

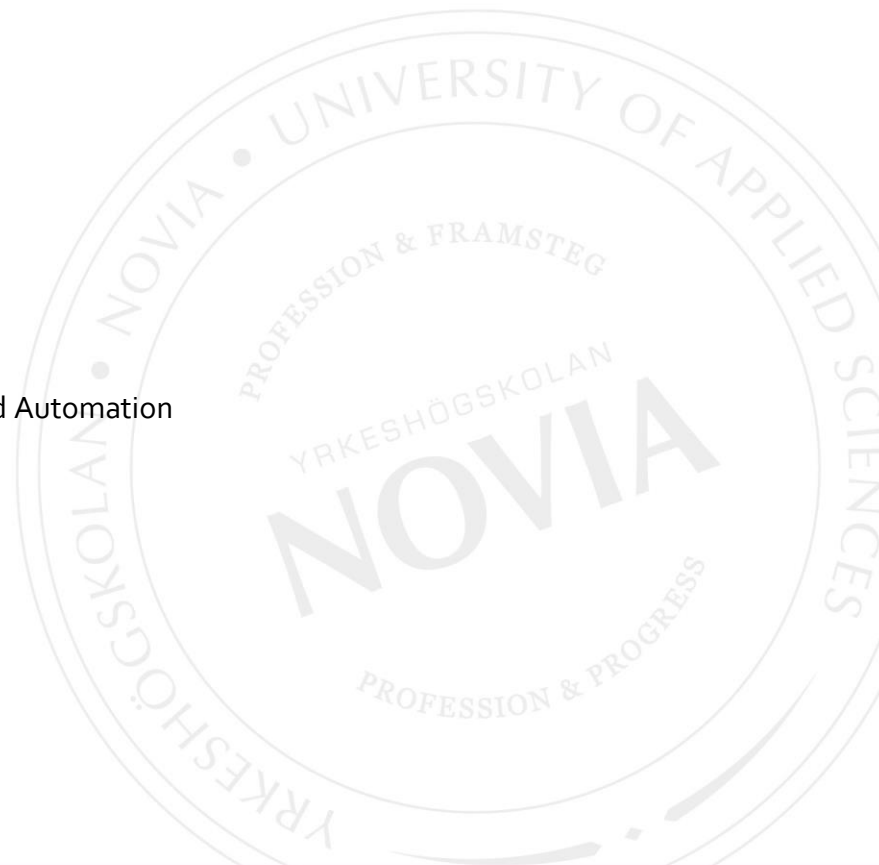
Analysis of Unit Testing Tools for Simulink Models

Johan Wahlman

Bachelor's thesis

Electrical Engineering and Automation

Vaasa 2019



BACHELOR'S THESIS

Author: Johan Wahlman
Degree Programme: Electrical Engineering and Automation
Specialization: Automation
Supervisors: Ray Pörn, Hanna Ylinen

Title: Analysis of Unit Testing Tools for Simulink Models

Date June 2, 2019

Number of pages 38

Appendices 1

Abstract

This thesis was written on behalf of the Engine Control System Development department at Wärtsilä in Vaasa. Wärtsilä's engines are controlled by many software application modules that have different functions, like controlling fuel injection, ignition, etc. Some of these applications are generated from models made in Simulink.

The purpose of this thesis was to get to know the functionality of the Simulink Test software and find out if it can be used to make unit tests for Wärtsilä's Simulink models, and if it can replace the existing unit testing functionality of Wärtsilä Simulink Development Environment. A task to implement an application's existing unit tests in Simulink Test was also given as a part of the thesis.

The theory chapter will explain what software testing is and why it is important. Furthermore, Wärtsilä's engine control system UNIC and some programs and software that the thesis will deal with, such as MATLAB and Simulink, will also be described.

The result of this thesis is a description of the relevant functionality of Simulink Test, as well as 54 unit test cases implemented in Simulink Test for an engine control application. In the discussion the question of whether Simulink Test can replace a part of WSDE will be answered.

Language: English

Key words: unit test, software testing, Simulink, model

EXAMENSARBETE

Författare: Johan Wahlman
Utbildning: El- och automationsteknik
Inriktningsalternativ: Automationsteknik
Handledare: Ray Pörn, Hanna Ylinen

Titel: Analys av testningsprogram för Simulink-modeller

Datum 2.6.2019

Sidantal 38

Bilagor 1

Abstrakt

Detta examensarbete gjordes på uppdrag av avdelningen Engine Control System Development vid Wärtsilä i Vasa. Wärtsiläs motorer styrs av många programvarumoduler eller applikationer som har olika funktioner, till exempel bränsleinsprutning, tändning och så vidare. Vissa av dessa applikationer genereras från modeller som är gjorda i Simulink.

Syftet med detta examensarbete var att få kunskap om programmet Simulink Tests funktionalitet och ta reda på om det kan användas för att göra enhetstester för Wärtsiläs Simulink-modeller, och om det kan ersätta den existerande funktionaliteten för enhetstestning som finns i Wärtsilä Simulink Development Environment. Utöver detta så skulle också en applikations existerande enhetstester implementeras i Simulink Test.

Teorikapitlet förklarar mjukvarutestning och varför den är viktig. Wärtsiläs motorstyrssystem UNIC och olika program som är relevanta för examensarbetet, till exempel MATLAB och Simulink, kommer också att beskrivas i detta kapitel.

Resultatet av detta examensarbete är en redogörelse för Simulink Tests funktionalitet samt 54 implementerade enhetstester för en applikation i Simulink Test. I diskussionen behandlas frågan om huruvida Simulink Test kan ersätta en del av WSDE.

Språk: engelska

Nyckelord: enhetstest, mjukvarutestning, Simulink, modell

Contents

1	Introduction	1
1.1	Background.....	1
1.2	Purpose and goal	1
1.3	Wärtsilä.....	2
1.3.1	History	2
1.3.2	Marine Solutions	2
1.3.3	Energy Solutions	3
1.3.4	Services.....	3
1.4	MathWorks	3
2	Theory.....	4
2.1	Software testing.....	4
2.1.1	Software testing levels.....	6
2.1.2	Black-box testing	6
2.1.3	White-box testing	7
2.1.4	Coverage.....	7
2.1.5	Usage of testing methods.....	8
2.1.6	Regression testing.....	8
2.1.7	Test automation	9
2.2	UNIC.....	9
2.3	MATLAB	10
2.4	Simulink	11
2.5	Wärtsilä Simulink Development Environment.....	11
2.6	Simulink Test	11
2.7	DevOps.....	12
2.8	Polarion	13
2.9	Operation Mode Control.....	13
3	Functionality of Simulink Test	14
3.1	Test harness	15
3.2	Test cases.....	16
3.3	Test Sequence and Test Assessment blocks	19
3.4	Test results.....	19
3.5	Coverage	21
3.6	Report	23
3.7	Requirements.....	24
3.8	DevOps.....	25
4	Implementation of unit tests.....	26

4.1	Making the test cases	28
4.2	MAT-files.....	31
4.3	Test Sequence and Test Assessment blocks	33
4.4	Problems.....	34
5	Discussion	35
6	References.....	37

Appendices

Appendix 1 Questions from initial meeting

Abbreviations

ECSD	Engine Control System Development
HIL	Hardware-in-the-loop
MCDC	Modified condition/decision coverage
PIL	Processor-in-the-loop
SIL	Software-in-the-loop
SUT	System under test
TDD	Test-driven development
UNIC	Unified Controls
WSDE	Wärtsilä Simulink Development Environment

List of Figures

Figure 1. Example processes embedded in the software development process.....	4
Figure 2. The cost of fixing bugs increases drastically over time.....	5
Figure 3. Levels of testing.....	6
Figure 4. Black-box testing concept.....	7
Figure 5. White-box testing concept.....	7
Figure 6. UNIC structure diagram.....	10
Figure 7. The Gartner toolchain.....	12
Figure 8. Operation mode control.....	13
Figure 9. The Test Manager.....	14
Figure 10. Links to external test harnesses in Simulink.....	15
Figure 11. The system under test settings.....	17
Figure 12. The coverage settings section.....	18
Figure 13. The results and artifacts view.....	20
Figure 14. Comparison of the baseline criteria and the simulated signals.....	20
Figure 15. A plot of signals from the result of a test.....	21
Figure 16. Aggregated coverage for all test cases in a suite.....	22
Figure 17. Coverage information window.....	22
Figure 18. MATLAB script for combining and displaying coverage.....	23
Figure 19. Test result report menu.....	24
Figure 20. A requirement's implementation and verification progress bars and links.	25
Figure 21. Some of the implemented unit tests in the Results and Artifacts view.....	26
Figure 22. Test harness 1.....	28
Figure 23. External input menu.....	29
Figure 24. The difference between linear and zero-order hold interpolation.....	30
Figure 25. Input data values in an Excel file.....	30
Figure 26. The Signal Editor.....	31
Figure 27. Adding MAT-file as input.....	32
Figure 28. The Test Sequence Editor.....	33
Figure 29. Inside the Test Assessment block.....	34

1 Introduction

This thesis was written for the Engine Control System Development (ECSD) department at Wärtsilä, Vaasa. I started working at ECSD in May 2018. During my employment at ECSD I have done software unit testing for an engine control application written in C, and configured engine software packages.

Software testing is an important step in the software development process, as it helps to expose defects in the software and to ensure that the software is of high quality. Some control applications for Wärtsilä's engines are made in Simulink. This thesis will investigate a Simulink model testing tool called Simulink Test, and if it can be used to test these control applications.

1.1 Background

Wärtsilä Simulink Development Environment, or WSDE, takes a lot of time and resources to develop, one reason being that MATLAB and Simulink get new releases twice a year. Due to this and some other issues with WSDE, a desire has appeared to replace WSDE with other software. One feature of WSDE is unit testing of Simulink models.

In 2015, MathWorks released Simulink Test, which has caught my department's attention. The question arose: Could we make unit tests for our models with this software, and replace this functionality in WSDE? A task to investigate this was given to me as a part of my bachelor's thesis.

To determine the most important functionality that needed to be investigated, a meeting with some application developers at ECSD was organized. During the meeting, the application developers got to express what they wanted to know about Simulink Test. The full list of questions from this meeting can be seen in Appendix 1.

1.2 Purpose and goal

The purpose of this thesis is to get to know the functionality of Simulink Test and find out if it can replace the unit testing part of WSDE and if it will improve and bring value to our software development process. Factors that will be investigated are ease of use, test creation, speed of testing, coverage calculation, report generation, requirement linking, flexibility and support for DevOps practices.

Additionally, I was also given as a challenge to take all the WSDE unit tests for an application called Operation Mode Control and try to implement them in Simulink Test.

1.3 Wärtsilä

Wärtsilä Oyj Abp is a Finnish technology company and a global leader in smart technologies and complete lifecycle solutions for the marine and energy markets. Wärtsilä's net sales was almost 5.2 billion euros with around 19 000 employees in 2018. The company operates in over 200 locations in more than 80 countries worldwide. Wärtsilä consists of two businesses: Marine Solutions and Energy Solutions. Wärtsilä Services, which was previously its own business, has been incorporated into the Marine and Energy businesses. (Wärtsilä, n.d.).

1.3.1 History

Wärtsilä was founded in 1834 and has existed for over 180 years. It started out as a sawmill in Karelia in a village called Wärtsilä. In the middle of the 1800s, Wärtsilä built an ironworks. In the 1920s, after financial problems due to a collapsed Soviet market, Wärtsilä branched out into other markets, by acquiring ship yards and engineering works in Finland. During World War II when Finland was at war with the Soviet Union, Wärtsilä produced ammunition and other metal products. In 1938, an agreement was signed with Friedrich Krupp Germania Werft AG in Germany which gave Wärtsilä permission to start manufacturing diesel engines, and the first engine was built in 1942 in Turku. After the war, Finland had to pay war reparations to the Soviet Union, and Wärtsilä became the biggest supplier of items for paying the reparations in Finland. By the end of the 1940s, Wärtsilä was the biggest company in Finland with over 11000 employees. Wärtsilä later started developing its own engines, and the first one was built in 1959 in Vaasa. In the 1970s Wärtsilä invested heavily in diesel engine technology, and later also started supplying engine powered power plants. In the 1980s, after a crisis in the marine industry, Wärtsilä closed its ship building yard in Turku. Wärtsilä started selling its smaller businesses and started focusing more on engines and power solutions. In 1997, Wärtsilä acquired Sulzer, a Swiss low-speed diesel engine maker, which made Wärtsilä a world leader in ship engines. (Wärtsilä, n.d.).

