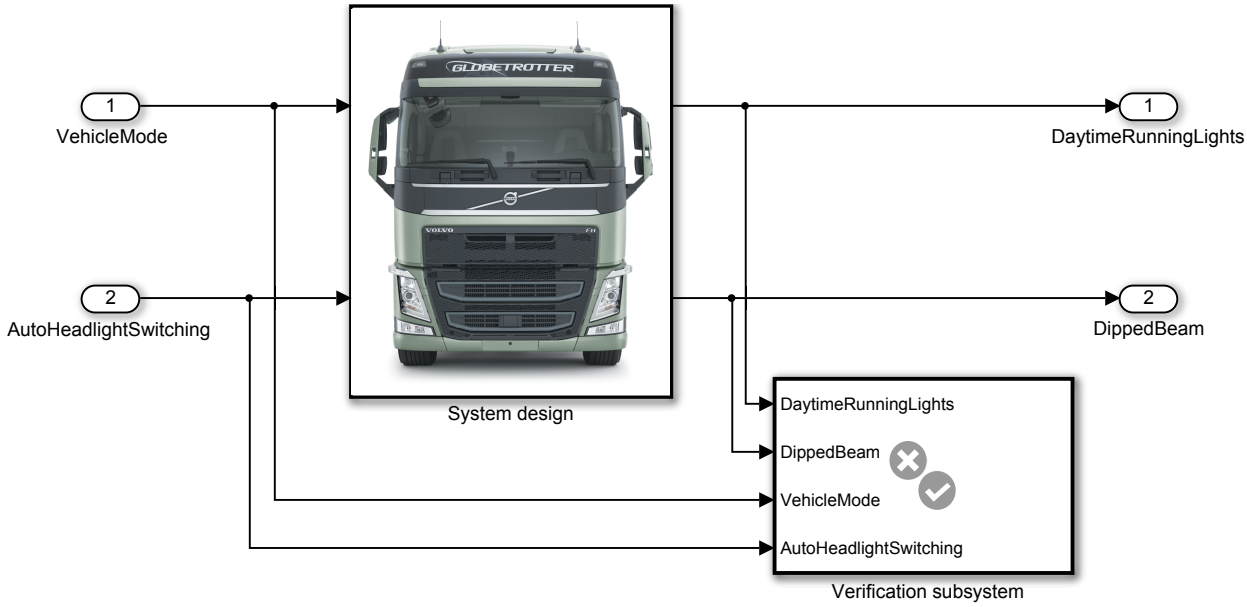
CHALMERS



Model-Based Testing with Simulink Design Verifier

A case study on property proving and automatic test case generation for automotive embedded systems

Master's thesis

Marcus Liliegård
Viktor Nilsson

Department of Signals and Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2014

REPORT NO. EX02b/2014

Model-Based Testing with Simulink Design Verifier

A case study on property proving and automatic test case generation for automotive embedded systems

MARCUS LILIEGÅRD
VIKTOR NILSSON

Department of Signals and Systems
Chalmers University of Technology
Göteborg, Sweden 2014

Model-Based Testing with Simulink Design Verifier
A case study on property proving and automatic test case generation for automotive embedded systems
MARCUS LILIEGÅRD
VIKTOR NILSSON

Technical report no EX02b/2014
Department of Signals and Systems
Chalmers University of Technology
SE-412 96 Göteborg
Sweden

Cover:
A Simulink® model with a system design model and a verification subsystem.

Chalmers University of Technology
Göteborg, Sweden 2014

"Testing is the process of systematically experimenting with an object in order to detect failures, measure its quality, or create confidence in its correctness."
- *Stephan Weileder and Holger Schlingloff*

Abstract

The amount of features in vehicles is rapidly increasing. Today the self-driving car is no longer a futuristic vision, but instead a highly tangible reality. To achieve a high product quality without errors, for a complex system, efficient testing methods are needed. The purpose of this thesis is to investigate how the Model-Based Testing (MBT) tool Simulink® Design Verifier™ can be applied to solve problems regarding error detection, automation and possibilities for earlier testing. Three methods for modeling requirements and applying MBT are considered: property proving with proof objectives; automatic test case generation with cause-effect graphs and test objectives.

The research questions of the thesis covers: (i) the types of software requirements for an embedded system, (ii) how to model these according to the three MBT methods, (iii) which method that is most appropriate to use for each type.

To answer these questions a case study was performed. The case covered the exterior lights of a truck, where focus was put on Automatic Headlight Switching. The case was selected mainly due to its appropriate complexity level. However, no automatic control was part of the system, which limits the applicability of the results.

Based on the case study answers to the research questions are provided. First requirements are grouped into three main types and a classification of the functional requirements, based on temporal properties and modeling similarities, is introduced. Secondly, models of each requirement type, based on the three methods , are described in detail. Lastly, an appropriate method for each requirement type is presented, based on an analysis regarding model complexity, applicability for different test systems and error detectability.

The analysis regarding appropriate methods puts emphasis on the functional requirements, since these were the only main type where MBT could be applied. For the case study it was found that requirements describing undesired behavior are suitable to verify with property proving. Test objectives were only useful for one small type of functional requirements that describes when a function shall be possible to activate. For the other functional requirements, test case generation with cause-effect graphs is recommended, due to the possible application on different test systems.

MBT can help find requirement errors since requirements are modeled in a formal manner. Testing can also start early since the models can be created during the requirement writing. When selecting an appropriate MBT method it is useful to analyze the requirements and behavior of the System Under Test.

KEYWORDS: Model-Based Testing, Simulink, Simulink Design Verifier, property proving, automatic test case generation, cause-effect graphs.

Preface

This thesis was conducted as the completion of the Masters Program *Systems Control and Mechatronics* at Chalmers University of Technology. The thesis work was intended for and supervised by the Test and Development department at Infotiv AB. The actual case study was carried out at Volvo Group Trucks Technology.

Author contributions

Marcus Liliegård has contributed to the following chapters: Model Based Testing, Simulink Design Verifier, Modeling requirements for use with Model-Based Testing and Conclusions.

Viktor Nilsson has contributed to the following chapters: Introduction, Testing at Volvo GTT, The case study, Requirement types of an automotive embedded system and Evaluation of Model-Based Testing methods.

Acknowledgments

We would like to thank our examiner Knut Åkesson for his support in developing our idea about the thesis and also for his feedback and opinions on the content of this report.

We would like to acknowledge Dan Olsson, our supervisor at Infotiv, for his support and guidance. We also thank Peter Thorngren, our supervisor at Volvo, for his contribution in enabling this thesis project and his enthusiasm during our work.

We also appreciate the help from others at Infotiv, especially Tomas Sjögren, Peter Haglund, Tariq Khalaili and Kristoffer Abrahamsson. The support from various people of the Integration and Verification section and especially Pierre Lange, was also highly appreciated.

Finally we thank the people at The MathWorks and especially Fredrik Olsson for enabling this thesis by providing the necessary software.

Göteborg, July 1, 2014

Marcus Liliegård, Viktor Nilsson

Contents

List of Abbreviations

AHS Automatic Headlight Switching

DRL Daytime Running Lights

E2E End-to-End

E2EF End-to-End Function

ECU Electronic Control unit

FRS Functional Requirement Specification

HiL Hardware-In-the-Loop

HMI Human-Machine Interface

I/O Input/Output

LCP Light Control Panel

LDC Logical Design Component

MBD Model-Based Design

MBT Model-Based Testing

MCDC Modified Condition/Decision Coverage

MiL Model-In-the-Loop

SiL Software-In-the-Loop

SLDV Simulink® Design Verifier™

SUT System Under Test

VMCU Vehicle Master Control Unit

Volvo GTT Volvo Group Trucks Technology

1 INTRODUCTION

Vehicles are becoming increasingly automated and complex to meet the requirements of everything from customer needs to environmental laws. Due to this the automotive industry is integrating more and more software into its products (Hanselmann 2008). The cost for developing an electric and electronic system can become one-third of the total development costs (Weber and Weisbrod 2003). It is desired that the final product contains little or no software errors, especially with advanced safety critical functionality, such as self-driving cars (Volvo Car Group 2013), (Gibbs 2014).

1.1 Background

With the increasing software complexity the number of requirement specifications increases. A problem will arise when specifications are not clear enough and the developers, who might be separated in many ways from the requirement writer, have to make guesses about how the system shall function. These guesses can lead to unintended behavior and errors, which can result in problems later in the development process. These issues especially occur when the software is being integrated in the complete system and tested with parts from other developers.

Finding and fixing a requirement error at the design phase can cost three to eight times more than correcting it at the requirement writing phase (Stecklein *et al.* 2004). Instead finding the error as late as the system test and integration phase can cost around 20 to 70 times more than it would cost to fix it at the writing phase (Stecklein *et al.* 2004). Thus to be able to develop complex products and maintain the control over costs, it is necessary to find errors as early as possible in the development process (Zander *et al.* 2011, p. 551). When requirements are correct it is also desired to verify these at an early stage to be able to correct design errors. To reduce costs and development time further it is desirable to perform the verification in an efficient manner. The methods and processes used must also be able to handle product changes during the development.

To help solving these problems of the development process the concepts of Model-Based Testing (MBT) (Apfelbaum and Doyle 1997), (Zander *et al.* 2011, p. 549), (The MathWorks Inc 2014) and Model-Based Design (MBD) (Friedman 2006) can be used. MBD involves creating a system design model and using this for simulations, early testing and code generation. A feature of MBT is automatic generation of test cases based on a model of the system behavior. This can help finding contradictory specifications to a higher extent than manual tests (Pretschner *et al.* 2005). Automatic test case generation is also cost efficient, compared to manual test writing (Utting *et al.* 2007, ch. 2).

Another MBT feature is property proving, where a model of requirements is used together with a system design model to prove that the requirements are always fulfilled (The MathWorks Inc 2014).

By combining MBD and MBT the verification of an embedded system can start early during the development, since no hardware is required. The combination of MBD with extensive early testing increases the number of errors found compared to only using models for design purposes (Broy *et al.* 2012). To summarize, MBT is a topic that shows possibilities for improving the development process and therefore make it possible to maintain control over costs and time consumption when creating an automotive embedded system.

1.2 Research questions

The purpose of this thesis is to investigate how MBT can be used to improve the development process for an automotive embedded system, and that this can benefit from making connections to the types of system behavior and requirements.

The objectives of the thesis are to evaluate possibilities for improvement concerning:

1. Finding errors such as:
   (a) Requirement errors such as unclear, incomplete or contradictory requirements.
   (b) Implementation errors resulting in violation of requirements.
2. Automation of the processes regarding:
   (a) Creation of test cases that covers a group of requirements, a single requirement or specified part of a requirement.
   (b) Creation of executable test scripts.
3. Initiating the testing earlier in the development process, regarding:
   (a) Creation of both abstract and detailed low level tests.
   (b) Testing on multiple test systems used in different phases of the development process.
   (c) Reusing information from a previous development process

The purpose and objectives form the basis for the following research questions which this thesis will answer:

- What types of software requirements exist for an automotive embedded system and which are most common?
- How can these requirements be modeled for use with different MBT methods in the MBT tool Simulink® Design Verifier™ (SLDV)?
- Which MBT method is the most appropriate for each requirement type?
  - At which system levels and test systems can the different methods be applied?
  - How well does the requirement models scale with respect to increased number of input signals and requirement complexity?
  - What types of errors can be detected with the different methods?

The MBT methods considered are: property proving with proof objective models; automatic test case generation with test objective models and cause-effect graphs. These will be described in detail in Chapter 4.

To answer these questions and gather the necessary data a case study together with an initial literature review was chosen as method. This enabled acquiring data from a real project and therefore take different anomalies, such as unclear requirements, into account. The case study, presented in Chapter 5, was performed at the Integration and Verification section of the Electronics and Electrics department at Volvo Group Trucks Technology (Volvo GTT). The specific case was chosen due to an already existing system model that enabled focus on MBT. It also consisted of several subsystems which allowed a more industrial workflow consisting of both component and system verification.

The approach in this thesis is as follows:

1. Analyze the requirements of an automotive embedded system and group these into types that are relevant for modeling with the considered MBT methods.
2. Model requirements of each identified type, for each of the MBT methods and identify factors in the requirement formulation that simplifies the modeling process.
3. Apply the corresponding SLDV feature to the modeled requirements and explore the tool.
4. Analyze the modeling and application by addressing factors relevant to problems that exist in an industrial context.

1.3 Scope and Limitations

The scope of the thesis is the MBT tool SLDV and its features property proving and automatic test case generation. The selection of this one tool was made for practical reasons. A relation with the supplier of SLDV (The MathWorks Inc) was already established thus making it possible to use the software necessary for the investigation of MBT. The existing system model was built in Simulink® and by choosing SLDV as the MBT tool no translation of the system model was needed.

Including software configuration parameters when applying property proving and generating test cases was also considered within the scope, due to its relevancy for the studied case. A limitation has still been made to only consider one vehicle variant configuration during the case study. The reason for this was to cope with the limited time and be able to produce reliable results about modeling different types of requirements. However, since there exists support for handling variants in Simulink® future research could possibly cover this gap.

There also exists limitations about in which types of test systems the generated test cases can be applied. Since only Model-In-the-Loop (MiL) tests are performed no data about executing the tests in other environments, such as Software-In-the-Loop (SiL) or Hardware-In-the-Loop (HiL), could be obtained. This area could be covered by constructing an algorithm to create executable test scripts from the generated test cases or by running a model containing the test cases inside a testing environment.

Simulink® blocks incompatible with SLDV is not considered within the scope. This implies limitations about continuous time behavior, since this is an unsupported feature. By not considering more modeling methods there exists further limitations. This only affects the results regarding the evaluation of MBT methods, the types of requirements and how to model those are still valuable results.

1.4 Outline

The proper background for the case is presented in Chapter 2. Here the current testing process at Volvo GTT is described, together with the different existing test systems. Current problem areas are also indicated. After this, the methodology for Model-Based Testing is described in Chapter 3. Connections are made to MBD which is a necessary part for two of the MBT methods.

When the current testing process and the basics of MBT are explained, possibilities for conquering the problem areas by using MBT are presented in Section 3.2. The following section introduces other available MBT tools and briefly compares those to SLDV. Related work is then presented to position the thesis and explain the contributions it will make. Lastly, theory about requirements is introduced to form a foundation for answering the research question about requirement types.

Chapter 4 describes the tool Simulink® Design Verifier™ and its available features to understand how it is used. In the last section general modeling techniques for the methods are explained and illustrated.

After the theoretical foundation is established Chapter 5 introduces the case and explains how it was selected. To later understand the results about requirement types the main functionality is described. The system specifications at different levels are also introduced. The components that implement the functionality are explained and connected to the system specification. The limitations of the case are also presented in this chapter, to give an indication of where the results are applicable or might be inaccurate.

For each of the research questions a result is presented and discussed. Starting with Chapter 6: Requirement types of an automotive embedded system, where the identified types are presented together with how they are distributed. The effect of requirement formulation is discussed together with how well the analyzed system represents the average embedded system.

Chapter 7: Modeling requirements for use with Model-Based Testing, explains how each requirement type is modeled as proof objective, test objective and cause-effect graph respectively. Lastly Chapter 8 evaluates the three MBT methods and an appropriate method for each requirement type are presented and motivated.

Finally conclusions are made about the usefulness of the selected approach and MBT in general. Connections are also made to the conclusions of related research. Raised questions which could be the base for future work are also presented.

2 TESTING AT VOLVO GTT

At the Integration and Verification section of the Electronics and Electrics department complete system test are designed and executed. Here the software of the vehicle is the focus for the tests. Testing is also carried out at other groups, e.g. the Powertrain group, and at other levels, e.g. testing of individual software components or Electronic Control units (ECUs). Due to the case study being carried out at the Integration and Verification group this thesis and chapter will consider the testing performed there. This chapter only provides the details necessary for the thesis, in reality the process is far more complex.

2.1 Testing process

The tests are performed at two phases, integration and verification. The main goal for the integration phase is to make sure that the test system is operable for the verification phase, which tests the different system level functions. Both phases includes planning, development and execution of tests. An overview of the process is shown in Figure 2.1.

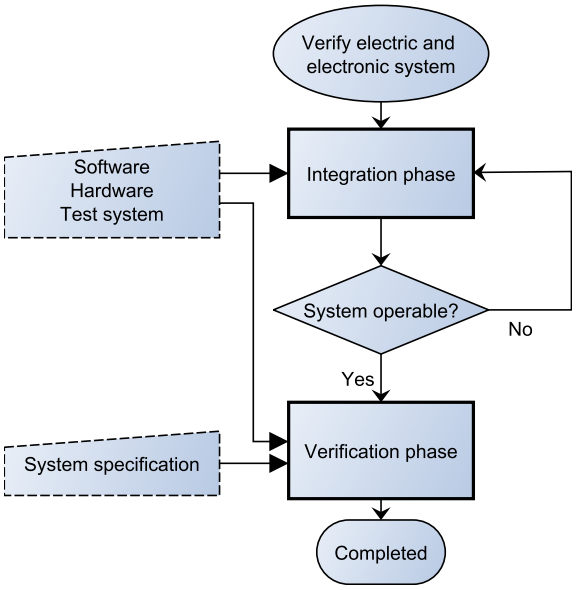


Figure 2.1. Overview of the system level testing at Volvo GTT, showing the two phases, integration and verification, together with the information inputs.

The integration phase includes checking that it is possible to download software to the ECUs and that the communication is working properly. The main vehicle functionality is also tested, for example engine start and braking.

Based on the changes in the tested software, which can be new functionality, bug fixes or changed hardware, strategic test cases are selected to address these matters specifically. The tests are also adapted based on the test system and also the vehicle variant, which can be for example different equipment levels.

When the test system is considered operable the verification phase starts. The goal of this phase is to verify end user functionality, which consists of the whole chain from user input to system output. For example, to verify part of the main beam headlights a test is executed where the vehicle is put in a running state, lights are activated and the main beam stalk is pulled to check that the lights are activated.

To create a functional test case information from use cases and scenarios, which have a high abstraction level, are used. To determine which actual signals that shall be part of the test a sequence diagram is used. The sequence diagram connects scenarios and use cases to signals that are present in the system. All this information is stored in a database and accessed via a SystemWeaver platform called SETool[1].

If extra details are needed for the test case, lower level specifications, describing the underlying software components, are used as the source of information. The test cases are also adapted based on the test system which it will execute on.

All gathered information is used to manually create a test script using the software PNTool[2]. The script describes which input signals that shall be set and which output signals that shall be read. The outputs are also compared to the expected values. Timing can be checked at this stage, by including a waiting time for the comparison. A test script for verifying a part of the main beam headlight's behavior can be seen in Figure 2.2.

