X.SF301	14A23	X	SF301		3	2	6	Meduim	80%		Vital parts are proven in use.	Focus unit test on parts not covered by normal usage
X.SF302	14A23	X	SF302		3	2	6	Meduim	80%		Proven in use	

APPENDIX B – Testing old method

Unit ID	Baseline	Model	Subfunction	Unit functional description	C	L	Risk	Prioity	CCT	Consequence assumption	Likelihood assumption	Comment
X.	**14A23**	**X**										**The code is not used**
X.SF10	14A23	X	SF10		1	1	1	Wait	0%			The code is not used
X.SF20	14A23	X	SF20		3	1	3	Low	60%		Simple. Not changed. Proven in use.	Can be covered by code review
X.SF301	14A23	X	SF301		3	2	6	Meduim	80%		Vital parts are proven in use.	Focus unit test on parts not covered by normal usage
X.SF302	14A23	X	SF302		3	2	6	Meduim	80%		Proven in use	
X.SF303	14A23	X	SF303		3	2	6	Meduim	80%			
X.SF304	14A23	X	SF304		3	2	6	Meduim	80%			
X.SF305	14A23	X	SF305		3	3	9	High	100%		More complex. Subjected to change.	
X.SF306	14A23	X	SF306		3	2	6	Meduim	80%		Proven in use. Well used by many projects.	
X.SF307	14A23	X	SF307		3	1	3	Low	60%		Simple. Proven in use. Well used. Maybe 3->2.	
X.SF308	14A23	X	SF308		3	2	6	Meduim	80%			
X.SF32	14A23	X	SF32		3	1	3	Low	60%		Simple. Code not changed. Mapping functionality well used. Calibration dependent.	

APPENDIX B (*continued*) – Testing old method

X.SF33	14A23	X	SF33		3	1	3	Low	60%			
X.SF34	14A23	X	SF34		3	2	6	Meduim	80%		More complex.	
X.SF35	14A23	X	SF35		3	2	6	Meduim	80%		Well used. Any override problem is verified by later tests.	
X.SF36	14A23	X	SF36		1	1	1	Wait	0%			Not used.
X.SF37	14A23	X	SF37		3	1	3	Low	60%		Proven in use. Not complex.	
Y.	**14A23**	**Y**										
Y.SF10		Y	SF10		3	1	3	Low	60%		Not complex. Proven in use. No calibration.	Can be covered by code review
Y.SF20		Y	SF20		3	1	3	Low	60%		Not complex. Proven in use. No calibration.	
Y.SF30		Y	SF30		3	2	6	Meduim	80%		Combines several units. The added logic is limited.	A future change impact will benefit from unit test on this unit.
Y.SF301		Y	SF301		3	1	3	Low	60%		Not complex. Proven in use. No calibration.	
Y.SF302		Y	SF302		3	1	3	Low	60%		Not complex. Proven in use. No calibration.	
Y.SF303		Y	SF303		3	1	3	Low	60%		Not complex. Proven in use. No calibration.	
Y.SF304		Y	SF304		3	1	3	Low	60%		Not complex. Proven in use. No calibration.	
Y.SF305		Y	SF305		3	1	3	Low	60%		Not complex. Proven in use. No calibration.	
Y.SF306		Y	SF306		3	2	6	Meduim	80%		Proven in use. No calibration.	

APPENDIX C – New risk prioritization matrix

Consequences	1	2	3	4	5
Performance	No error messages. Driver does not notice. No reduction to performance.	Possible error message. Driver does not need to react to fault. Minor, unnoticeable reduction of performance.	Error message. Driver needs to act upon fault. Noticeable reduction of performance.	Driver needs to seek help with fault immediately. Severe reduction of performance. Risk of operation failure.	Driver must stop immediately. Engine performance is reduced close to 0. Operation failure. Risk of damage to machinery.
Legal	No code faults that could affect legal violation	Code fault could lead to legal violation at a later stage. Barriers in code are preventing legal violation.	Error in code would lead to legal violation. Barriers in code are preventing legal violation.	Not applicable	Code failure leads to legal violation. Barriers in code to prevent legal violation might fail.

