CHALMERS
UNIVERSITY OF TECHNOLOGY

Formal Verification of Charging Control Software Using Simulink Design Verifier

Master's thesis in Systems, Control & Mechatronics

AXEL ANDERSSON

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020:NN

Formal Verification of Charging Control Software Using Simulink Design Verifier

AXEL ANDERSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2020

Formal Verification of Charging Control Software Using Simulink Design Verifier
AXEL ANDERSSON

Supervisor: Fredrik Hansson, Volvo Car Corporation
Supervisor: Johan Lidén Eddeland, Industrial PhD student, Volvo Car Corporation
Examiner: Knut Åkesson, Department of Electrical Engineering

Master's Thesis 2020:NN
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

Formal Verification of Charging Control Software Using Simulink Design Verifier
AXEL ANDERSSON
Department of Electrical Engineering
Chalmers University of Technology

Abstract

The software complexity in the automotive industry has grown quickly in the last years. Formal verification can be used to verify requirements in such complex systems, but the size of the software also causes issues for verification tools. This thesis carries out a case study in formal verification of the charging control software, modeled in Simulink, at Volvo Car Corporation using Simulink Design Verifier (SLDV). The intent of the case study is to investigate how test environments and requirements should be modeled to allow SLDV to easily verify or falsify the software, as well as how requirement models can be altered to isolate potential software errors, to help the developers of the software.

The purpose of the software is to handle communication between the vehicle and charging station according to certain charging standards, compute charging currents and voltages, and detect faults during the charging process. Although some states and signals in the software are continuous, the majority are discrete, and the main program flow is controlled by Stateflow charts in the software.

The case study presents general actions taken in verifying or falsifying requirements; these are done to allow SLDV to reach a conclusion faster. The actions involve isolating parts of the software to build a smaller model, remodeling temporal logic using SLDV temporal operator blocks, lowering the lengths of the temporal logic, or replacing temporal transitions in Stateflow charts with input triggered transitions. The case study also presents methods for isolating errors, in order to help developers to correct faults in the software. The case study presents these results through a series of requirements for the charging control software at Volvo Cars.

In general, the actions proposed above yield significantly higher performance of SLDV in the case study. Methods for isolating software components in order to minimize the test environment are presented, as well as examples for how requirements can be divided into sub-requirements; allowing SLDV to prove each sub-requirement individually can be very efficient. The case study suggests that up to a certain level, SLDV can be used for formal verification of requirements for the charging control software using the proposed methods.

Keywords: Simulink, Simulink Design Verifier, Formal Verification, Falsification, Discrete, Model-based, Software.

Acknowledgements

I would like to thank my supervisors, Fredrik Hansson and Johan Lidén Eddeland, for their support and guidance in both practical software development and testing, as well as the theory and academic aspects of the subject. Moreover, I would like to thank Volvo Cars for having me and for the welcoming environment, and a special thank you to my manager Sajed Miremadi for giving me the opportunity to do this thesis project at Volvo Cars.

I would also like to thank my examiner Knut Åkesson for his support with strategies for the thesis project, as well as his guidance in writing this report.

Finally, I would like to thank Fredrik Håbring, Chris Setiadi and Timian Goldbeck-Wood from MathWorks, for their guidance and support in Matlab, Simulink and Simulink Design Verifier throughout the entire thesis project.

Axel Andersson, Gothenburg, July 2020

Contents

List of Figures

List of Tables

Abbreviations

CHAdeMO	–	CHArge de MOve
EV	–	Electric Vehicle
EVSE	–	Electric Vehicle Supply Equipment
MiL	–	Model-in-the-Loop
SiL	–	Software-in-the-Loop
HiL	–	Hardware-in-the-Loop
SLDV	–	Simulink Design Verifier
ECU	–	Electronic Control Unit
SWC	–	Software Component
CI	–	Continuous Integration

1

Introduction

With the growing amount of software in automotive vehicles, the use of formal verification becomes a bigger challenge. Formal verification allows proving models fulfill requirements, and can automate the process of verifying requirements. This thesis investigates the possibilities of adding formal verification tools to the development process in the automotive industry, and will through a case study present a guidance for how this can be done in a more general case.

1.1 Background

The amount of software in vehicles has exploded in the last decades. Software sizes in cars from Volvo Car Corporation (Volvo Cars) increased from 10.9 MB in 2006 to over 900 MB in 2014 [1]. The growing complexity and importance of software demand consistent and accurate verification. System models are often tested through simulations, using sets of inputs test engineers deem representative for the conditions the systems will be exposed to. Through these simulations, the behavior of the system can be investigated.

Simulations for a limited set of inputs is not a fool-proof method as mistakes can be made, especially when systems are highly complex. The complexity can make it difficult to come up with inputs that tests software thoroughly. Unintended behavior can be difficult to find yet very important to test. For instance, advanced driver assistant systems can be used to optimize fuel or battery usage, but it is important that the car does not counteract the driver pressing the brake pedal. When charging an electrical vehicle, it is important to know that the charging will stop if there is a risk of fire due to increasing temperatures. Unintended behavior in these scenarios can be severe safety hazards. Sophisticated methods should be used to as accurately as possible verify that the software is not subject to faulty behavior.

Formal verification refers to the subject of proving or disproving correctness in a system, with respect to a set of requirements. It is more powerful than simulations of limited sets of test inputs, but it is also more difficult to carry out. Formal verification will be explained further in section Subsection 4.4.3. In this thesis, a requirement which a system fulfills will be denoted *verified*, whereas the opposite will be denoted *falsified*.

Consider the example of a system consisting of an input signal x, a parameter θ, an `OR` gate and an output y. All signals and parameters are Boolean; Boolean algebra is further explained in Appendix A. The output is determined by

$$y = x \vee \theta \tag{1.1}$$

The equivalent Simulink model for this is shown in Figure 1.1. The requirement for this example is that the output should always be *true*. If parameter θ is *true*, the system fulfills the requirement as a $\vee$ operator returns *true* if one of the operands are *true*. However, if θ is false instead, the system can be simplified as

$$y = x \tag{1.2}$$

A simple counterexample which falsifies the requirement is $x = \mathit{false}$.

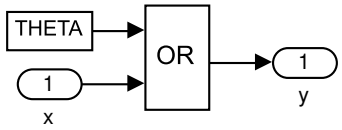


Figure 1.1: Simple `OR` example.

The previous example is very small and could easily be carried out by hand. However, this is generally infeasible in larger models which may have hundreds of inputs and outputs. To formally verify large models, tools must often be used to perform this process within a reasonable amount of time. There are different software tools for formal verification, such as the free tools Uppaal [2] and Supremica [3], or the licensed Simulink Design Verifier (SLDV) for MathWorks' Simulink software [4]. Efficient verification early in the development process can greatly reduce costs through early detection of errors in the designed system, as development costs typically increase the later in the process the error is detected [5]. Moreover, it can also mathematically prove that the software design is valid and will fulfill the specified requirements, or find a counterexample where the requirement is not fulfilled.

The charging control software for electrical vehicles at Volvo Car Corporation is currently tested using a limited set of test cases. The software is modeled in Simulink, where part of the testing is performed. The size and complexity of the software Simulink model prevent test engineers from designing test cases, to verify that all possible computations and state combinations fulfill the requirements. The addition of formal verification into the development pipeline can greatly improve the quality of the software as it allows proving correctness, or finding counterexamples showing how to falsify the requirements. Since the charging control software is already modeled in Simulink, SLDV is a clear candidate for carrying out the formal verification.

1.2 Purpose

In this thesis, SLDV is used to formally verify a subset of requirements for the charging control software at Volvo Car Corporation. The case study of the requirements and software is used to draw conclusions regarding how requirements should

be modeled, how test environments with the requirements and parts of the software should be set up, and how formal verification using SLDV can incorporated in a development pipeline in the industry. The case study is qualitative and focuses on steps taken to allow SLDV to come to conclusions for the requirements studied. The case study mainly targets the addition of SLDV into the testing process and how to resolve some of the potential issues encountered in this addition.

1.3 Scope and Limitations

The thesis considers how large models in Simulink can be formally verified using SLDV. This is done in a case study using Simulink models of the charging control software developed at Volvo Cars. The scope includes verification of the charging control software; no other examples of software will be formally verified. The general case of formal verification using SLDV is discussed with the case study as a baseline. The charging control software is mostly discrete, meaning a clear majority of the signals and states in the software have a discrete domain such as Boolean or enumeration signals, although some continuous signals exist too. The requirements studied are mainly discrete; only one requirement which deals with continuous signals is partially investigated. The charging control software is described more in detail in Chapter 3.

There are studies that show that, *e.g.* UPPAAL provides higher performance than SLDV [6]. However, the size of the charging software makes it infeasible to remodel the software in such tools for the scope of this thesis. Hence, the thesis is limited to the usage of SLDV.

The charging control software has a large number of requirements. A subset of the requirements are investigated and included in this thesis. The requirements are selected to represent a variety of requirement types which are typically present at Volvo Cars. These requirements are investigated qualitatively; there is no study on the performance of SLDV with respect to various metrics or requirement types. Instead, detailed analyses and processes for isolating errors and optimizing for SLDV are described. Due to time constraints, only requirements for the CHAdeMO charging standard are investigated; the charging standards are presented in Section 3.1.

The thesis only includes Simulink models of software. No source code or compiled application will be looked at; the thesis is limited to the Simulink models in the early stages of the development pipeline. Due to confidentiality, detailed examples from the charging control software are not be presented in the thesis. Typically, reduced models with similar or equivalent structures are presented to assist the explanations in the case study. Signal and state names are removed and replaced by arbitrary names instead.

Prerequisite knowledge of Simulink is assumed in this thesis, and no introduction will be given for it. However, SLDV, which is a toolbox for the Simulink software, is explained in Chapter 5 and in Appendix C.

1.4 Research Questions

To assist in fulfilling the purpose stated above, the following research questions are formed. They serve as a guidance to the problem of formally verifying mostly discrete software, and are formalized to allow for conclusions regarding large Simulink software models in general, while also being applicable to the charging control software at Volvo Cars which is the case study of the thesis.

1. *How should requirements be modeled in Simulink to allow Simulink Design Verifier to come to conclusions faster?*

The ability to use SLDV to formally verify software is not black and white. Some requirements can be tough for SLDV to handle. Answers to this question can be very helpful for testers in designing requirement models which are optimized for SLDV, but can also be a guidance to which requirements are suitable for use with SLDV.

2. *How should Simulink requirement models be altered to isolate errors and show absence of other errors, with the use of Simulink Design Verifier?*

In the case of a falsification, it is possible that one would want to temporarily alter the requirement to isolate the error to try and ensure the error is limited in the software; this could be of great help for developers of the model.

3. *How should test environments be set up to allow for Simulink Design Verifier to easily formally verify or falsify the requirements, without altering the behavior of the software?*

Requirements are not everything. It is important to consider the actual software which is to be formally verified. This research question deals with the problem of creating test environments consisting of the whole or parts of the software, to allow SLDV to come to conclusions as easily as possible. Optimally, the behavior of the software should not be changed in this test environment.

2

Related Work

Currently, there are numerous articles regarding the subject of formal verification. This chapter will briefly introduce a few of them which are relevant either for being applicable and comparable to the results presented in this thesis, or for analysis on how the results in this thesis could have been improved through a different methodology. Moreover, it aims at explaining the current state-of-the-art of formal verification to the reader.

Studies of how SLDV can be used in the automotive industry have already been carried out. A master's thesis from 2014 [7] presents a case study in how SLDV can be used for model-based testing. The case study consists of an Automated Headlight Switching system at Volvo Trucks. The thesis mainly focuses on how to model different types of software requirements, and when these requirement models should be used. The thesis considers both formal verification and test case generation, and presents modeling examples and suggestions for different types of requirements.

Another master's thesis from 2015 [8] investigates how to incorporate formal verification methods into the model-based development at Scania, in contrast to the simulation-based testing carried out at the time of writing the thesis. The thesis presents suggestions for how to model various types of behavior in SLDV, and contains many examples for the case study. The thesis is recommended for readers interested in complementary examples and discussions.

It can also be useful not only to test models of software but actually software itself. In [9], the verification of C software using Frama-C is discussed. Frama-C is a software analysis platform [10], which can be used to formally verify specifications of C-code. The report discusses the use of Frama-C in a case study, and concludes that it can be used but is only recommended for highly safety critical requirements and functionality. The report is suggested for readers interested in applied formal verification of software (rather than models of software).

3

Charging control software

The charging control software in Electric Vehicles (EVs) from Volvo Cars has many purposes. It handles communication with the Electric Vehicle Supply Equipment (EVSE, charging station), understanding and following protocols for different charging standards, computing charging currents, and detecting faults are some of the tasks of the software. It is mostly discrete, meaning a clear majority of the states and signals have a discrete domain, such as Boolean or enumeration signals. There are also continuous signals, represented by integers, single-precision floating-point or fixed-point values; although integers are not strictly continuous, they can still be used for computations which is not typically the case for Booleans and enums. The charging control software will be explained more in detail in this chapter.

3.1 Charging Standards

Much like regular electrical power sockets in houses, charging connectors for electrical vehicles also have different standards in various areas of the world. The charging standards have different connectors and protocols. The charging control software must be able to handle all of the different standards and protocols.