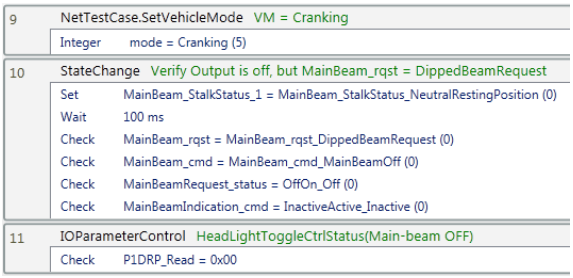


Figure 2.2. Example of a test script in PNTool. The script first sets the vehicle in a state where the engine is about to start. Then the signal for the main beam stalk is set to *neutral*. A check is performed to verify that the signal for the main beam is set to *off*. Finally the output from the corresponding ECU is checked.

The tests in both the integration and verification phases are executed on several test systems that are described in the following section.

[1] `http://systemite.se/content/products-services/systemweaver-platform`
[2] PNTool is developed by Volvo

2.2 Test systems

There are three types of test systems for the system level tests, Hardware-In-the-Loop (HiL) rigs, Box trucks and Complete vehicle rigs. Many of the tests are carried out at HiL rigs, which contain most of the real hardware but simulates the physical vehicle movement and sensor inputs.

The hardware included in a HiL rig are: the complete Human-Machine Interface (HMI) consisting of buttons, stalks, dashboard, indicator lamps and various displays; the lights, both interior and exterior; selected parts of the pneumatic system; all ECUs that are not part of the powertrain.

A CANoe[3] testing environment, running on a PC, is connected to the rig. This environment can observe signals in the system via a CAN-bus interface[4]. The inputs from the hardware can also be sent via this interface instead of applying the inputs manually (with stalks and buttons, etc.). The testing environment allows automatic execution of tests by applying inputs and monitoring outputs according to the written test script. Figure 2.3[5] shows a HiL test rig where system tests are performed.

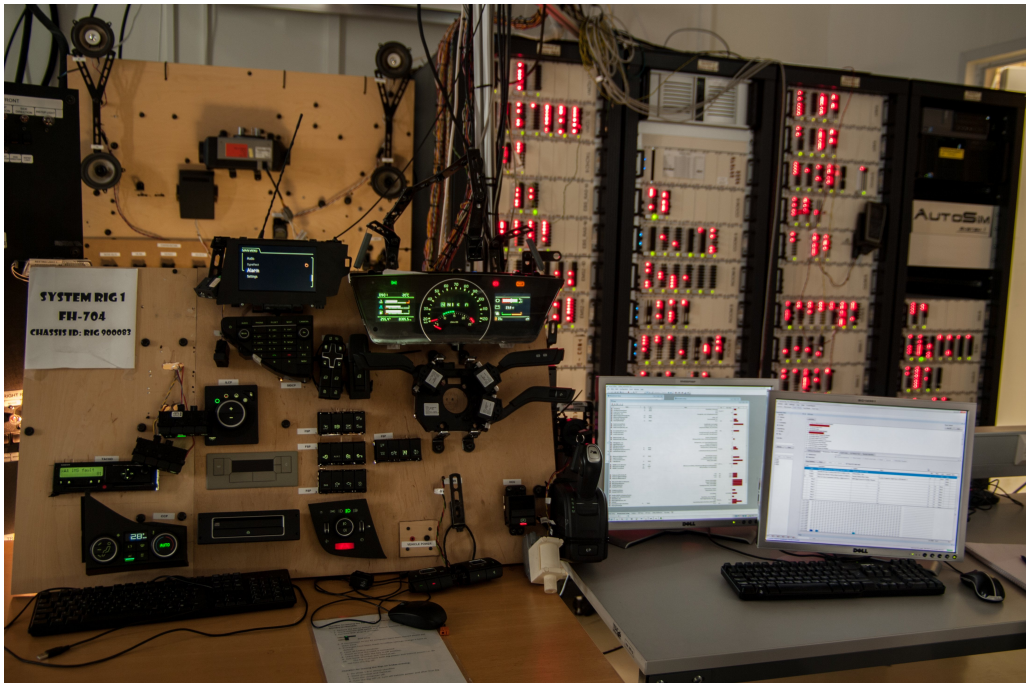


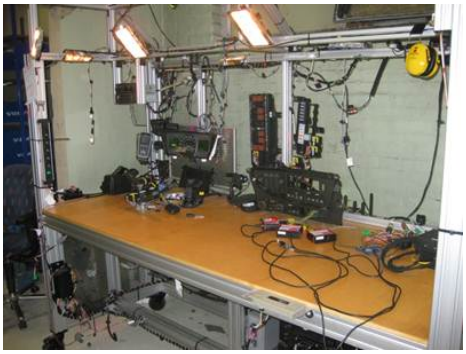
Figure 2.3. Photo of a HiL test system at Volvo GTT. The HMI is shown to the left, monitors displaying the testing environment to the right. The rigs for simulation and signal monitoring are located in the background to the right.

[3] http://vector.com/vi_canoe_en.html
[4] http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can_fd_spec.pdf
[5] Source: Volvo Group Trucks Technology

Simulating parts of the vehicle allows efficient testing of different vehicle variants in terms of models, equipment, number of axles, etc. It is also possible to test failure handling behavior and robustness, since errors such as short circuits and disconnection of components can easily be introduced. Since the truck body is simulated dynamic and safety critical tests can be executed in the HiL test rig as well. For example, running a test that disconnects the ECU responsible for braking while the vehicle is running 90 km/h and making an evasive maneuver, is easily done in the test rig but is unsafe to test with a real vehicle. All the features for simulation and fault injection can be part of a test script to allow for automation.

The goal is to automatically execute most of the HiL tests, but currently semi-automated and manual tests are also used. When running manual tests, test engineers will perform input actions and evaluate all outputs. The semi-automated ones contains automated sequences but can also ask the test engineer to perform an action, e.g. push a button or check that a light is activated.

The second type of test system, Box trucks, also consists of the real vehicle hardware, including the complete wiring harness. But in comparison to a HiL rig, it has no simulation possibilities. This type of test system also lack connection to a testing environment thus automated testing is not possible. Instead box trucks are used for various integration tests. Since the wiring harness is identical to the one in the real vehicle, box trucks are highly suitable for testing voltage and current levels. Software download and diagnostic tests are also performed on this type of test system. Figure 2.4a[6] shows a photo of a box truck used at Volvo GTT.



(a) Box truck.

(b) Complete vehicle.

Figure 2.4. Test systems for system level tests at Volvo GTT

Complete vehicle rigs, as seen Figure 2.4b[7], are used by the Integration and Verification section to prepare the rigs for final tests, called complete vehicle tests. These tests are mainly performed to validate that the right kind of functionality is included in the vehicle and that it behaves as the user would expect. For this test system the integration team initially focuses on the most important parts that make the vehicle drivable and safe to operate.

[6]Source: Volvo Group Trucks Technology
[7]Source: Volvo Trucks Image Gallery

2.3 Problem areas

During the functional verification implementation errors are sometimes found. These are related to the software not following the requirements, or requirement issues, such as ambiguities and contradictions, which has caused an incorrect interpretation. Incomplete requirements, which can be for example missing error handling, are also sometimes found.

There exists no requirements stating the behavior of an ECU in terms of its Input/Output (I/O) signals. Requirements are only specified at lower levels (software components) and higher levels (end user functionality). This creates a problem when designing functional test cases. Since test cases can only include signals that are present at ECU I/O, this lack of specification increases the time consumed by a test engineer to create the test. This also introduces possibilities for errors when trying to understand the combined behavior of different software components, since these include details that might not be considered at the functional test.

There is a need for more time and cost efficient creation of test cases. The current process requires one test script to be created for each input signal permutation, which is highly time consuming. Since the test scripts are created manually it is also time consuming to update them when a requirement is changed.

To decrease the time consumed by the complete development process there is also a need for a more efficient reuse of information from previous development. This includes possibilities for adapting old test cases together with reuse of scenarios and use cases for creating tests for the new system.

2.4 Summary

This chapter has put testing and the problems that can arise during system development, in a specific context to provide more detailed background information for the topic covered by this thesis.

The problem areas at Volvo GTT connects to the objectives presented in Section 1.2 and will be used as a foundation when connecting to related research and introducing how Model-Based Testing (MBT) can be used to improve the development process, presented in Chapter 3.

3 MODEL-BASED TESTING

The demand for automatic verification have made it necessary to create formal methods to find test cases that are relevant to the system specification and certain coverage criteria. One approach on this is the methodology of Model-Based Testing (MBT). How MBT can solve some of the problems described in Section 2.3 is indicated in Section 3.2. A comparison of MBT tools for Simulink® models are then presented. Related work in the area of MBT is presented in Section 3.4. Section 3.5 provide information about system requirements.

3.1 The Fundamentals of Model-Based Testing

Test procedures are being automated to a large extent (Wahler *et al.* 2012) and Model-Based Testing is one methodology describing how to do this. MBT is a concept that is defined and approached in a number of different ways.

A common approach is to create an abstract model of a system. The system can be anything from a model itself to a complete embedded system. This abstract model can be seen as a formal specification of the system (Zander *et al.* 2011, ch. 1). An alternative is to use models of certain behaviors or properties of the system, e.g. a signal shall stay within a certain range of values.

By using various algorithms(Utting *et al.* 2007, p. xiii), it is possible to generate test cases based on a model. Test cases usually consist of input signal vectors and vectors for the expected outcome. A common objective when generating test cases from a model is to achieve a certain model coverage criteria, which is described in more detail in Section 4.1.

Certain MBT tools can also, by using formal methods, perform property proving (The MathWorks Inc 2013, ch. 12). This means that a model of a property is mathematically proved to always be fulfilled with respect to a model of the system. It is for example possible to verify if variables exceed certain values or that under certain conditions some functions are not able to activate.

Property proving requires that there exist a system design model. The system design model may be created by a developer using Model-Based Design. The purpose of the model can be simulation and generating code to be implemented in a real system, e.g. an embedded system. Properties that have been fulfilled on the model will also be fulfilled in the real system unless new errors occurs due to for example latencies or faults in the code generation.

The most important differences between property proving and test cases are where they are applicable and how extensive they are. Property proving is a much more thorough verification method since it actually proves that a property is fulfilled at all times. Property proving is however limited to Model-In-the-Loop (MiL) testing while test cases can be used anywhere from MiL to Hardware-In-the-Loop (HiL) environments by the use of an adaptor model for creating executable test scripts (Utting *et al.* 2007, ch. 2.2).

3.2 Possibilities for improvements

MBT can help finding requirement errors in an early phase, since a formal and precise model has to be created based on the system requirements (Utting *et al.* 2007, ch. 2.7). When creating a model of a requirement it is reviewed from a different perspective, which can help find ambiguities. Model-based tests can also find more requirement errors than manually created tests, when executing the test cases in a later phase (Pretschner *et al.* 2005). This indicates that MBT could be part of the solution for finding ambiguous, contradictory or incomplete requirements.

A model of the system behavior can be created based on a test scope, therefore becoming more abstract than the System Under Test (SUT) (Utting *et al.* 2007, ch. 3.1),(Broy *et al.* 2005, ch. 10). The model could then act as a specification at the Electronic Control unit (ECU) level which could solve the problem concerning specifications at this level. The behavior model can also be created at higher abstraction levels such as combining ECUs to subnets or considering the complete vehicle as the SUT. An adaptor for the generated test can then be used to concretize the test cases for execution on the test system (Utting *et al.* 2007, ch. 2.2),(Broy *et al.* 2005, ch. 15).

Different case studies have shown that MBT can lead to savings by reducing development costs by 20-30% (Zander *et al.* 2011, ch. 1.4). MBT also has possibilities for time saving when the requirements change. Instead of updating all tests manually, the test model, which is much smaller, is updated and the new tests are automatically generated (Zander *et al.* 2011, ch. 6.1). By using a system design model as the SUT and applying MBT the verification can start early in the development, before any hardware exists.

3.3 Tools for Model-Based Testing

The tool used in this study is Simulink® Design Verifier™ (SLDV), which is a toolbox for Simulink® and is presented in detail in Chapter 4. There do however exist alternative software for applying MBT on Simulink® models. Some will be presented in this section. A more in-depth analysis of different tools have been done by Zander-Nowicka (2009, ch. 3).

Reactis from Reactive Systems[1] allows for automatic test case generation and property proving. Test cases can be generated from state machines that are modeled in the tool. For property proving it is possible to model properties using Simulink® or as c-code like syntax of Reactis.

TPT (Time Partition Testing) from PikeTec GmbH[2] have support for test case generation. Test cases are specified as automata. TPT can automatically generate a test harness in Simulink® and simulate the test cases on a system design model.

T-VEC from T-VEC Technologies[3] analyze Simulink® models and identify modeling faults such as dead logic. Test cases that achieve a certain coverage objective can be generated from the system design model.

SLDV is capable of both automatic generation of test cases and proving properties. The advantage of using SLDV is that both property and system modeling can occur in the same environment. Properties and models used for test case generation can be created in a wide verity of ways. It can be done in either Simulink®, Stateflow® or as MATLAB® code.

3.4 Related work

How to work with MBT and the process from requirement all the way to test execution is covered by several books and papers (Utting *et al.* 2007, ch. 2), (Broy *et al.* 2005, ch. 10). These cover a general approach to software testing using MBT but there also exist material with more focus on Simulink® and the automotive industry (Bringmann and Kramer 2008), (Zander *et al.* 2011, ch. 19).

The generation of test cases is also a well researched area. Test cases from Simulink® and Stateflow® models can be generated by translating the model to an extended finite automaton and analyzing its paths (Li and Kumar 2012). Another way to generate test cases which have been proven useful, is the use of a model checking tool (Micskei and Majzik 2006), (Mohalik *et al.* 2014).

The common factor for these studies are that the actual system is not the main focus. This thesis aims to contribute by making further connections to the actual SUT and instead focus on different types of specifications and behavior. By doing this the usefulness of MBT in an industrial context can be further evaluated.

Using a model checker for Stateflow® models, a model can be verified by defining system properties in a model checker (Chen *et al.* 2012). This method can also find design errors and provide counterexamples for debugging. Similar to (Chen *et al.* 2012), the work in this thesis also cover verification of properties (with property proving) but instead has an approach that reduces the number of tools needed.

[1] `http://www.reactive-systems.com/products.msp`
[2] `http://www.piketec.com/products/tpt/matlab-simulink.php`
[3] `http://www.t-vec.com/solutions/simulink.php`

This is achieved by using Simulink® and Stateflow® for both system and property modeling together with SLDV for verification of the properties.

A case study used SLDV on a train tracking function for an automated train protection system (Etienne *et al.* 2010). Different proof strategies were presented such as proof by induction. A property is proven to hold in the first time step and when proving that it holds in the next time step it is assumed that it holds for all the previous time steps. The case study showed that property proving on a variety of behavior is possible, including safety properties. This is closely related to the presented case in terms of procedure and tool selection. The train tracking function have a somewhat different specification which is more mathematically complex compared to the presented case, which has a more event based specification.

To summarize, the processes and techniques behind Model-Based Testing are relatively widely researched. Different tools have been created and studied, but the practical use of Simulink® models together with MBT in Simulink® Design Verifier™ has not gained much attention. The topic of this thesis is chosen for this reason.

3.5 Requirement specifications

Requirements can exist in two domains; the problem and the solution domain (Hull *et al.* 2011, ch. 1.9). The first of these focuses on the stakeholder's (owner's, customer's, etc.) view, therefore the requirements in this domain explains what problems the stakeholders want to solve. The requirements of the solution domain instead puts focus on what the system will do, hence they are often referred to as system requirements (Hull *et al.* 2011, ch. 1.9). All requirements for lower level parts, e.g. subsystems and components, belong to this domain.

It is common to separate the system requirements into functional and non-functional requirements (Glinz 2007). Since the definition of a non-functional requirement is unclear (Chung and Prado Leite 2009),(Glinz 2007), a different separation will be used. This separation, created by Glinz, is based on a more detailed taxonomy that splits up the non-functional requirements into more clearly defined types based on concerns (Glinz 2007). These types are shown in Figure 3.1.

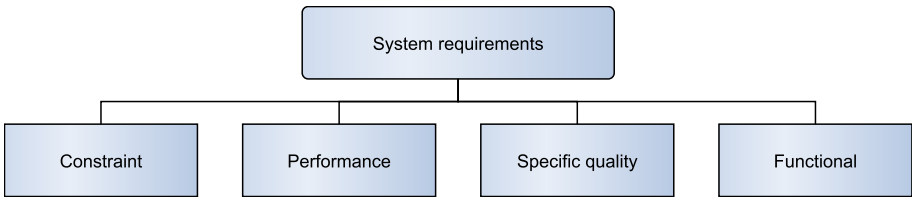


Figure 3.1. Separation of system requirements into different types based on concerns.

A functional requirement concerns the behavior of the system and the expected output for some input. Performance requirements concerns timing, speed, volume and throughput. A specific quality requirement concerns the qualities described in ISO/IEC 9126-1 (ISO/IEC 2001), which describes for example system security and usability. Constraints limits the possible solutions further than what is required to meet the functional, performance and quality requirements. This can be for example physical, legal or interface constraints.

Since the standard ISO/IEC 9126-1 has been superseded by ISO/IEC 25010, Glinz's (2007) taxonomy will be used with the exception of quality requirements instead concerning the qualities described in ISO/IEC 25010 (ISO/IEC 2011).

3.6 Summary

This chapter presented different tools and related work to give a broader picture on the subject of MBT. To form a foundation for answering the research question about requirement types, Section 3.5 explains different aspects of this topic. Section 3.1 explained the fundamentals of MBT which provides necessary background information for Chapter 4.

4 SIMULINK DESIGN VERIFIER

This chapter will cover Simulink® Design Verifier™ (SLDV), which is a toolbox for Simulink® that is meant for finding design errors and formal verification. The first version of SLDV was released in 2007 by The MathWorks. Simulink® Design Verifier™ is capable of both property proving and automatic test case generation (explained in Section 3.1). To perform property proving and test case generation SLDV uses Prover Plug-In[1] from Prover Technology (The MathWorks Inc 2013, ch. 1). Another feature in SLDV is design error detection which can identify design errors such as dead logic and division by zero. This feature will however not be further mentioned since this thesis focus on formal verification against requirements.

Section 4.1 will explain some of the features of SLDV that are necessary as background information for coming sections. Three methods of modeling requirements that utilizes different features in SLDV are introduced in Section 4.2. These are: the proof objective model, test objective model and cause-effect graph.

Not all Simulink® models are compatible with SLDV. One of the limitations is that a fixed step solver must be used. Certain blocks are not supported, e.g. the blocks in the continuous library. Some blocks that are not supported can however be replaced using automatic block replacement (The MathWorks Inc 2013, ch. 3). A consequence of block not being supported is that a lot of work might be necessary to redesign the model to get it compatible, e.g. in control systems the controller have to be replaced with a discrete controller.