APPENDIX C (*continued*) – New risk prioritization matrix

Likelihood	Error prone	Code confidence	Proven in use	Dataset parameters	Value of development test	Complexity
1	Error unlikely to occur. Fool proof design.	Stable and well known code. Very small change in code.	Used by costumer for long time (>1 year). Proven in use. Old code.	Code is not affected by dataset-parameters.	Error is easy found with code review, but time consuming to find with other tests.	5
2	Small probability of error occurring. Well known and used code design.	Stable and well known code. Moderate change in code.	Out in production for some time (<1 year). Well tested code. In use.	A few dataset-parameters exist in the code.	Failures are normally discovered in office test rig. (Mini-rig)	10
3	Moderate probability of errors occurring. Model has had previous errors recently.	Minor unknown code. Large change to code.	Tested code. Unit test exists for previous versions of code. Not proven in use.	Multiple dataset-parameters. A single togglable function.	A certain likelihood of discovery with Unit test. Fault might be found in engine room.	20
4	High probability of errors to occur. Previous, repeated errors.	Unknown code. Small or moderate change of code.	Fairly new code. Tested by developer. Not in production.	Togglable functions in code that could affect other functions. High number of dataset-parameters.	Failures in code are not very likely to be discovered at later stages of development. Expensive to test at later stages, e.g. Vehicle test. Low frequency faults.	50
5	Code has had recent, severe errors. Very high probability of errors to occur.	Mostly unknown code. Large changes to code.	New code. No unit test exists.	Several togglable functions that could affect several other functions. Many different and complex dataset-parameters.	Not at all likely that the failure will be discovered in another test. Hard to test later, might be found in extensive vehicle testing.	100

APPENDIX C (*continued*) – New risk prioritization matrix

1. Conduct ASIL analysis

2.Conduct Risk Assessment
  2.1 Maximum Consequence
  2.2 Average Likelihood

3. Take Maximum value of Risk Assessment and ASIL Risk Value.

Consequences	**1**	**2**	**3**	**4**	**5**
Performance	No effect	Possible error message. Driver does not need to react to fault. Minor, unnoticeable reduction of performance.	Error message. Driver needs to act upon fault. Noticeable reduction of performance.	Driver needs to seek help with fault immediately. Severe reduction of performance. Risk of operation failure.	Driver must stop immediately. Engine performance is reduced close to 0. Operation failure. Risk of damage to machinery.
Legal	Not legal violation	Code fault could lead to legal violation at a later stage. Barriers in code are preventing legal violation.	Error in code would lead to legal violation. Barriers in code are preventing legal violation.	Not applicable	Code failure leads to legal violation. Barriers in code to prevent legal violation might fail.

Likelihood	**Error prone**	**Code confidence**	**Proven in use**	**Dataset parameters**	**Other Verification**	**Complexity**
1	Error unlikely to occur. Fool proof design.	Stable and well known code. Very small change in code.	Used by costumer for long time (>1 year). Proven in use. Old code. Well tested with this configuration.	Code is not affected by dataset-parameters.	Fault is easy found with code review	5
2	Small probability of error occurring. Well known and used code design.	Stable and well known code. Moderate change in code.	Out in production for some time (<1 year). Well tested code with this configuration. In use.	A few dataset-parameters exist in the code.	Expected and unexpected behaviour is easily detected by other documented tests. Low cost (time and money) to correct.	10
3	Moderate probability of errors occurring. Model has had previous errors recently.	Minor unknown code. Large change to code.	Tested code. Unit test exists for previous versions of code. Not proven in use.	Multiple dataset-parameters. A single togglable function.	Expected and unexpected behaviour is most likely to be detected by other documented tests. Low cost (time and money) to correct.	20
4	High probability of errors to occur. Previous, repeated errors.	Unknown code. Small or moderate change of code.	Fairly new code. Tested by developer. Not in production.	Togglable functions in code that could affect other functions. High number of dataset-parameters.	Expected and unexpected behaviour are most likely to be detected by other verification activities or a detected fault is expensive (time and money) to correct.	50
5	Code has had recent, severe errors. Very high probability of errors to occur.	Mostly unknown code. Large changes to code.	New code. No unit test exists.	Several togglable functions that could affect several other functions. Many different and complex dataset-parameters.	Expected and unexpected behaviour are not likely to be detected by other verification activities or a detected fault is expensive (time and money) to correct.	100