1.3.2 Marine Solutions

Wärtsilä Marine Solutions is a leading provider of ship machinery, propulsion and maneuvering solutions. Some of the products that Marine Solutions supplies are large

engines, generating sets, reduction gears, propulsion equipment, control systems and sealing solutions for various types of vessels. Marine Solutions contributed to 24% of Wärtsilä's net sales in 2018. (Wärtsilä, n.d.).

1.3.3 Energy Solutions

Wärtsilä Energy Solutions is a leading supplier of power plants comprising of engine-based flexible power plants and hybrid solar power plants. They also provide liquid gas systems, energy management systems and storage and integration solutions. Wärtsilä has an installed power plant capacity of 70 GW in over 177 countries. Energy Solutions contributed 29% of Wärtsilä's net sales in 2018. (Wärtsilä, n.d.).

1.3.4 Services

Wärtsilä Services support their customers over the lifecycle of their installations. They provide service, maintenance and reconditioning for power plants and ships. Wärtsilä Services contributed 47% of the total net sales in 2018. (Wärtsilä, n.d.).

1.4 MathWorks

MathWorks is the leading company for developing mathematical computing software. Its biggest products are MATLAB and Simulink. MathWorks was founded in 1984 and in 2017 its revenue was 900 million United States dollars. MathWorks has over 4000 employees, with 30 % of its employees located outside the United States. (MathWorks, 2018).

2 Theory

This chapter will explain relevant theory for the thesis. This includes software testing and why it is needed, the Wäertsilä embedded engine control system UNIC and some of the programs that are used or mentioned in this thesis.

2.1 Software testing

There are several definitions on what software testing is. One definition of software testing is that it is a group of procedures carried out to evaluate an aspect of some software. Another definition is that testing is a process for revealing defects in some software, and to determine that the software has reached a certain degree of quality. Testing should be a part of the software development process, which is a series of phases and procedures that produces a software product. Testing can consist of both verification and validation activities. An example of the software development process can be seen in Figure 1. (Burnstein, 2003, p. 6).

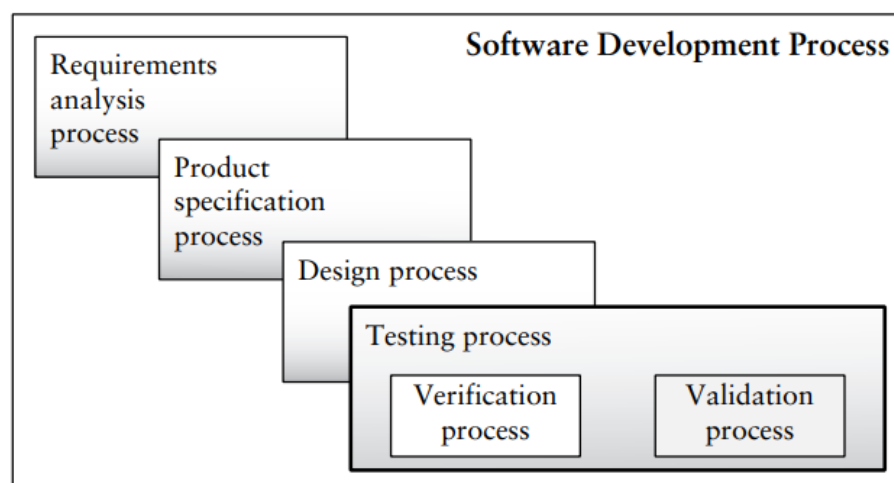


Figure 1. Example processes embedded in the software development process (Burnstein, 2003, p. 7).

Verification is a process of evaluating some software during a development phase to ensure that it meets the specified requirements. Validation is a process of evaluating some software at the end of development to ensure that it meets the customer's needs and that the specifications were appropriate to begin with. (Software Testing Fundamentals, n.d.).

According to Myers, Sandler and Badgett (2012, p. 6), software testing is the process of executing a program with the intent of finding errors. It is described that when testing a program, you want to add value to it. By testing the program to find and remove errors, you are improving the reliability and quality of the program and therefore also its value. It is also

mentioned that if the tester's intent is to show that the program has no errors, the tester will sub-consciously write tests that have a lower probability of causing an error. On the other hand, if the tester has the intent of exposing errors, the probability of finding errors will be higher, and therefore this approach will add more value to the program.

There are various types and methods of software testing. A program usually undergoes several different testing methods under its development. Most software testing techniques function in a similar way: Given a certain set of inputs, the program is expected to produce some outputs. A test is made by defining the inputs and outputs, or test data, in a test case.

A test case should contain statements of expected outputs, or it is of no value. An observed variable will be compared to an expected value, and if they match, the test case will pass. If the observed variable does not match with the expected value, the test case will fail. This will allow the tester to see if an error has been found. Test cases should be written for both valid and invalid inputs. This is because a program might receive invalid inputs for several reasons, and finding and fixing defects this way will make the program more robust. (Burnstein, 2003, p. 29).

It is important that testing is begun early and done throughout the development process so that bugs will be found as soon as possible. The later a bug is found under the development process the more costly it will be, which is illustrated in Figure 2. (Patton, 2001, p. 18).

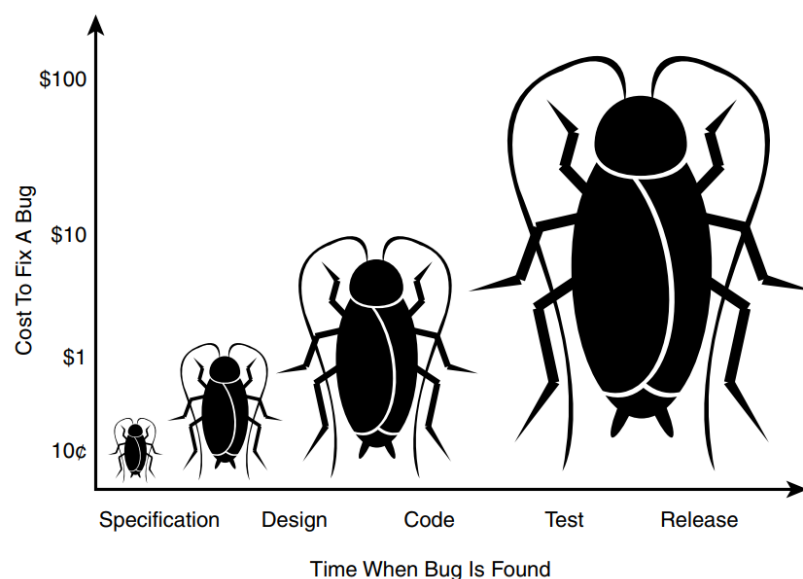


Figure 2. The cost of fixing bugs increases drastically over time. (Patton, 2001, p. 18).

2.1.1 Software testing levels

There are various levels of software testing. The primary levels of testing are unit testing, integration testing, system testing and acceptance testing. Unit testing is the lowest level of testing and focuses on an individual software component. After unit testing the individual software components and fixing any bugs that have been found, the components are combined into groups, and the next level of testing, called integration testing, can be done. After this comes system testing which focuses on the system as a whole. This level usually requires the most resources of all the levels. When system testing the tester looks for defects in the software, just like in the lower levels of testing, but the main focus of system testing is to evaluate the quality of the program, such as the performance, usability and reliability. Finally, we have acceptance testing which aims to show that the software meets all the customer's requirements. The different levels of testing can be viewed in Figure 3. (Burnstein, 2003).

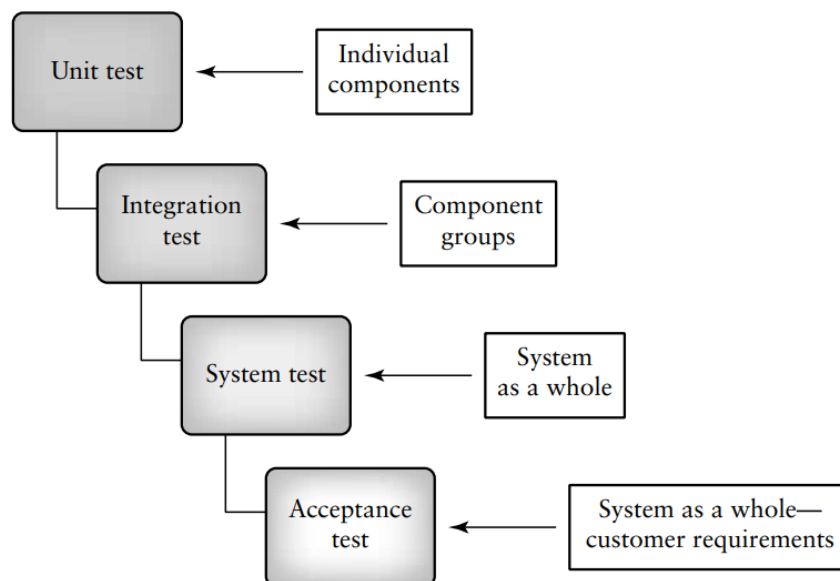


Figure 3. Levels of testing (Burnstein, 2003, p. 134).

2.1.2 Black-box testing

Black-box testing is a testing strategy in which the program is viewed as a black box. During the testing the tester should not be concerned about the internal structure of the program. The test data of the program are obtained solely from the program's specifications. (Myers, et al., 2012, pp. 8-9). The black-box testing concept can be seen in Figure 4.

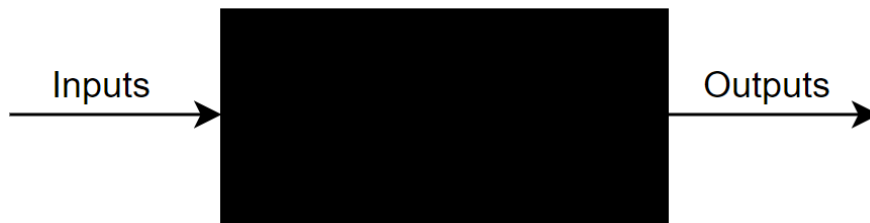


Figure 4. Black-box testing concept.

2.1.3 White-box testing

White-box testing is another testing strategy which permits the tester to look at the internal structure of the program. The test data is obtained by examining the program's logic. The tester should make sure all the statements in the program are tested, but to fully test a program with the white-box method, every possible path in the program would have to be tested. This can be near impossible to accomplish, depending on how large the program is. (Myers, et al., 2012, pp. 10-12). The white-box testing concept can be viewed in Figure 5.

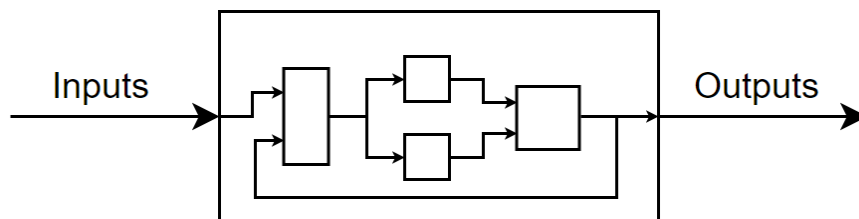


Figure 5. White-box testing concept.

2.1.4 Coverage

In white-box testing, test coverage is a way of measuring how much of a program has been executed during testing, and there are different ways of measuring this. It is measured with a percentage from 0 – 100% and the goal is to get 100% coverage.

Executing every statement in a program might seem like a good goal. This is not really sufficient as it will not find all errors, because there are usually many different paths through a program. To measure coverage this way is called statement coverage or execution coverage. (Myers, et al., 2012, pp. 43-44).