4.1 Features and Fundamentals

SLDV comes with special blocks such as the proof objective block and the test objective block to instruct that a property shall be proved or a test case shall be generated. It is also possible to generate test cases by specifying a model coverage objective that shall be obtained by the test cases. Coverage criteria describes how well a test suite covers different parts of a model. In SLDV the following control-flow based coverage criteria (The MathWworks Inc 2013, ch. 15), (Zander *et al.* 2011, p. 87) (Utting *et al.* 2007, ch. 4.1.1) is used:

Decision coverage - The percentage of the possible decision paths through a model. For a test suite of a state chart to achieve full decision coverage all entry and exit paths through a state must be evaluated by test cases. In Simulink® decision paths can be for example Switch-blocks. The model in Figure 4.1 has two decision paths. To achieve full decision coverage both these paths need to be taken by test cases.

[1] `http://www.prover.com/products/prover_plugin/`

Condition coverage - The percentage of the total number of logical inputs and transitions that are covered. All guards in a state chart must be evaluated to both true and false by different test cases to achieve full condition coverage. In Simulink® the parts affected by condition coverage are for example Stateflow® transitions and blocks with logical functions. In the model in Figure 4.1 there are two guards, `In1` and `In2`. To achieve full condition coverage both `In1` and `In2` have to be as true and false.

Modified Condition/Decision Coverage (MCDC) - Extends decision and condition coverage by showing the independence of logical input and transitions. For a test suite to achieve full MCDC every input to a logical function or a transition must be changed independently while the other inputs remain fixed. To achieve full MCDC `In1` and `In2` have to be as evaluated as true and false independently.

There are three alternatives when choosing coverage criteria objective: decision, decision and condition or MCDC. Decision coverage gives the least extensive test cases while MCDC gives the most extensive. Using decision coverage objective on the state chart in Figure 4.1 gives the test case in Table 4.1. If MCDC objective is used then the test case in Table 4.2 is given.

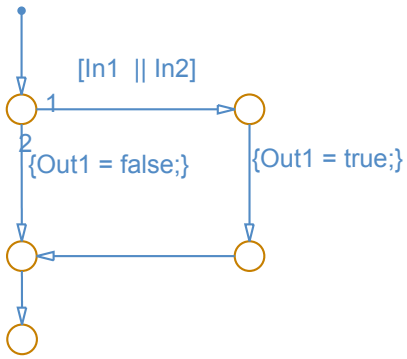


Figure 4.1. An small Stateflow® model used to explain coverage criteria. Decision path 1 is taken when `In1` or `In2` is *true* and path 2 is taken when both `In1` and `In2` are *false*.

Table 4.1. Generated test case from the model in Figure 4.1 using full decision coverage objective.

	Step	**1**	**2**
Generated Input Data	In1	0	0
	In2	1	0
Expected Output	Out1	1	0

Generated test cases can be simulated in a test harness, as seen in Figure 4.2. After test cases have been generated this harness can be automatically generated. Simulating the test cases will provide a report containing how much of the system was covered.

Table 4.2. Generated test case from model in Figure 4.3 using full MCDC objective.

	Step	**1**	**2**	**3**
Generated Input Data	In1	0	1	0
	In2	1	0	0
Expected Output	Out1	1	1	0

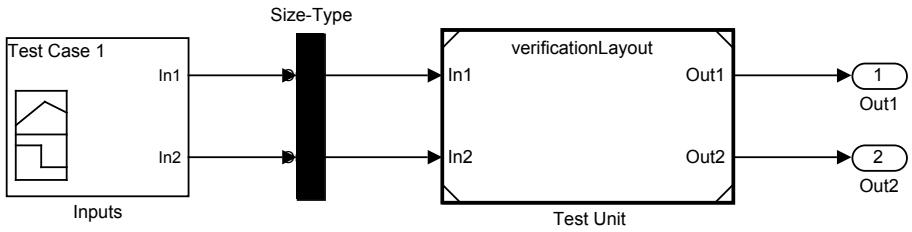


Figure 4.2. Harness model automatically generated by SLDV. The signal builder contains the test cases and the model used to generate the test cases is included by reference as the "Test Unit".

The test objective block (Figure 4.3a) enable test case generation without specifying a coverage objective. This block is used to define values on a signal. The value is indicated above the block. Simulink® Design Verifier™ will attempt to construct as test case that will give the signal the defined value. Only one test case will be provided and it will be as short as possible. The test condition (Figure 4.3b) is used to put constraints on signals during test case generation. Constraints can be placed anywhere in the system on any signal.

(a) Test objective **(b)** Test condition

Figure 4.3. The test objective and test condition block.

An example with the test objective and the test condition is depicted in Figure 4.4. The example consist of two inputs, one output and an `xor` logical operator. One of the inputs have a test condition that constrain the signal to the value *false*. A test objective is used on the output signal of the `xor` block with the specified value *true*. Performing test case generation provides the test cases shown in Table 4.3. The test cases consist of input data and expected output.

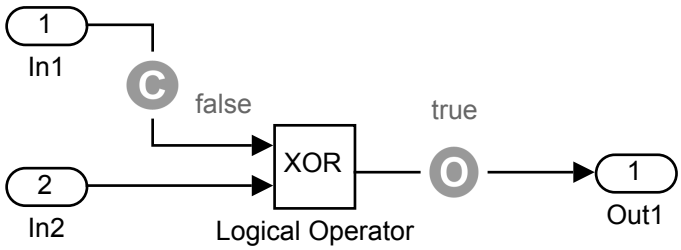


Figure 4.4. A small example with the test objective and the test condition. The objective is to generate a test case where the output from the `xor` block is *true* with one of the inputs constrained to *false*.

Table 4.3. Generated test case from model in Figure 4.4.

	Step	**1**
Generated Input Data	In1	0
	In2	1
Expected Output	Out1	1

The proof objective block (Figure 4.5a) is used to prove that a signal always have the values that are specified. No simulations are made to perform this task, instead a mathematical proof is constructed to verify that the objective is fulfilled at all times. If the objective can not be proven valid a counterexample is generated, containing a test case as short as possible that moves the system to a state where the objective is not fulfilled. The assumption block (Figure 4.5b) is used to constrain signals just as the test condition block. The difference is that the assumption block is ignored when performing test case generation and the test condition block is ignored when performing property proving.

(a) Proof objective **(b)** Assumption

Figure 4.5. The proof objective and assumption block.

In Figure 4.6 a model with a proof objective and assumption is shown. The assumption block constrains one of the inputs to *true*. In this example Simulink® Design Verifier™ will attempt to prove that the output from the `xor` block always is *true*. Performing property proving will prove the objective unsatisfiable and provide a counterexample. The counterexample can be seen in Table 4.4. The counterexample is an input vector where both input signals are *true*. This makes the output of the `xor` block *false* which is a violation of the proof objective.

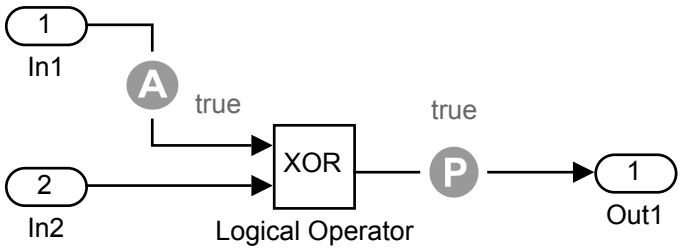


Figure 4.6. A small example with the proof objective and the proof assumption. The objective is to prove that the output from the `xor` block always is *true* when one of the inputs are constrained to be *false*.

Table 4.4. Counterexample from model in Figure 4.6.

	Step	**1**
Counterexample	In1	1
	In2	1
Expected Output	Out1	0

4.2 Modeling requirements

This section will present three methods of modeling requirements in Simulink® for use with formal verification such as property proving and automatic test case generation. The three methods use different features of Simulink® Design Verifier™. The proof objective model is meant for property proving. The test objective model is meant for test case generation using a test objective block and the cause-effect graphs are also meant for test case generation but uses a coverage objective to do so.

The layout used for performing some of the Model-Based Testing (MBT) features often contain a system design model and a verification subsystem, as shown in Figure 4.7. Inside the verification subsystem proof and test objectives are modeled in terms of the Input/Output (I/O) signals of the system model. Cause-effect graphs are separate models where the generated test cases are independent of the system design model.

4.2.1 Proof objective models

The proof objective model is a formal representation of a requirement that is used to perform property proving on a system model. A requirement is modeled inside a verification subsystem (Figure 4.7) using the inputs and outputs of the system. The requirement can be modeled in Simulink® directly or as Stateflow® charts or Matlab-functions. (The MathWorks Inc 2013, ch. 12)

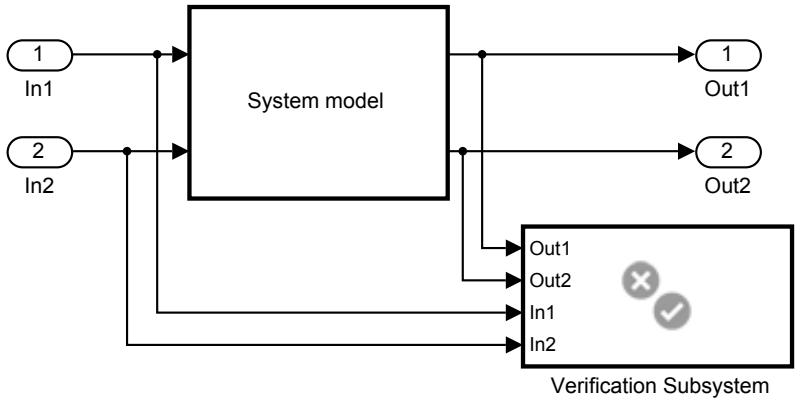


Figure 4.7. Common model layout for Model-Based Testing tasks using Simulink® Design Verifier™.

A generic model of a requirement can be seen Figure 4.8. The condition and the expected result are modeled using the signals of the system and logic operators. With the generic proof objective model the task for Simulink® Design Verifier™ is to prove that

$$Condition \Rightarrow Expected\ result \tag{4.1}$$

always is true.

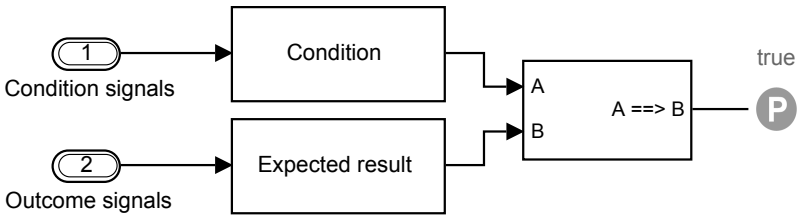


Figure 4.8. Generic proof objective model.

If the condition cannot be fulfilled by the system model then the proof objective will never be violated. To make sure that the condition can be fulfilled a test objective block can be placed after the condition with the specified value *true*. If a test case can be generated then the condition can be fulfilled.

4.2.2 Test objective models

The concept behind this method is to use the system model and the test objective block. The layout when working with test objective models are similar to the one for proof objective models. The signals from the system model are used to create the conditions and expected results. In Figure 4.9 an example of a test objective model is depicted.

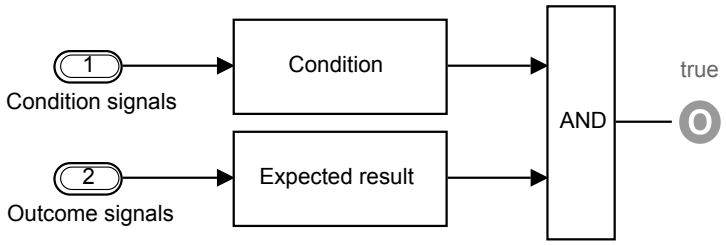


Figure 4.9. Example of a test objective model. The test objective model is placed beside the system model and is connected with its signals.

A test case will be generated that fulfills the condition and expected result. A test case generated from the system design model and a test objective model will contain input signals and expected output for all inputs and outputs of the system. Since other signals are involved in the test case it can not be said for sure that the condition caused the expected result to occur. One way of handling this is to constrain input signals that shall not be involved in the test case.

Unlike the proof objective model that proves that a property cannot be violated, the test objective model will only test if the property can be fulfilled.

4.2.3 Cause-effect graphs

Cause-effect graphs (Figure 4.10) is a common approach in software testing. The concept is to use graphs to connect different causes to different effects. The cause-effect graphs are based directly on a requirement and provides a formal way of verification. It is suitable for requirements that state the relationship between input and output signals. In this thesis the basis for applying cause-effect graphs are the approach presented by Lee and Friedman (2013). With SLDV it is possible to generate test cases based on the cause-effect graphs by selecting a coverage objective, e.g. decision, decision/condition or MCDC (see Section 4.1).

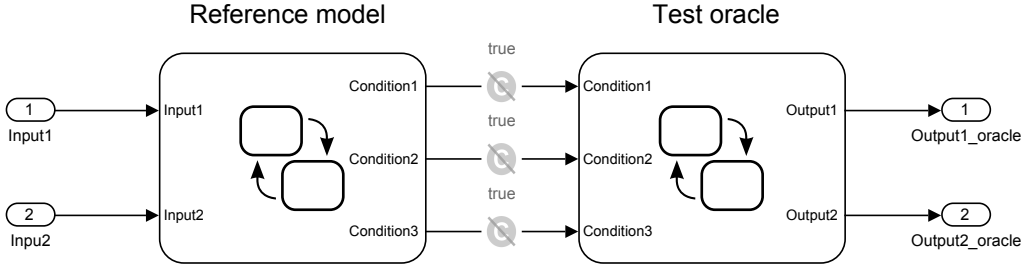


Figure 4.10. A cause-effect graph. The input signals causes conditions to be fulfilled. The oracle determine the expected output based on active conditions (effects).

The model contains two directed tree graphs; the test reference and the test oracle. Both of these are modeled in Stateflow®. The reference model(Figure 4.11a) determine if a condition (effect) is fulfilled based on input signals (causes). The oracle (Figure 4.11b) decides the system output, based on fulfilled conditions. The requirement in Figure 4.11 states that:

- If Input1 and Input2 are *true* then

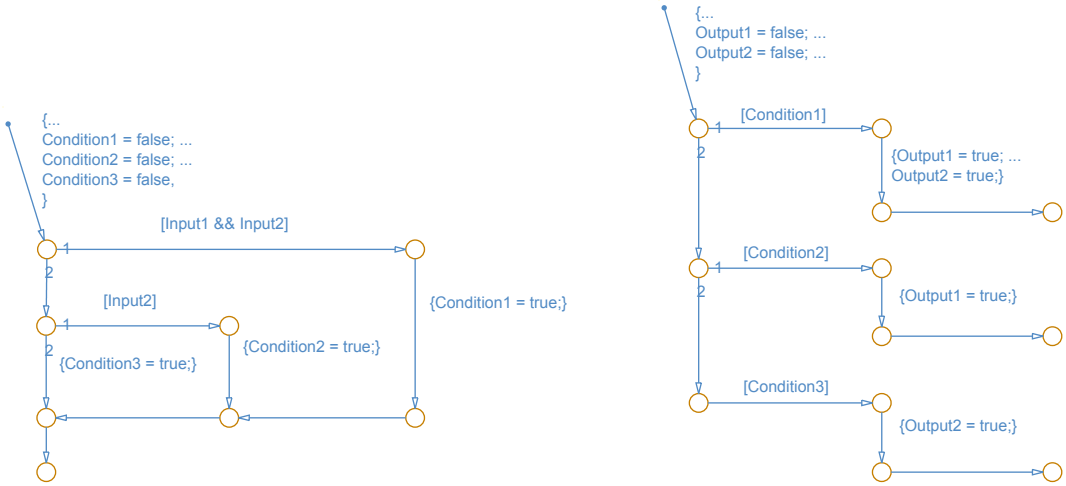
  Output1 and Output2 shall be *true*.

- Else if Input2 is *true* then

  Output1 shall be *true* and Output2 shall be *false*.

- Else

  Output1 shall be *false* and Output2 shall be *true*.



(a) Chart in Reference model **(b)** Chart in Test oracle

Figure 4.11. Details of the different parts in a cause-effect graph.

There are three different outcomes specified in this requirement. This determines how many effects (conditions) there are. The different decision paths in the reference model are directly related to the three statements in the written requirements.

A benefit of separating the requirement between the reference model and test oracle is to give more control over the test case generation. With the separation it is for example possible to only generate test cases for when only one condition is active or when one condition is always false. All of this is done by using test condition blocks on the signals between the reference model and test oracle. To generate test cases where any of the conditions is fulfilled an `or` block and test condition block with the signals can be used.

Unlike the test objective models, only the signals that are specified in the requirement are included in the generated test cases. This assures that it is the desired inputs that causes the expected outcome. With the test objective model only one test case are generated per model. With a cause-effect graphs it is possible to generate either a long test case or several test cases that cover the whole requirement.

To do Model-In-the-Loop (MiL) verification a test harness is created. A harness can be created with a command after test case generation has been performed. The test harness contain the cause-effect graph along with the test cases that are placed inside a signal builder. A copy of the system design model is also included and simulated side by side with the cause effect graph that functions as a test oracle. The setup is depicted in Figure 4.12. If the output from the system model differ from the oracle then the test case is not fulfilled and thus the corresponding requirement is not fulfilled.

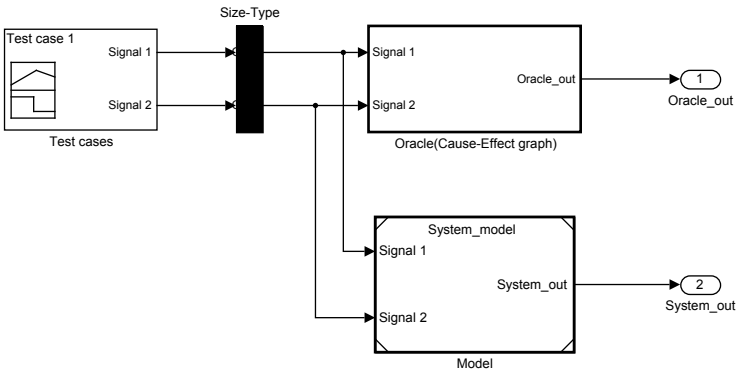


Figure 4.12. Test harness for executing test cases on system model and test oracle.

4.3 Summary

The purpose of this chapter was to present the SLDV toolbox and introduce the three different modeling methods. These methods will be used in Chapter 7 where the research question about how requirements are modeled to be used for MBT is answered.

5 THE CASE STUDY

To answer the research questions a case study that contained both modeling and testing of a vehicle was conducted. The basis for the case was an existing Simulink® model that implemented some of the functionality for the exterior lights system of a truck. Focus was put on Automatic Headlight Switching (AHS), a system that automatically activates the headlights during nighttime.