Different parts of the world typically use different charging standards, although some overlap exists. Different charging standards also have different connectors. The charging standard with largest global coverage is CHAdeMO (CHArge de MOve) [11], originally created by a group of Japanese companies. China has their own charging standard called GB/T which is mandatory in new EVs in China, but is also limited to China. There are two types of CCS (Combined Charging System) standards in use: type 1 which is used in the U.S., and type 2 which is adopted in Australia and Europe (these two types have different connectors). Additionally, the car manufacturer Tesla has its own charging system. The different systems have various specifications, for instance for supported voltages and currents. The charging control software must be able to support these standards and detect the type of EVSE the EV is connected to (with the exception of the Tesla standard). Each charging standard defines the procedure for communication and charging between EV and EVSE. The charging control software must handle the communication with the EVSE, detect signals from the EVSE and answer according to the standard. Much of this behavior is discrete; continuous variables are used to determine charging

speeds through voltages and currents, but the signals which determine in which part of the charging process (*e.g.* startup, charging or termination) are typically discrete.

For this thesis, only requirements for the CHAdeMO standard will be considered. There are requirements for other standards as well, and for parts of the software related to all or none of the supported standards, but the requirements dealt with in the case study in Chapter 6 are all for the CHAdeMO standard; this is due to time constraints for the thesis project.

3.2 Charging control software at Volvo Cars

The purpose of charging control software can be summarized as handling communication with the EVSE according to its charging standard, following the charging standard protocol, computing charging currents and voltages, and handling faults which can occur during charging. Although part of the software signals and states are continuous, the majority of the software is discrete. For instance, the charging procedures specified by the charging standards are modeled using Stateflow charts which are inherently discrete.

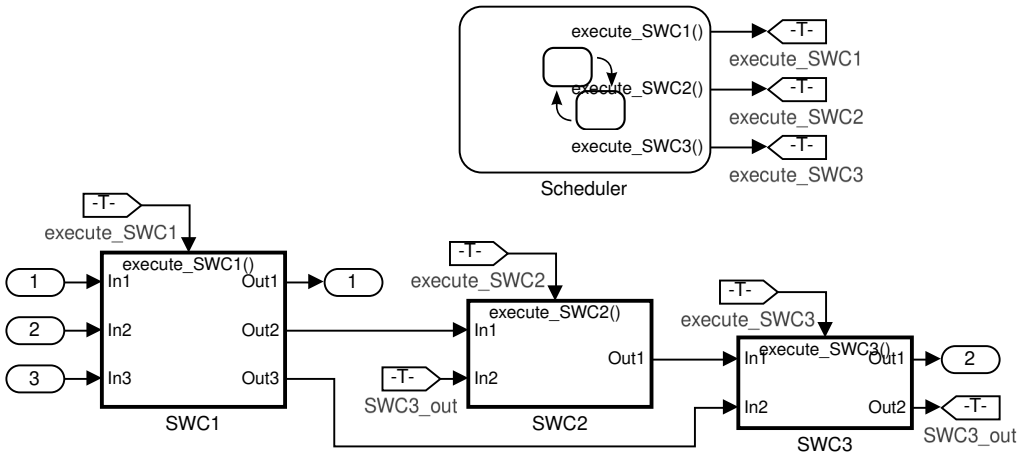
3.2.1 Software model structure

The software model is split into multiple *Software Components* (SWCs). Some SWCs are used for specific charging standards, whereas others handle logic unrelated to charging standards or multiple charging standards simultaneously. The time-step length for the charging control software Simulink model is 0.01 seconds, *i.e.* it is updated 100 times per second. In each time-step, the SWCs are executed in a specific order. This allows the developers to control the program flow in each time-step, and also how the software should handle closed loops within the software. The charging control software structure on a top-level is visualized in Figure 3.1.

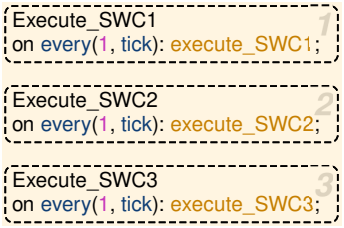
3.2.2 Model-based testing at Volvo Cars

Currently, the model-based testing carried out at Volvo Cars is closed-loop and simulation based. This means that a plant model of an EVSE is connected to the software, and sets of test inputs are used to trigger certain behavior and types of tests in the closed-loop test environment. Models of the requirements are used to determine whether the requirements are fulfilled or not. Test environments are created for the different supported charging standards, and the requirements for that specific charging standard are used in the observers. Tests are also performed on the generated software (as opposed to the Simulink software models dealt with in this thesis) in Software-in-the-Loop (SiL) test environments, and running on actual hardware in Hardware-in-the-Loop (HiL). Open-loop and closed-loop testing, including Model-in-the-Loop (MiL), SiL and HiL, is explained more in-depth in Section 4.2.

In the development process, some tests are run automatically. When a change is made and sent to the central version control system at Volvo Cars, tests are



(a) Software root, containing all SWCs and the Scheduler chart.



(b) Inside of the Scheduler Stateflow chart, set to parallel decomposition. The states are executed in the order as determined by the numbers to the right in each state.

Figure 3.1: Structure of the charging control software, with the SWC interconnections and the Scheduler Stateflow chart used to control the execution order of the SWCs.

automatically run to find errors. If no errors are found, the change is accepted and is submitted. Otherwise, the change is blocked and developers will need to resolve the issues in the software model. This allows for safe version control, but requires reliable tests.

3.2.3 Problems in the testing process

Although there are charging standards, the charging standards do not define all the behavior in EVSEs. As a result, there are many different implementations of the charging standards. Thorough testing with EVSEs would require a multitude of plant models to be developed and tested for each charging standard. It is unlikely that all different implementations of the charging standards in EVSEs can be modeled and tested by Volvo Cars. Moreover, faulty behavior in an EVSE can be difficult to test in these test environments.

Currently, the software is simulated using a predefined set of test inputs. The predefined set of test inputs are created manually by software testers. These can be useful to find simple errors in the software, but testing only a small fraction of the entire input space will not prove correctness, it can only disprove it. Therefore, even if the software model passes all tests, there could still be faulty behavior for

other inputs. Additionally, the same test inputs on an EVSE with a different implementation of the charging standard than tested at Volvo Cars could produce an error in the observer models. To summarize, the uncertainty in the EVSEs and the inherent inability to prove correctness with a few sets of test inputs are issues in the model-based testing of the charging control software at Volvo Cars, which could be solved with requirement proving methods.

4

Testing

There are many different methods used to test software. The purpose of testing is done to try and prove or verify correctness of a system, to ensure the requirements specified for the system are fulfilled. Software correctness means the software is implemented such that it fulfills the requirement for the software. Testing can be done at many different stages in the development process, but should be carried out often as the costs for fixing errors increase the further into the development process they are discovered [5]. Testing can help developers find simple errors such as data type mismatches or missing minus signs in software, or very complex behavior which appears for very specific unintended inputs but lead to dire consequences for the system. Testing can be done in different orders of magnitude, on different levels of the system and in different test environments. This chapter will explain some widely used testing methods.

4.1 Requirement and testing levels

Requirements can be specified on different levels, meaning the requirements can be applied to very small parts of the system or the entire system. Testing methods can differ a lot between different levels. For a small subsystem, coming up with appropriate tests can be easy. However, higher level requirements deal with larger and more complex systems and are therefore usually harder to test, as the input range and state space is increased.

Testing small units of a system is called *unit testing* [12]. Typically, unit level requirements specify the desired functionality for functions in source code, or individual subsystem blocks in Simulink. With unit testing, the very basics of the software can be partially or fully verified. For instance, custom library blocks can be unit tested so that their behavior is correct in the entire software. Since these units are often re-used, verification of these units can be very important. Additionally, the small size and (usually) low complexity of the units suggests the units are easy to test or prove to fulfill requirements. The unit tests are typically testing the implementation through small pieces of the system, but rarely touch on the general behavior of the system as a whole.

Component-level requirements (also called integration requirements) [13] are specified to define the behavior of the separate software components. For the charging

control software, these are the individual SWCs as shown in Figure 3.1. Testing these are more difficult than unit level testing as the input range and state space is usually much bigger than for unit testing. Component-level requirements define parts of the functionality of the system, rather than very small pieces of the implementation. The component separation is often partially implementation based, and the requirements for the entire system do not necessarily mention the specific components. However, component-level requirements can still be used to test part of the behavior which is expected of the system as a whole. There are still limitations to tests on these levels, as they must look at single components, but testing some of the system behavior is possible on this level.

System-level requirements [14] define the behavior of a system from input to output. Optimally, they should not be implementation specific. The purpose of system-level tests is to define the overall functionality of the system. Tests for requirements on this level are very important to conduct, as these tests usually define the entirety of the system, including how the components should interact with each other. However, testing on this level is difficult as the full input range and state space is included, and the complexity of the system can require a large multitude of tests to be conducted in order to get some confidence in the correctness of the system.

Testing on different levels have different benefits and shortcomings. As such, different methods can be used on different levels. The best approach depends on the application: the details of the system, and what fits best into the development process of the system as well.

4.2 Open and Closed Loop testing

There is an important distinction between so-called *open-loop* and *closed-loop* systems. In many cases, especially in automatic control, a system can be split into two parts: the controller and the plant. The plant is the actual system one is trying to control. The controller on the other hand is the system which attempts to control the plant, achieving desired behavior in the plant. In this master's thesis, what constitutes the plant and controller is slightly less clear. The plant in this case is the charging station the vehicle is connected to, as well as the other ECUs (Electronic Control Units) in the vehicle, whereas the controller is the charging control software.

A closed-loop system with an observer can be seen in Figure 4.1. The software and plant work independently from the observer; the purpose of the observer is to determine that the software's behavior and input to the plant fulfills a set of requirements. For instance, a requirement could state that the software should stop charging and enter a specific state whenever the plant outputs an error. If modeled correctly, the observer can detect whether the requirement is fulfilled at a specific time.

The plant has a defined behavior. The signals from the plant into the software depend in most cases on the signals from the software to the plant. This means that for a given output from the software into the plant, the plant can only supply

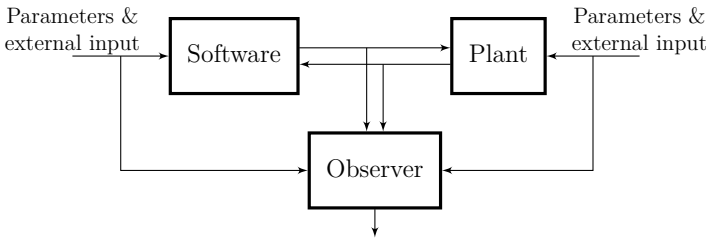


Figure 4.1: Model of a closed-loop system with an observer.

the software with a subset of its input space. An example of such a plant can be seen in Figure 4.2, where the plant outputs the input twice; in linear state space representation, this can be written as

$$\boldsymbol{y}(t) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \boldsymbol{u}(t) = \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} \tag{4.1}$$

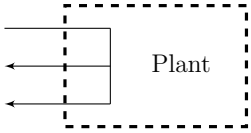


Figure 4.2: Example of plant which only outputs a subset of the range of possible inputs to the software.

Assume now that there is a requirement that a signal from the software to the observer must always be false, where the signal is

$$s(t) = y_1(t) \oplus y_2(t) \tag{4.2}$$

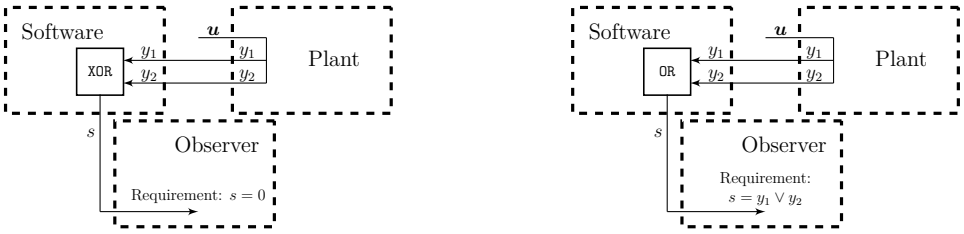
where $\oplus$ is the `XOR` operator. For the given plant in the closed-loop system, $y_1 = y_2$ meaning the requirement will always be fulfilled. However, a counterexample of $y_1 = 1, y_2 = 0$ can be used in the open-loop system, to show that the requirement is not fulfilled in the open-loop case.

Assume instead that the signal is

$$s(t) = y_1(t) \vee y_2(t) \tag{4.3}$$

and the requirement specifies that the signal should only be false whenever both inputs are 0, and true in all other cases. Trivially, the requirement is fulfilled for all Boolean y_1 and y_2, *i.e.* the requirement is fulfilled in the open-loop case. Hence, the subset of inputs possible in the closed-loop case is also fulfilled, which implies that verifying the requirement in an open-loop setting also verifies it in a closed-loop setting. These two examples are visualized in Figure 4.3.

In the general case, verifying a requirement in an open-loop setting is *sufficient* to verify its corresponding closed-loop system. Contrarily, falsifying a requirement in the open-loop system is *not necessary* to falsify it in the closed-loop setting. Summarized, verifying a requirement in the open loop is sufficient to show that it if fulfilled



(a) Requirement example *a*.

(b) Requirement example *b*.