A better way to measure coverage is called decision coverage or branch coverage. This type of coverage measures that every decision statement must be executed at least once. For

example, an if-else statement must get both a true and a false outcome for full coverage. (Myers, et al., 2012, p. 44).

Condition coverage is an even stronger measurement. This way, every condition in a decision must go through all possible outcomes at least once for full coverage. (Myers, et al., 2012, p. 45).

2.1.5 Usage of testing methods

White-box testing requires that the tester knows a lot about the internal structure of the program and how it functions. White-box testing might not be very feasible in large software components where many different software developers might have contributed. Therefore, white-box testing is better suited for the unit testing level. Black-box testing can be used for both small and large programs.

It is recommended to use a combination of both black-box and white-box testing, as this makes the testing more thorough (Myers, et al., 2012, p. 83). A combination of black- and white-box testing can be called gray-box testing (Homés, 2012, p. 144).

2.1.6 Regression testing

Regression testing is the act of rerunning previously made test cases to make sure that the program still works as it should after a change to the software. Regression testing can be used to make sure that previously tested functionality has not changed, and that no previously fixed bugs can re-enter the software. Regression testing is not a testing level and can be used at all testing levels (Burnstein, 2003, p. 176).

According to Rasmusson (2010, p. 206), when a bug is found it is tempting to start trying to fix it right away. Instead, the bug should be captured by writing a failing unit test, and then it should be fixed. This is to make sure that the bug is understood and actually gets fixed. Also, by saving the unit test and running it whenever a change is made, the bug will never re-enter the software.

2.1.7 Test automation

If some software has hundreds of tests, rerunning all the test cases manually can be impossible. This is where a testing tool comes in handy. A testing tool will run all tests automatically and compare the outputs with the expected outputs. There are many benefits of using a testing tool. It is faster, more efficient and more accurate. After running a few test cases a person will get tired and mistakes become more likely. A testing tool does not get tired and as such the testing will be more accurate and precise. (Patton, 2001, pp. 220-221).

With a testing tool, a test report is important. After running all the tests, the tool will make a readable report with the test data and results, so it can be checked that the tests were run correctly or why a test failed.

2.2 UNIC

UNIC is an embedded engine control system for Wärtsilä's four-stroke engines. The name UNIC comes from Wärtsilä Unified Controls.

UNIC is a modular system with various hardware modules with different functionality. The type and number of modules used for an engine depends on the model, so that only the required modules are used. The modularity reduces unnecessary hardware and thus cost is reduced.

The UNIC software consists of three main parts: platform software, control applications and an engine specific configuration. The platform software is the lowest level software and is the foundation for the control applications and the configuration, and it handles the hardware communication.

The control applications are also modular. The various functions of the engine are controlled by different application modules. The application module that is tested later in this thesis is called Operation Mode Control and handles the operation mode of the engine. Other application modules control functions like fuel injection, ignition etc. There are around 80 different application modules for UNIC. Application modules are written in C or made in Simulink. The hierarchy and structure of the UNIC control system can be seen in Figure 6.

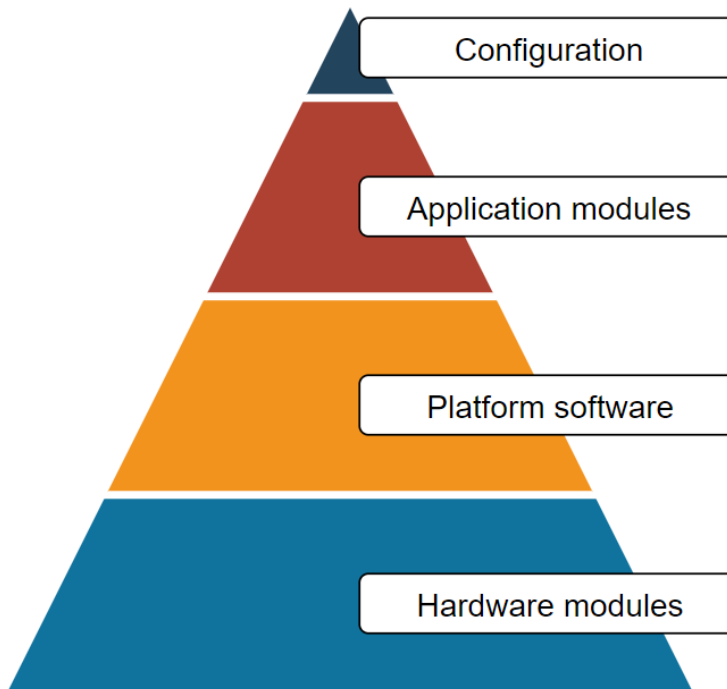


Figure 6. UNIC structure diagram.

2.3 MATLAB

MATLAB is a programming environment made by MathWorks. Its first commercial release was in 1984, and by 2018 it had 3 million users worldwide. It is designed for engineers and scientists. (Wikipedia, 2019).

MATLAB can be used for many different tasks, such as analyzing data, making complex calculations, developing algorithms and writing programs. MATLAB is also the name of the programming language that is integrated with the MATLAB software. The MATLAB programming language can express matrix and array mathematics directly which makes it well suited for computational mathematics. (MathWorks, n.d.).

MATLAB is commonly used by typing commands into a command window or running text files with MATLAB code in them. MATLAB scripts are stored in M-files and data is stored in MAT-files. MATLAB also has graphical features, such as plotting two- and three-dimensional graphs. It is also possible to mathematically manipulate and display images files. (Wikipedia, 2019).

There are many add-ons and tools for MATLAB that extends its functionality. These are usually their own products and are licensed separately.

2.4 Simulink

Simulink is a graphical programming environment made by MathWorks. Simulink uses block diagrams to make a program. These programs are called models. A model can be simulated to see how it works and values in the model can be viewed and plotted during simulation. Simulink is integrated with MATLAB, which enables the use of MATLAB algorithms in models and the exportation of simulation results to MATLAB for analysis. It is possible to automatically generate program code from a Simulink model. (MathWorks, n.d.).

There is a tool for Simulink called Stateflow, which provides a graphical programming language for making transition diagrams, flow charts and more within a Simulink model. (MathWorks, n.d.).

2.5 Wäertsilä Simulink Development Environment

WSDE is a tool for developing UNIC application modules in Simulink and it has been developed by Wäertsilä. WSDE has libraries with blocks that are used for connecting with the platform software, for example to read sensor values and to communicate with other applications. WSDE supports various functions, like hardware-in-the-loop (HIL) testing and unit testing. WSDE documentation also has guidelines on how the structure and layout of the model should be made for easy readability and uniformity between applications. (Karjalainen, 2016).

The unit testing part of WSDE has its own interface, where it is possible to add inputs and outputs, write the test data values, calculate coverage and generate reports. The tests are saved in a MAT-file and can be rerun later as regression tests.

2.6 Simulink Test

Simulink Test is a tool for testing Simulink models. With Simulink Test one can author, manage and execute systematic tests for Simulink models. These tests can be based on simulation of the model, generated code or simulated or physical hardware. Simulink Test can be used to perform functional, unit, regression and comparison testing, and these can be performed using normal, processor-in-the-loop (PIL), software-in-the-loop (SIL) or HIL modes. (MathWorks, n.d.).

2.7 DevOps

DevOps is a hot topic in software development right now. DevOps is a set of practices in software development that are supposed to make development cycles shorter and increase deployment frequency while still ensuring high quality releases. The name DevOps comes from the desire to unify software development (Dev) and software operation (Ops). DevOps comprises of tools or toolchains. There are different versions of the DevOps toolchain, but they are all quite similar. (Mala, 2019, pp. 16-17).

One example of the DevOps toolchain is the Gartner toolchain which consists of the following steps: Plan, Create, Verify, Package, Release, Configure and Monitor. These can also be divided into four major building blocks: Continuous Integration, Continuous Delivery, Continuous Deployment and Operate (De Vleeschauwer, 2017). The aforementioned can be seen in Figure 7 below.

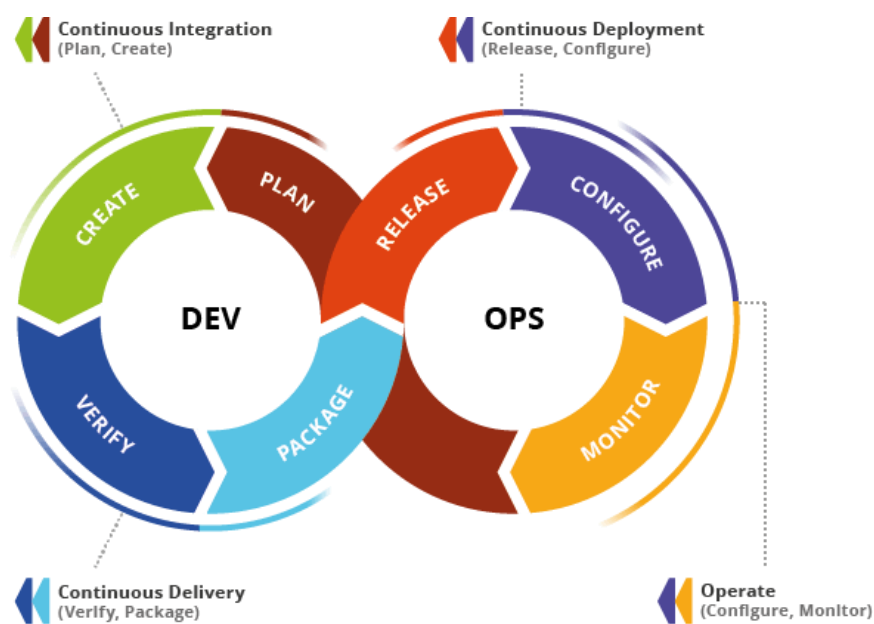


Figure 7. The Gartner toolchain (De Vleeschauwer, 2017).

DevOps advocates the use of automation to reduce manual labor and speed up development. The verify step in the toolchain above comprises of several practices such as software testing and test automation (De Vleeschauwer, 2017).

Another DevOps practice is test-driven development, or TDD.

TDD is a software development principle that helps to incrementally design software by using short development cycles. It works by first writing a failing unit test that shows exactly what the software should do. Then the code is implemented that will make the test pass. It is

recommended not to over-engineer the solution and only add the necessary code. When the test is passing, the code is cleaned up to make it more organized and readable. This part is called refactoring. (Rasmusson, 2010, p. 228).

2.8 Polarion

Polarion Software was founded in 2004, and since then they have made software to help companies with the development, governance and maintenance of software. Polarion Software was acquired by Siemens in 2016. (Siemens, n.d.).

One of their products is Polarion ALM, which is a browser-based application lifecycle management tool that Wärtsilä uses so to keep track of work items, requirements and more. The requirements for the UNIC application modules are stored in Polarion.

2.9 Operation Mode Control

Operation Mode Control is an application module made in Simulink for UNIC. It sets the currently active operation modes for the engine.

Operation Mode Control is quite small compared to many other applications and consists mainly of Stateflow state transition diagrams. There exists 54 WSDE unit tests for this application. Conforming with WSDE guidelines, the application consists of three main subsystems: Inputs, Control and Outputs. These can be seen in Figure 8.

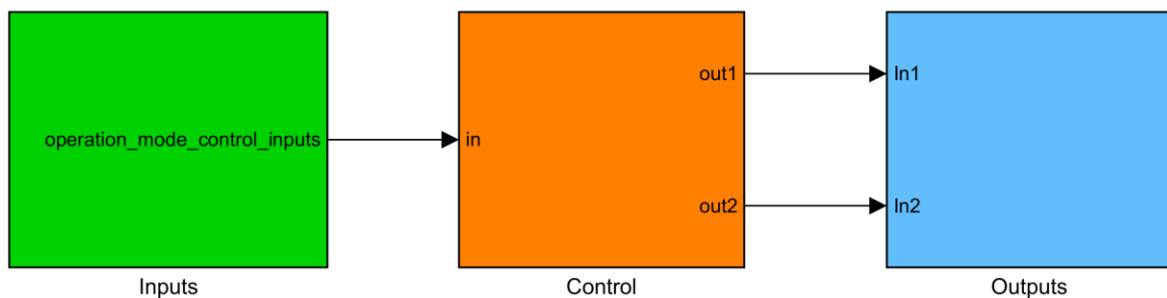


Figure 8. Operation mode control.