5.1 Case selection

The case was selected since it allowed an investigation of the problems described in Section 2.3. These concerned incorrect requirements, test case creation, non-existing requirements on the Electronic Control unit (ECU) level and the need for earlier testing.

Another factor in choosing the case was the system's complexity regarding functionality. It was complex enough to contain different types of behavior and a high amount of requirements, but not so complex that low-level details had to be disregarded. The available information and its efficient structure was also crucial when selecting the case. This information consisted of requirements on both low and high level together with test cases on system level. All this allowed to easier understand the system structure and its behavior.

The functionality for AHS was under development at the time when the case study was conducted. Due to this there existed possibilities to find more errors, which might not have been the case for a highly matured functionality. This corresponded well with the research question concerning error detection.

The previously made Simulink® model allowed to focus on modeling the system functionality and requirements instead of its framework and interfaces. Since the model also contained some functionality, the requirement modeling for the different methods and the exploration of Simulink® Design Verifier™ (SLDV) could be initiated early in the study.

5.2 The system for Automatic Headlight Switching

The functionality for AHS is a distributed system where several ECUs and components are included. Multiple light sources also have a connection to AHS. The most important parts are visualized in Figure 5.1[1].

[1]Sources, component images: Volvo Trucks Image Gallery and Volvo Group Truck Technology

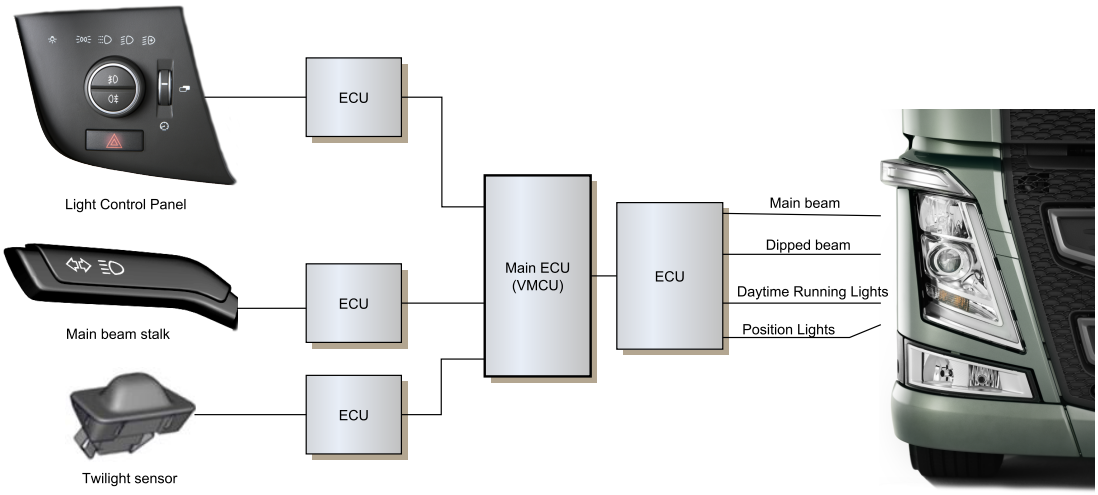


Figure 5.1. Connection between the main parts of the AHS. The ECUs also have connections to other parts contributing to different functionality than the AHS, but these are not shown here.

The twilight sensor measures the ambient light and uses this to request activation of dipped beam or Daytime Running Lights (DRL). The Light Control Panel (LCP) is a freewheel switch used to select light mode, where different modes activates and deactivates certain functionality of the exterior lights. The main beam stalk is used to toggle between main and dipped beam. It can also be used to activate the main beam temporarily, called Flash-to-pass. The ECUs connected to these components contains basic functionality and forwards commands to the Vehicle Master Control Unit (VMCU). The VMCU makes decisions about which light sources to activate based on the twilight sensor, LCP and main beam stalk. The light sources which have a connection to AHS are briefly described in Table 5.1.

Table 5.1. Light sources affected by the AHS

Component	Description
Position lights	Creates basic conspicuousness for the vehicle. Also indicates width and height.
Daytime Running Lights	Increases the conspicuousness for the vehicle during daytime driving.
Dipped beam headlights	Used when meeting oncoming traffic during nighttime driving.
Main beam headlights	Creates better visibility for the driver at nighttime driving.

5.3 Specification of the system

The system is specified using End-to-End (E2E) functions, describing the functionality from the users point of view, and several Logical Design Components (LDCs), which represents different software components. The LDC specifies which Input/Output (I/O) signals that are present, both in terms of signal names and encoded values, and the behavior based on these signals. Combinations of LDCs realizes an E2E function and the combination of End-to-End Functions (E2EFs) and LDCs realizes a Collaboration, which represents the complete functionality.

The AHS function is mainly distributed over four LDCs, one for the Twilight sensor and the LCP respectively as well as two LDCs which are both part of the VMCU:

- ExteriorLightsCtrl - Sends commands to control the different light sources and corresponding dashboard indicators. These commands are based on requests from TwilightSensor and ExteriorLightsHMICtrl and the Main Beam Stalk.

- ExteriorLightsHMICtrl - Uses information from the LCP together with status from ExteriorLightsCtrl to request changes in light mode and activate indicators corresponding to the currently selected mode.

The connections between these LDCs, the relevant E2EFs and Collaboration are visualized in Figure 5.2.

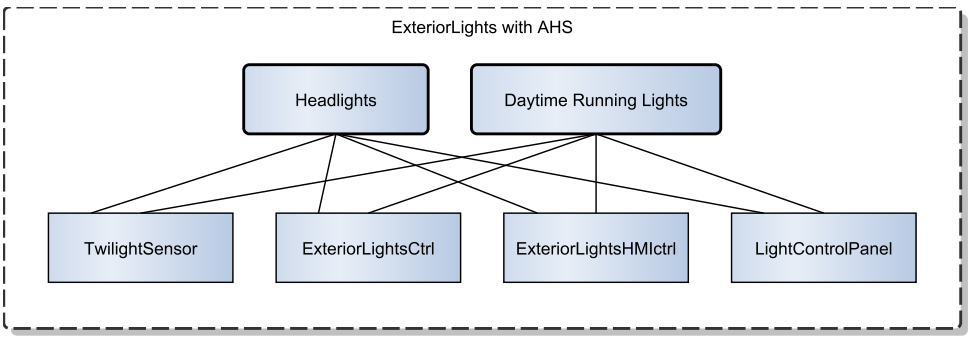


Figure 5.2. Connections between Collaboration (top level), E2E functions (middle level) and LDCs (bottom level)

A key part in specifying the LDC behaviors is the use of vehicle modes to represent the different overall states, that are described in Table 5.2. Another important part of the specification is the light mode (controlled by the LCP) which is described in Table 5.3.

Table 5.2. The different vehicle modes available. These are closely related to the position of the ignition key.

Vehicle mode	Description
Hibernate	The vehicle is in production or being delivered to end customer.
Living	The vehicle is parked and the driver is inside. Only some functionality is available, to reduce electrical energy consumption.
Accessory	Same as for Living but more functionality available.
Prerunning	The driver is about to start the engine. Functionality needed for the powertrain is available.
Cranking	The engine is starting. This mode is only active for a short time when moving between Prerunning and Running.
Running	The engine is running and the vehicle is in motion or performing standstill operations.

Table 5.3. The different light modes that can be active.

Light mode	Description
Park	Activates the parking lights.
OFF/DRL	Deactivates main and dipped beam headlights. Activates AHS.
Drive	Activates the dipped beam and allows activation of main beam.
Drive+	Activates the dipped beam and allows activation of main beam together with extra main beam.

5.4 Existing Simulink model

To perform the Model-Based Testing tasks a previously made Simulink® model was used. This model mapped all LDCs (which are part of the Collaboration) to corresponding subsystems and connected them with the specified signals. The part of the model relevant for AHS is shown in Figure 5.3.

The subsystems included the main behavior for Vehicle modes Running or Prerunning, but these modes were assumed and a way of alternating between modes was not modeled.

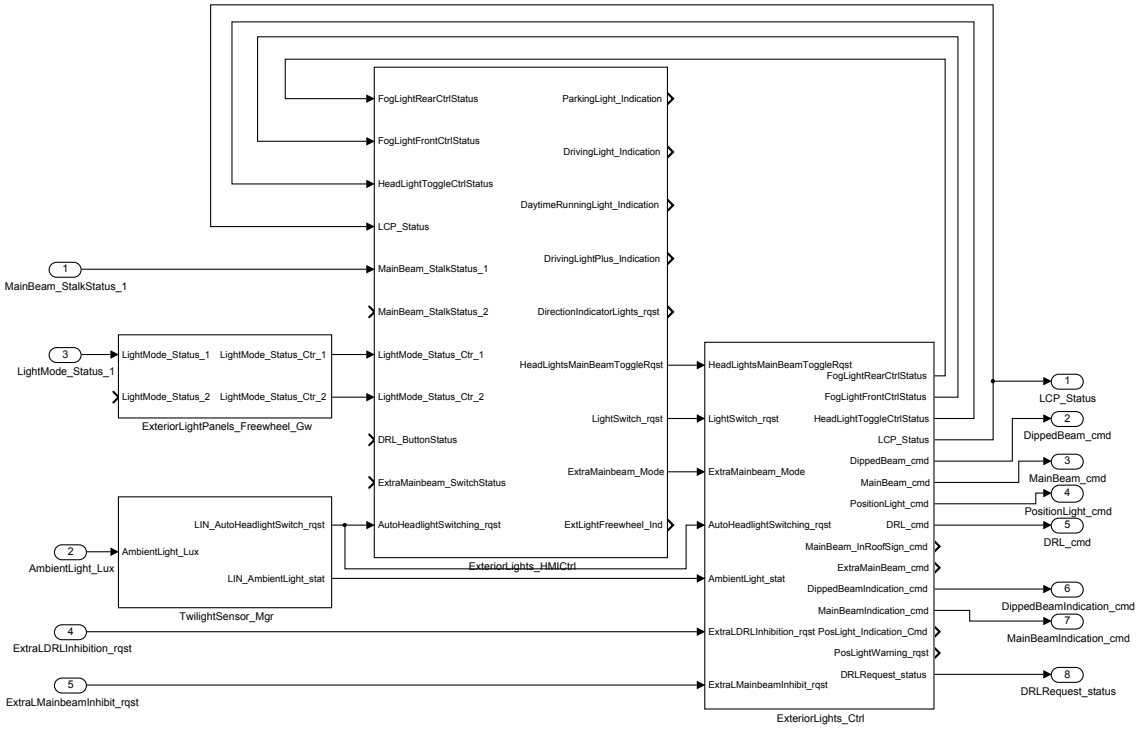


Figure 5.3. Part of the previously made Simulink® model, where the subsystems for AHS are shown.

5.5 Limitations

There exists limitations both regarding system behavior and specification. Since no dynamical behavior, such as filtering or automatic control, is part of the system no results can be obtained regarding Model-Based Testing of this area. An automotive functionality that would cover this could be for example cruise control or filtering of radar measurements for an active safety system.

There is also no data processing, in terms of mathematical calculations, in the system. This further limits the applicability of the results. All behaviors are possible to model with the standard Simulink® blocks supported by SLDV. For this reason to data could be obtained about requirement modeling and applying SLDV features to user defined functions.

Even though the specification covers higher level behavior, the number of requirements on this level limited the amount of data that could be obtained, thus making the results less reliable. There are also no specifications considering decision making at an abstract level which limits the results regarding the different method's applicability for system level testing.

5.6 Summary

This chapter has provided a motivation for selecting the case as well as the case's limitations. This information is necessary when putting the answers to the research questions in a wider context and discussing how well the case represents other embedded systems.

The explanation of the system structure, its components and its specification provides a foundation for understanding the answers to the research questions makes it easier to connect the details of these answers.

Chapter 6, 7 and 8 will now cover the research questions presented in Section 1.2, which regards: the types of requirements for an embedded system, how they are modeled and what Model-Based Testing (MBT) method that is appropriate for each type.

6 REQUIREMENT TYPES OF AN AUTOMOTIVE EMBEDDED SYSTEM

This chapter covers the research question:

- What types of software requirements exists for an automotive embedded system and which are most common?

The requirements for the Collaboration, End-to-End Functions (E2EFs) and Logical Design Components (LDCs) described in Chapter 5 have been manually read and analyzed. They have then been grouped according to the theory described in Section 3.5. For the most common type different subtypes are identified, based on definitions provided in this thesis.

The answer to the question about what types of requirements exist is presented in Section 6.1 and in the following sections the types are described and the most common types are mentioned. The answer is summarized in Section 6.5.

For confidentiality reasons, the complete data for the analysis can not be presented in this thesis. However, the examples for the requirement types are real data, to provide a hint of how the requirements of the system are formulated.

6.1 Identified requirement types

The requirements for each the three system levels are specified at different abstraction levels. LDC requirements are detailed and relates to the system implementation. At this level explicit signal names and encoded values are used. Behavior for different truck variants are also specified. The requirements on End-to-End (E2E) level are written on a higher level in such way that they do not use explicit signal names and do not relate to a specific truck configuration. Requirements on Collaboration level are based on use cases and scenarios, explaining some expected outcome for certain system level conditions.

Of the main requirement types, based on Glinz's (2007) taxonomy explained in Section 3.5, all types exists in the specifications for the Exterior lights system. For the first three main requirement types only one subtype is present for each one. For the functional type the requirements are formulated in a number of different ways, therefore this main type is split into new subtypes which are defined in this thesis. Figure 6.1 shows the identified and created requirement types connected to the main types.

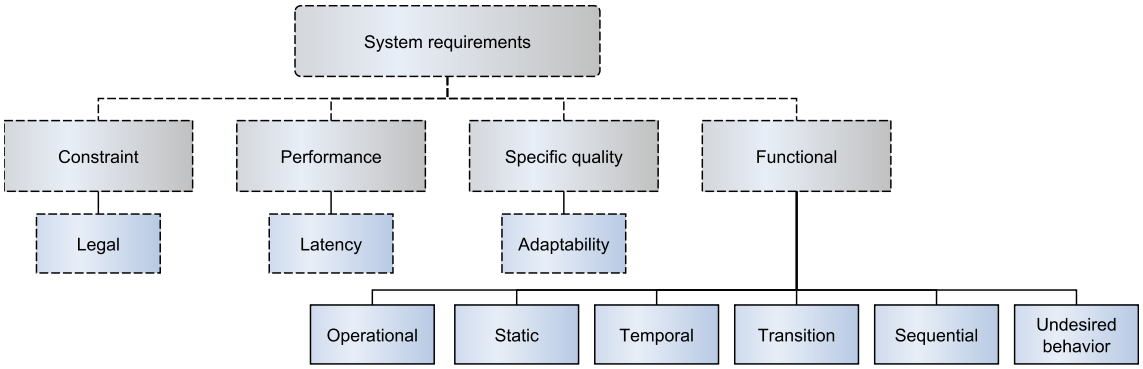


Figure 6.1. Connection between the main requirement types and the identified subtypes. Types defined by others have dashed outline and the types defined in this thesis (which are based on requirements from the case study) have solid outline.

The separation of functional requirements relates closely to modeling similarities in terms of block types. Both the textual requirement statements and the block types has differences in how time takes part which makes this a useful separation when answering questions about modeling in later chapters.

6.2 Description of requirement types

The identified subtypes of the four main requirement types are described in detail below. For each type of functional requirement the created definition of the identified type is also presented.

Constraints
Only one constraint was present in the specification, this was a legal constraint prohibiting certain combinations of light sources being active. Due to the low quantity, this type will not be further analyzed. The outcome of the constraint will still be present through a undesired behavior-requirement (stated as an example below).

Example:
The vehicle shall conform to "United Nations Regulation ECE48-3 - Lighting and light signalling devices".

Performance requirements
This main type only consisted of latency requirements stating a required maximum response time from input condition to output action.

Example:
The delay from driver requesting activation of Daytime Running Lights (DRL) to switching on the DRL lights shall be maximum 250ms.

Specific quality requirements
Of this main type only adaptability requirements existed, based on the definitions of ISO/IEC 25010 (ISO/IEC 2011). These requirements are stating that it shall be possible to control a certain behavior with a configuration parameter.

Example:
It shall be configurable whether the DRL is enabled or disabled.

Operational
Specifies that it shall be possible to activate a signal or function at some conditions, often in which Vehicle mode(s) and without specifying a complete condition based on all input signals that can affect the behavior.

Definition: A requirement stating output signal values for a condition which is not narrow enough to consider the requirement fully tested if the condition is fulfilled.

Example:
Daytime Running Lights shall be operational in VehicleMode Running.

Static
For the collaboration level this type specifies an expected output for some specific conditions and a change in system input. At E2E level this requirement type explains when a functionality shall become activated, but at a higher abstraction level without details about specific signals or values. At LDC level these requirements are often formulated as a nested if-statement, where a certain combination of input signals and vehicle modes shall lead to a certain combination of output signals.

Definition: A requirement stating output signal values for a condition purely based on the current input signal(s).

Example:
If MainbeamCmd or DippedBeamCmd is ON,
HeadLampStat shall be set to ON

Temporal
Specifies a condition for when an output shall become activated, based on both signal values and time. This type mostly consists of failure handling requirements stated as the example.

Definition: A requirement stating output signal values for a condition being fulfilled: for, after or before some time interval or value.

Example:
If Input signal is received with the value Error, then the previous value shall be assumed during the first Delay seconds. If no previous value exists, the value Default shall be used. If the signal still has value Error after this timeout, the value Default shall be used.

Transition
Specifies a behavior when input signals change or changes between specific values.

Definition: A requirement stating output signal values for a condition based on the time instant when a signal changes value.

Example:
When VehicleMode transitions from Running to Cranking the Main beam shall be deactivated.

Sequential
This type is used to specify sequences for a Human-Machine Interface (HMI) consisting of various buttons and stalks. Requirements if this type are formulated as state machines, where different values for an output signal is set for different states. Conditions on signals specifies the transitions.

Definition: A requirement stating output signal values for a condition specifying a certain sequence of input signal values.

Example: Figure 6.2 shows an example of a sequential requirement. This represent toggling between main and dipped beam by pushing the main beam stalk and then releasing it back no its neutral position.

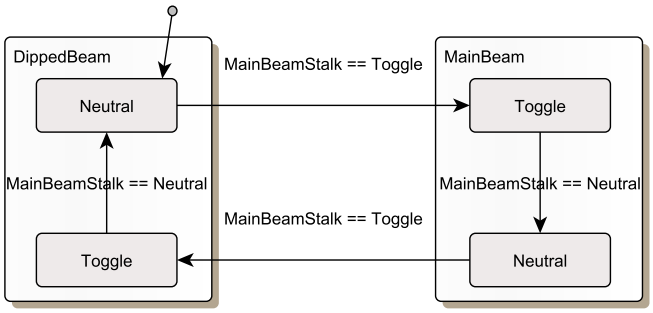


Figure 6.2. Example of a sequential requirement. In addition to the state machine output signals for each state is specified.