Figure 4.3: Requirement examples.

in the closed loop, but an open-loop falsification requires further investigation in the closed-loop setting.

In this thesis, the system structure depicted in Figure 4.4 will be used. The observer contain requirements and checks whether the requirements are fulfilled or not in the open-loop. The inputs from the plant to the software in the closed-loop structure will now instead be supplied through inputs to the system.

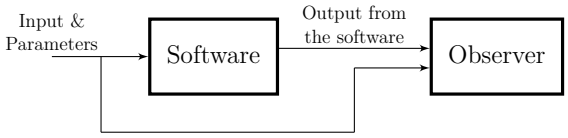


Figure 4.4: Open-loop structure used in this thesis.

4.2.1 MiL, SiL and HiL

In closed-loop testing, there are different stages in the development process. Model-in-the-Loop (MiL), Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL) [15] are common terms used in the automotive industry for testing in various stages. MiL testing involves simulations of models of the software and the plant. In the case of the charging control software, MiL testing involves the Simulink models of the software and plant models for EVSEs. The verification done in this thesis is closely related to MiL, although the "loop" is missing; the open-loop setting makes the plant model obsolete as only inputs and outputs to the software are observed by the requirement observers. MiL testing allows developers to test their systems early in the development process on a plant model, which should at the very least closely resemble the plant in reality.

SiL testing (Software-in-the-Loop) is similar to MiL testing, as the plant is simulated. However, in this stage the software is generated and compiled from the model. This stage allows developers to test the correctness of the software compared to the model. This can be seen as a preparatory step for Hardware-in-the-Loop (HiL) testing. In HiL testing, the simulations of actuator and sensor components are replaced by actual hardware for these components. This testing stage is performed to see how the software works in a real environment without simulated hardware. Note that the implementation of the plant in HiL testing can still be simulated, but this is done on the "other end" of the hardware. By going through the MiL, SiL and HiL

testing steps, the software can be gradually tested and verified, and the systematic approach can help developers isolate and debug design errors.

4.3 Continuous integration

In the development process of large software, it is common to use *Continuous Integration* (CI) [16]. For large software projects with many developers working on it, the risk of merging conflicts is big if commits and merges are not carried out frequently. Continuous integration is the practice in which the software is developed and submitted to the shared code base several times a day. This is followed by an automated process to check for conflicts and errors in the software. The testing is performed to ensure no errors are merged into the other developers' local copies of the code base. The automated testing in the CI process must therefore be robust and be simple enough to be executed in the expected frequencies; if the tests are too computationally heavy, the version control server will lag behind and queued tests and merges will pile up, causing the master branch to fall behind. Thus, the developers will end up working on an ageing code base, at which point the CI process is nullified. Hence, the testing method used for the CI process must be designed to be fast enough for the version control servers to handle.

4.4 Testing methods

There are many different testing methods. They have different benefits and shortcomings, and selecting the best methods for an application can be difficult. This section will present a few different testing methods and their usage in different situations.

4.4.1 Manual and automatic test case generation

Systems can be tested against their requirements using sets of test cases. Test cases in this thesis means sets of inputs for which the system is simulated to obtain the system outputs. If the outputs for the simulated system do not fulfill a requirement, then the test case is said to falsify the requirement.

A drawback of using test cases is that unless a huge amount of test cases are used which span the entire input and state space of the system, the requirements cannot be known to be fulfilled. For instance, a system with 20 independent Boolean inputs have over a million input combinations in each time-step. When the input space is sampled by simulating a limited amount of test cases, only a certain confidence about correct functionality of the system can be gained. However, a falsifying test case will always disprove correctness in the system.

A clear benefit of using test cases is that in general, no knowledge of the internal parts of the system is needed. The test cases can be generated and then simulated or run on hardware, and the outputs can be observed. Thus, testing with the use of

test cases can be simple to carry out, to gain some confidence in the correctness of the system.

Test cases can either be generated automatically or manually. Random testing, which is an example of automatic test case generation, is presented in the next subsection. A benefit of automatic test case generation is efficiency [17], as tools for this purpose can generate tests faster than a test engineer or developer. Additionally, algorithms for test case generation can produce test cases which span a wide variety of inputs. As such, much of the logic and dynamics in the system can be tested for these test cases. A test engineer is more likely to miss certain behavior, especially in large systems when there are many details to consider.

However, automatic test cases are not necessarily the best approach. While the test cases can span much of the input range and state space, it might not fit the typical inputs expected from a plant system. Of the entire input range, a very small part can be expected to occur in practice. A test engineer generating the test cases manually is likely to have the knowledge to produce a variety of realistic test cases. Since testing through test cases only samples the correctness of the system, importance sampling near expected inputs can be very beneficial. Without specific rules for the automatic test case generation, this can be difficult to achieve.

4.4.2 Random testing

One approach for automatic test case generation is *random testing*. In random testing, random independent inputs are generated. The system is then tested using these random inputs, to try and find a falsifying test case, *i.e.* a counterexample to the requirement(s).

To exemplify the usage of random testing, a small software example is used. The C code for an absolute value function, *i.e.* $y = |x|$, is shown below. The function contains an error, where x is returned instead of $|x|$.

```
int abs(int x)
{
  if (x < 0)
    return x; // Should be -x.
  else
    return x;
}
```

The function can be random tested against a requirement stating that the output of the function is to be non-negative. An example of such source code can be seen below.

```
void testAbs()
{
  int i;
  for (i = 0; i < NUM_TEST_CASES; i++)
  {
    int x = generateRandomInput();
    assert(abs(x) >= 0);
```

```
  }
}
```

Since all negative inputs to the system will fail this test, about 50% of all possible inputs should fail this test. Thus, if the number of test cases is large enough, the error will be detected consistently. The source code for a correct implementation of the software can be seen below.

```
int abs(int x)
{
  if (x < 0)
    return -x;
  else
    return x;
}
```

Now, a drawback with random testing can be seen. The test that $x \geq 0$ will pass for every input, except one. 32-bit integers using two's complement can represent values from -2^{31} through $2^{31}-1$. Here, an interesting question arises: what happens when $x = -2^{31}$? Typically, two's complement integers are negated according to

$$-x = \neg x + 1 \tag{4.4}$$

where $\neg$ is used here to denote a bitwise `NOT` operation. This works for all values of x, except for the most negative value that can be represented by the integer. In this specific case, the negation returns the value itself, *i.e.* $-x = x$. Thus, there is only one case in 2^{32} where the test $x \geq 0$ will fail. If the inputs are sampled uniformly, it is very unlikely that this issue will be detected. Note that the issue is created by the representation of integers and not the function itself, but it is nonetheless important for the developer to know of this edge case.

To detect these types of issues, it is possible to sample inputs non-uniformly. Inputs can instead be sampled more densely around 0 and the minimum and maximum integer values. It is also possible to test specific values which commonly create problems in software, such as 0, -1, 1, and minimum and maximum integer values. This is known as edge case testing, and can be very helpful as edge cases are usually a source of bugs in the code [18]. If the number of edge cases is small, it is possible to test these in addition to the regular random testing.

4.4.3 Formal Verification

The previous testing methods have an shared issue: they do not guarantee software correctness. Even if the requirements are fulfilled for all test cases, absence of errors are not guaranteed. Formal verification, which is the subject of using formal methods to verify or falsify that the system fulfills the requirements, solves this issue. There are different methods for this, such as Stålmarck's method [19]. With the use of formal verification, systems can be proven to conform to the requirements, or disproven, often with a counterexample. Because correctness can be proved or disproved with formal verification, it is very powerful. However, the

process of formally verifying systems with respect to requirements is usually much more computationally heavy than the use of test cases.

There are many different tools for formal verification. One of these is the Simulink Design Verifier (SLDV) [20] toolbox for the Simulink modeling software; the toolbox will be further explained in Chapter 5. It allows for formal verification of Simulink models. For software already modeled in Simulink, SLDV is a clear candidate for formal verification. While other tools may require remodeling of the software in those tools, SLDV can be used on the existing Simulink models of the software. Since Simulink is widely used in the automotive industry, there is much potential for the use of SLDV. However, the growing software sizes in the vehicle industry creates large problems for the use of formal verification, as the massive input range and state space can be too much for formal verification tools to handle, including SLDV. Hence, a study in the use of SLDV in the automotive industry and the process of creating test environments and requirement models is important.

5

Simulink Design Verifier

Simulink Design Verifier (SLDV) is a toolbox for MathWorks' Simulink software. Its purpose is to find design errors such as dead logic (*i.e.* logic which does not have any effect on the state or output of the model), but also proving that certain properties always hold or generate test cases to test the behavior of the model. SLDV uses formal methods for these purposes [21]. This chapter aims to describe the fundamentals, the work process, the basic building blocks, and the terminology used in SLDV. The details which are used in the thesis are presented here. Complementary information can be found in Appendix C.

5.1 Property Proving

One of the main features of SLDV is the ability to prove properties. This means that SLDV is able to prove that certain properties will always hold, regardless of the input to the system. The user can create objectives for certain signals, and SLDV will either prove that those signals always fulfill the objectives, or produce a counterexample which falsifies the objectives.

The property proving functionality is enabled through the use of the *Proof Objective* block in Simulink. The task of SLDV is to prove that the input to the Proof Objective block always coincides with the specified objective. The objective can either be a single value, or a set of values. The Proof Objective block is visualized in Figure 5.1. In regular modeling, it appears as a gray block with the letter 'P'. During and after the property proving process, the objectives can be highlighted in the model. Satisfied objectives have their objective values highlighted in green, falsified objectives in red and undecided objectives in orange. Proof Objectives can also be disabled, causing SLDV to ignore them. Disabled Proof Objectives are grayed out and crossed over.

(a) Regular **(b)** Satisfied **(c)** Falsified **(d)** Undecided **(e)** Disabled

Figure 5.1: SLDV Proof Objective block.

To visualize property proving, the example from Chapter 1 is re-used, where

$$y = x \vee \theta \tag{5.1}$$

where x is an input and θ is a parameter. The requirement that the output must be 1 still holds. As explained earlier, the requirement is only verified for $\theta = true$. However, if $\theta = false$ the requirement is falsified by the counterexample $x = false$. This is visualized in figure 5.2 where the Proof Objective connected to the `OR` gate with the *true* Constant block is verified by SLDV, while the other Proof Objective is falsified.

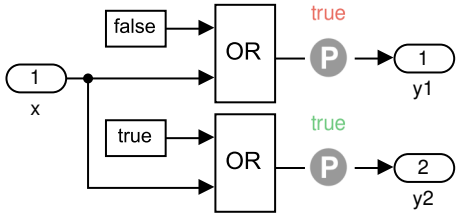


Figure 5.2: Model examples of the usage of Proof Objectives to prove properties using SLDV. The upper Proof Objective is falsified by input $x = false$, while the lower Proof Objective is verified by SLDV.

In some instance, one could want to add limitations to the behavior of the system for the sake of property proving. This is possible through *Assumption* blocks, as seen in Figure 5.3. Like the Proof Objective, the Assumption block can be enabled or disabled. The block imposes restrictions on signals and instructs SLDV to prove the properties in the model, assuming certain values on signals; note that only property proving in SLDV is affected by this block. This can either be used to replace behavior in other subsystems of the model, or to debug the falsification cause of a Proof Objective.

(a) Enabled **(b)** Disabled

Figure 5.3: SLDV Assumption block. The block lets SLDV assume the input signal is in the set $\{[1,\ 2], 3, [4,\ 5]\}$.

An example of the usage of Assumption blocks can be seen in Figure 5.4. In the left figure, the property is falsified by SLDV by the input -2. In the right figure, the Assumption block (marked with the letter 'A') forces non-negative inputs which causes the property to become *true*, hence proved valid by SLDV.

Requirements are usually much more complex than in the previous examples. Instead of simply verifying that a signal is in a specified set, there can be conditions which specify when parts of a requirement should be activated. This is easily modeled using the *Implies* block in SLDV. Its purpose is to represent the logical implies operator '$\rightarrow$'. With it, conditional requirements can be expressed as

(a) Model without Assumption block, falsified by SLDV.

(b) Model with Assumption block, proven valid by SLDV.

Figure 5.4: Assumption block usage example.

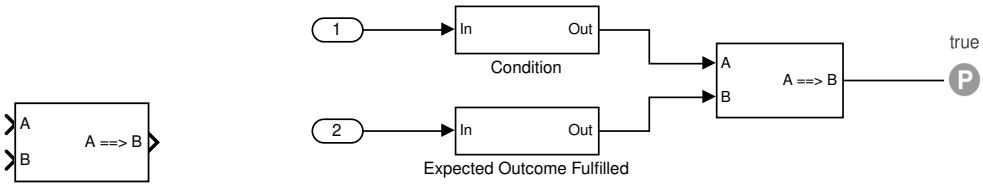
Condition → *Expected outcome fulfilled*, as visualized in Figure 5.5. More formally, the implication can be written with its operand names as *Antedecent* → *Consequent*. The Proof Objective block is combined with the Implies block to ensure the expected outcome is always fulfilled when the condition is satisfied. Its use can be exemplified with a simple addition. The plus operator ('+') can be considered a system with two inputs a and b, where the output c is

$$c = a + b \tag{5.2}$$

Assume there are three requirements for this system:

- Whenever the second input is zero, the output shall equal the first input.
- If the first input is strictly positive, the output shall be greater than the second input.
- If the first input is strictly positive, the output shall be smaller than the second input.