3 Functionality of Simulink Test

This chapter will present the main features of Simulink Test and how to use them. This information was gathered from extensive testing of the software and by reading documentation. Some information was also received from numerous evaluation meetings that were held with MathWorks spokespersons.

Some of the features will be described in more detail later in Chapter 4. The Simulink Test version that has been tested is 2017b, since this is the version of MATLAB and Simulink that ECSD uses. Thus, some of the functionality presented might have changed in a newer version.

A key component and the main user interface of Simulink Test is the Test Manager. One way to open the Test Manager is from the Analysis tab in Simulink. The Test Manager manages the creation of test files that contain test suites and test cases. To the left in the Test Manager it is possible to switch between two different views. One is called the Test Browser and the other is called Results and Artifacts. The test suites and test cases are created by right-clicking a level in the Test Browser and choosing which type of test to create. This can be seen in Figure 9.

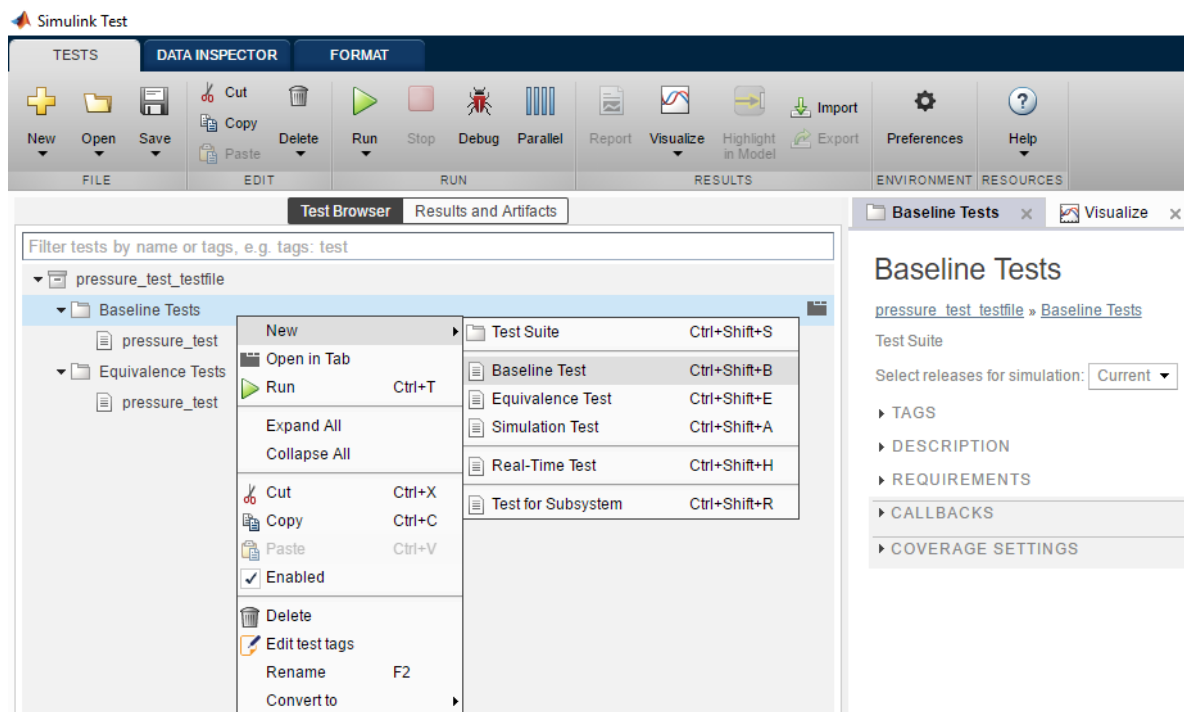


Figure 9. The Test Manager.

For example, the test suites would be placed under the test file level and the test cases would be placed under a suite. There are three main types of test cases: baseline test, equivalence test and simulation test. The test cases can be created to test an entire model or a test harness.

3.1 Test harness

Simulink Test gives the option to make test harnesses for a model. A test harness can be made for an entire model or a smaller model component, like a block or subsystem. Creating a harness will copy and isolate the component in its own model. When running a test case for a test harness, only the harness will be simulated. A test harness can be set to be synchronized with the model, so that if a change is made to the model, it will also be applied to the harness. This way, changing the model will not necessarily affect the tests. However, if inputs, outputs or signal names are changed in the model, then the test case settings and external input and output files might have to be updated with correct names. A test harness can be saved inside the main model file, or as a separate external model file. Navigation between the model and the harness is done by clicking links on the subsystem block or in the bottom right corner of the Simulink window. Links from a subsystem to some test harnesses can be seen in Figure 10.



Figure 10. Links to external test harnesses in Simulink.

Since the test harness is a separate model, it is possible to add blocks and logic to the harness without affecting the main model. This is important because unnecessary blocks are not wanted in the main model and its generated code.

A harness is created by simply right-clicking a subsystem and then choosing to create a test harness in the context menu. There are a few different types of harnesses to choose from and

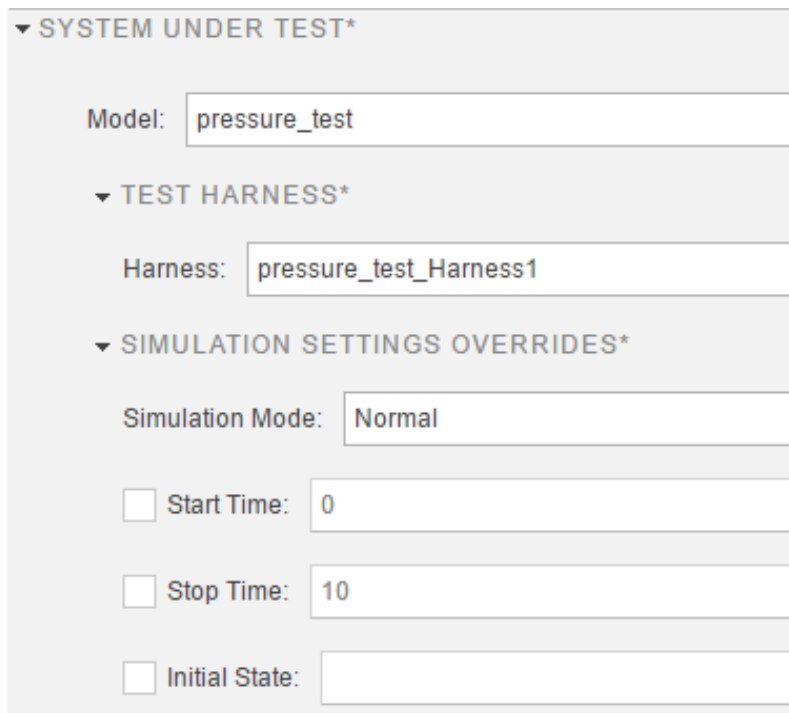
there are settings for automatically adding input ports, outputs ports, Test Sequence blocks and more.

3.2 Test cases

A baseline test case is used for comparing the outputs of the system under test (SUT) with some expected values, or baseline. The inputs and expected outputs of the test case are defined in external files of either Microsoft Excel file or MATLAB MAT-file formats. When a baseline test case is clicked in the Test Manager, all the settings of that test case will be viewed. The test case settings are well organized into different sections. Next to these sections there are clickable question marks that opens documentation for that particular section.

For the baseline test case, the settings are divided into the following sections:

1. **Tags.** Tags or keywords can be added for the test case here. There is a search feature in the Test Browser where test cases can be searched and filtered based on their tags.
2. **Description.** Here a description can be added for the test.
3. **Requirements.** Any requirement can be linked to the test case here. For this Simulink Requirements will be needed.
4. **System under test.** Here the SUT is specified. This is the model that will be tested and is either the actual model or a test harness. Here one can also override the simulation mode and the start and stop time of the test case simulation. The simulation mode can be set to be normal, SIL or PIL mode. The SUT section can be seen in Figure 11.



▼ SYSTEM UNDER TEST*

Model:

▼ TEST HARNESS*

Harness:

▼ SIMULATION SETTINGS OVERRIDES*

Simulation Mode:

Start Time:

Stop Time:

Initial State:

Figure 11. The system under test settings.

5. **Parameter overrides.** In this section one can choose to temporarily override any parameters that are defined in the model workspace.
6. **Callbacks.** There are three callbacks for a test case that can be used to run scripts at various times during a test. These times can either be pre-load, post-load or cleanup. For example, a script could be set to change some parameters or settings before a test, and then another script could clean up the changes after the test finishes. Callbacks are also available for test suites and test files.
7. **Inputs.** This is where the inputs for the test case are added from an external file. The structure of the input files and how to add them will be described in Chapter 4.
8. **Simulation outputs.** Any signal that needs to be investigated can be added here. It will then be included in the test results.
9. **Configuration setting overrides.** The model settings can be overridden by adding a configuration file here.
10. **Baseline criteria.** The external file for the expected outputs or baseline can be added here. It is also possible set tolerances for the baseline. There are four different tolerances that can be set: absolute, relative, lead and lag tolerance. It is possible to capture the baseline by clicking on capture in the baseline section. It captures the baseline by

simulating the SUT with the added inputs, recording the outputs and then saving them to a MAT-file.

- 11. Iterations.** Iterations can be used to repeat a test multiple times with different test data, instead of making many test cases.
- 12. Custom criteria.** In this section it is possible to write custom MATLAB code in an embedded editor window to make a custom test criteria script. The script is run after simulation.
- 13. Coverage settings.** In this section one can choose to select coverage for the test case. Some of the different coverage metrics that can be collected are execution coverage, decision coverage, condition coverage and modified condition/decision coverage (MCDC). The coverage settings can be set for test suites and test files as well, which will apply to all test cases under that level. The coverage settings can be seen in Figure 12.

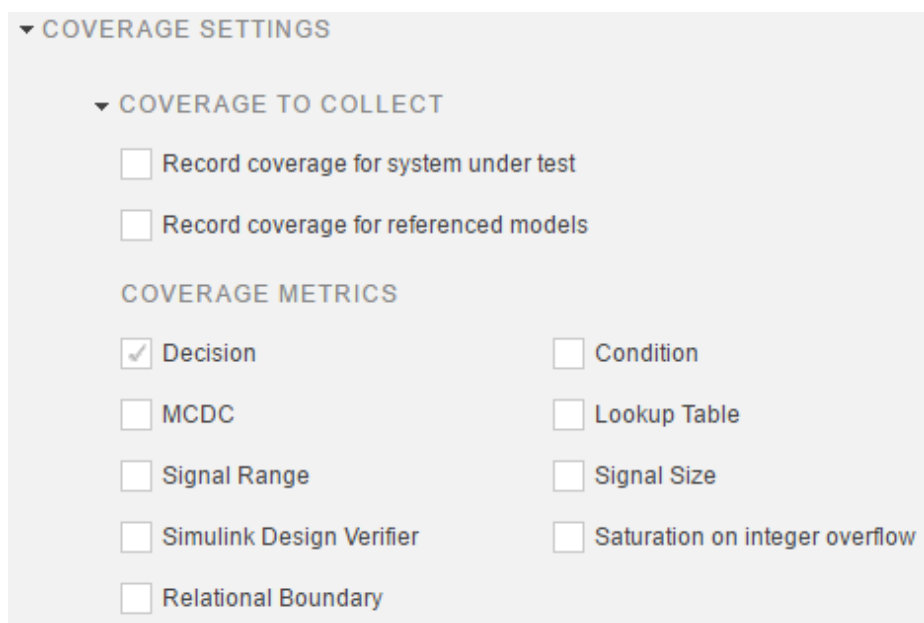


Figure 12. The coverage settings section.