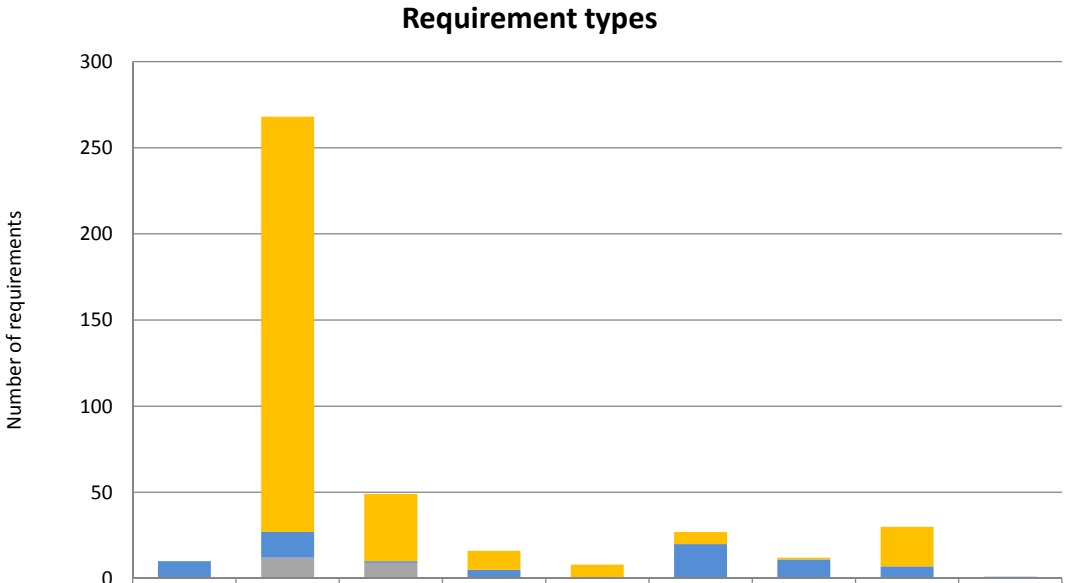
Undesired behavior
This type contains requirements stating that something shall not happen. This can be both a purely undesired behavior, e.g. stating a safety property or a prohibited behavior as a cause from a legal requirement, but also a part of some wanted behavior, e.g that a signal shall not change at certain conditions.

Definition: A requirement stating that output signals that shall not attain some value(s) for a certain condition that is not narrow enough to consider the requirement fully tested if the condition is fulfilled.

Example:
Dipped beam or main beam shall not be activated when the position lights are not activated.

6.3 Common requirements

The manual analysis resulted in a total of 420 requirement statements being grouped into the explained types. The simpler requirements which could be combined with one or-statement were counted as one requirement statement. The more complex requirements, e.g. nested if-statements of the static type, were counted as several statements based on the number output signal combinations and grouping in the textual specification. The number of requirement statements for each requirement level and type are presented in Figure 6.3.



	Operational	Static	Temporal	Transition	Sequential	Undesired behavior	Latency	Adaptability	Legal constraint
LDC	0	241	39	11	8	7	1	23	0
E2E	10	15	1	5	0	20	11	7	1
Collaboration	0	12	9	0	0	0	0	0	0

Figure 6.3. Number of requirements for each requirement type for the three specification levels.

The most common requirement types for the Automatic Headlight Switching (AHS) are the static functional requirements which represents 64 % of the total amount when combining requirements at all levels. This is also the most common requirement type at LDC level.

At E2E level the number of requirements are more evenly distributed over the different types than the other levels. Most of the undesired behavior is specified here, due to the signals in this type often being scattered over multiple LDCs. The requirement: *The main beam headlights shall not be deactivated as long as the driver makes a request for temporary activation (flash-to-pass) with the main beam stalk*, contains signals present in ExteriorLightsCtrl (*main beam headlights*) and ExteriorLightsHMICtrl (*main beam stalk*).

6.4 Discussion

If a requirement belongs to a certain type can be a matter depending on the surrounding requirements. For example, "*Signal X shall not be false when signal Y is true*" could be considered both a static and an undesired behavior requirement. The last part of the definition for the operational and undesired type ("...for a certain condition that is not narrow enough to consider the requirement fully tested if the condition is fulfilled") aims to take into account how many other signals that can change X (the output signal) while Y remains true (the condition). If Y is the only signal that is expected to change X, the requirement is to be considered belonging to the static type. However, if there exists a significant amount of other signals that might affect X while Y remains true this would then be considered belonging to the type undesired behavior. This because, when the requirement is to be tested it would be appropriate to also include the other signals.

The system is probably representative for most discrete-event systems based on combinational logic. Even though the amount of undesired behavior requirements most likely is higher for a supervisory control system. It is possible to have more requirements of this type in the Exterior lights-system as well, either by reformulating existing requirements or introducing new ones. An example can be: *No indicator light shall be activated unless the corresponding light source is active.*

A system more oriented to dynamical control probably has more temporal and also dynamical requirements, for example requirements regarding control signal activity. A higher amount of performance requirement also comes naturally for this type of system since the controlled variable(s) will change with respect to time.

A more safety critical system most likely has more undesired behavior in terms of safety properties as well as more performance requirements in terms of timing. This is the case for aircraft control systems, where systematic approaches to safety requirements are taken (Allenby and Kelly 2001).

6.5 Summary

Six different type of functional requirements have been defined and identified: operational, static, temporal, transition, sequential and undesired behavior. For each of the other three main types, one subtype was identified. These subtypes were: legal constraint, performance requirement regarding latency and specific quality requirement regarding adaptability. The most common type is the functional static one, which makes up 64% of the total number of requirement statements for the studied system.

The identified types will be used to provide a clear presentation of models and appropriate Model-Based Testing methods in the coming chapters.

7 MODELING REQUIREMENTS FOR USE WITH MODEL-BASED TESTING

This chapter covers the research question:

- How can these requirements be modeled for use with different Model-Based Testing (MBT) methods in Simulink® Design Verifier™ (SLDV)?

This question will be answered by using the three different types of models presented in Section 4.2 and the identified requirement types listed in Section 6.1. For each requirement type an attempt to model at least a few with each modeling method will be made. This is done to determine which types of requirements that are possible to model in Simulink®. Some practical tips and guidelines about modeling requirements can be found in Appendix A.

For the identified requirement type the first thing was to determine which types were suitable for verification with MBT. Requirements of the type adaptability was determined to not be suitable. The reason for this is that these requirements state the existence of a configuration variable and this is not a property that is appropriate to model or to verify with test cases. The requirements can however easily be verified by manual inspection.

Latency requirements are used when verifying the hardware and the delays when transmitting signals between multiple Electronic Control units (ECUs). Since only the model of the system is verified, and no communication delays are modeled, these requirements were disregarded. This means only requirements types under the category functional will be presented in this chapter.

The answer to the research about how to model requirements will be presented in Section 7.1, 7.2 and 7.3. The three sections cover one of the three modeling method each. These are the proof objective model, the test objective model and the cause-effect graph. In each section it will be answered if and how the different functional requirements can be modeled. Examples of each requirement type that was modeled is presented. Discussion about the result and work with the modeling is held in Section 7.4. In Section 7.5 the chapter is concluded and the answer will be summarized.

7.1 Proof objective models

Almost all identified functional requirement are possible to formulate as a proof objective model. The wide range of logic operators and the ability to use custom functions allow for complex properties to be specified.

Operational

This requirement was not formulated in a way that was suitable for property proving. It basically states that under a certain mode a function can be active or not active. This can be formulated as in Eq 7.1.

$$mode_1 \Rightarrow (function \vee notfunction) \tag{7.1}$$

When modeled like this the requirement can never be violated since the expected result will always be fulfilled.

Since this requirement is not applicable as a proof objective model directly an alternative formulation was attempted. The alternative formulation (seen in Eq 7.2) is to have the condition be that the function is active and the expected result is that one of the allowed modes is active.

$$function \Rightarrow (mode_1 \vee mode_2 \vee ...) \tag{7.2}$$

This formulation only works when there are one mode that the function can be active in, for example: *DRL shall only be operational in VehicleMode Running*. If more than one mode exists then this model can not verify that the function is operational in an individual mode.

Static

These requirements can be modeled in the same way as the generic proof objective model shown in Section 4.2. An example of a static output activation requirement is shown in Figure 7.1.

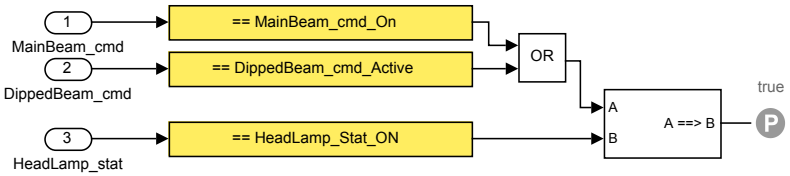


Figure 7.1. Example model of a static requirement. If the main beam or dipped beam is active then the signal HeadLamp_Stat have to be on.

Static requirements are common on both Logical Design Component (LDC) and End-to-End (E2E) level. LDC requirements are usually written as if-statements and as a result they are precise and easy to translate into a proof objective model. On E2E level requirements are in general more ambiguous and therefore harder to formalize with logic expressions. Without actual signal names it takes more effort to figure out what the condition and expected result are.

Temporal
Conditions that describe temporal properties are modeled using temporal operators from the SLDV block library. The detector block is used to model a condition where signals shall have certain values for a required amount of time.

Some temporal requirements can describe a complex behavior that can be decomposed to simpler behaviors. For example *"If condition_x is fulfilled the function_y shall activate after at least 1 second and at most 2 seconds"*. This can be decomposed into two properties:

(a) *Function_y shall be inactive before condition_x has been fulfilled for 1 second.*
(b) *Function_y shall activate before condition_x has been fulfilled for 2 seconds.*

Property (a) can be modeled as in Figure 7.2. The `Detect Increase` block outputs true for one time step when *condition_x* is fulfilled. The signal is then extended to 1 second with the `Extender` block. During this time the signal shall not be active. If *condition_x* becomes unfulfilled then the `Extender` is reset and outputs false since the requirement does not say what shall happen when the condition is not fulfilled.

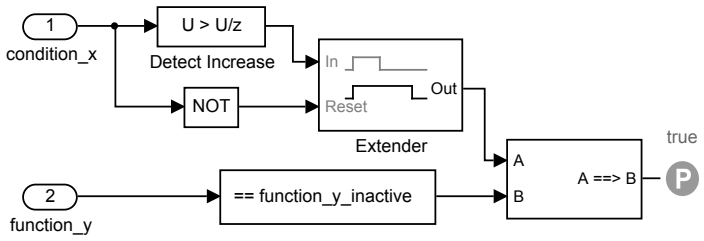


Figure 7.2. A property that states that a certain time after *condition_x* have been fulfilled *function_y* shall be deactivated.

Property (b) can be modeled as in Figure 7.3. The `Detector` block will output true when the *condition_x* has been fulfilled for at least 2 seconds. If the `Detector` outputs true then *function_y* shall be active.

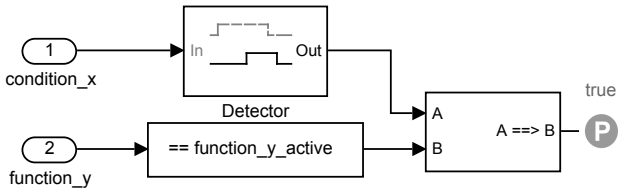


Figure 7.3. A property that states that if *condition_x* have been fulfilled for a certain time *function_y* shall be active.

Transition

The condition in this requirement is that a signal change to or from a specific value. This change is detected using a relational operator and a `detect change` block. Depending on if the condition is a change to or from a value the `detect increase` or `detect decrease` block is used respectively. In Figure 7.4 and example of an transition requirement can be seen.

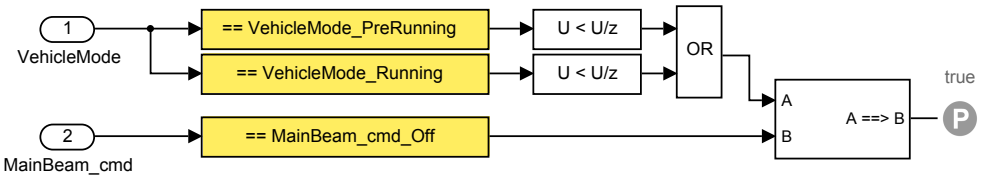


Figure 7.4. A requirement stating that the Main-Beam shall be off when transitioning out of vehicle mode running or pre-running.

Sequential

A state machine is a formal way of representing a requirement. Instead of using a standard proof objective model the state machine is remade in Stateflow®. The objective is then to prove that the system model will always follow the state chart, as seen in Figure 7.5. Sequences that are written in text can be reinterpreted as a state machine.

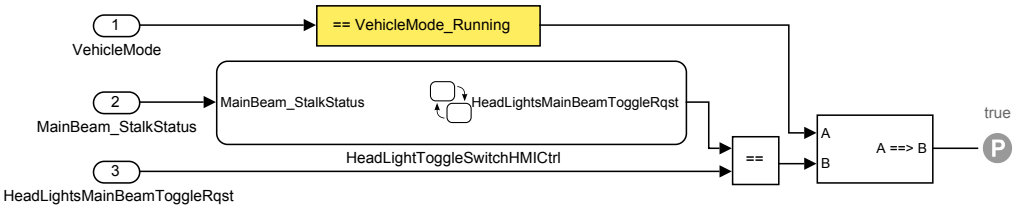


Figure 7.5. A sequential requirement specified as a state chart. The requirement is only valid when the vehicle mode is running. The expected result is the output from the state machine and the signal from the system (*HeadLightMainBeamToggleRqst*) are the same.

Undesired behavior

For requirements of the type undesired behavior the generic proof objective model with the implies block was not used. Instead an alternative formulation was used. The objective for Simulink® Design Verifier™ is to prove that

$$Condition \wedge Undesired\ behavior \tag{7.3}$$

always is false. The reason for using an alternative proof objective model was to make it more similar to the written requirement which states that something is forbidden. The alternative formulation also reduces the number of logical operators.

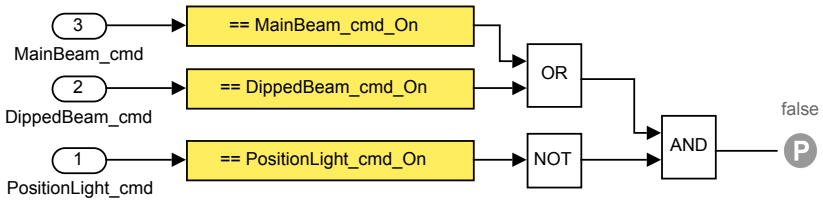


Figure 7.6. Proof objective model of an undesired behavior. The requirement states that dipped beam or main beam shall not be activated when the position lights are not activated.

Some properties that describe an undesired behavior require extra care when modeling. For example *"If condition_x is fulfilled then function_y shall not change value"*. This requirement have three things that have to be regarded when modeling:

(a) *Function_y is allowed to change value the same moment as condition_x is fulfilled.*
(b) *If function_y changes value then the detect change logic operator will output true for one time step.*
(c) *The standard detect change block will output true the first time step if the initial value of function_y is non-zero.*

To solve problem (a) and (b) a custom detector block that detects if the condition has been fulfilled for one time step is used. To solve problem (c) a custom detect change block was created that will always output false when the simulation time is equal to zero. The result can be seen in Figure 7.7.

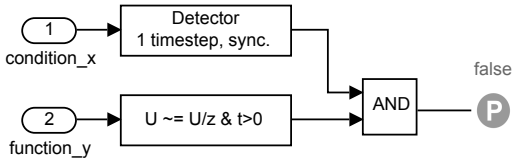


Figure 7.7. Model of a requirement where a signal shall not change when some condition is fulfilled

7.2 Test objective models

With the test objective model it is possible to model and generate test cases from properties. The models presented in this section did successfully generate a test case. No constraints on any input signals were used unless specified otherwise.

Operational
Requirements of this type where modeled by having the objective be that the function shall be active and the system shall be in a specific mode. An example of a complete test objective model for a requirement of type operational can be seen in Figure 7.8.

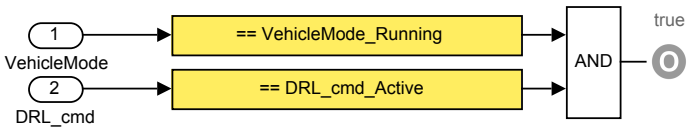


Figure 7.8. A test objective model for an operational requirement. The function is the daytime running lights and the operating mode is vehicle mode running.

Static
Requirements with a static condition are modeled as described in section 4.2.2. This model only cover one specific condition that is specified in a requirement. An example of a static requirement can be seen in Figure 7.9.

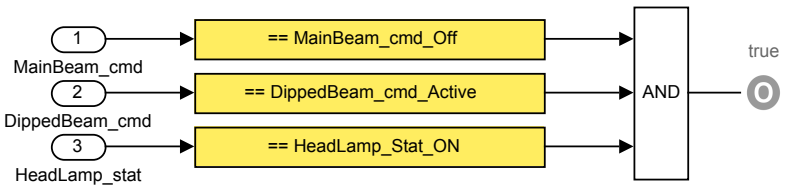


Figure 7.9. A static requirement as a test objective model. The condition is that the dipped beam is active and the expected result is that HeadLamp_Stat is on.

The HeadLamp_Stat will be on in main beam or dipped beam is active. To verify that the dipped beam is the cause for the HeadLamp_Stat to be on then the main beam must be off.

Temporal

An example of a temporal test objective can be seen in figure 7.10. The requirement is that *"function_y shall be active before condition_x has been fulfilled for 1 second"*.

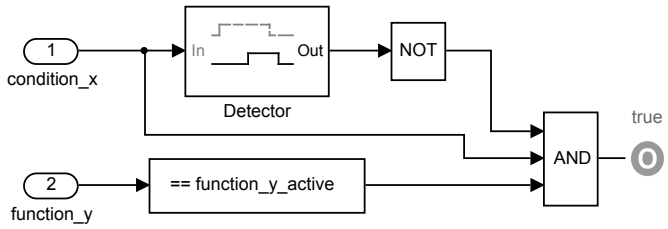


Figure 7.10. A temporal property as test objective model.

The test objective is constructed so that *function_y* and *condition_x* shall be true but *condition_x* shall not have been true for more than 1 second.

Transition

A test objective model of a transition requirement can be seen in Figure 7.11. The requirement states that "when transitioning out from vehicle mode running the main beam shall deactivate".

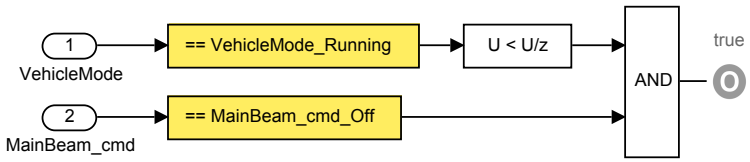


Figure 7.11. A test objective model for a transition requirement.