Consequences		**1**	**2**	**3**	**4**	**5**
Likelihood		**MAXIMUM**				
1	AVERAGE	🟩 1	🟩 1.5	🟩 2	🟨 2.5	🟨 3
2		🟩 1.5	🟩 2	🟨 2.5	🟨 3	🟨 3.5
3		🟩 2	🟨 2.5	🟨 3	🟨 3.5	🟥 4
4		🟨 2.5	🟨 3	🟨 3.5	🟥 4	🟥 4.5
5		🟨 3	🟨 3.5	🟥 4	🟥 4.5	🟥 5

APPENDIX C (*continued*) – New risk prioritization matrix

Unit ID	Baseline	Model	Subfunction	Unit functional description	Complexity	C	L	Risk	CCT	ASIL	Consequence assumption	Likelihood assumption	Comment
X.	**14A23**	**X**											**The code is not used**
X.SF10	14A23	X	SF10		3	1	1	1	60%				The code is not used
X.SF20	14A23	X	SF20		1	3	1	2	70%			Simple. Not changed. Proven in use.	Can be covered by code review
X.SF301	14A23	X	SF301		7	3	2	2,5	75%			Vital parts are proven in use.	Focus unit test on parts not covered by normal usage
X.SF302	14A23	X	SF302		19	3	2	2,5	75%			Proven in use	

APPENDIX D – ASIL analysis

	0	1	2	3	4
Severity (S)	No injuries.	Light and moderate injuries.	Severe injuries, possibly life-threating, survival probable.	Life-threating injuries (survival uncertain) or fatal injuries.	
Exposure (E)		Very low probability.	Low probability of hazard. <1% of average operating time.	Medium probability of hazard. 1%-10% of average operating time.	High probability of hazard. >10%of operating time.
Controllability (C)	Controllable in general.	Simply controllable. 99% or more of other traffic participants are usually able to avoid a specific harm.	Normally controllable. 90% or more of all drivers or other traffic participants are usually able to avoid a specific harm.	Difficult to control or uncontrollable. Less than 90% or more of all drivers or other traffic participants are usually able to avoid a specific harm.	

		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

ASIL	QM	A	B	C	D
Coverage	CCT% from "Unit list" Condition or Decision coverage	100% Condition or Decision coverage	100% Decision coverage	100% Decision coverage	100% MC/DC coverage
Risk value	1	4	4	4.5	5

APPENDIX E – Complexity test of safe and unsafe models

Unsafe	**Complexity**	
X	232	
X	299	
X	54	Allowed to control engine functions
X	300	
X	759	
X	869	
X	244	
Safe		
X	28	
X	49	
Random unknow		
X	99	
X	212	
X	142	
X	24	

APPENDIX F – Complexity test of units

Unit ID	Complexity
X.SF10	3
X.SF20	1
X.SF301	7
X.SF302	19
X.SF303	43
X.SF304	15
X.SF305	106
X.SF306	8
X.SF307	0
X.SF308	10
X.SF32	0
X.SF33	0
X.SF34	8
X.SF35	16
X.SF36	0
X.SF37	3
X.SF30	
Y.SF10	0
Y.SF20	2
Y.SF30	0(28)
Y.SF301	0
Y.SF302	1
Y.SF303	3
Y.SF304	4
Y.SF305	4
Y.SF306	16
Y.SF31	22(44)
Y.SF313	22
Y.SF32	22
Y.SF40	8
Y.SF41	9
Z.SF10	3
Z.SF11	0
Z.SF20	7
Z.SF30	4
Z.SF31	4
W.SF10	6
W.SF20	16
W.SF30	9
W.SF40	0
V.SF100	8
V.SF101	28(47)
V.SF1013	19
V.SF102	14
V.SF110	3
V.SF111	2
V.SF112	4
V.SF113	4
V.SF114	4
V.SF115	4
V.SF116	4