The simulation test case has most of the options of the baseline test, but it does not have the baseline criteria section. Instead this type of test case is meant to mainly be used with the Test Sequence and Test Assessment blocks that will be explained later.

The equivalence test case will compare two simulations. It has settings for two different simulations, and instead of baseline criteria it has something called equivalence criteria. When the test is run, it checks if the second simulation has the same outputs as the first one.

This can be useful for checking that generated code behaves the same way as the model. Generated code can be tested by setting the simulation mode in the SUT section to SIL mode. This will automatically generate and test the code when the test is run.

With a real-time test case it is possible to do HIL testing. HIL testing is typically done after SIL and PIL testing have been done. In real-time test cases, you can add baseline or equivalence criteria or use test assessment blocks in your harness.

When asked how a new version of MATLAB and Simulink will affect saved test cases, MathWorks assured that new versions will not affect them, and they will still work in a newer version.

Simulink Test has a debugging feature, which can be found in the top of the Test Manager. Clicking debug for a test case will allow you to step through the test case simulation of the model and find out why a test case is failing.

3.3 Test Sequence and Test Assessment blocks

The Test Sequence and Test Assessment blocks come included with Simulink Test. The Test Sequence block can have inputs and outputs, and the output signals can be edited by writing MATLAB program code. This is done in the Test Sequence Editor, which is opened by double-clicking the block. The Test Sequence block functions by switching, or transitioning, between different steps and substeps, and each step can execute a set of commands.

The Test Assessment block functions in the same way. In this block, “assert” and “verify” commands can be used to check the value of model outputs. If the SUT in a test case has a Test Assessment block, these commands will determine whether the test case fails or passes.

The usage of Test Sequence and Test Assessment blocks will be explained more in Chapter 4 along with some visual examples.

3.4 Test results

The Results and Artifacts view displays the results from the tests. Things that can be seen are pass or fail status and relevant signals of the test case such as inputs and outputs. The results can be exported and saved to a file and can then be imported and examined later. A view of the Results and Artifacts view can be seen in Figure 13.

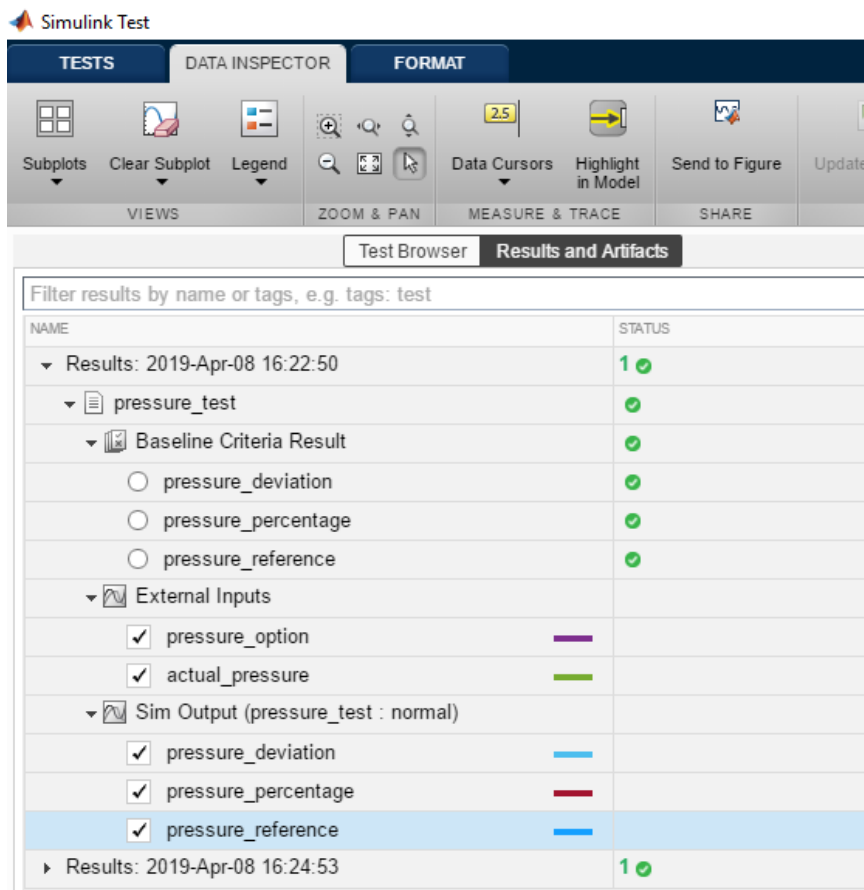


Figure 13. The results and artifacts view.

If the status field has a green ball, the test has passed, and if there is a red ball, the test has failed. When the round buttons are clicked next to the signals in the baseline criteria result, the difference between the expected and simulated signal, as well as the tolerance, will be displayed in a plot. This kind of plot can be seen in Figure 14.

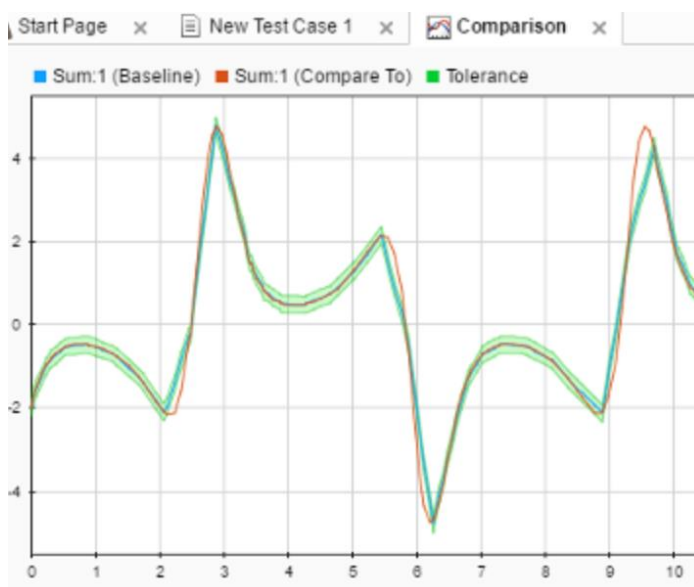


Figure 14. Comparison of the baseline criteria and the simulated signals (MathWorks, n.d.).

If the checkbox next to a signal is checked, it will plot the signal in a window. Data cursors can be used to better examine these values. A plot of some signals from a test case can be seen in Figure 15.

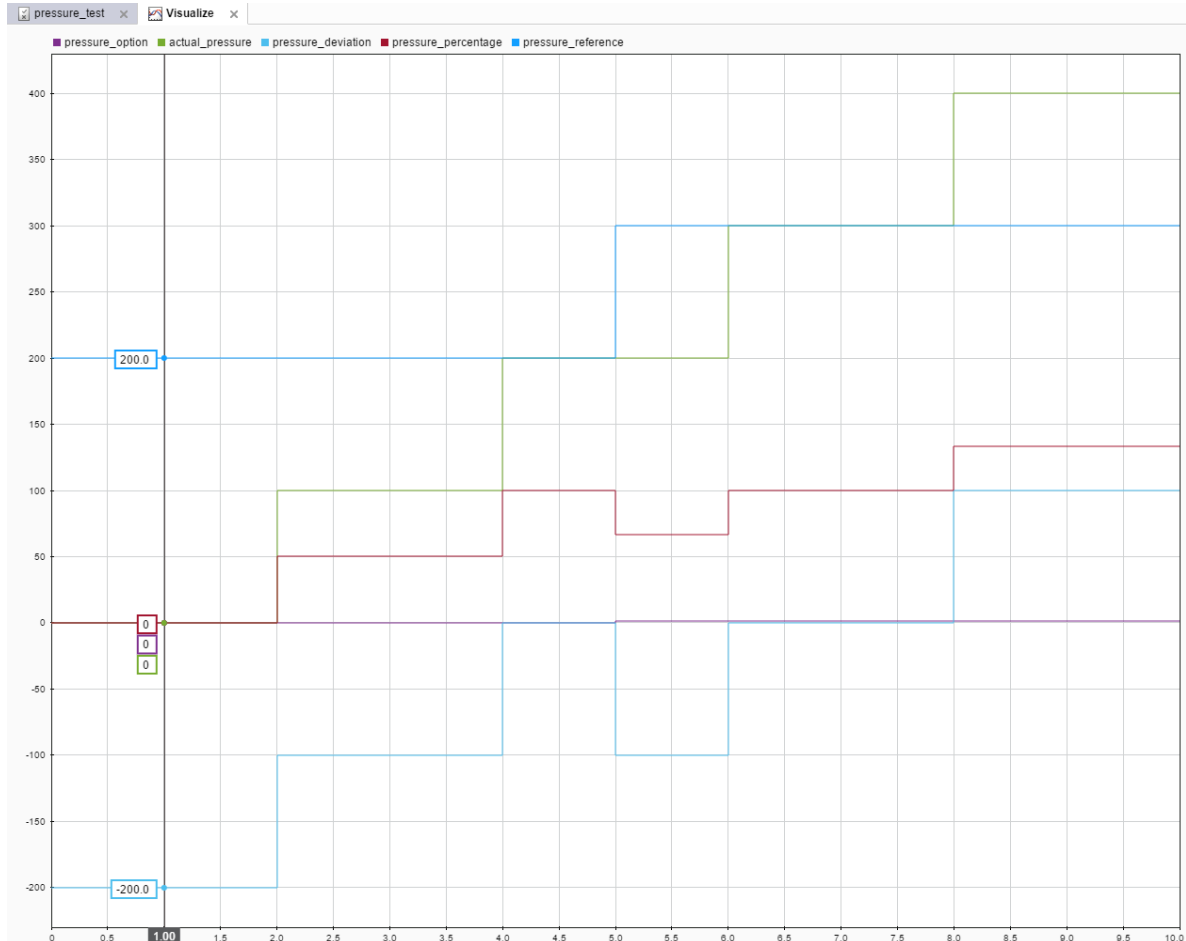


Figure 15. A plot of signals from the result of a test.

3.5 Coverage

Coverage from test cases can be collected for a SUT by enabling it in the settings of a test file, test suite or test case. After the tests have been run, the coverage can be viewed in the Results and Artifacts view. It is possible to see the combined coverage of a higher level or just the coverage of a single test case. The coverage for a test suite can be seen in Figure 16.

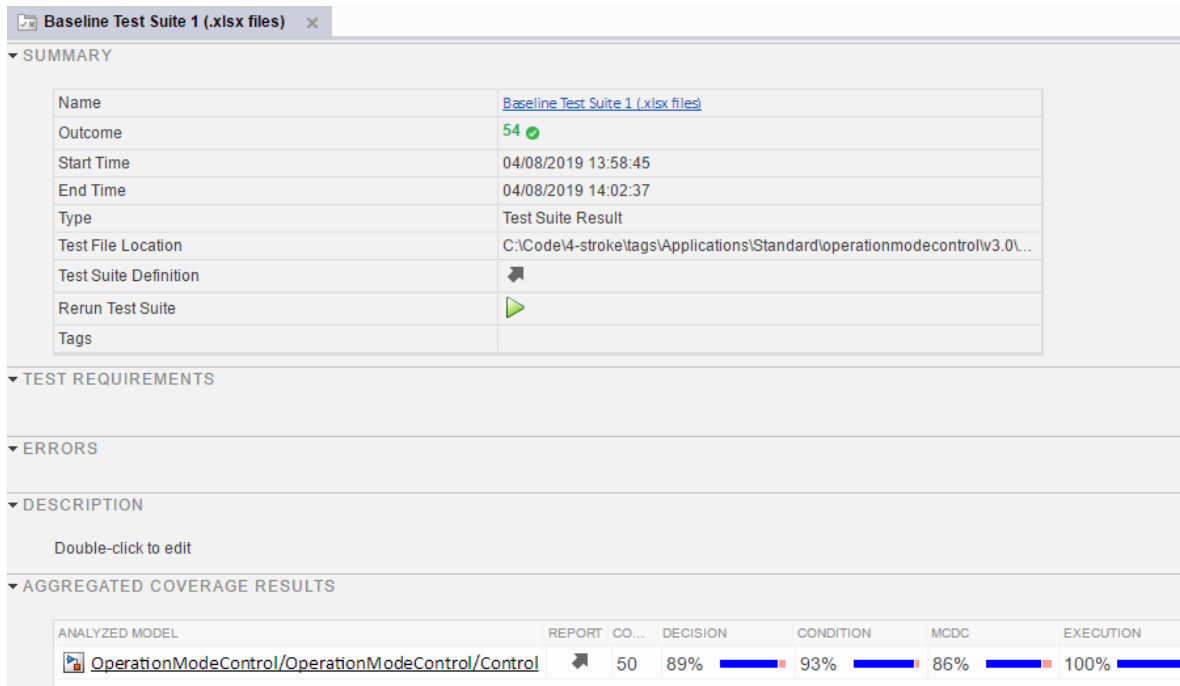


Figure 16. Aggregated coverage for all test cases in a suite.