The first two requirements are *true* whereas the third is *false*. The requirements can be modeled as in Figure 5.6. The upper two requirements are proven valid by SLDV, whereas the third requirement is falsified. The counterexample given by SLDV is $A = 6$, $B = -1$. Note that this specific example requires certain constraints on the inputs to achieve this result; this will be explained in Section 5.3.



(a) Implies block.

(b) Usage example of the Implies block.

Figure 5.5: Example of how the *Implies* block can be used in SLDV.

5.1.1 Proving strategies

In Property Proving with SLDV, there are three Proving Strategies: *Prove*, *FindViolation* and *ProveWithViolationDetection*. With *Prove*, SLDV will attempt to prove the Proof Objectives always hold. With Proving Strategy set to *FindViolation*, SLDV will instead try to find falsifying counterexamples, limited by a maximum

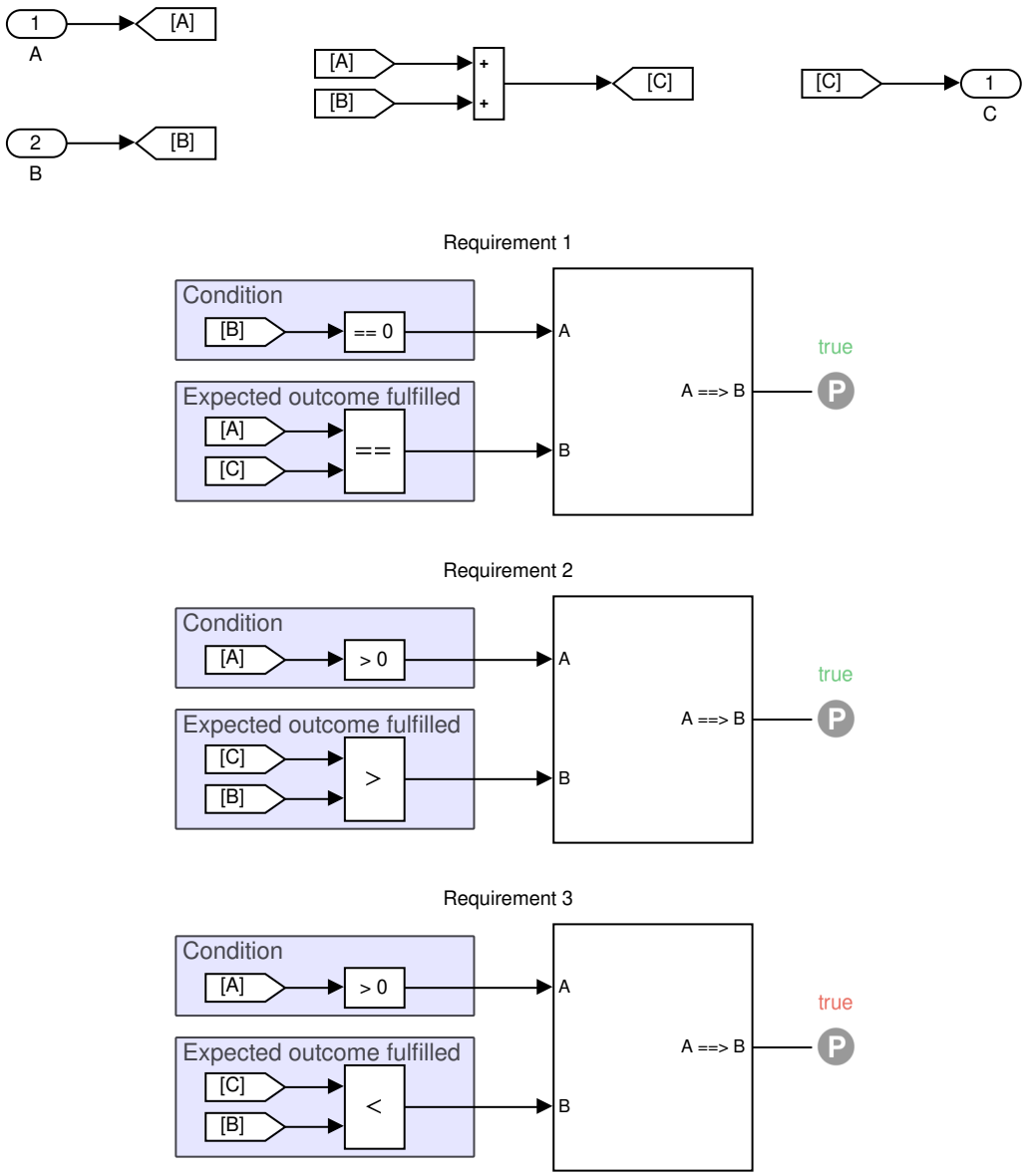


Figure 5.6: Example of how three conditional requirements can be modeled in Simulink to allow SLDV to verify or falsify them. Requirements 1 and 2 are proven valid, whereas requirement 3 is falsified.

number of violation steps. When using the *FindViolation* or *ProveWithViolationDetection* strategies, another parameter is enabled: maximum number of violation steps. This parameter allows the SLDV user to specify the maximum length of the counterexample that should be possible when running violation detection. If a counterexample is found within this bound, the Proof Objective is falsified and the counterexample is returned. If there exists no counterexamples within the specified maximum number of violation steps, the Proof Objective is *Valid within bound*.

The proving Strategy *ProveWithViolationDetection* is a combination of the other two. First, SLDV will perform a violation detection (*i.e.* same as *FindViolation*) for the specified maximum number of violation steps. If the Proof Objective is valid

within bound, then SLDV switches to the *Prove* strategy, and will attempt to prove the Proof Objective always holds.

5.2 Temporal Operators

Temporal logic refers to logic which is time dependent. Note that the systems in this master's thesis are time-invariant in the sense that in a given state, it will react identically to a set of inputs, regardless of when the system was initiated. However, there are initial states or variables read from a virtual memory in the initialization of the system. Temporal logic in this thesis generally refers to detecting Boolean signals which are *true* for a specified amount of time, without becoming *false* in this interval. There are three temporal operators in SLDV, all of which work in Boolean signals. The purpose of this section is to explain the behavior and usage of these temporal operators.

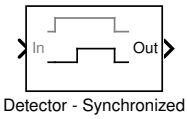
The *Detector* block detects Boolean inputs which are *true* for a certain period of time. It will output *true* whenever the input has been *true* for the last x time-steps. However, the *Output type* setting for the Detector block defines the behavior more in-depth. Possible values for this setting are:

- *Synchronized*: the output is *true* whenever the input has been *true* for the last x time-steps, including the current time-step. The *Time steps for input detection* parameter can be specified in the block parameters dialog. If set to *e.g.* 7, the output will be *true* when the input was *true* for the last 6 time-steps and is *true* in the current time-step. Note that there is no delay; the output will be *true* immediately in the 7^{th} time-step. As soon as the input is *false*, the output will be *false* as well.

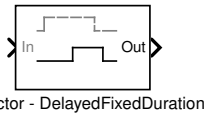
  In short: *A Synchronized Detector block with x detection steps outputs true if the input has been true for the last x consecutive time-steps, including the current time-step.*

- *Delayed Fixed Duration*: the output is *true* after detection for a fixed duration, with an optional delay. The detection is similar to *Synchronized*, where the input must be *true* for x time-steps. However, this does not count the current time-step. There is also a setting for delaying the output, so that after detection the output will not be *true* until a specified number of time-steps have passed. Finally, the output will only be *true* for a set amount of time after detection and delay. After that duration, the output will be *false* for at least one time-step.

  In short: *A Delayed Fixed Duration Detector block set to x detection steps, y delay steps, and z output duration steps will after detecting a true input for x consecutive time-steps wait for y time-steps, and then output true for z time-steps before the output becomes false again. Note that the output will always become false for at least one time-step after the z time-steps.*



(a) Synchronized



(b) Delayed fixed duration

Figure 5.7: SLDV Detector block, set to different output types.

The usage of the Detector block can be seen in Figure 5.8. A step input is supplied at $t = 3$, with step length 1 s.

- The *Synchronized* block has its *Time steps for input detection* parameter set to 3. This means that in the 3rd consecutive time-step with a *true* input, the output will be *true* until the input is *false*. Thus, 2 time-steps after the step in the simulation (*i.e.* at $t = 5$), the synchronized Detector block will output *true*. Since the input never becomes *false* again, the Detector output is constantly *true* after $t = 5$.
- The *Delayed fixed duration* block has its *Time steps for input detection* parameter set to 2, the *Time steps for delay* parameter set to 1 and *Time steps for output duration* set to 3. This means that after the input has been *true* for 2 time-steps (at $t = 4$), input detection has passed and the output should be *true* in the next time-step ($t = 5$). However, the delay of 1 time-step delays the *true* output until $t = 6$. The output duration is set to 3, thus the output becomes *false* at $t = 9$. Then, the process repeats itself.

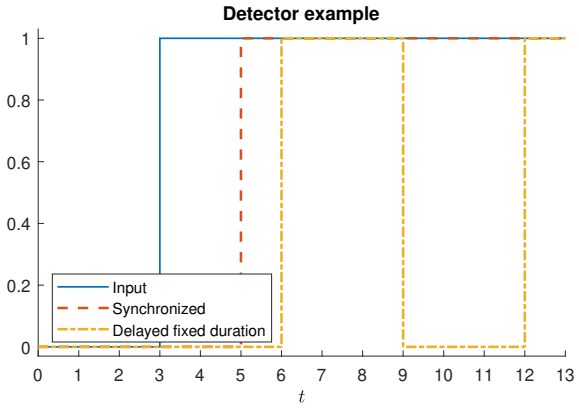


Figure 5.8: Simulation example of Detector block behavior.

The *Extender* block (shown in Figure 5.9) is a less complex block. It simply extends *true* inputs for x time-steps, as defined by its *Time steps for extension* parameter. In other words, the output will be *true* whenever the input is *true*, and for x consecutive time-steps after the last *true* input. An example can be found in Figure 5.10, with an extension of 2 time-steps.

The third and final temporal operator in SLDV is the *Within Implies* block. Its purpose is to detect whether or not an observed signal is raised (*i.e.* set to *true*) in intervals. The block, which can be seen in Figure 5.11, has two inputs. The *In*

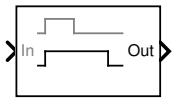


Figure 5.9: SLDV Extender block.

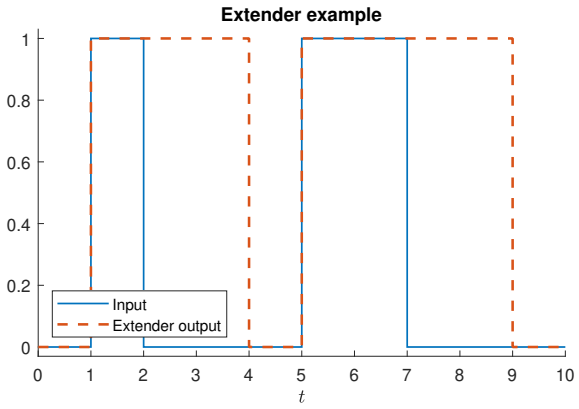


Figure 5.10: Simulation example of Extender block behavior.

input specifies the intervals. An interval is the period in which this signal is *true*; setting the In input to *false* ends and interval. The other input is the *Obs* signal, *i.e.* the observed signal. The Within Implies block will detect whether or not the observed signal is *true* in at least one time-step in the interval. The default output of this block is *true*. If the observed input is not *true* in at least one time-step in an interval, the output will be *false* for one time-step when the interval ends. The usage of the Within Implies block is exemplified in a simulation in Figure 5.12.

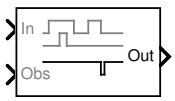


Figure 5.11: SLDV Within Implies block.

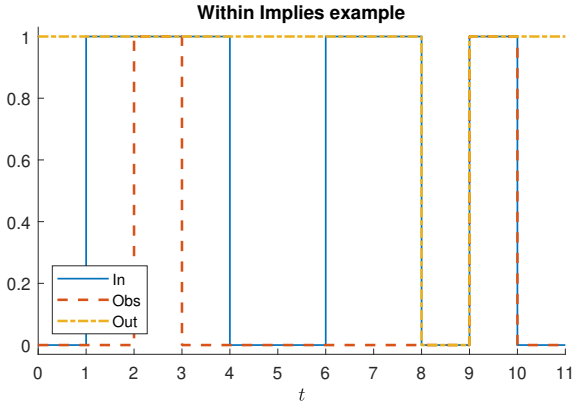


Figure 5.12: Simulation example of Within Implies block behavior.

5.3 Continuous Signals in SLDV

Signals in Simulink can either have discrete or continuous domains. Inherently, digital representations of continuous signals can create issues such as accuracy limitations, or numerical errors in arithmetical operations. SLDV will sometimes approximate floating-point values as rational numbers. In many instances this is not an issue, but since floating-point values cannot represent all real values, the approximations can lead to incorrect results. An example of this is shown in Figure 5.13, where the comparison $1 + x = 1$ is done where $x \in [10^{-10}, 10^{-8}]$. For any number, this is incorrect. However, 32-bit floating-point values do not have the accuracy to represent decimals of the magnitude of x in 1, *i.e.* x is too small compared to 1 such that $1 + x = 1$ is indeed *true* when the values are represented by 32-bit floats; note that 32-bit floating-point values are also called single-precision floating-point values.