APPENDIX F (*continued*) – Complexity test of units

Unit ID	Complexity
V.SF117	4
V.SF301	23
V.SF302	16
V.SF303	13(29)
V.SF3031	6
V.SF3035	10
V.SF304	27
V.SF305	30
V.SF306	12
V.SF310	6(21)
V.SF3101	15
V.SF311	11
V.SF312	12
V.SF313	13
V.SF314	1
V.SF32	5
U.SF11	0
U.SF12	10
U.SF30	44
Q.SF211	8
Q.SF212	9
Q.SF213	21
Q.SF214	12
Q.SF215	3
Q.SF216	4
Q.SF217	11
Q.SF218	20
Q.SF23	1
Q.SF31	25
Q.SF32	26
Q.SF41	13
Q.SF42	8
Q.SF43	7
Q.SF44	10
Q.SF45	10(23)
Q.SF451	13
Q.SF46	10
Q.SF47	2
Q.SF48	4
T.SF10	0
T.SF11	2
T.SF12	10
T.SF13	3
T.SF30	32

APPENDIX G – Interviews

Interview with Ingemar Eckerström

Questions:
Models
- *From a security perspective, which of the models that the group is responsible for, do you consider to be especially important to test thoroughly?*
- *Are there any "safe" models? Models that you have a good knowledge of and rarely have problems with.*
- *Do you have any models that are complex and difficult to grasp?*
- *Are there any models that historically have had a lot of problems?*

Bugs and Errors
- *Is it usual that some bugs in the models are recurring? What kind of bugs?*
- *Are there any types of mistakes that "newbies" often makes when working with the models?*
- *Have you had any serious / dangerous bugs or errors? How did you find and fixed them?*
- *Are there any other issues you would like to be able to test for, that you can't test at the moment?*

Unsafe models?	**Safe models?**	**Common bugs?**
Arbetar I X mappen.	X	Ofta svårt att avgöra innan testning av fel om det beror på:
X		Hårdvara
X		Diagnostik funktioner eller
		Dåliga sensorer

Testing ideas	**Common errors, severe faults etc**
Som ny är de lätt att missa några saker:	Allvarliga fel är om man råkar fylla katalysatorn med urea, annars inget speciellt.
Korrekt användande av xml.	
Diagnostik kedjan	
Förstå sig på den stora statemaskinen.	
Tester som skulle kunna hjälpa är regressionstester av statemaskinen.	

APPENDIX G (*continued*) – Interviews

Interview with Henrik Nilsson

Questions:
Models

- *From a security perspective, which of the models that the group is responsible for, do you consider to be especially important to test thoroughly?*
- *Are there any "safe" models? Models that you have a good knowledge of and rarely have problems with.*
- *Do you have any models that are complex and difficult to grasp?*
- *Are there any models that historically have had a lot of problems?*

Bugs and Errors

- *Is it usual that some bugs in the models are recurring? What kind of bugs?*
- *Are there any types of mistakes that "newbies" often makes when working with the models?*
- *Have you had any serious / dangerous bugs or errors? How did you find and fixed them?*
- *Are there any other issues you would like to be able to test for, that you can't test at the moment?*

Unsafe models?	**Safe models?**	**Common bugs?**
X		Samma som nämnts av Martin.
X		Typfel mm.
Ofta är det vanligare att hårdvaran felar istället för mjukvaran.	Flesta modeller är någorlunda säkra, mycket diagnosfunktioner som funkat länge.	Andra missar är mer runt att man inte förstått kravet/beskrivningen.

Testing ideas	**Common errors, severe faults etc**
Mycket miljöberoende testing är beroende av input från sensorerna.	Lätt att göra misstag vi beskrivning av funktioner som ny. Att förstå beskrivningar.
Datasatser kan störa mycket.	Lätt att missförstå info om emmissioner.
Testa med data från vagn, motorrum eller liknande är bra!	Illa om det flera system missar, leder till överhettning.
En brödbacks rigg med mer möjlighet att ange givarsignaler, motormoment och avgassmassflöde hade varit bra.	
Bra att kunna ta data direkt från tex motorum och stoppa in i matlab, men detta går ej just nu på något enkelt sätt.	(Till befintlig funktion ”SB_Signal_Loader”): Kanske också ska ha möjlighet att välja ett visst intervall i mitten av en signal.