It is possible to display what has been executed in the model by clicking the link to the analyzed model in the figure above. This will show the model with blocks and Stateflow colored in green if it has been executed with full coverage and red if it has not been executed or it does not have full coverage. Clicking blocks or Stateflow will show a window with information about the coverage for the component. This window can be seen in Figure 17.

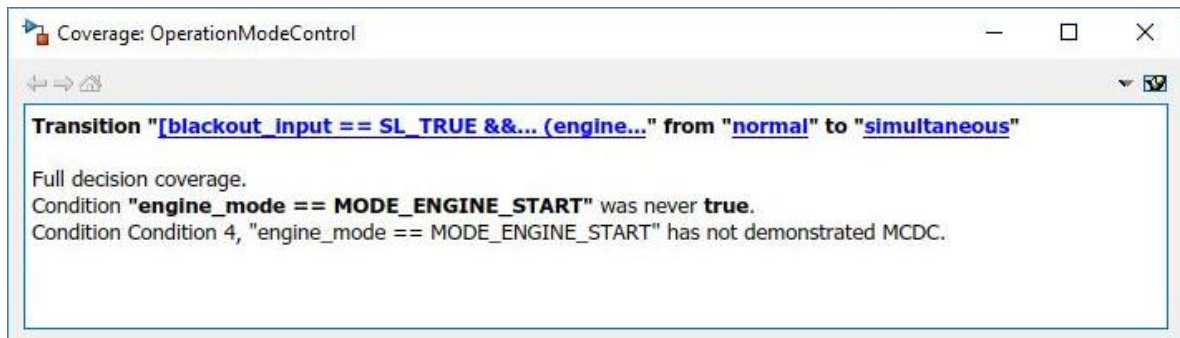


Figure 17. Coverage information window.

When making and running a new test case to increase coverage, there is no way in the user interface to combine the coverage results for the new test case to the previously run test cases. To see the new total coverage, all the test cases would have to be run again, which can take a long time depending on how many test cases there are. However, it is possible to combine the coverage from different test runs by using the MATLAB command window or making a MATLAB script. An example script that combines coverage can be seen in Figure 18.

```
% get results from Results and Artifacts list
rsList = sltest.testmanager.getResultSets;

% get the coverage from the two latest test runs
cov_run1 = rsList(end-1).CoverageResults
cov_run2 = rsList(end).CoverageResults

% combine the coverage
cov_tot = cov_run1 + cov_run2

% display the combined coverage on model
cvmodelview(cov_tot)

% generate report
cvhtml('cov_tot.html',cov_tot)
```

Figure 18. MATLAB script for combining and displaying coverage.

3.6 Report

It is possible to generate reports from the test results that can include a plethora of information. A report is created by right-clicking some test results in the Results and Artifacts view and then clicking on Generate Report. The test report menu can be seen in Figure 19.

Create Test Result Report ? X

Title Page Information

Title:

Author:

Include MATLAB version

Include in Report

Results for:

Test requirements

Plots of criteria and assessments

Plots for simulation output and baseline

MATLAB figures

Error and log messages

Simulation metadata

Coverage results

Output Options

File Format:

File Name:

Customization

Template File:

Report Class:

Figure 19. Test result report menu.

The report of the test results can be customized to some extent in the report generation menu by checking some checkboxes on what to include. The report can be customized even more by creating a custom report class in a MATLAB script. With a custom class it is possible to choose what to include in the report, such as plots and images, and to change font and layout.

3.7 Requirements

Simulink Test has integration with Simulink Requirements, and in Simulink Requirements version 2017b there is no Polarion integration.

Simulink Requirements lets you add requirements to a model. It is then possible to link them to a block or subsystem that implements the functionality of the requirement. Requirements can also be linked to a test case in Simulink Test. This also adds a link to the requirement from the test case in the Test Manager.

Progress bars of the implementation and verification of the requirements can be seen for requirement groups and different sublevels of requirements. The progress bars of a requirement and links can be seen in Figure 20.

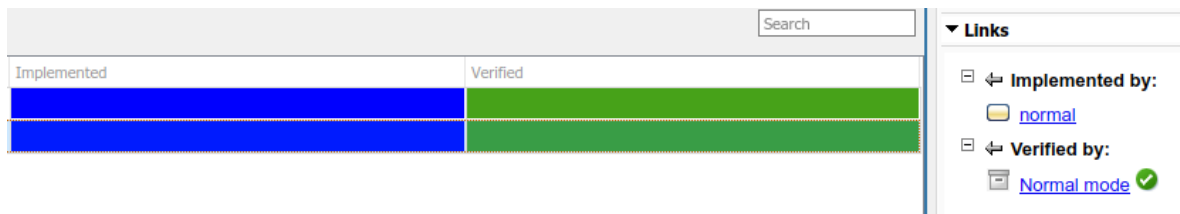


Figure 20. A requirement's implementation and verification progress bars and links.

The verification bar is green if the test has passed, yellow if the test has not been run, and red if the test failed.

To link the requirement to Polarion one could just add a hyperlink to its web address in the requirement description.

However, newer versions of Simulink Requirements work better with Polarion. In Simulink Requirements version 2018b it is possible to import ReqIF files, which is a standard XML format for requirements. The requirements in Polarion can be exported to a ReqIF file. In version 2019a this import works even better because Polarion can be set as the source. This will automatically map all the attributes of the file correctly. In version 2019a it is also possible to navigate from an imported requirement to Polarion.

There is also a Simulink extension by Polarion called Polarion Connector for Simulink. With this tool bi-directional links can be made between Polarion requirements and model elements, so that one can navigate from a block in a model to Polarion and vice versa.

3.8 DevOps

For TDD, it is possible to create an empty subsystem with inputs and outputs, and then write failing test cases for it according to the requirements. Then the functionality can be implemented to make the test pass.

Simulink Test has a programmatic interface and all its functionality can be used entirely from the MATLAB command window. It is possible to write a script that performs tasks automatically, like creating or running tests and creating reports. An automated test script could also be scheduled to run automatically. For example, the script could be set to run in the night, and then a report of the results could be read in the morning. The script could also be set to be triggered and automatically executed when committing model changes to version control.

The tests can be set to execute externally on a dedicated server, cluster or cloud with MATLAB Parallel Server, and they could also be run in parallel, with the MATLAB Parallel Computing Toolbox. This could greatly reduce the time it takes to run the tests and also completely eliminate the downtime of a developer's computer.

4 Implementation of unit tests

This chapter will present how the WSDE unit tests were implemented in Simulink Test. Several ways of making tests and some problems that were met will also be described.

The 54 WSDE unit tests for Operation Mode Control were successfully implemented in Simulink Test. The results for some of these test cases can be seen in the Results and Artifacts view in Figure 21.

NAME	STATUS
Results: 2019-Apr-08 13:58:41	63 ✓
testfile	63 ✓
Baseline Test Suite 1 (.xlsx files)	54 ✓
Normal mode	✓
Limp mode (COM-1 failure)	✓
Baseline Criteria Result	✓
AppSetOperationMode	✓
Write_IS821	✓
External Inputs	
ModuleFailureStatus	—
Sim Output (OperationModeControl : normal)	
Write_IS821	—
AppSetOperationMode	—
Limp mode (COM-2 failure)	✓
Limp mode (CCM-A1 failure)	✓
Limp mode (CCM-A2 failure)	✓
Limp mode (CCM-A3 failure)	✓
Limp mode (CCM-A4 failure)	✓
Limp mode (CCM-A5 failure)	✓
Limp mode (CCM-B1 failure)	✓
Limp mode (CCM-B2 failure)	✓
Limp mode (CCM-B3 failure)	✓
Limp mode (CCM-B4 failure)	✓
Limp mode (CCM-B5 failure)	✓

Figure 21. Some of the implemented unit tests in the Results and Artifacts view.

All the unit tests simulate and test the whole model, but they are still considered as unit tests because each UNIC application model is a smaller component, or unit, of a bigger system.

Most of the tests are derived from the requirements and specifications of the application, which characterizes the black-box testing method. However, the white-box method is also used, because the tester looks at the logic of the model and coverage of the model is also calculated, and a 100% coverage is strived for. Therefore, it can be said that the unit tests are written using a gray-box testing method.

Running all the unit tests for Operation Mode Control in WSDE took four minutes and 25 seconds and the same unit tests made in Simulink Test took three minutes and 52 seconds. Based on these results, Simulink Test was 33 seconds or 13% faster, while also collecting more coverage metrics. Simulink Test collected execution, decision, condition and MCDC coverage, while WSDE only collected the first two.

Simulink Test has many ways of testing a model. Based on how the unit tests are written in WSDE, the unit tests in Simulink Test were implemented using baseline test cases. These were quite similar to the test cases in WSDE, because of how the test inputs and expected outputs are declared. All the test cases were made using Excel files for the inputs and expected outputs. The inputs and expected outputs had their own Excel files, and every test case had its own sheet in these files.

A couple of test cases that used MAT-files were also made to see how the usage of the two file formats differed. It was later revealed that the application developers at ECSD would prefer to use MAT-files, so testing the usage of MAT-files was important.

As described in Chapter 2, Operation Mode Control has three main subsystems: Inputs, Control and Outputs. Because of this structure, a test harness for the Control subsystem was created. The test harness made it easier to set the inputs and expected outputs in the test cases. The test harness can be seen in Figure 22.

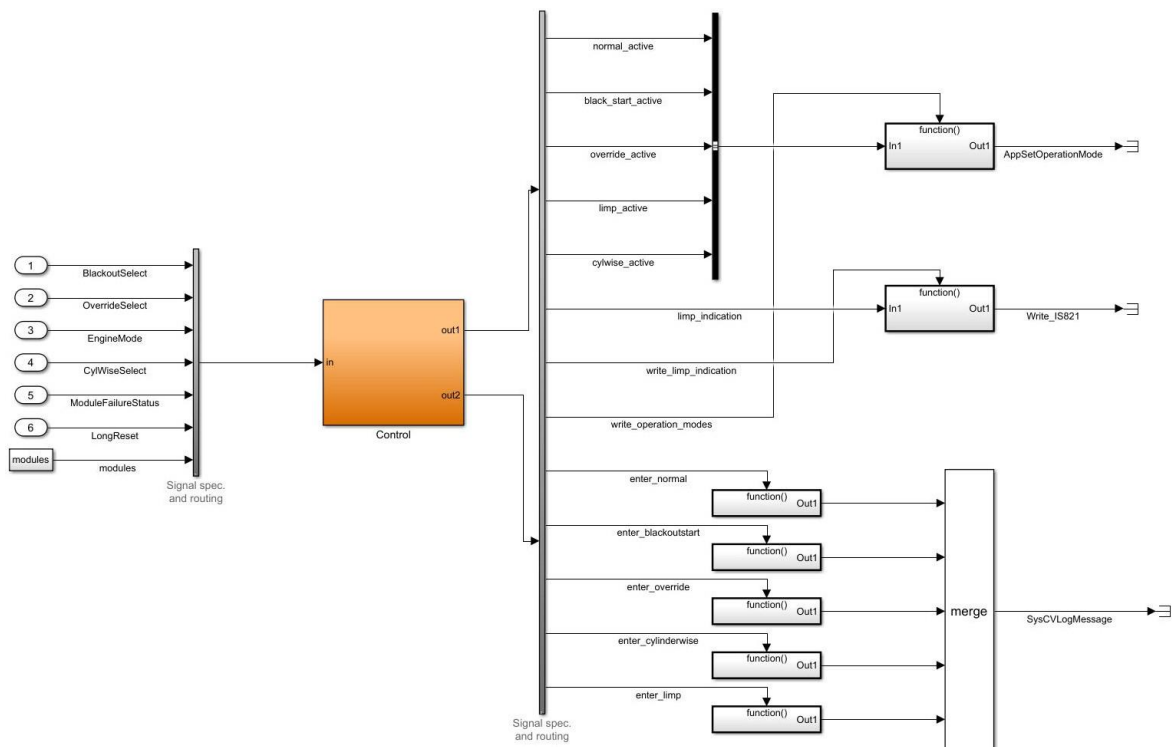


Figure 22. Test harness 1.