When using this test objective model it will provide one test case with a transition from vehicle mode running to an other vehicle mode. Individual models have to be created if test cases for transitioning to every other vehicle mode is wanted. If a transition from a specific vehicle mode to another then this can be modeled as seen is Figure 7.12. The objective is to transition from vehicle mode running into vehicle mode living.

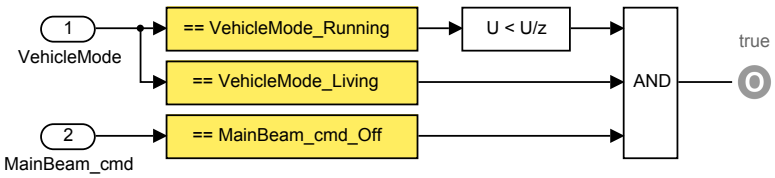


Figure 7.12. A transition from one mode to a specific other.

Sequential
Modeling sequences with the test objective model was done by chaining test objectives after each other. This was made possible with the infinite extender block. The infinite extender block is a custom block that outputs true endlessly after true is received on the input. In Figure 7.13 a sequence property is modeled. The property describe the effect when toggling the main beam stalk and how it effects the main beam.

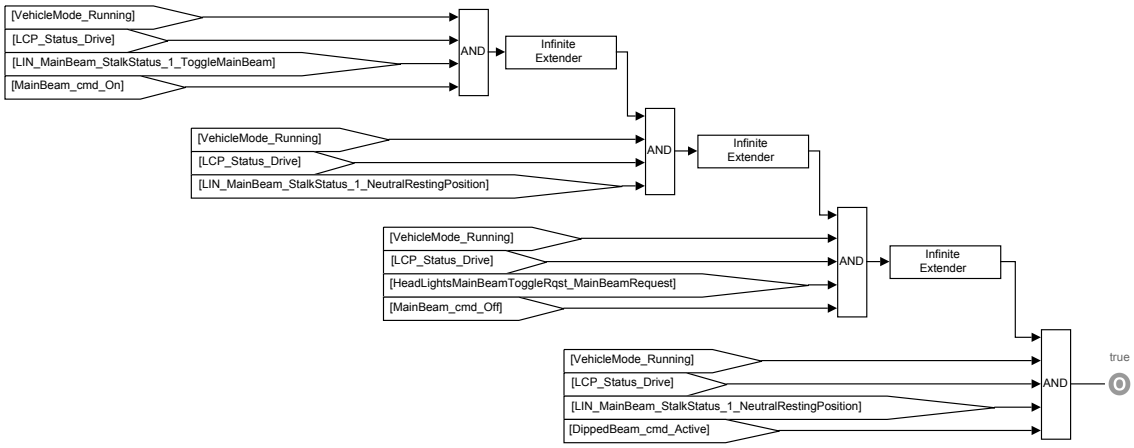


Figure 7.13. A test objective model of a requirement describing the toggling of the main beam stalk.

Undesired behavior
Using test cases to verify that an undesired behavior does not occur requires a large amount of test cases. Since only one test case is generated per test objective model this would require a lot of models. This means that the requirement is not useful as a test objective model.

7.3 Cause-effect graphs

This section covers the modeling of cause-effect graphs. The focus in this section is mainly the models and not the actual test case generation. For each of the modeled requirement it have however been confirmed that the models generate working test cases when using a coverage objective. More results about the test case generation is found in Chapter 8.

Operational
This requirement does not have a specific enough cause. Since specifying the cause is a main part of this method it was determined that this requirement type is not suitable as cause-effect graphs.

Static

Static requirements can be modeled as described in Section 4.2.3 without any modifications. No issues were identified for this requirement type. Requirements written as if-statements were easily adapted to cause-effect graphs. One branch in the cause-effect graph corresponds to one if-statement. Cause-effect graphs for requirements on E2E level become much like a simplified model of the system with fewer inputs and outputs.

In Figure 7.14 the reference model of a static requirement is depicted. The inputs to the graph is the vehicle mode, mean beam stalk position and the light mode status. Different combinations of inputs leads to decision paths that activate different effects.

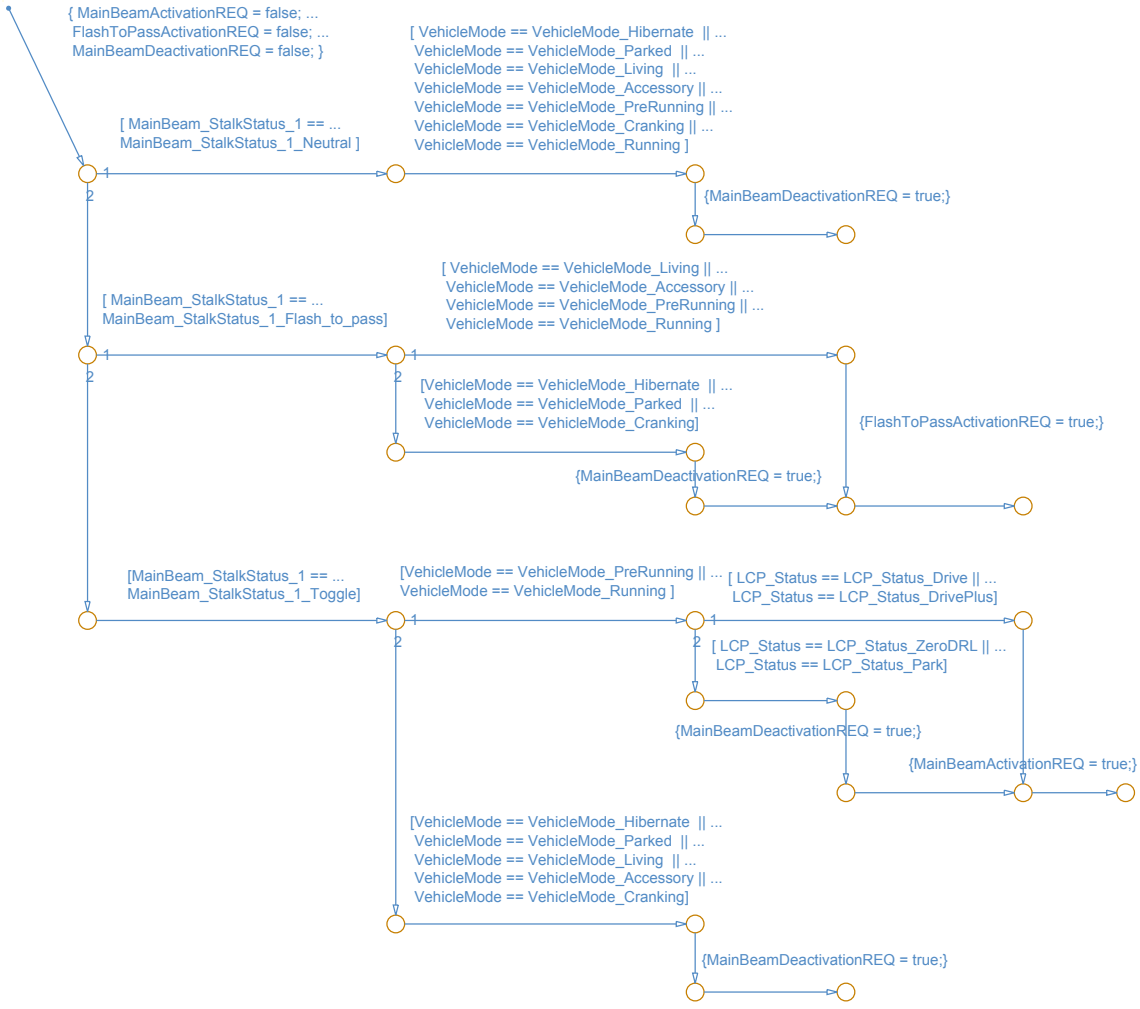


Figure 7.14. Example reference model of a static requirement.

Temporal
The main issues with modeling temporal requirements as a cause-effect graphs are:

(a) The test reference model need to be augmented with states and timed events.
(b) Adding states will result in that the test reference model has a different structure than graphs for other requirements.

The solution to this to model the temporal properties in separate graphs outside the main test reference model.

Example: *"If signal_y has been true for n seconds and signal_w is true the function_x shall be active"*

The temporal behavior of *signal_y* can be modeled in a separate state chart, as seen in Figure 7.15.

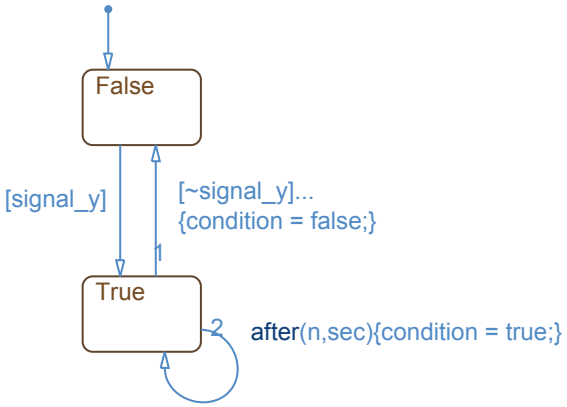


Figure 7.15. Temporal behavior modeled separately. The output condition will be true when *signal_y* has been true for *n* seconds

The output (signal named *condition*) from the chart is a signal that can be used as an event in the main cause graph, as seen in Figure 7.16.

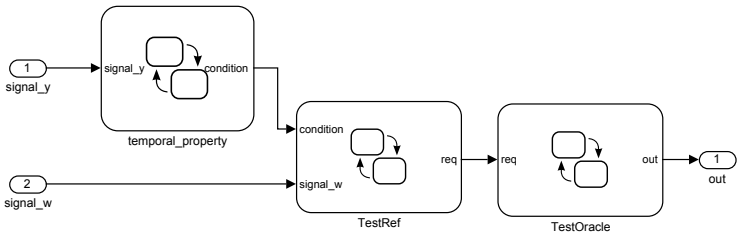


Figure 7.16. Cause-effect graph with temporal property modeled separately.

Transition

Modeling the transition property is done as in the proof objective model. The transition can be modeled using standard logic operators or with state charts. An example with logic operators can be seen in Figure 7.17.

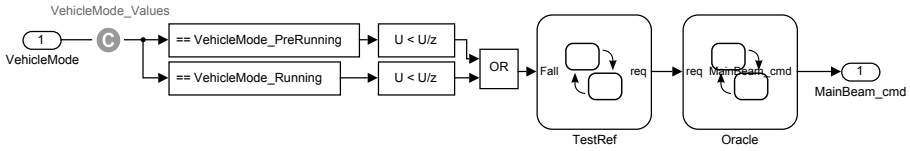


Figure 7.17. Cause-effect graph with logic operators to detect a transition.

The main issues with transition requirements as cause-effect graphs are:

(a) The condition can not be fulfilled at the first time step.
(b) The requirement does not state what the output signal is before the condition is fulfilled.
(c) When generating test cases the output must be known at all times.

A solution to the problem described above is to combine the transition requirement with other requirements regard the same output. It can, for example, be combined with a static requirement, as seen in Figure 7.18.

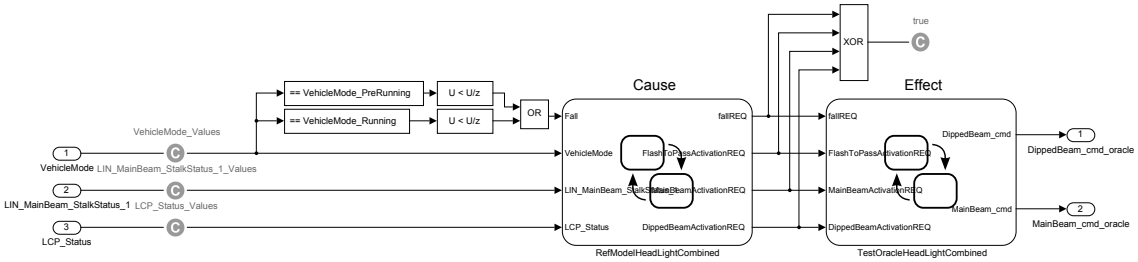


Figure 7.18. Transition requirement combined with a static requirement that describes the behavior of the same output.

The transition event is an input to the test reference graph. The branches from the transition cause graph is added to the static cause graph. When using a model coverage objective to generate test cases it will give some test cases where one of the static requirements will be active first and then the transition requirement will be active.

Sequential

When modeling sequences in Stateflow® states are used to allow the different steps in the sequence to occur. If the requirement is specified as a state machine then it is possible to extract sequences from it. To keep the structure of the reference model intact the sequence is modeled in a separate state chart. The steps in the sequence is modeled using states. The sequence chart output a boolean that is used as an event in the reference model. In Figure 7.19 a requirement that states the effect when toggling the main beam stalk is depicted. It describes the sequence of pushing the main beam stalk to toggle position and then back to neutral position. A state is required between these two events since the main beam stalk can not be in both positions at once.

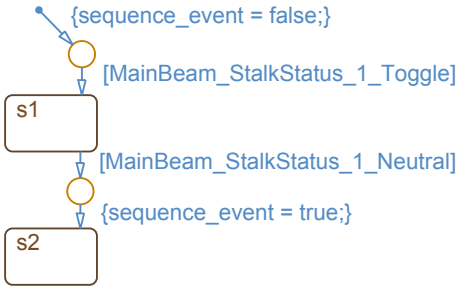
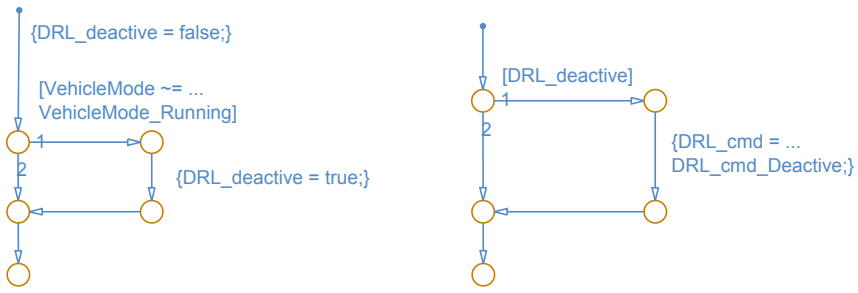


Figure 7.19. A state chart that will output an event when a sequence is completed.

Undesired behavior

To verify that something shall not occur with ordinary test cases can be very hard since a lot of test cases would be required to gain confidence that it will not occur. It could be possible to generate test cases that cover all static input combinations. But all sequences and temporal behavior are not feasible to cover with test cases. In Figure 7.20 the test reference model for a requirement of type undesired behavior.



(a) Test reference **(b)** Test oracle

Figure 7.20. Test reference model and test oracle for an undesired behavior.

7.4 Discussion

The difference in time spent working with the three modeling methods is of significance. The largest portion of the time was spent working with the proof objective models. The test objective model was deprioritized in favor for the cause-effect graphs which looked more promising. Because of this some of the results regarding test objective models are not as developed as for the other modeling methods. For example the part about temporal properties was only briefly investigated. For other methods enough requirements were modeled of each type to get an assessment of how it should be modeled. An exception to this is that only one sequential requirement was used for property proving. The reason for this is the infrequency of this requirement type. The requirement that was modeled covered a local behavior in the system model. For this it does work and it can be assumed to work for similar situations. If more requirements of each type had been modeled then maybe some exceptions to the modeling suggestions made in this chapter could have been identified.

The proof objective models were almost exclusively modeled with Simulink® blocks. Using state charts or MATLAB® code could perhaps have been an improvement in some cases.

The time it takes to model and generate one test case with the test objective model compared to the cause-effect graphs is significantly longer. With the cause-effect graph generating a test case takes less then a minute and with the test objective model it can take up to an hour and sometimes more. The time it takes to generate the test cases is related to the size of the model. In a cause-effect graph the model is relatively small compared to the test objective model which also include the system design model when generating test cases.

An issue with many requirements on LDC level is that properties for a output signal is spread out in different requirements. In many cases there are a main requirement that describe conditions and expected results. And in some cases there are separate requirements that describe exceptions to these conditions. This causes problem when only one requirement is taken into account since that conditions is incomplete. If everything is not taken into account in the condition then false violations can appear when performing property proving.

An alternative method to use when verifying undesired behavior is to use the test objective model. The objective is to generate a counterexample by stating that the condition and the undesired behavior shall be true simultaneously. If a test case is successfully generated then the requirement is violated. No benefits of using this method over the proof objective model have been found however.

7.5 Summary

The proof objective model is applicable on all functional requirement types except operational. The generic proof objective model with the implies block and proof objective is used for almost all requirement types except for undesired behavior where an alternative modeling strategy closer to the written formulation were used. Some custom blocks were created to be able to model certain properties.

The test objective model is applicable on all functional requirements except undesired behavior. This method is the only one that can test operational requirements in an easy way. The infinite extender block is used to link test objectives together and create sequences.

The cause-effect graphs is applicable on all functional requirements excepts operational and undesired behavior. To give better readability and lower complexity some properties can be modeled outside the test reference graph. For example transition properties are best modeled with logic operators and temporal properties are best modeled in separate state charts and using timed events.

In Chapter 8 the different modeling methods are evaluated in depth and an appropriate method for each requirement type in purposed.

8 EVALUATION OF MODEL-BASED TESTING METHODS

This chapter cover the research question:

- What Model-Based Testing (MBT) methods are appropriate to use for each requirement type?
  - At which system levels and test systems can the different methods be applied?
  - How well does the requirement models scale with respect to increased number of input signals and requirement complexity?
  - What types of errors can be detected with the different methods?

The answer is presented in Section 8.4. Motivations to the answer is given in the sections about the different criteria, i.e. applicability, scalability and error detectability. The answer is summarized in Section 8.6.

Data regarding scalability was obtained by modeling requirements and visually compare the models. For the question regarding error detection, both requirement modeling and application of Simulink® Design Verifier™ (SLDV) features served as methods for gathering data. As to the question regarding application at different system levels and test systems, data was gathered by experimenting with SLDV as well as analyzing models of requirements.

A relative comparison of the methods, based on a general analysis for each criterion, is presented in Table 8.1. Details of the analysis will be presented in the following sections. For each criteria the analysis combines data from all requirement types, but when a specific type deviates from the common analysis it will be handled separately.

A test objective model can only cover one branch in a static requirement, therefore this model type increases fast in complexity when the number of signals in the requirement increases. Since every branch has to be modeled independently the complete model will increase significantly in complexity for an increasing number of requirements, resulting in bad scalability.