Newer versions of SLDV also include the option for additional analysis to reduce floating-point approximation errors. Enabling this option will increase analysis times but will also correctly detect that using 32-bit floating-point signals will satisfy the Proof Objective.

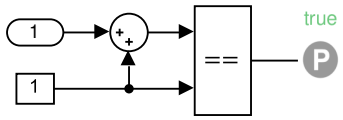


Figure 5.13: Example of continuous signals in SLDV, where representation discrepancies can cause incorrect conclusions by SLDV.

5.4 Block replacements

When running SLDV, it is possible to allow SLDV to replace blocks in the model automatically. Some blocks in the model might be incompatible with SLDV, can be difficult for SLDV to handle, or the tester could simply want to test a slightly different behavior for the model. Using the automatic block replacement feature of SLDV, it is possible to create script files which find and replace specified blocks. When this option is enabled, the model will be copied and the block replacements are performed on the new copy of the model; the original model is not affected by this feature.

In Appendix D, an example of a block replacement script file is shown. It is possible to create multiple, different such files and SLDV can use all of them when performing the block replacement.

6

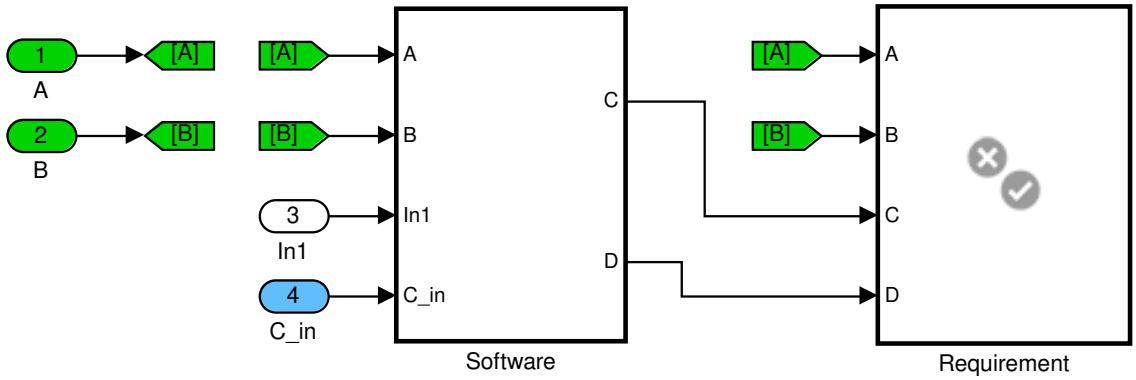
Case Study – Charging Control Software

The Simulink model of the charging control software at Volvo Cars is currently being tested using test case simulations. This allows for many tests to be carried out on the full software model without much difficulty. However, the testing is very dependent on test cases which look at many different aspects and functionalities of the software. Since the software is very big and complex, it is very difficult to robustly test the behavior of the software against the requirements. This chapter presents the methods used in the thesis project for creating test environments and requirement models to allow SLDV to reach a conclusion regarding verification or falsification of the requirements. Further steps taken to isolate errors in the software and possibly resolve them are presented as well. The case study is qualitative and will go through a few requirements of different types in detail. The requirements in the case study are used as a basis for the discussions carried out later in the thesis, and although many of the actions taken are very specific to the charging control software at Volvo Cars, the strategies used for these results can be helpful in integrating SLDV into other software development environments.

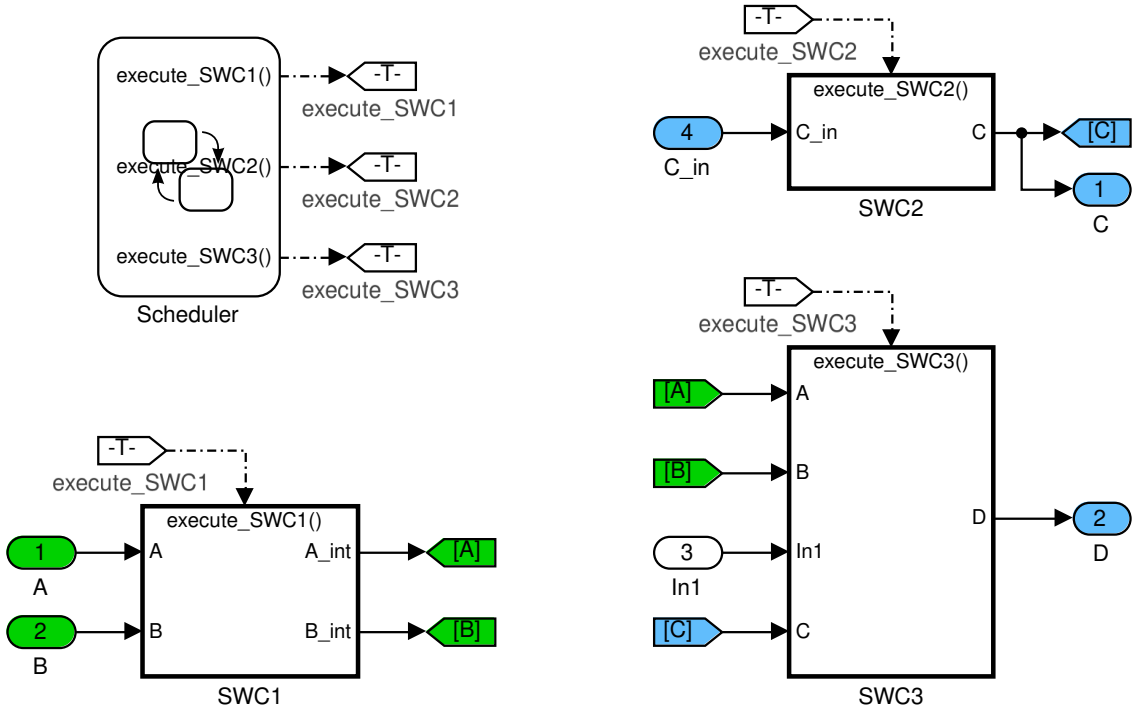
This chapter will first include some general findings in the charging control software at Volvo Cars. Then follows a series of requirements, and how test environments and requirement models are built to allow for SLDV to verify and falsify them is presented. The test environments for the requirements typically follow the pattern as shown in Figure 6.1. However, if test environments contain only a single SWC, then the Scheduler is omitted. The results obtained here are used as a basis for the discussions and answers to the research questions in the following chapters.

The case study with the charging control software is carried out on a laptop computer with an Intel i5-6300U CPU and 8 GB RAM. Due to the low number of cores on the computer (2), analysis times are heavily dependent on other processes running simultaneously. Therefore, exact comparisons between analysis times for the charging control software are not necessarily accurate. Large differences in computation times are meaningful, but even relatively large discrepancies for short analysis times can be unimportant.

Some analysis is done on reduced models with some equivalent structure and behavior as the charging software; these reduced models are shown in figures for the



(a) Test environment model root.



(b) Test environment software subsystem. The Scheduler determines the execution order of the SWCs. In each time-step, a so-called *function-call* to the SWCs, triggering atomic execution of the SWCs in the order specified in the Scheduler.

Figure 6.1: Test environment structure used to verify or falsify the requirements for the software.

different requirements. These analyses are instead carried out on a desktop computer with an AMD Ryzen 7 3700X CPU and 32 GB RAM. Due to the high core count (8), this computer is much less sensitive to other processes running in the

background, and the analysis times are therefore more consistent. Although the single-core computation capabilities are vastly different between the two systems, the results indicate the order of magnitude in computation times that can be expected from running similar models in SLDV.

6.1 General actions for SLDV optimizations

This section will go through common actions and changes to the test environments and requirement models done to assist SLDV in coming to conclusions faster, and what blocks and logic that are typically difficult for SLDV to handle.

6.1.1 Temporal Logic

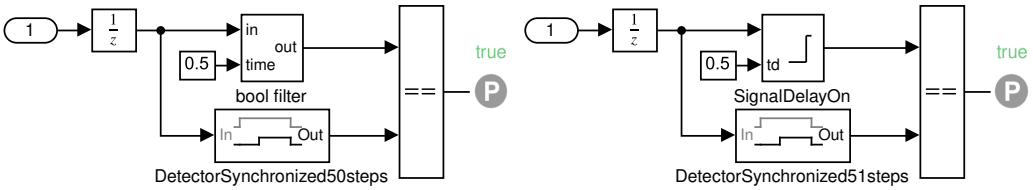
Temporal logic refers to logic which is time-dependent. Note that in this case study, the system is time-invariant in the sense that in a given state, it will react identically to a set of inputs, regardless of when the system was initiated. However, there are initial states or variables read from a virtual memory in the initialization of the system. Temporal logic in the system usually refers to detecting signals which are raised for a certain period of time, or triggering timeouts when a response has not been received in an expected time frame. Here, common temporal logic bottlenecks are presented and how to mitigate them.

6.1.1.1 Temporal Detector blocks

An issue with temporal logic is the increased size of the state space which in turn increases computational times. When running property proving on models containing logic to detect consecutive *true* or *false* inputs, computation times can increase rapidly if not modeled correctly. The charging control software contains multiple implementations for this purpose, all of which work fine for simulation but the internal closed loops and floating-point values cause problems for analysis with SLDV. However, this logic can be replaced by equivalent SLDV temporal operators.

To detect consecutive *true* inputs, a Detector block set to *Synchronized* can be used; this block is explained in Section 5.2. However, in order not to change the behavior of the software, equivalence between the Detector and the custom detector blocks should be proven. If the blocks are equivalent, then the temporal logic in the software can be replaced by the SLDV Detector block.

Two examples of equivalence tests are shown in Figure 6.2. The purpose of the blocks are to detect *true* consecutive inputs, and output *true* after consecutive inputs corresponding to the time specified by the second input. The implementations of this behavior differ slightly between the blocks. For a detection time input of 0.5, the *bool filter* block is shown to be equivalent to a Detector block set to *Synchronized* and 50 time-steps. For the *SignalDelayOn*, the Detector block must instead be set to 51 time-steps. This shows the importance of proving block equivalence before replacing blocks in the test environment.



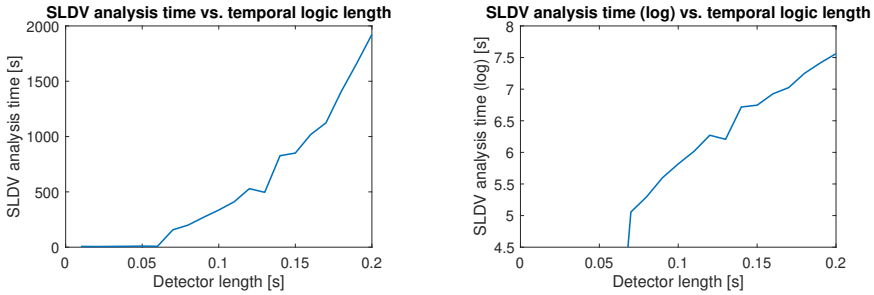
(a) Equivalence example one: 0.5 seconds is equivalent to 50 Detector time-steps.

(b) Equivalence example one: 0.5 seconds is equivalent to 51 Detector time-steps.

Figure 6.2: Example of equivalence tests, showing that the custom temporal detector blocks can be replaced by SLDV Detector blocks.

The Unit Delay blocks are added to avoid a bug in the *bool filter* block (this bug is not present in the *SignalDelayOn* block). If the input is *true* in the very first time-step, then the detection implementation in the *bool filter* is bypassed and the output is set to *true* instantly. The Unit Delay blocks are added to ensure the input to the *bool filter* in the first time-step is *false*. With the Unit Delay block added, equivalence is proven by SLDV.

To shed more light on the issue with the temporal detection subsystems, a small scale version of the test environment in Subsection 6.2.1 is investigated. Property proving is performed for detection times from 0.01 s (*i.e.* one time-step) to 0.2 s, all of which proves the requirement valid. The analysis times for SLDV Property Proving can be seen in Figure 6.3. Although noisy, it is clear from the plots that the analysis time in SLDV increases with the temporal logic length. The logarithmic plot in Figure 6.3b suggests the analysis times increase exponentially as a function of the temporal logic length, when temporal logic starts to become the major bottleneck in the property proving. This can be very troublesome in large models with extensive temporal logic.



(a) Linear analysis times.

(b) Logarithmic analysis times.

Figure 6.3: SLDV analysis times for temporal logic equivalence tests.

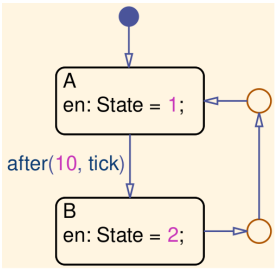
An important thing to note is the implementation of custom temporal detector blocks. The detection times in both versions in Figure 6.2 are specified in seconds. Therefore, they must either be converted to number of time-steps, or computations are done with floating-point signals to keep track of how long the input signal has been *true*. The floating-point values causes SLDV to approximate these signals

as rational numbers. It is possible to perform additional analysis to reduce these approximations. Unfortunately, this can take some time, especially with double-precision floating-point values. When running an equivalence test on one of the custom detector blocks for a detection time of 0.02 seconds, the additional analysis for single-precision values is complete after 30 seconds, whereas the double-precision analysis does not conclude after 1 hour. These results show that floating-point signals should use single-precision in favor of double-precision whenever possible.