4.1 Making the test cases

When adding the Excel file to the input menu, it is possible to choose which sheet in the Excel file to use for the inputs. Then the inputs need to be mapped to signal name, block name, port order or some other option. This matches the names of the signal data in the Excel file or MAT-file with the corresponding signal name or block name in the model. The menu for this can be seen in Figure 23.

INPUT FILE SPECIFICATION

File: ..\inputs.xlsx

Add iterations to run this input

▼ SHEETS AND RANGE SPECIFICATION

SHEETS	RANGES
<input checked="" type="checkbox"/> Limp_mode_COM_1_failure	
<input type="checkbox"/> Limp_mode_COM_2_failure	
<input type="checkbox"/> Limp_mode_CCM_A1	
<input type="checkbox"/> Limp_mode_CCM_A2	
<input type="checkbox"/> Limp_mode_CCM_A3	
<input type="checkbox"/> Limp_mode_CCM_A4	
<input type="checkbox"/> Limp_mode_CCM_A5	

▼ INPUT MAPPING

Mapping Mode: Signal Name

Compile the system under test

▼ MAPPING STATUS

Successfully mapped inputs.

PORT	BLOCK NAME	MAPPED SIGNAL	STATUS
1	OperationModeControl_Harness 1/In1	BlackoutSelect	✓
2	OperationModeControl_Harness 1/In2	OverrideSelect	✓
3	OperationModeControl_Harness 1/In3	EngineMode	✓
4	OperationModeControl_Harness 1/In4	CylWiseSelect	✓
5	OperationModeControl_Harness 1/In5	ModuleFailureStatus	✓
6	OperationModeControl_Harness 1/In6	LongReset	✓

▶ ADVANCED

Figure 23. External input menu.

Adding the baseline Excel file is done in a similar way, but mapping the outputs is not required. Tolerances for my test cases were not needed, as the declared output values were exact.

The default interpolation for the signal data values in Simulink Test is linear. In WSDE, the unit tests for Operation Mode Control is set to interpolate the data values in a zero-order hold way. Instead of a linear and gradual change of a value, zero-order hold interpolation holds a value for a time period and then jumps directly to another value resulting in a square-wave-shaped signal. The difference between the two interpolation methods can be seen in Figure 24.

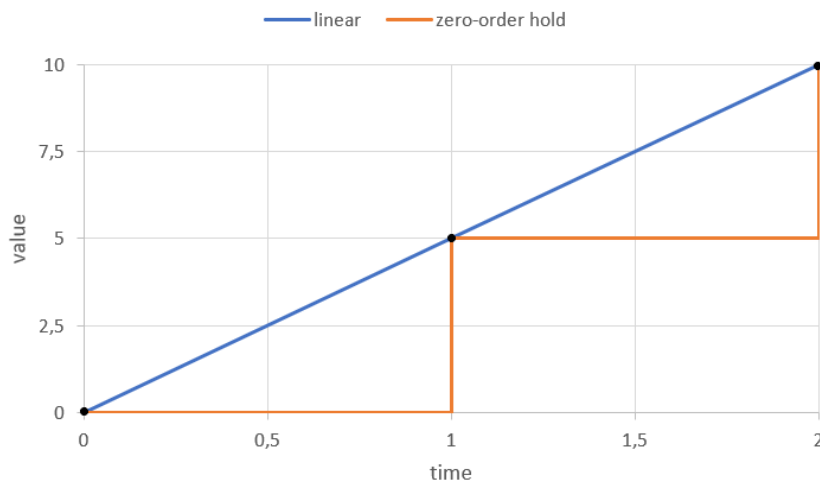


Figure 24. The difference between linear and zero-order hold interpolation.

The interpolation setting for a signal could be declared in the Excel file by writing “Interp: zoh” in a cell before the data values. It works the same way for declaring the data type, for example by writing “Type: uint8” in a cell. The data type in Excel needs to match with the data type for the signal in the Simulink model. For Simulink Test to correctly read the data from the Excel file, it needed to be structured in a certain way. The structure of the input test data for a test case can be seen in Figure 25.

	A	B	C	D
1	t	ModuleFailureStatus	EngineMode	LongReset
2		Type: uint32	Type: uint8	Type: uint8
3		Interp: zoh	Interp: zoh	Interp: zoh
4	0	1	32	0
5	5	0	1	1
6	10	0	1	1

Figure 25. Input data values in an Excel file.

The time values are declared in column A. In the first row there are names of the input signals, in the second row there are declarations of data type and the third row says what interpolation should be used. Then in the rows 4 – 6 the data values for the corresponding time are put in.

The examined outputs for each test case had to be added to the simulation output menu to make them appear in the results view. It was also desired to see the input signals in the results. This could be accomplished by checking the “Include external inputs/signal builder data in test result” in the inputs menu.

4.2 MAT-files

The MAT-files can be made by either manually creating the signals in the command window in MATLAB or by using the Signal Editor. In the Signal Editor, the data values are put into cells, much like in Excel. In the Signal Editor it was also possible to set the data type and interpolation method. When making the signals they can be visualized in a plot. Each test case could have its own scenario with signals, much like the sheets in the Excel files. The creation of the signals with the Signal Editor can be seen in Figure 26.

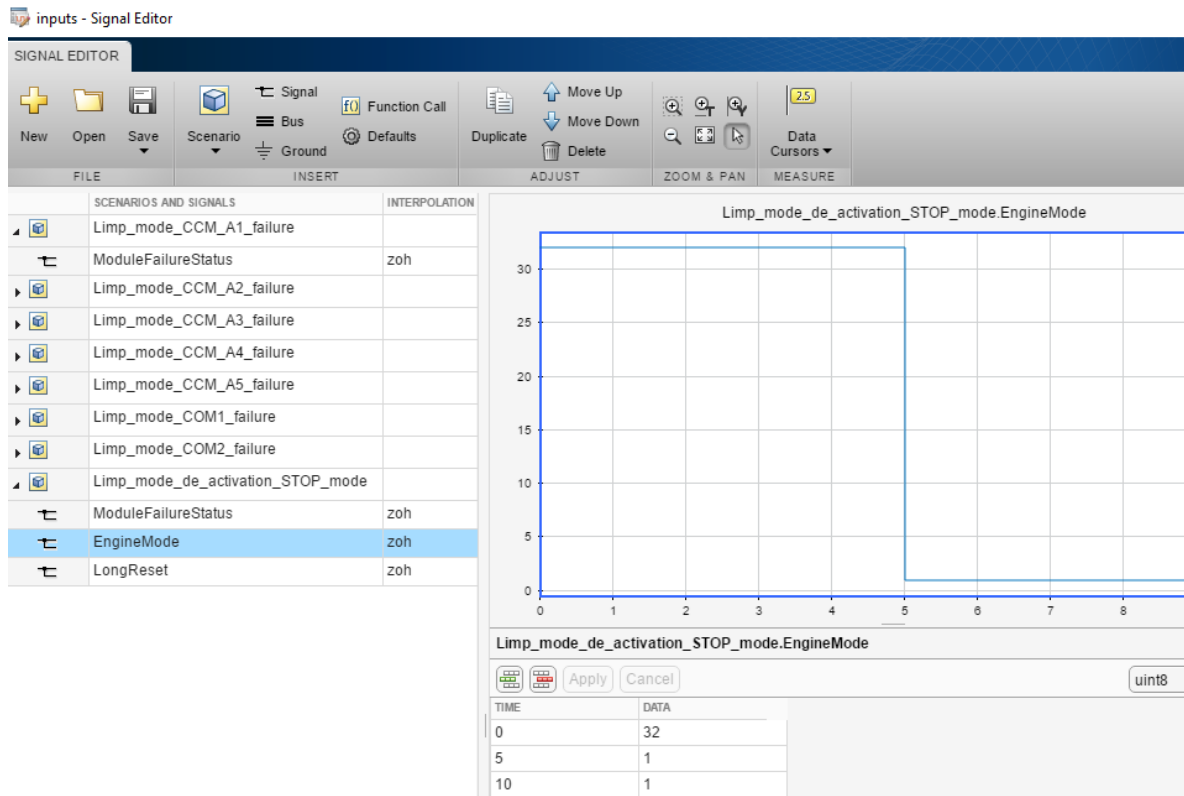


Figure 26. The Signal Editor.

When adding a MAT-file in the input menu of a test case, it was not possible to choose which scenario to use for the test case like it was for the sheets in an Excel file. However, it was still possible to make it work by editing the input string in the advanced section of the menu. The input menu for adding a MAT-file can be seen in Figure 27.

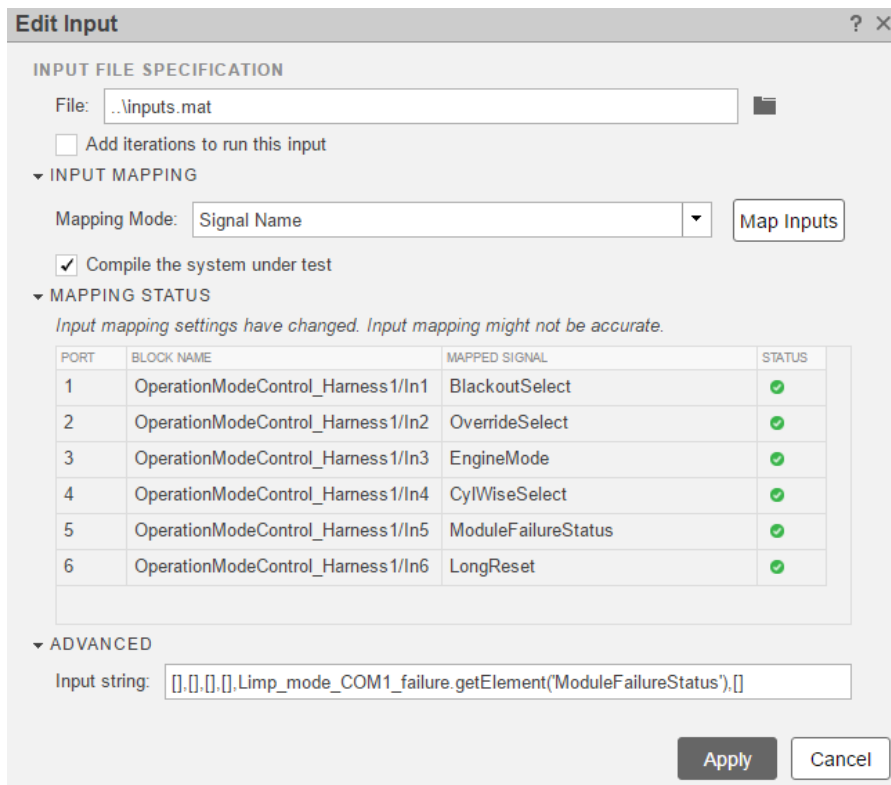


Figure 27. Adding MAT-file as input.