With the test objective models all signals of the system is included in the generated test cases which makes it harder to identify the purpose of a test case. The test cases generated from cause-effect graphs only include the input and output signals that are relevant to the requirement. Input signals disregarded in the test objective model has to be constrained and therefore the model complexity is high for all requirements except the operational type. Since the method only has acceptable

Table 8.1. Relative ranking of the different methods based on the three evaluation criteria. Lower number indicates higher ranking. Details describing each criterion are presented in the following sections.

Criterion	Method		
	Proof objective models	Cause effect graphs	Test objective models
Applicability	2	1	
Complexity & Scalability	2	1	3
Error detectability	1	2	

performance for this type, which represents a small amount of the total, it is not considered further when going in to details of the different criteria.

8.1 Applicability

By creating a cause-effect graph based on abstract events, test design can be performed on a higher more abstract level. No system signals are then included, instead the abstract events will be translated into system signals by blocks outside the graph. These translations can be included when the details of the system are created and the graph can then be used to generate a useful test case. Figure 8.1 shows an example of a cause-effect graph with abstract events.

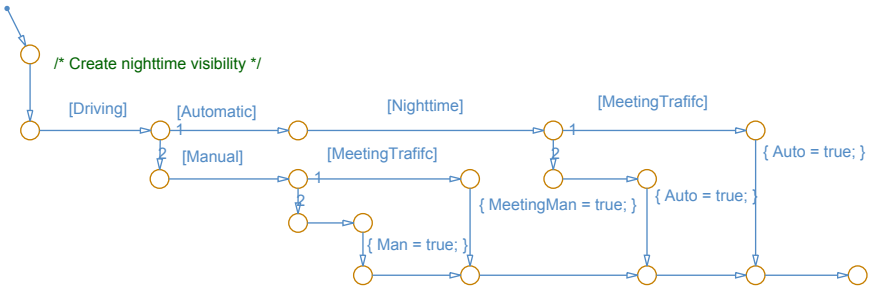


Figure 8.1. Example of a cause graph with abstract events. The events are translated into system signals with blocks outside the graph.

The test cases generated from a cause-effect graph are possible to use for both model and Hardware-In-the-Loop (HiL) tests. The proof objective models are efficient when working in a model environment but does not provide the opportunity to generate stimuli to a test rig. This grants a higher applicability for cause-effect graphs and is a main advantage of this method.

Both proof objective models and a cause-effect graphs are also possible to use as observers when running tests. They can then trigger assertions when requirements are violated or expected result does not match the actual one. This makes both methods applicable for verification on different system levels. Both model types can easily be connected to a system model in Simulink® and they can also run inside a testing environment for HiL tests, such as CANoe.

Only generating test cases based on coverage criteria is not considered sufficient at all times. The reason for this can be that the selected criteria does not result in a high enough coverage of the intended behavior of the system. Figure 8.2 shows an example where this is the case. Generating a test case based on Modified Condition/Decision Coverage (MCDC) coverage will not result in VehicleMode being Running at the same time as LCP_Status is either Drive or DrivePlus hence it will miss a main part of the intended behavior.

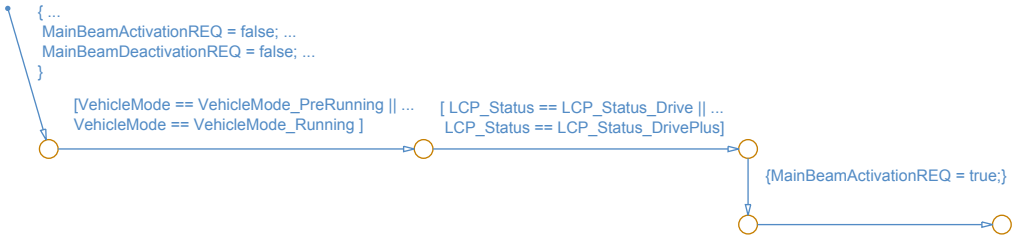


Figure 8.2. Example of a cause graph where MCDC coverage might not be considered sufficient. The resulting test case will never activate main beam due to VehicleMode being Running and LCP_Status being Drive or DrivePlus.

Another reason the selected coverage criteria might not be sufficient is that the generated test case is hard to follow. This can lead to not understanding the cause of an error if the System Under Test (SUT) fails the test case. Figure 8.3 shows the result of generating a test case based on MCDC coverage for a cause-effect graph modeling main beam activation.

By instead combining a coverage criteria and test conditions on the input signals, test cases with specific combinations of these signals can be generated. To generate test cases in a systematic way a MATLAB® function has been created (see Appendix B for the code). It loops through levels of test conditions, here called a test strategy, and generates test cases for each combination. For a combination that is not covered by the cause-effect graph (it might not be within the test scope) the model will be contradictory and no test case will be generated. The test cases for each input combination can then be combined into a long test case.

The result of a systematic test case generation can be seen in Figure 8.4. The reason why some combinations of inputs that are missing is because these combinations are prohibited by the cause-effect graph.

Cause-effect graphs for static requirements are easy to read and give a good overview of the different decision paths, which makes them easier to understand for someone

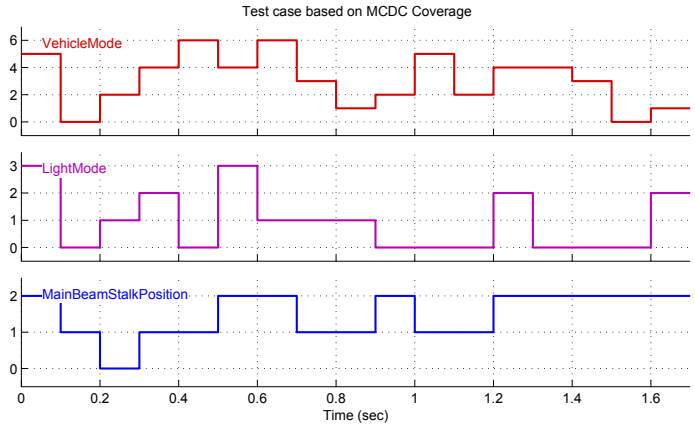


Figure 8.3. A test case generated from MCDC coverage criteria.

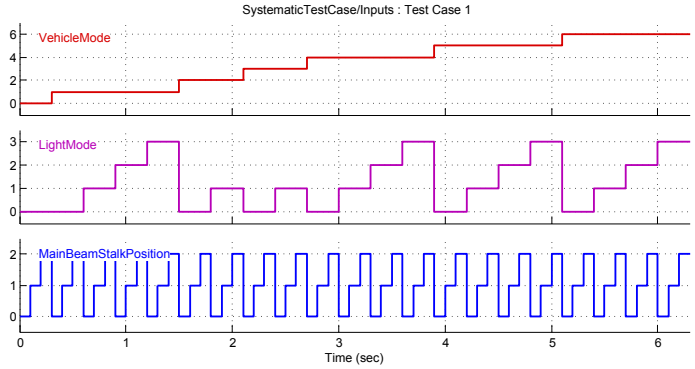


Figure 8.4. The resulting input vectors after a systematic test case generation. The strategy aims to cover three values for MainBeamStalkPosition for each value of LightMode, for each value of VehicleMode.

not familiar with Simulink®. This makes them useful as an alternative representation for a textual requirement or for communication purposes. Proof objective models are more difficult to understand since they require more experience with logics and are less readable than the cause-effect graphs.

8.2 Model complexity and scalability

Here model complexity has two parts, the amount of blocks, subsystems and Stateflow® states together with the amount of necessary custom defined functions and more complex Simulink® blocks such as temporal operators. The model complexity is also used as a measure of scalability by analyzing how it changes for increasing requirement complexity, such as increasing number of signals or signal values. The time consumption for modeling is also briefly analyzed.

The cause effect graph can be separated into three parts: complex conditions (temporal, transition and sequential), reference model and test oracle. This decreases model complexity but since the complex conditions also have to be included in the graph at the correct place these will increase time consumption. For proof objective models the complex conditions will consume less time for modeling since they can be separated from the other requirements.

The efficient way of modeling decision paths in cause effect graphs help reduce complexity and improve scalability. It also has a positive effect on the time consumption for modeling since the graphical structure can be created automatically for many cases. Creating a test strategy for systematic generation of test cases increases modeling time and complexity as well as lowering scalability.

For the proof objective models the complexity is high for the static requirements due to the way if-statements are modeled. Since the expected result has to be modeled nearby the conditions this has a negative effect on scalability. For the least complex requirements, based on only a few signal values, the model instead has low complexity.

8.3 Error detectability

This section presents the possibilities to detect different errors for each of the methods. When errors related to a specific requirement are detectable for a certain method, this is also indicated.

Property proving has good error detectability for violations to undesired behavior requirements since any counterexample can be provided. The main advantage of property proving is that the absence of an error can be shown, where test cases only can show the presence of an error. Applying property proving also helps finding signals missing from the implementation but part of the requirement. From the counterexample the missing signal can often be identified as the one that is changing when other signals remain constant.

Since the proof objective models have to be very specific (and also cover all exceptions when a modeled behavior shall not be active) it requires a review of the requirements from a different perspective than a purely textual review. This can help find incomplete, unclear and contradictory requirements. During the case study 19 requirement issues was identified. The different types of issues and number of issues found, are shown in Table 8.2.

Five of the conflicting requirements were *else*-statements describing normal behavior, that covered the signal values being subject of failure handling. This lead to the normal behavior stating one output for the system and the failure handling another. The last conflicting issue was an End-to-End (E2E) requirement describing a behavior not included in an Logical Design Component (LDC).

Table 8.2. Number of requirement issues found for different issue-types.

Requirement issue type	No. of issues
Conflict	6
Incompleteness	1
Ambiguities	12

The issue regarding incompleteness was a requirement not stating when to set an output signal to value *Error*, when this value was being handled in another requirement. Most of the ambiguities regarded use of incorrect language or undefined terms. For example, some failure handling requirements stated a behavior for when an input signal was "erroneous", without stating the corresponding enumerated values.

These issues were found when first modeling proof objectives and performing property proving. It was confirmed that the conflicts also could be found with cause-effect graphs if all requirements concerning the same outputs were grouped together. The ambiguities were found when reading and translating requirements into models, which gives an indication that these could have been found when modeling cause-effect graphs as well, if the method had been investigated before proof objectives.

The cause effect graphs also help find incomplete requirements since it presents decision paths in a clear way. This makes it possible to check that all paths are covered. When generating systematic test cases with conditions, all input combinations that part of the strategy will be covered. If there is no valid cause-path leading to the activation of an effect the model will be contradictory, due to the condition stating that one and only one effect must be active at all times. This will be an indication that some part of the input space is not covered.

To find uncovered decision paths a proof objective can be used as shown in Figure 8.5. A simple constructed example can be seen in Figure 8.6.

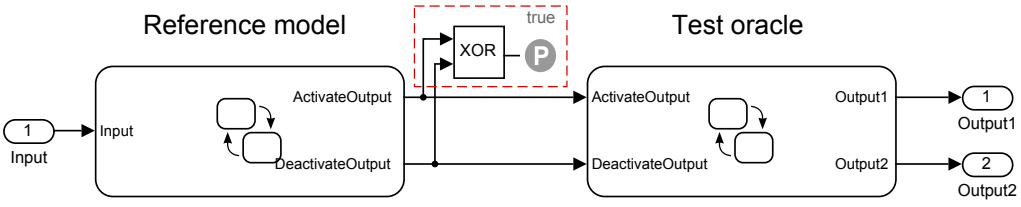
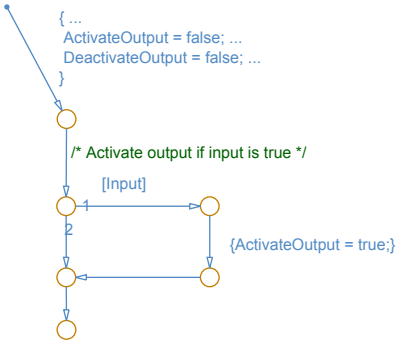
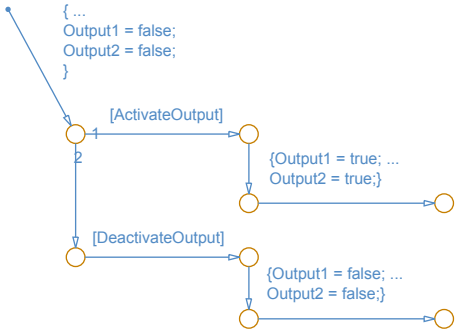


Figure 8.5. Cause effect graph model with a proof objective to check that there is always an active effect.

Looking at a the simple example, where the input being false does not activate any effect, it is clear that the requirement this is based on is incomplete. This is the case because it is unclear what the system shall do if the input signal is false. By trying to prove that there is always an active effect, a counterexample will be provided



(a) A simple reference model with an if-statement. It indicates that the effect *ActivateOutputs* shall become active when the input is active. Note that there is no effect active when the input is not active.



(b) The corresponding test oracle. It activates and deactivates two output signals based on the cause graph.

Figure 8.6. Reference model and test oracle that exemplifies an incomplete requirement.

stating that the input being false will violate the property. If the model can then be verified to match the requirement, the requirement is incomplete (assuming that there are no constraints on the input signal).

8.4 Evaluation

For each requirement type an appropriate MBT are proposed, based on the evaluation criteria. The result is presented in Table 8.3.

For the operational requirement type a test objective model is most suitable since it is the only method that can fully handle this type. It also provides a low model complexity for this specific type.

The proof objective model is most suitable to use for the undesired behavior since it provides superior error detectability. The error detection is an advantage when applying property proving to other types of requirements as well, but since no test cases can be generated to serve as stimuli for other test systems it fails due to the lower applicability.

As discussed in Section 7.4 no method was used to model latency requirements, instead this could be verified by using a cause-effect graph as an oracle. The requirement is verified by allowing a maximum time gap between the oracle and the SUT. However, the model must contain the corresponding requirement(s) stating output signal activation and must be used as an test oracle. Since a configurability requirement was not possible to model with any of the methods, none of them were found to be appropriate.

Table 8.3. The proposed choice of MBT method for each requirement type. For latency requirements the cause effect graph is to be used as an observer. None of the MBT methods were found appropriate to use for the configurability requirements.

Requirement type	Method		
	Proof objective model	Cause effect graph	Test objective model
Operational			✓
Static		✓	
Temporal		✓	
Transition		✓	
Sequential		✓	
Undesired behavior	✓		
Latency		✓	
Configurability			

The other requirement types, static, temporal, transition and sequential were found appropriate for use with cause effect graphs. For all these types the method performs well regarding complexity, scalability, modeling time and required skill level. The method provides good applicability due to the possibilities for testing in multiple environments. It also allows for finding incomplete requirements.

8.5 Discussion

Even though the test objective models are not useful for most of the identified requirement types, the test objective block can be a valuable tool when doing property proving. This block can be applied to the condition at an implies block to test that the implication is not proven valid due to the condition being permanently false.

When error detection is crucial property proving might be more appropriate than cause effect graphs when working in an model environment. An alternative is to have a detailed cause effect graph and use proof objectives to verify equality between oracle and model output. For any of these alternatives to be useful for a safety critical system, target code must be generated from the system model. This is to ensure that the code running on the real system is as similar to the model as possible to create assurance that the property is fulfilled also by the real system.

For a system relying on discrete events, states and sequences the proposed choice of MBT methods are most likely applicable, with the previously mentioned exception for safety critical behavior.

A system involving more dynamic behavior, such as control systems, the property

proving is probably the most useful method, due to requirements stating signal properties instead of values. For example, the requirement "The output signal shall be in the range 0-20V", is more suitable for property proving due to being more easily modeled with this method.

Even though some unclear and contradictory requirements were found during the case study these might also be possible to find by a textual review. However, they were not found during the textual review thus MBT provides possibilities to find unclear and incomplete requirements.

Blocks incompatible with SLDV analysis was not within the scope, but this would most likely not affect the proposed methods. This since the cause effect graphs were possible to model with compatible blocks and no system model (that might contain incompatible blocks) is used when generating the test cases.

The aspect of system variants was not covered but since Simulink® provides possibilities for variant handling by selecting a variant subsystem based on a variable, and SLDV can treat this variable as a configuration variable, it is probably possible to include variants when proving properties and generating test cases.

The proposed requirement types has the benefit of giving rise to similarities when generating test cases. This can be useful for further investigations regarding attributes and usefulness of automatically generated test cases.

It would also be possible to group the requirements based on their complexity in terms of number of signals in the condition and expected outcome. This can be more useful if investigating computing time for test generation or property proving.

8.6 Summary

Cause-effect graphs were found to be the most appropriate MBT method for all functional requirement except the ones regarding undesired behavior. This is mostly because of the possibility to apply generated test cases on different test systems and model both abstract and concrete requirements.

Test objective models were found to only be appropriate for the operational type, but was also the only method that could handle it properly.

For undesired behavior, property proving with proof objective models was found most appropriate due to the excellent error detectability.

The last chapter will present conclusion based on the answers to all research questions and connect to the purpose and objectives presented in Section 1.2.

9 CONCLUSION

This thesis have differed from related work by focusing on the actual System Under Test (SUT) and its requirements. With the selected approach it has been possible to evaluate the usefulness of Model-Based Testing (MBT) and in an industrial context and give an answer to when and how it is appropriate to use.

Dividing requirements into categories was a decisive factor for the work with the remaining research questions. With the different requirement types it was possible to present coherent and practical modeling results. With a classification of just functional and non-functional requirements this would not have been possible.

The result of modeling requirements was the basis for the evaluation of which modeling method was the most suitable for each requirement type. Including applicability as a criteria elevated the evaluation and gave benefit to test case generation since it applicability spans the whole development process.

9.1 Improving the development process with Model-Based Testing

The case study confirmed the conclusions made in related works: that MBT can help find unclear and ambiguous requirements. When working with proof objective models or cause-effect graphs incomplete, unclear and contradictory requirements were found, which fulfills Objective 1a. Ambiguous requirements are not only a problem in the verification process. Formal and concise requirements give the developers a clear target and reduce the risk of unintended behavior occurring in the system. This is especially important since implementation errors due to ambiguous requirements have been shown to be a main problem area. Using a model as a formal specification is one way of solving this problem.

Creating cause-effect graphs of system behavior and generating test cases can be a more efficient alternative to writing test scripts manually and can help automate the development process as stated in Objective 2a. Since the test cases are present in a general form (as vectors with signal values) this introduces opportunities for automated creation of test scripts. Even though the complete process of moving from a generated test case to an executable script was not covered, the area has been investigated and therefore Objective 2b is considered fulfilled.

Modeling of requirements can be initiated at the same time as the first requirement is written. Test design can also be performed at a higher more abstract level by using cause-effect graphs with abstract events. This allows for starting the design of tests before the actual system is designed. It can be concluded that the MBT methods

covered by this thesis grants opportunities for testing earlier in the development process, therefore fulfilling Objective 3a.