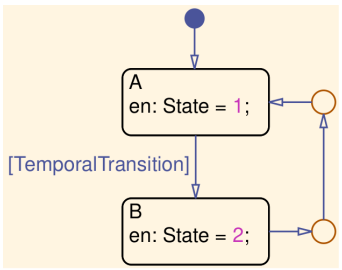
6.1.2 Stateflow charts

Much of the program flow in the charging control software is controlled using Stateflow charts. These contain many temporal transitions, which can increase computation times greatly; this is exemplified later in the report. Typically, they are implemented as shown in figure 6.4a, where the Stateflow chart will make the transition from A to B 10 steps after entering A. The chart in Figure 6.4a is the chart named After in Figure 6.4c.

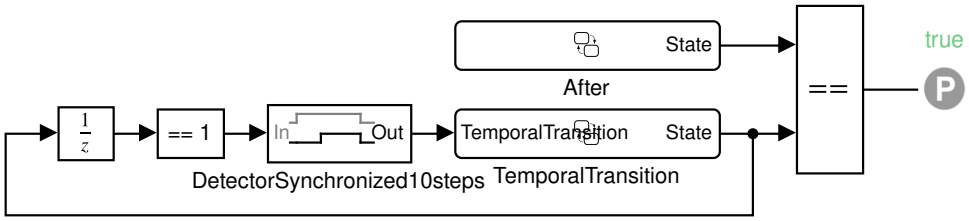
The temporal logic can also be moved from the Stateflow chart to regular Simulink blocks. In this case, an extra input *TemporalTransition* is added which is used to trigger the transition instead, as shown in Figure 6.4b. A closed-loop is also added to trigger *TemporalTransition* when *State* has been 1 for 10 consecutive time-steps, using a Detector block set to *Synchronized* and 10 time-steps. The Unit Delay is added to avoid algebraic loops in the model. Equivalence is shown using SLDV, as shown in Figure 6.4c, where the TemporalTransition chart is the one shown in Figure 6.4b.



(a) Typical temporal logic structure in Stateflow charts.



(b) More general temporal logic structure in Stateflow charts.



(c) Simulink structure to mimic the behavior of the After chart with the TemporalTransition chart. Equivalence is verified by SLDV after a few seconds.

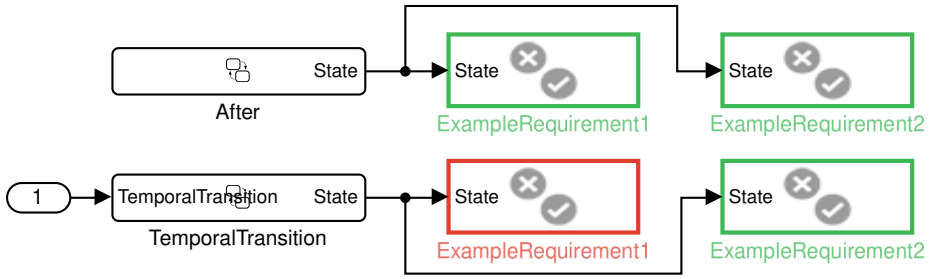
Figure 6.4: Different approaches to modeling temporal logic in Stateflow charts.

It is also possible to remove the closed-loop, and make *TemporalTransition* an input

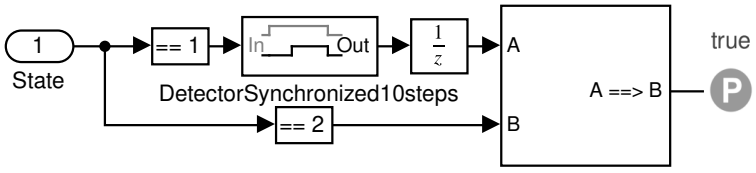
to the system. This allows SLDV to trigger the temporal transition at all times, which removes the logic that triggers the temporal transition after a specific amount of time-steps, but also makes the model more general; the temporal transition can still be triggered in the "correct" time-step. The more general model can be more useful to use with SLDV as it is simpler and contains less logic, and could therefore provide a conclusion quicker. Note that this is only useful for requirements which do not test a temporal transition precisely. For instance, consider the two requirement examples stated below.

- *If the Stateflow chart was in state A the last 10 time-steps, then its current state shall be B.*
- *If the Stateflow chart was in State B in the previous time-step, then its current state shall be A.*

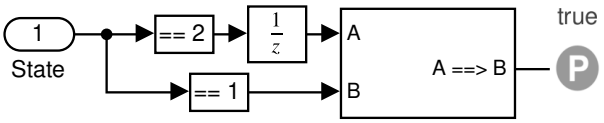
The first one relies on the temporal transition whereas the second one does not. The first example requirement is falsified for the more general model but still verified for the original model. The counterexample given by SLDV is to simply never set *TemporalTransition* to *true* until the requirement is falsified. Contrarily, the second example requirement is verified for the general model which is a stronger proof than for the original model, hence automatically verifies the original model as well. This is shown in Figure 6.5.



(a) Example requirement model. Both requirements verified for the After chart. The first requirement is falsified, and the second requirement is verified for the TemporalTransition chart.



(b) Example requirement 1.



(c) Example requirement 2.

Figure 6.5: Stateflow temporal logic example requirements.

When the Stateflow chart contains multiple temporal transitions, one input for every

transition is not necessary. Instead, a single *TemporalTransition* input can be used to trigger all temporal transitions. It is important to know that this can cause issues in states with multiple possible temporal transitions. This is exemplified in Figure 6.6a. If *in* is never set to *true*, then the chart will first enter A2 from A1, and finally enter C. However, if both transitions are replaced by a *TemporalTransition* input as in Figure 6.6b, A2 will never execute. Thus, dead logic is introduced. To solve this, a local variable *TemporalTmp* can be introduced which enforces an execution order on the temporal transitions. This variable is kept *false* at most times. However, in a state and substate where multiple temporal transitions are possible, the variable is set to *true* after the first transition is made, and a condition for *TemporalTmp* to be *true* can be added to the second temporal transition. This is shown in Figure 6.6c. Note that *TemporalTmp* should be set to *false* for all transitions out of the state with multiple temporal transitions, to enable its use in other such states.

6.2 Requirement 1 – Fault detection

There are many errors that can occur during charging. For instance, there could be malfunction in the hardware, or a fault is intentionally generated due to communication problems. It is important for the software to find these faults during runtime and handle them accordingly. The purpose of this requirement is to ensure that the software will signal that a fault has been detected, for a subset of the errors which can occur.

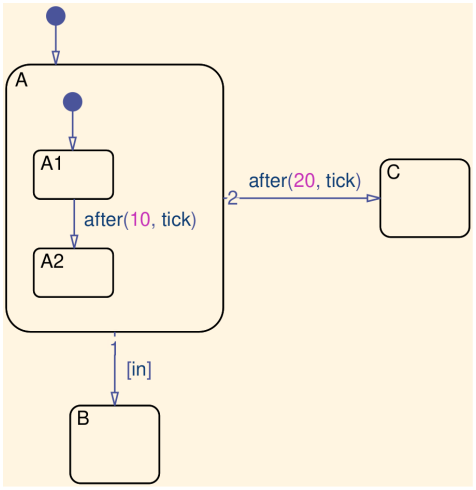
Requirement 1

If signal A has enumeration value 2, signal G shall be set to enumeration value 1 when at least one of the following conditions is true:

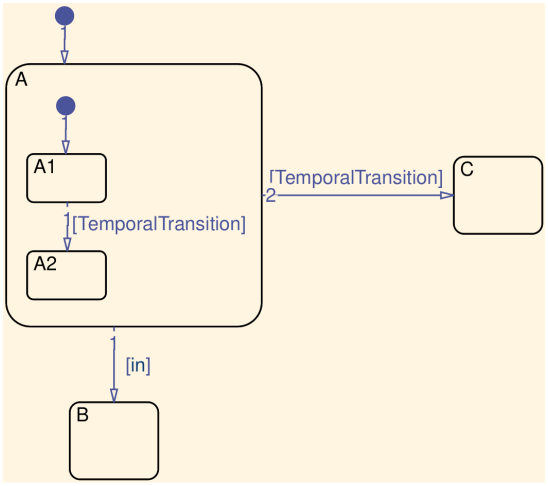
- *signal B is set to true*
- *signal C has been consecutively true for at least 0.5 seconds and at least one of signals D, E, F is true*

All signals B...F are boolean, and signals A and G are enums of different types. Signals A, B, G are outputs from the software. Signals C...F are inputs to the software.

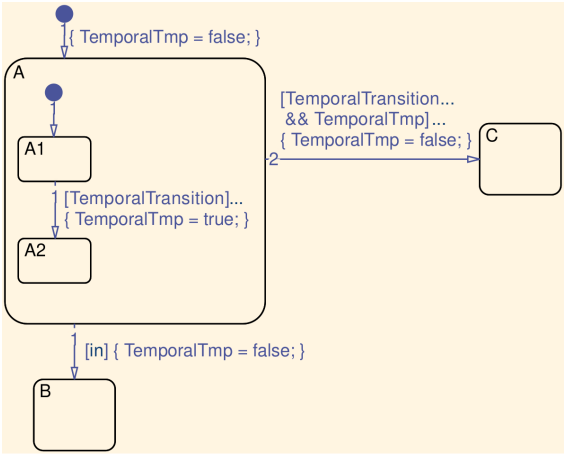
The requirement in its original form, used today to test the software, can be found in Figure 6.7. The output *Ver* is a verdict signal, specifying whether or not the requirement is fulfilled. If *Ver* is *false*, the requirement is falsified in the current time-step. Signal G will only be tested if the conditions are fulfilled. The *CustomDetector* block is a subsystem implemented by Volvo Cars to detect consecutive *true* inputs in the first inport for the duration specified by the second inport, in this case 0.5 seconds. This is equivalent to an SLDV Detector block set to *Synchronized* and 51 detection steps. An equivalent requirement model which is optimized for SLDV can be seen in Figure 6.8. The replacement of the Switch to an Implies block has resulted in shorter analysis times in some instances, although no difference between



(a) Original Stateflow chart, having explicit temporal transitions.



(b) General Stateflow chart, with *TemporalTransition* input triggered transition. State A2 can never be entered as a *true TemporalTransition* input in State A takes the chart into state C.



(c) Updated version of (b), with an added local parameter *TemporalTmp* to ensure temporal transitions are executed in the correct order.

Figure 6.6: Examples showing how temporal logic can be represented in Stateflow charts.

Switch and Implies modeling was found for this requirement; the replacement is done for consistency and explicitness.

For this requirement, different approaches for verification will be presented. Then, a comparison and conclusion will be presented. Note that the metrics for the different test environment models will be presented in the summary and comparison.

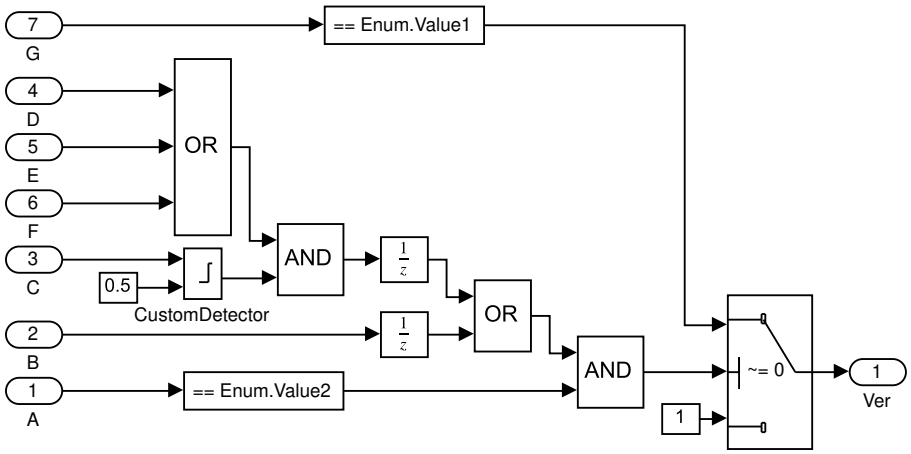


Figure 6.7: Original model for Requirement 1.

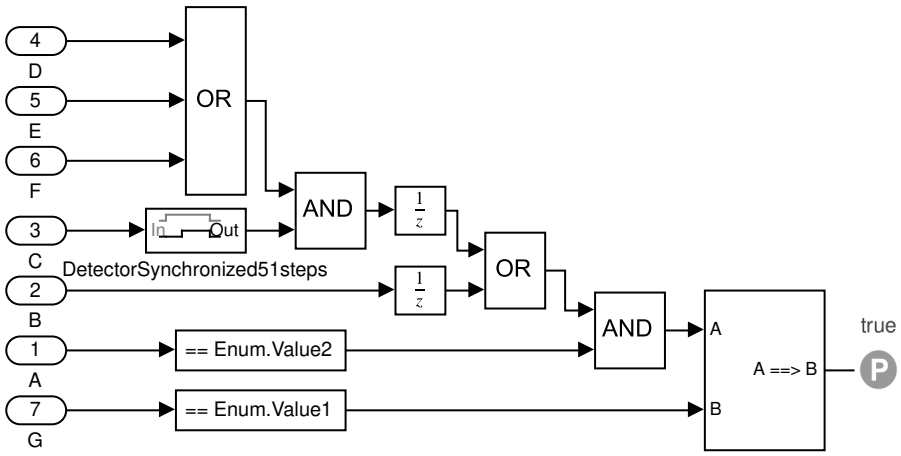


Figure 6.8: Original requirement 1 model optimized for SLDV.