APPENDIX G (*continued*) – Interviews

Interview with Martin Wilhelmsson, Questions:

Models

- *From a security perspective, which of the models that the group is responsible for, do you consider to be especially important to test thoroughly?*
- *Are there any "safe" models? Models that you have a good knowledge of and rarely have problems with.*
- *Do you have any models that are complex and difficult to grasp?*
- *Are there any models that historically have had a lot of problems?*

Bugs and Errors

- *Is it usual that some bugs in the models are recurring? What kind of bugs?*
- *Are there any types of mistakes that "newbies" often makes when working with the models?*
- *Have you had any serious / dangerous bugs or errors? How did you find and fixed them?*
- *Are there any other issues you would like to be able to test for, that you can't test at the moment?*

Unsafe models?	Safe models?	Common bugs?
Hög konsekvens kan regenererings ”kedjan” ha. Kolla bla på:	X	De flesta ”vanligare” buggar orsakas av okunskap från nybörjare.
-X (ca 3 år gammal funktion)		Detta då designprocessen är svår, samt även reviewen.
-X		Exempel på fel:
		Fel datatyp, tex: Bool/Float
		Insignaler i fel ”ordning”, skillnad mellan targetlink modellen och rapsody modellen.
X är också en modell som kan ha allvarliga konsekvenser, då den har tillåtelse att styra vissa motorfunktioner.	Många modeller är idag mer säkra, då mycket arbete gjorts med att bygga stabilare och bättre modeller.	Skalnings misstag, för hög nogrannhet som i längden gör att minnet ”tar slut” efter en tid och orsakar konstiga, exponensiella hopp.

Testing ideas	Common errors, severe faults etc
Viktiga tester: Regressions tester för att skydda mot nybörjarmisstag och stressade ändringar.	Tid att lära upp en ny på funktionsdesign: Några månader → ett halvår beroende på arbetstakt.
Detta då mycket kod finns i flera upplagor över många projekt, och vissa ändringar kanske funkar i ett projekt men inte i ett annat.	Saker som kan orsaka allvarligare fel är: hög personalomsättning, väldigt hög arbetsbelastning.
	Måste få utrymme att göra fel(Nybörjare).
Andra eftersökta verktyg är ett grafiskt compareverktyg för targetlink.	Tidigare upptäckta fel är billigare fel.
Vecu är bra för regressiontestning (Egentligen ett ”Checking” vertyg, kräver god kunskap om modellen av testaren)	Allvarligare fel är tex: switch som ska skicka ut float skickar en bool. Fel som gör de svårt för andra typer av testning.
Simulink unit test bra för utforskande testning, testning som kräver att saker ska snurra i cycler mm.	Många av felen kan gå långt upp i kedjan, men typ aldrig ut i kundens vagn. Upptäcks i motorrum mm.

APPENDIX H – Meeting notes, unit selection meeting

Sammanfattning, Möten. Unit urval.

2013-11-22

2013-11-17

2014-01-08

Deltagare: Mikael Thorvaldsson, Stefan Eisenberg, Martin Willhelmsson, Mattias Johansson, Christer Beskow (Närvarande: 22 november), Simon Börjesson, Erik Andersson.

Områden som påverkar sannolikhet för konsekvens att inträffa:

- Proven In Use
  - Om koden är gammal och välprövad är den säker
  - Ny och kompliserad kod eller lite äldre men okänd kod är en risk faktor
- Hur bra man kan sin kod påverkar bedömningen, är du säker på att den inte brukar krångla så sätter man lägre risk
- Hur bra förtroende man har för koden (Har den haft problem tidigare? Är de lösta?)
- Dataset parametrar kan göra det svårt att bedöma risken. Ska man bedöma från utgångspunkten att han som sätter datasattsen kommer göra ett perfekt eller dåligt jobb?
- Det finns redan vissa typer av test som fångar kända fel. Men ligger de rätt i processen?