Applying MBT together with Model-Based Design (MBD) and Simulink® can save time and streamline the verification process. This is partially thanks to the extensive verification that can be done early in the development process with property proving on software components created in Simulink®. According to Objective 3b it has been shown that MBT is possible to apply on multiple test systems in different phases of the development, for example proof objectives as observers in a Hardware-In-the-Loop (HiL) rig or with property proving in a model environment.

Requirement models are more abstract than an executable test scripts and are easier to update and modify, therefore models provide possibilities for reuse in coming development processes, especially for cause-effect graphs with abstract events. Based on this Objective 3c is considered fulfilled.

Previous research has shown that MBT is useful for generating test cases based on coverage criteria. However, this thesis can conclude that when generating test cases for a more complex system these become hard to understand and connect to the intended behavior. Thus automatic test case generation is not useful on its own. Without manually composed test cases or a human that can inspect and verify the correctness of the generated test case the connection between the human and the product can be lost. This reduces the possibilities for early validation during the development process.

9.2 Raised questions

This section lists of new problem formulations that where found and not solved during the work.

How well implementation errors can be detected by using MBT has only been briefly covered (finding missing signals with property proving) which is why Objective 1b is not considered fulfilled. By applying property proving on a larger amount of models or running generated test on both models and other test systems, future research has the opportunity to fully cover this topic.

Future research could also further cover automatic test case generation according to a more functional coverage criteria, as proposed by "systematic test case generation". Alternative methods for generating test cases, such as using state charts, and comparing those to cause-effect graphs is also an interesting topic for future work.

Another raised question is if cause-effect graphs for software components can be combined to model a complete Electronic Control unit (ECU) and how the combined behavior then can be limited.

How to execute a test generated by Simulink® Design Verifier™ (SLDV) in a HiL testing environment is also a raised question. Possibilities that have been stumbled upon are: generating an executable test script from the SLDV test case or running the test case in a Simulink® environment connected to a HiL environment.

Finally it can be concluded that when selecting an appropriate MBT method it is valuable to analyze the types of requirements and behavior of the SUT.

Bibliography

Allenby, K. and Kelly, T. (2001). Deriving safety requirements using scenarios. In: *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. pp 228–235.

Apfelbaum, Larry and Doyle, John (1997). Model based testing. In: *Software Quality Week Conference*. pp 296–300.

Bringmann, E. and Kramer, A. (2008). Model-based testing of automotive systems. In: *Software Testing, Verification, and Validation, 2008 1st International Conference on*. pp 485–493.

Broy, M., notes in computer science (e-book collection), Lecture and (e-book collection), SpringerLink (2005). *Model-based testing of reactive systems: advanced lectures*. vol. 3472. Springer. Berlin.

Broy, Manfred, Sascha, Kirstan, Helmut, Krcmar and Bernhard, Schtz (2012). What is the benefit of a model-based design of embedded software systems in the car industry?. In: *Emerging Technologies for the Evolution and Maintenance of Software Models*. pp 343–369. IGI Global.

Chen, Chunqing, Sun, Jun, Liu, Yang, Dong, Jin S. and Zheng, Manchun (2012). Formal modeling and validation of stateflow diagrams. *International Journal on Software Tools for Technology Transfer*, **14**(6), 653–671.

Chung, Lawrence and Prado Leite, Julio Cesar (2009). Conceptual modeling: Foundations and applications. Chap. On Non-Functional Requirements in Software Engineering, pp 363–379. Springer-Verlag. Berlin, Heidelberg.

Etienne, J.-F., Fechter, S. and Juppeaux, E. (2010). Using simulink design verifier for proving behavioral properties on a complex safety critical system in the ground transportation domain. In: *Complex Systems Design & Management*, Marc Aiguier, Francis Bretaudeau and Daniel Krob (Eds.). pp 61–72. Springer Berlin Heidelberg.

Friedman, J. (2006). Matlab/simulink for automotive systems design. In: *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*. vol. 1. pp 1–2.

Gibbs, Samuel (2014). Google's self-driving car: How doeas it work and when can we drive one?. *The Guardian*29 May 2014 Available: `http://gu.com/p/3ptht` [Accessed: 2014-05-30].

Glinz, M. (2007). On non-functional requirements. In: *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*. pp 21–26.

Hanselmann, Herbert (2008). Challenges in automotive software engineering. In: *28th International Conference on Software Engineering : proceedings : 20-28 May 2006, Shanghai, China*. ACM. pp 888–888.

Hull, Elizabeth, Jackson, Ken, Dick, Jeremy and (e-book collection), SpringerLink (2011). *Requirements engineering*. Springer. London.

ISO/IEC (2001). Software engineering – product quality – part 1: Quality model. ISO/IEC 9126-1:2001. International Organization for Standardization. Geneva, Switzerland.

ISO/IEC (2011). Systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models. ISO/IEC 25010:2011. International Organization for Standardization. Geneva, Switzerland.

Lee, Chien-Chang and Friedman, Jon (2013). Requirements modeling and automated requirements-based test generation. *SAE International Journal of Aerospace*, **6**(2), 607–615.

Li, Meng and Kumar, R. (2012). Model-based automatic test generation for simulink/stateflow using extended finite automaton. In: *Automation Science and Engineering (CASE), 2012 IEEE International Conference on*. pp 857–862.

Micskei, Z. and Majzik, I. (2006). Model-based automatic test generation for event-driven embedded systems using model checkers. In: *Dependability of Computer Systems, 2006. DepCos-RELCOMEX '06. International Conference on*. pp 191–198.

Mohalik, Swarup, Gadkari, Ambar A., Yeolekar, Anand, Shashidhar, K.C. and Ramesh, S. (2014). Automatic test case generation from simulink/stateflow models using model checking. *Software Testing, Verification and Reliability*, **24**(2), 155–180.

Pretschner, A., Prenninger, W., Wagner, S., Khnel, C., Baumgartner, M., Sostawa, B., Zlch, R. and Stauner, T. (2005). One evaluation of model-based testing and its automation. In: *Proceedings of the 27th international conference on software engineering*. ACM. pp 392–401.

Stecklein, Jonette M., Dabney, Jim, Dick, Brandon, Haskins, Bill, Lovell, Randy and Moroney, Gregory (2004). Error cost escalation through the project life cycle. Technical report.

The MathWorks Inc (2013). *Simulink Design Verifier 2 User's Guide*.

The MathWorks Inc (2014). Model-based testing. `http://www.mathworks.se/discovery/model-based-testing.html` [Accessed: 2014-01-20].

The MathWworks Inc (2013). *Simulink Verification and Validation User's Guide*.

Utting, Mark, Legeard, Bruno and (e-book collection), ScienceDirect (2007). *Practical model-based testing: a tools approach*. Morgan Kaufmann Publishers. Amsterdam.

Volvo Car Group (2013). Volvo car group initiates world unique swedish pilot project with self-driving cars on public roads. Press Release. `https://www.media.volvocars.com/global/en-gb/download/136182/pdf/pdf`.

Wahler, M., Ferranti, E., Steiger, R., Jain, R. and Nagy, K. (2012). Cast: Automating software tests for embedded systems. In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. pp 457–466.

Weber, M. and Weisbrod, J. (2003). Requirements engineering in automotive development: experiences and challenges. *Software, IEEE*, **20**(1), 16–24.

Zander, Justyna, Schieferdecker, Ina, Mosterman, Pieter J. and (e-book collection), CRCnetBASE (2011). *Model-based testing for embedded systems*. CRC Press. Boca Raton, Fla.

Zander-Nowicka, J (2009). *Model-based Testing of Real-Time Embedded Systems in the Automotive Domain*. Fraunhofer IRB Verlag. Berlin.

Appendix A

Modeling guidelines

This appendix presents some guidelines regarding modeling when using Simulink® Design Verifier™. Where statements about the impact on computing time are made, the corresponding actions are not thoroughly investigated and might not be beneficial for other cases than the one presented in this thesis.

MATLAB 2012b with the following toolboxes were used:

- Simulink® version 8.0
- Stateflow® version 8.0
- Simulink® Design Verifier™ version 2.3
- Simulink® Verification and Validation™ version 3.4

General tips

When using extenders and detectors, be sure to provide external reset to avoid output being held true even when conditions are not met.

The detector in the SLDV library supplies the number of time steps from the mask to the underlying MATLAB® function as an `int8`. This significantly reduces the possible number of time steps that are detectable. Since the underlying MATLAB® function instead uses an `int16` this can be fixed my making a copy if the block and changing the variable type in the mask. This problem was reported to MathWorks and might be corrected in later versions of SLDV.

A property stating that an integer signal shall be zero, which is a common value for *Off*, or that a boolean signal shall be false for some condition, can lead to a property being proven valid if this signal is not connected. For example, the properties shown in Figure A.1 can be proven satisfied if the two output signals are not connected.

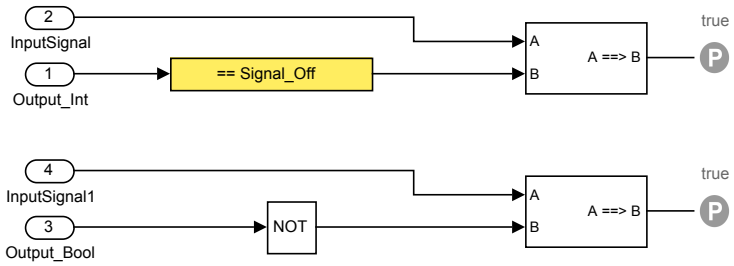


Figure A.1. Example properties which can be proven satisfied if the signals *Output_Int* or *Output_Bool* are not connected

Computing time

When computing times are long and objectives still are undecided, try the method presented in:
`http://www.mathworks.se/help/sldv/ug/prove-properties-in-large-models.html`

Use integers instead of doubles as far as possible to significantly reduce computing time for property proving.

Avoid using the clock block to reduce computing time.

Removing dead logic before property proving can reduce computing time.

Increasing the sampling time based on the lowest used timing parameter (e.g. used in a detector block) can reduce computing time significantly. The sampling time should be set as long as the shortest timing parameter but short enough so that no extender or detector block gets an output duration of zero.

Decoupling an input signal where only specific values shall be used in a Stateflow® chart can reduce computing time. E.g. if the chart only uses *VehicleModeRunning*, it can be more efficient to check the value of VehicleMode outside the chart and only have a boolean input, as exemplified in Figure A.2.

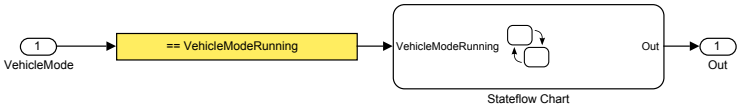


Figure A.2. Creating a boolean input to a Stateflow chart instead of making the check inside of the chart. This can reduce computing time when performing property proving

Using a MATLAB® chart instead of the standard Stateflow® chart often leads to significant increases in computing time performing property proving.

Appendix B

Systematic test case generation

The following type of structure was used when generating systematic test cases:

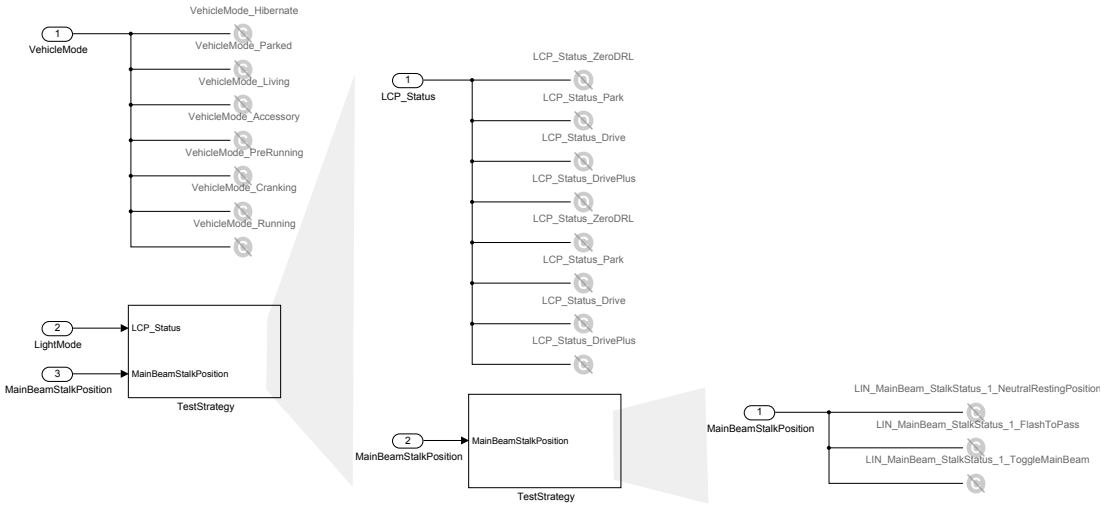


Figure B.1. Test strategy for generation of a systematic test case.

The following MATLAB® code was created to generate systematic test cases.

AutoHarness_Strategy.m

```
function [status, fileList] = ...
    AutoHarness_Strategy( path , init, separateTestCases)

%Inputs: Path to a TestStrategy subsystem

%Files
filename = [bdroot(gcb) 'AutoHarness.slx'];
filename_temp = [bdroot(gcb) 'AutoHarness_temp.slx'];
file = [pwd '\' filename];

%SLDV Options
```

```
opts = sldvoptions;
opts.Mode = 'TestGeneration';
opts.ModelCoverageObjectives = 'mcdc';
%opts.ModelCoverageObjectives = 'ConditionDecision';
opts.TestSuiteOptimization = 'CombinedObjectives';
opts.OutputDir = ['autoharness' bdroot(gcb)];
if(separateTestCases)
    opts.SaveHarnessModel = 'on';
    opts.SaveDataFile = 'off';
else
    opts.SaveHarnessModel = 'off';
    opts.SaveDataFile = 'on';
    opts.SaveExpectedOutput = 'on';
    longTestFile = [pwd '\' 'autoharness' bdroot(gcb) '\'...
        bdroot(gcb) '_LongTestCase.mat'];
    %Used when creating harness from sldvData
    harnessopts = sldvharnessopts;
    harnessopts.harnessFilePath = file;
    harnessopts.modelRefHarness = false;
end

opts.DisplayReport  = 'off';
opts.HarnessModelFileName = filename_temp;
%get(opts)

fileList={};

constraintBlockName = 'req';
strategyBlockName = 'TestStrategy';

%Remove the final Harness file if this is the first function call
if(init)
    if exist(file, 'file')
      delete(file);
    end
end

SysList = find_system(path,'SearchDepth',1,'BlockType','SubSystem');

%Find systems at current level (not at higher levels)
SysListCurrentDepth = strrep(SysList,path,'');
constraintIndex = ~cellfun(@isempty,strfind...
    (SysListCurrentDepth,constraintBlockName));
constraintPath = SysList(constraintIndex);
constraintNum = length(constraintPath);

%Deactivate all constraints (precaution)
for i = 1:constraintNum
    set_param(constraintPath{i},'enabled','off')
end

% activate  one by one
for i = 1:constraintNum

    set_param(constraintPath{i},'enabled','on')
```

```
    %Find next level of TestStrategy subsystem(s) at current level
    strategyIndex = ~cellfun(@isempty,...
        strfind(SysListCurrentDepth,strategyBlockName));
    strategyPath = SysList(strategyIndex);

    %Loop through all strategy subsystems, if length(strategyPath) is zero,
    %we are at the bottom level of the branch

    if(isempty(strategyPath))
        %Generate test cases
        [status,filenames] = sldvrun(bdroot(gcb), opts, true);

        %Check that a test case was generated
        %(since status, is not zero for contradictory model)
        %load(filenames.DataFile)
        %if (status) %If constraints are not contradictory
            delete(filenames.LogFile)

            %Save file name (for merge at last return)
            if(separateTestCases)
                close_system([opts.HarnessModelFileName '/Inputs'], 0)
                close_system(filenames.HarnessModel, 1)
                fileList = [fileList ; filenames.HarnessModel];
            else
                fileList = [fileList ; filenames.DataFile];
            end
        %end

    else
        for j=1:length(strategyPath)
            %Recursive call to activate next TestStrategy subsystem
            [status, List] = AutoHarness_Strategy(strategyPath{j},...
                false,separateTestCases);
            if(~status)
                return
            end
            fileList = [fileList ; List];
        end
    end

    set_param(constraintPath{i},'enabled','off')
end

disp(['TestStrategy completed: ' path])

if(init)%If about to exit first function call

    if(separateTestCases)
        sldvmergeharness(file, fileList);

        close_system([file '/Inputs'], 0)
        close_system(file, 1)
    else
        MakeLongTestCase(fileList, longTestFile);
```

```
        sldvmakeharness(bdroot(gcb),longTestFile,harnessopts);
    end

    delete([opts.OutputDir '/' bdroot(gcb) 'AutoHarness_temp*'])
end
status = true;
end
```

MakeLongtestCase.m

```
function MakeLongTestCase(sldvDataFileList, fileName)


load(sldvDataFileList{1})

%Copy sldvData to obtain correct struct
sldvDataLong = sldvData;
sldvDataLong.TestCases = sldvDataLong.TestCases(1,1);
%Clear testcase
sldvDataLong.TestCases(1,1).timeValues = [];
sldvDataLong.TestCases(1,1).dataValues = ...
    cell(size(sldvDataLong.TestCases(1,1).dataValues));
sldvDataLong.TestCases(1,1).expectedOutput = ...
    cell(size(sldvDataLong.TestCases(1,1).expectedOutput));

Ts = sldvData.AnalysisInformation.SampleTimes;

%Combine test cases
for i=1:length(sldvDataFileList)
    load(sldvDataFileList{i})
    %Check if a testcase exists (no test case if e.g. contradictory model)
    if(isfield(sldvData,'TestCases'))
        for j=1:length(sldvData.TestCases)
            %Extend existing timeValues with new that are moved forward in time
            if(i==1 && j==1) %If this is the first tc, just copey timeValues
                sldvDataLong.TestCases(1,1).timeValues = ...
                sldvData.TestCases(1,j).timeValues;
            else
            sldvDataLong.TestCases(1,1).timeValues = ...
                [sldvDataLong.TestCases(1,1).timeValues ...
                sldvData.TestCases(1,j).timeValues + ...
                sldvDataLong.TestCases(1,1).timeValues(end)+Ts];
            end

            %Extend dataValues
            for k=1:length(sldvData.TestCases(1,j).dataValues)
            sldvDataLong.TestCases(1,1).dataValues{k} = ...
                [sldvDataLong.TestCases(1,1).dataValues{k} ...
                sldvData.TestCases(1,j).dataValues{k}];
            end

            %Extend expectedOutput
            for k=1:length(sldvData.TestCases(1,j).expectedOutput)
            sldvDataLong.TestCases(1,1).expectedOutput{k} = ...
```

```
                [sldvDataLong.TestCases(1,1).expectedOutput{k} ...
                sldvData.TestCases(1,j).expectedOutput{k}];
            end
        end
    end


end
save(fileName,'sldvDataLong')

end
```