6.2.1 SingleSWC – Testing a single software component

Of the seven signals mentioned in Requirement 1, three of them are outputs from the software. However, two of them are conditions in the requirement and used to compute the third. Additionally, these two output signals are also inputs to the SWC mainly tested for this requirement. Therefore, a very simple test environment can be set up where signals A and B, which are the two condition signals in the requirement, are treated as inputs. Thus, the only functionality left in the software in this test environment is the one used to compute signal G.

Normally, there are several SWCs to detect faults in the software, and Requirement 1 is used to test one of these SWCs, denoted SWC1 for the scope of this requirement. The detected faults are then propagated into another SWC (SWC2) which outputs signal G. SWC2 contains more logic than detecting faults into signal G and will also detect more faults into G than what is found by SWC1, but it can be shown that whenever the output from SWC1 is *true*, signal G has enumeration value 1, *i.e.* $H \rightarrow (G == 1)$ where H is the output from SWC1. Therefore, a slight restructuring of the software can be done. A Switch block is added to H which outputs enum

value 1 if H is *true*, otherwise enum value 0. This single SWC test environment can be seen in Figure 6.9.

The single software component can now be tested against Requirement 1. However, since all signals except G are treated as inputs, there is no need for synchronization using the Unit Delay blocks in the requirement. Hence, the Unit Delay blocks from the model in Figure 6.8 should be removed to form a requirement model for the single SWC. Additionally, the single SWC removes the need of a Stateflow scheduler, and the subsystem can be converted from a trigger subsystem to a simple atomic subsystem. However, the signal timings in this test environment are incorrect from the full software. The logic in SWC2, represented by the Switch, is performed before SWC1 in each time-step, hence the timings of the signals are incorrect.

The resulting requirement model is shown in Figure 6.10. In this test environment, the requirement is verified in 20–30 seconds. Including the Unit Delay blocks in Figure 6.8 causes the requirement to be falsified by SLDV after a few seconds, as the synchronization is not needed in this setup. The verification in this test environment shows that it is possible to verify requirements by treating software outputs as inputs if they are condition outputs, *i.e.* outputs which are only present in the requirement condition. Note, however, that this is not always the case. Additionally, simplifications can be done if they have been shown to never fulfill the requirement when the original model does not.

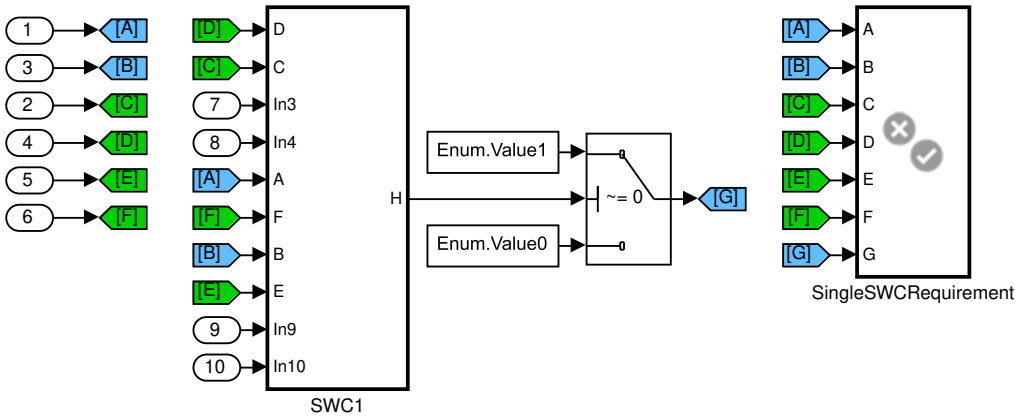


Figure 6.9: SingleSWC test environment.

6.2.2 DoubleSWC – Test environment with two software components

In the SingleSWC test environment used above, the signal timings for signals G and H are incorrect. The reason for this is that SWC2, which the Switch block is representing, is executed before SWC1 in each time-step. The incorrect signal timings can be fixed by removing the Switch after SWC1 and adding SWC2 back into the software, to get an isolated version of the software consisting of two SWCs. Thus, proper behavior is retained in the software. The proper execution order is also implemented, such that SWC2 is executed before SWC1 in each time-step. The

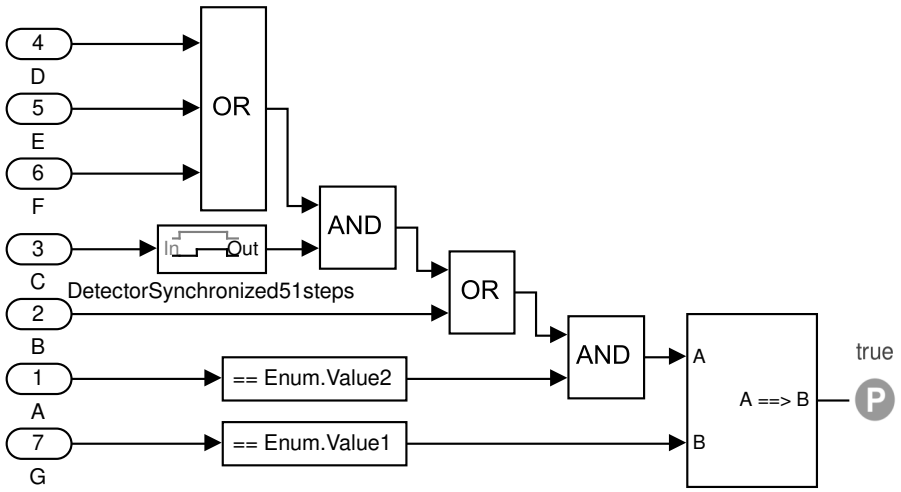


Figure 6.10: SingleSWC requirement model, with removed Unit Delay blocks to resolve signal timings.

test environment in this case can be seen in Figure 6.11. It should be noted that SWC2 is reduced. Firstly, all outputs apart from G and the logic to evaluate these are removed. Secondly, SWC2 contains several other signals used to compute G. However, these culminate in a large `OR` gate close to the output. Hence, the logic not related to H is removed and replaced by the two other signals as seen in the previously mentioned figure, to make a much smaller but more general SWC. The two other subsystems, SWC3 and SWC4, are added for correct signal synchronization, and do little else than passing the signals straight through the subsystems. The execution order is the same as the output order from the Scheduler chart.

With the added SWC, the signal timings in the requirement must be resolved. In the original requirement, there were two Unit Delay blocks in place for this purpose, but that requirement model is falsified by SLDV. Another Unit Delay block must be added to signal A. This can be counter intuitive as signal A is evaluated earlier in each time-step by the scheduler than H. However, since SWC2 takes in H from SWC1 from the previous time-step, the requirement must also look at the value of A in that time-step as well, hence a Unit Delay block should be added to A in the requirement. This results in the requirement model seen in Figure 6.12, which is verified in just over 30 seconds. The test environment shows that it is indeed possible to verify requirements in some instances where software outputs are treated as inputs, if they are part of the antecedent in the implication, *i.e.* they only lead to the A input in the Implies block in the requirement. Compared to the single SWC setup, this test environment is more "correct" with respect to the full software, although the reductions done for the single SWC setup are correctly done and verifications for the reduced system also verifies the original system.

6.2.3 MultiSWC – Test environment with 4 components

In the previous test environments, signals A and B have been treated as inputs to the software and requirement. However, they are outputs from the software. The

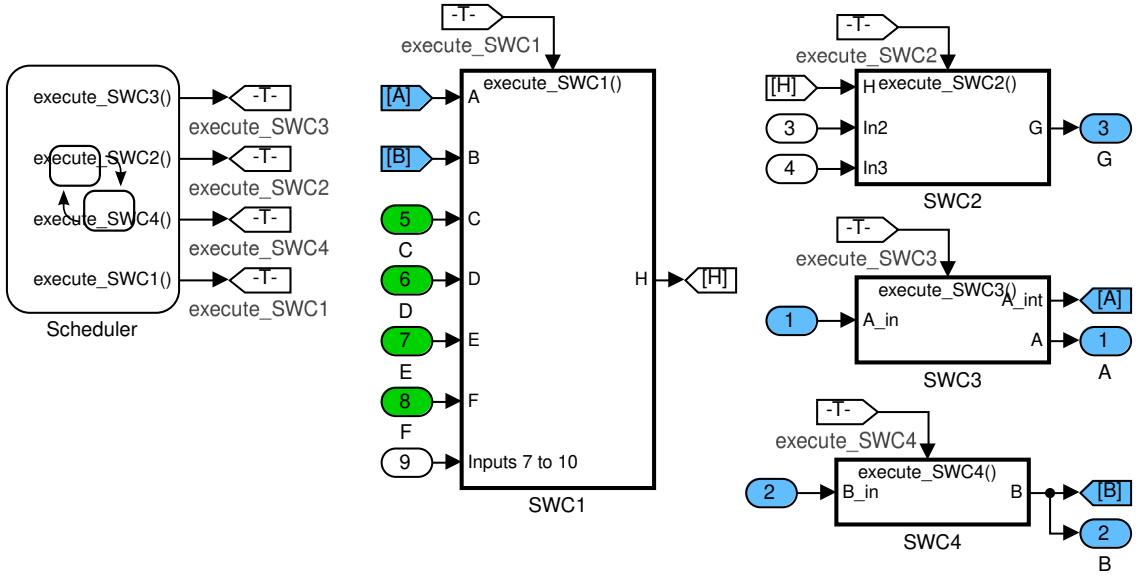


Figure 6.11: DoubleSWC test environment. SWC3 and SWC4 are included to synchronize signals A and B. Signal A has different enumeration types internally in the software and externally, but this has no effect on the current test environment and the internal and output signals can for simplicity be regarded to be of the same types.

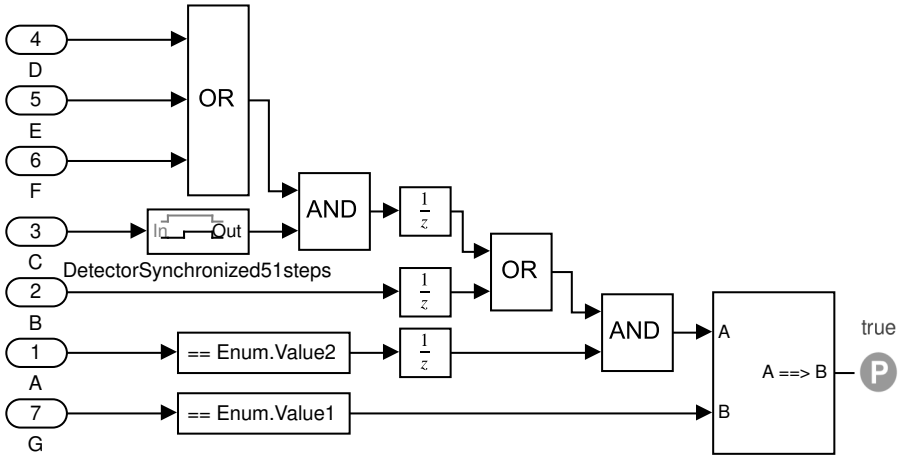


Figure 6.12: Requirement model for the DoubleSWC test environment. A Unit Delay block is added to signal A with respect to the requirement model seen in Figure 6.8.

following tests show that it is possible to verify the requirements for test environments in which the SWCs (SWC3 and SWC4) which output signals A and B are included. SWCs 3 and 4 are not only dependent on software inputs but on outputs from other SWCs as well, but for the sake of this test their inputs are treated as inputs to the software. SWC3 consists mainly of Boolean operators, comparisons and switches. SWC4 contains some temporal logic but mainly consists of a big Stateflow chart. This produces the test environment in Figure 6.13. The requirement is the

same as before, as shown in Figure 6.12.