Each set of brackets in the input string corresponds to an input signal in the mapping status list. It was possible to add the signals from the correct scenario by manually writing “scenario_name.getElement(‘signal_name’)” in place of the brackets. The application developers at ECSD would like to use a single MAT-file for the test data, but since the scenarios cannot be selected easily like the Excel sheets, it seems that in Simulink Test it would perhaps be best to use separate MAT-files for each test case, and separate files for the inputs and baseline also.

Another issue with MAT-files was that when creating them with the Signal Editor, the signals turned up with the name “unnamed” in the test results. This was because the timeseries within a signal created by the Signal Editor is called “unnamed” and Simulink Test uses that name instead of the signal name. It was possible to fix this by manually changing the timeseries’ name in the MATLAB command window.

MathWorks was contacted about these issues and they confirmed that it is like this in the newer versions of Simulink Test as well, and that they appreciated the feedback.

4.3 Test Sequence and Test Assessment blocks

A couple of unit tests were also implemented using the Test Sequence and Test Assessment blocks to see how they worked. A separate test harness was made for this. The Test Sequence Editor can be seen in Figure 28.

Step	Transition	Next Step
Initialize_1 BlackoutSelect = 0; OverrideSelect = 0; EngineMode = 32; Cy/WiseSelect = 0; LongReset = 0; modules = 128963; ModuleFailureStatus = 0;	1. true	Normal_mode
<input type="checkbox"/> Normal_mode	1. Active_Step_Out == Active_Step_Enum.Normal_mode_end	Limp_mode_COM_1_failure
<input type="checkbox"/> Normal_mode_1 %no inputs	1. after(tbs,sec)	Normal_mode_end
<input type="checkbox"/> Normal_mode_end		
<input type="checkbox"/> Limp_mode_COM_1_failure		
<input type="checkbox"/> Limp_mode_COM_1_failure_1 ModuleFailureStatus = 1;	1. after(tbs,sec)	Limp_mode_COM_1_failure_2
<input type="checkbox"/> Limp_mode_COM_1_failure_2	1. after(tbs,sec)	Limp_mode_COM_1_failure_end
<input type="checkbox"/> Limp_mode_COM_1_failure_end		

Figure 28. The Test Sequence Editor.

In the first initialization step some signals going out from the block are declared. The transition for that step is set to true, which makes the execution jump to the next step called “Normal_mode”. By writing “after(tbs,sec)” in the other transitions, it jumps to the next step after a certain number of seconds. In this case, “tbs” is a defined constant with the value one and stands for time between step. A signal called “Active_Step_Out” was also configured. This signal connects the Test Sequence and the Test Assessment blocks, which makes it easy to check which step is currently active. This is useful when assessing the examined signals in the Test Assessment block. The editor for the Test Assessment block can be seen in Figure 29.

Step	Transition
Assessments	
<ul style="list-style-type: none"> └─ Normal_mode 	1. Active_Step_In == Active_Step_Enum.Normal_mode_end
<ul style="list-style-type: none"> └─ Normal_mode_1 when Active_Step_In == Active_Step_Enum.Normal_mode_1 <li style="padding-left: 40px;">verify(AppSetOperationMode == 1); └─ Normal_mode_2 	
<ul style="list-style-type: none"> └─ Limp_mode_common_baseline 	
<ul style="list-style-type: none"> └─ Limp_mode_common_baseline_1 when Active_Step_In == Active_Step_Enum.L <li style="padding-left: 40px;">verify(AppSetOperationMode == 32); <li style="padding-left: 40px;">verify(Write_IS821 == 1); └─ Limp_mode_common_baseline_2 	

Figure 29. Inside the Test Assessment block.

A “When decomposition” was used in the Test Assessment block. The “when” command checks if a statement is true, and if it is, it activates that step. In this case, it checks if the right step is active in the Test Sequence block, and then runs some “verify” statements which checks the value of some signals.

4.4 Problems

To simulate the model, a WSDE initialization script is run to declare some variables to the workspace. Some WSDE global variables are not found in this script and Simulink Test was not able to find these. A second initialization script that declared these global variables was created to be able to run the test cases for the model.

Getting the zero-order hold interpolation to work was problematic. Even after declaring the interpolation method as zero-order hold in the Excel file the signal value would still increase linearly. It turned out that the properties of the input ports in the test harness model had a “Interpolate data” checkbox checked by default, which forced the data to be interpolated linearly. Unchecking this made the zero-order hold interpolation work as it should.

At first, it was not possible to run the created unit tests. An error occurred that said a setting called “ReturnWorkspaceOutputs” was turned off. With this option enabled, Simulink creates a Simulink object containing the simulation outputs. Simulink Test was unable to access the simulation outputs with this setting turned off. It was not possible to turn this setting on either as it was locked. This was caused by a WSDE custom code generation TLC (Target Language Compiler) file, that calls a script that changes and locks a lot of settings

in the configuration parameter settings. This was fixed by changing a line of code in this script so that the “ReturnWorkspaceOutputs” setting was turned on. After doing so the test cases were able to be run.

The SIL mode simulation was not successfully run for the Operation Mode Control model because of errors that were caused by incompatibility with some WSDE code generation settings. However, when it was tried on another model that was made for testing it worked fine. SIL mode can only be run on the actual model or a test harness of the highest level in the model, and not for a subsystem. When a test case is run in SIL mode, a code generation report will be shown.

5 Discussion

The purpose of this thesis was to discover the functionality of Simulink Test and determine if it can replace the unit testing part of WSDE and if it could bring value to our software development process. Some of the investigated factors were ease of use, test creation and speed of testing. I was also tasked with the implementation of some unit tests in Simulink Test for the application Operation Mode Control.

The user interface of Simulink Test is easy to use. It looks modern and polished, and things are easy to find. The difficulty of making test cases is about the same in Simulink Test as in WSDE. As described in Chapter 3.2, 4.1 and 4.2, the inputs and expected outputs in Simulink Test are defined in external MAT- or Excel files. It can be argued that the way the test data are written in external Excel files gives a better immediate overview of the signals and how they relate to each other, as they can be seen side by side. In WSDE every signal must be clicked and opened individually to edit or view the values.

Speed is an important factor for the developers at ECSD as the unit tests for some of the larger applications can take hours to run. As stated in Chapter 4, the speed of running the unit tests in Simulink Test was around 13% faster than in WSDE. Every unit test for the Operation Mode Control application must simulate the model for at least 10 seconds. Because of the small difference in speed between Simulink Test and WSDE, it can be concluded that the time it takes to run the tests are determined by how the unit tests are written and how long they take to simulate. To reduce the time it takes to run the tests, they could be written in a way that shortens the time the model has to be simulated. Also, if the

tests would be scheduled or run on a server which is possible for Simulink Test, as mentioned in Chapter 3.8, it could further speed up the testing process.

As further development, if Simulink Test is taken into use, the rest of the UNIC applications' WSDE unit tests would need to be implemented in Simulink Test. This would of course take some work, as there are many applications and the number of unit tests is large. However, because WSDE unit tests are stored in a MAT-file, and the fact that Simulink Test has a programmatic interface, the move of the existing tests could be automated by writing a script. The script could read the WSDE MAT-file data, create input and baseline files and then create the test cases for Simulink Test. The problems mentioned in Chapter 4.4 with Simulink Test's incompatibility with some of the WSDE model settings would also have to be solved and fixed, but the current plan is to completely replace WSDE with other software eventually.

During the work on this thesis I have learned a lot about software testing, and I have considerably improved my skills in Simulink and MATLAB. Software testing is very important and a big part of software development, and Simulink and MATLAB are widely used by engineering companies, so the knowledge and skills I have acquired from this project are truly valuable and will definitely be useful for me in the future.

In conclusion, based on the results in Chapter 3 and 4, I think that Simulink Test could well be used to replace the unit testing part of WSDE. It possesses all the functionality of WSDE and has a lot of other useful functionality that could be used to improve and bring value to the software development process.

The goals of this thesis have been fulfilled, as the functionality and usability of Simulink Test have been established and all the initial questions in Appendix 1 have been investigated and answered. The unit tests for the application Operation Mode Control have also been implemented successfully in Simulink Test.

6 References

- Burnstein, I., 2003. *Practical Software Testing: A Process-Oriented Approach*. New York: Springer-Verlag.
- De Vleeschauwer, R., 2017. *Agile and DevOps (and BizDevOps)*. [Online]
Available at: <https://www.bluebridgesoftware.com/blog/26-devops-the-gartner-toolchain.html>
[Accessed 8 May 2019].
- Homés, B., 2012. *Fundamentals of Software Testing*. London; Hoboken: ISTE Ltd; John Wiley & Sons, Inc..
- Karjalainen, J., 2016. *WSDE Guide*. Vaasa: Wärtsilä.
- Mala, D. J., 2019. *Integrating the Internet of Things Into Software Engineering Practices*. Hershey PA: IGI Global.
- MathWorks, 2018. *Company Overview*. [Online]
Available at: <https://www.mathworks.com/content/dam/mathworks/tag-team/Objects/c/company-fact-sheet-8282v18.pdf>
[Accessed 3 May 2019].
- MathWorks, n.d. *Examine Test Failures and Modify Baselines*. [Online]
Available at: <https://se.mathworks.com/help/sltest/ug/examine-test-failures-and-modify-baselines.html>
[Accessed 14 May 2019].
- MathWorks, n.d. *Simulink*. [Online]
Available at: <https://se.mathworks.com/help/simulink/index.html>
[Accessed 5 May 2019].
- MathWorks, n.d. *Simulink Test*. [Online]
Available at: <https://se.mathworks.com/help/sltest/>
[Accessed 5 May 2019].
- MathWorks, n.d. *Stateflow*. [Online]
Available at: <https://se.mathworks.com/help/stateflow/index.html>
[Accessed 5 May 2019].
- MathWorks, n.d. *What is MATLAB?*. [Online]
Available at: <https://se.mathworks.com/discovery/what-is-matlab.html>
[Accessed 5 May 2019].
- Myers, G. J., Sandler, C. & Badgett, T., 2012. *The art of software testing*. 3rd ed. Hoboken: John Wiley & Sons, Inc..
- Patton, R., 2001. *Software testing*. 1st ed. Indianapolis: Sams Publishing.
- Rasmusson, J., 2010. *The Agile Samurai*. Raleigh: The Pragmatic Bookshelf.
- Siemens, n.d. *About us*. [Online]
Available at: <https://polarion.plm.automation.siemens.com/company/index>
[Accessed 13 May 2019].

Software Testing Fundamentals, n.d. *Verification vs Validation*. [Online]
Available at: <http://softwaretestingfundamentals.com/verification-vs-validation/>
[Accessed 5 May 2019].

Wikipedia, 2019. *MATLAB*. [Online]
Available at: <https://en.wikipedia.org/wiki/MATLAB>
[Accessed 5 May 2019].

Wärtsilä, n.d. *About Wärtsilä*. [Online]
Available at: <https://www.wartsila.com/about>
[Accessed 3 March 2019].

Wärtsilä, n.d. *The History of Wärtsilä*. [Online]
Available at: <https://www.wartsila.com/about/history>
[Accessed 5 May 2019].

- How user-friendly is the user interface?
- How easy is it to make unit tests?
 - How do you define inputs and expected outputs?
- Can you see the difference between expected and simulated values in a plot?
- Is it possible to get the unit test coverage of the model?
 - If you make a new test and run it, will the total coverage be updated, or do you have to run all the tests again?
- Can you visualize the execution path in the model for a unit test?
- If you modify the model, how will it affect the tests?
- Is it possible to prioritize tests so only the most important ones are included in the report?
- Will it be faster to run the unit tests compared to in WSDE?
- Is it possible to debug why a test fails?
- How easy is it to test the generated code?
- How will a new version of Simulink/MATLAB affect the tests?
- Does Simulink Test have Polarion integration?
 - Is it possible to somehow link the requirements to the tests?
- Is it possible to only test a small part of the model without simulating the whole model?
- Does Simulink Test have any functionality that supports DevOps practices, like:
 - Is it possible to make tests from just a requirement before you make the model?
 - Is it possible to run the tests automatically?