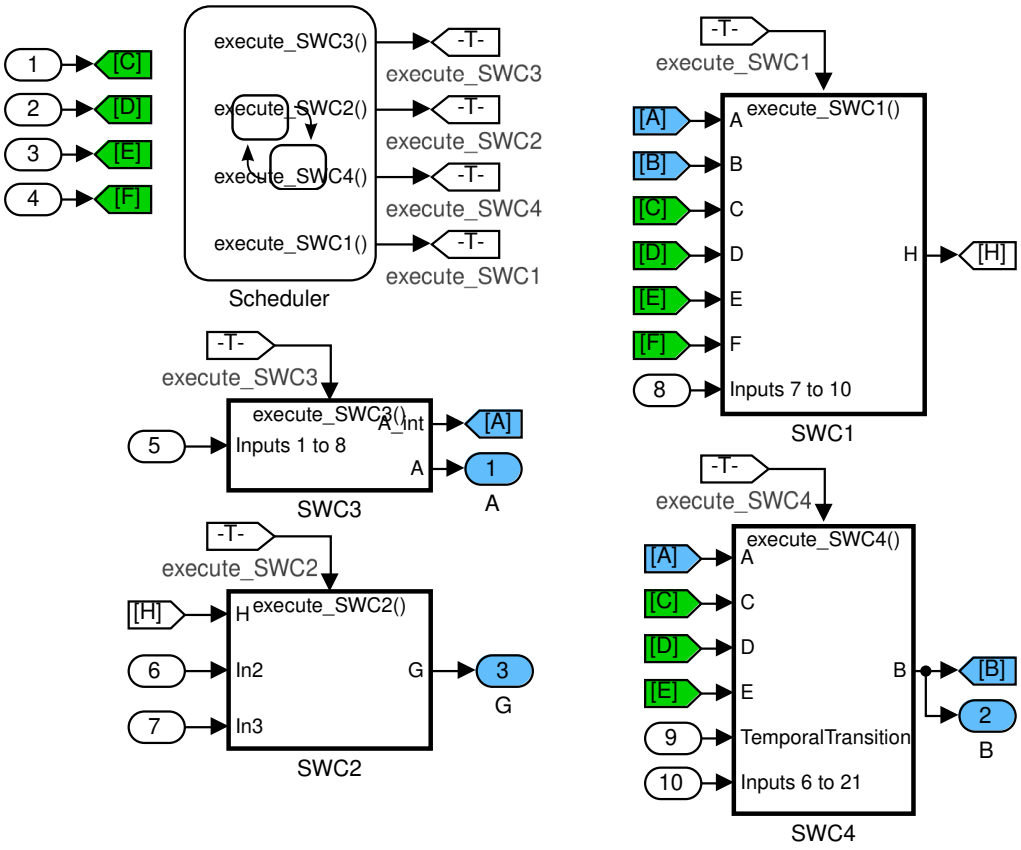


Figure 6.13: MultiSWC test environment, used together with the same requirement model as in the DoubleSWC test environment (Figure 6.12). When SWC3 or SWC4 are bypassed, they are replaced by their equivalent subsystems as seen in the DoubleSWC environment in Figure 6.11, which are only used for synchronization of inputs.

An apparent issue related to SWC3 and SWC4 causes problems for this test environment. Both components take in floating-point and fixed-point values which are subject to data type conversions and comparisons. Initially, SLDV will approximate these signals as rational numbers, and produce a result under approximation. It is possible to specify in the Configuration Parameters for SLDV to perform additional analysis to reduce the rational number approximations. However, this can take a very long time. When verifying Requirement 1 with SWCs 3 and 4, an approximated result is reached in approximately 28 minutes, but the additional analysis reaches 10 hours without coming to a conclusion. This behavior can also be shown to cause greatly increased analysis times for smaller models. When bypassing SWC3, such that an input is only passed straight through for synchronization, an approximated result is achieved in two minutes, and the additional analysis is complete after another three minutes. The same bypass but for SWC4 results in a valid requirement under approximation in under one minute, while the additional analysis lasts for over 15 minutes.

To alleviate this issue, the floating-point and fixed-point signals in this test environment can be converted to to the "correct" type to avoid data type conversions. In these SWCs, no computations are done, only comparisons between values. Thus, these signals can be converted to Boolean inputs which simply represent if comparisons return *true* or *false*. Hence, full coverage is achieved, and the model is optimized for SLDV. With these optimizations, SLDV is able to verify Requirement 1 in approximately the same amount of time as the approximated results for the non-optimized test environment.

6.2.4 Conclusion

The tests and results from the test environments show that Requirement 1 is fulfilled, if correction of the signal timings is allowed in the requirement model. Different approaches were conducted, all of which were successful apart from the MultiSWC model using both SWC3 and SWC4 which takes too long to analyze, and the MultiSWC setup with SWC4 bypassed where an approximation result is achieved relatively quickly but the additional analysis to reduce the rational number approximation does not finish in a relatively close amount of time.

A summary of the analysis times for the different models is listed in Table 6.1 and the metrics for these models are found in Table 6.2. The test results show that isolating behavior and SWCs in the model greatly improves performance, despite generating a more general model. Although these isolations can be difficult to automate, the tests show that even with more logic in the model than required, it is still possible to reach a conclusion in a reasonable amount of time. Floating-point and fixed-point signals can cause long analysis times to reduce rational number approximations, although this can be optimized for by removing Data Type Conversion blocks and using consistent data types.

Model	**Analysis time (approximation)**	**Analysis time (full)**	**Analysis time (full, FP optimized)**
SingleSWC	0:24	0:25	-
DoubleSWC	0:27	0:34	-
MultiSWC, SWC3 bypassed	2:02	5:18	1:50
MultiSWC, SWC4 bypassed	0:46	>15 min	0:48
MultiSWC, full	28:10	>10h	27:26

Table 6.1: Analysis times for verifications using SLDV. The requirement models are proven valid in all cases, except for the cases with a '>' symbol, which are undecided. The abbreviation FP means floating-point.

Model/Subsystem	Number of blocks
SWC1	57
SWC2	79
SWC3	145
SWC4	706 (256)
SingleSWC	116
DoubleSWC	238
MultiSWC, SWC3 bypassed	737 (256)
MultiSWC, SWC4 bypassed	212
MultiSWC (full)	1067

Table 6.2: Model metrics for the test environments for Requirement 1. Numbers in parentheses in the *Number of blocks* column are the number of Stateflow objects in the test environment or subsystem.

6.3 Requirement 2 – Start charging

To handle the many different stages in the hand shaking and synchronization between the car and a CHAdeMO charging station, a Stateflow chart is used. The charging process is initiated using two signals: A and B, which are inputs to the Stateflow chart. During this process, the Stateflow output signal C should be set to *true*, until the process is terminated, at which point the software will set up to allow the charging process to start yet again. To illustrate this, a small example of a reduced similar Stateflow structure is shown in Figure 6.14. All states where signal C is set are present, and the other states have been merged into the Intermediate state. Note that all temporal transitions have been replaced by a *TemporalTransition* input triggered transition as shown in Subsection 6.1.2 unless stated otherwise, including the *after(100, tick)* transition. The purpose of the requirement is to check if signal C is set for the right A and B inputs.

Requirement 2

If signal A is true and signal B changes from false to true, signal C shall be set to true. Signal A is an enumeration signal, and signals B and C are boolean signals. Signal B is an input to the software, and signals A and C are outputs from the software.

It should be noted that in the original requirement, signal A is an enum where its value should be 2. However, since only comparisons to the enumeration value 2 are made, it is replaced with a boolean signal instead, which represents whether the enumeration value is 2. Additionally, although signal A is an output from the system it is also used as input to many SWCs in the software, such as the SWC which outputs signal C. Therefore, it is treated as an input to the system in this case,

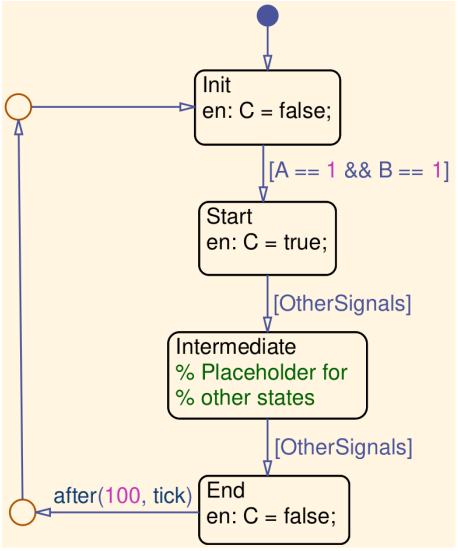


Figure 6.14: Software CHAdeMO Stateflow structure.

meaning the test environment will only consist of one SWC and the requirement.

6.3.1 Original requirement model

In its original form, the requirement was modeled as seen in figure 6.15. The infinite extender was originally modeled as an SR-latch with a constant *false* as R input; these can easily be proven to be equivalent. There are two interesting things to note in this original model. First, there is no detection for when signal B goes from *false* to *true*, as Requirement 2 states. However, since signal B is always *true* whenever it changes from *false* to *true*, this form is a stronger proof. In other words, if the requirement model is verified for whenever B is *true*, then it is also verified for whenever B changes from *false* to *true*. This can be shown with SLDV, as done in Figure 6.16.

The other interesting thing to note is the infinite Extender on signal C. This causes the requirement to be fulfilled for the rest of the simulation, as soon as signal C is set to *true*. Since C is only set to *true* once in the software, the model only checks if the requirement is fulfilled until it has been actively fulfilled once (*i.e.* the Implies block outputs *true* and the condition is active). Note that there is no specification in the requirement regarding if signal C should be set to *true* whenever signals A and B are *true*, or only the first time.

Using the original requirement model, the requirement was falsified with a simple counterexample after approximately 30 seconds. In the counterexample, signals A and B are set to *true* in the very first time-step. This falsifies the requirement since the Stateflow chart will perform the default transition into the Init state in the first time-step, and is therefore unable to transition into the Start state to set C to *true*. To ensure the requirement is only tested after the Stateflow chart is initialized, Assumption blocks can be used. However, a simpler approach is to add Unit Delay blocks to the inputs of the test environment. This ensures the model is

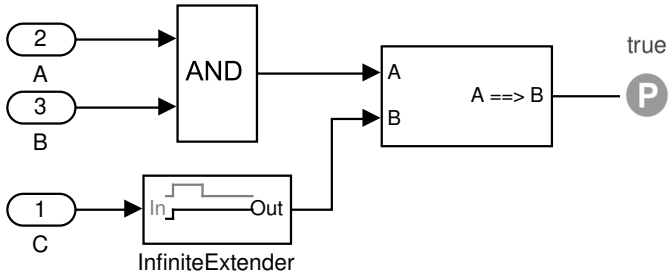


Figure 6.15: Original Requirement 2 model.

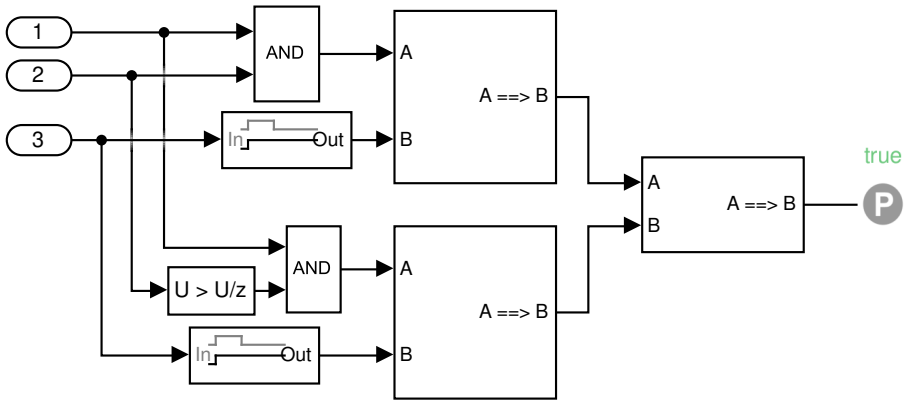


Figure 6.16: Implication example, showing that removing the positive edge detection provides a stronger proof.

initialized in the first time-step, and does not ignore input from the first time-step. The initial conditions can easily be set to not trigger the requirement condition, in this case the initial conditions for A and B are set to *false*. With these Unit Delays, the requirement is instead proven valid by SLDV after about 30 seconds. The test environment structure for these tests can be seen in Figure 6.17.

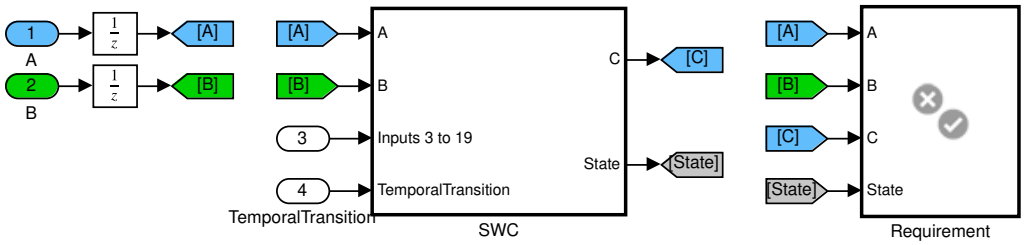


Figure 6.17: Test environment structure for Requirement 2.

6.3.2 Removing the infinite Extender block

At this point, the infinite Extender block was removed to see if the requirement would be verified on each charging process iteration, or if there were any issues found when the software has gone through one or multiple charging processes. This caused the requirement to be falsified in 30 seconds. The counterexample goes through the charging process normally and reaches the End state. However, upon

reaching this state, signal C is set to *false* which means an active condition (*i.e.* signals A and B set to *true*) would falsify the requirement. The Stateflow chart cannot react to this active condition until it reaches the Init state after one second, meaning the requirement is falsified for one second if signals A and B are set to *true* when the software reached the End state.

In this specific case, the software can be remodeled, so that signal C is never set to *false* in the End state but only in the Init state. This is shown in Figure 6.18. This almost solves the issue, but the requirement is still falsifiable. A counterexample was given after 30 seconds where the requirement is falsified if A and B are set to *true* in the same time-step the Stateflow chart transitions from End to Init. Thus, C is *false* in this time-step. To solve this, an exemption can be added to the requirement, so that the requirement is deactivated when signal C goes from *true* to *false*. This alteration can be seen in figure 6.19. With this exemption in the requirement, it is proven valid in about 20 seconds.

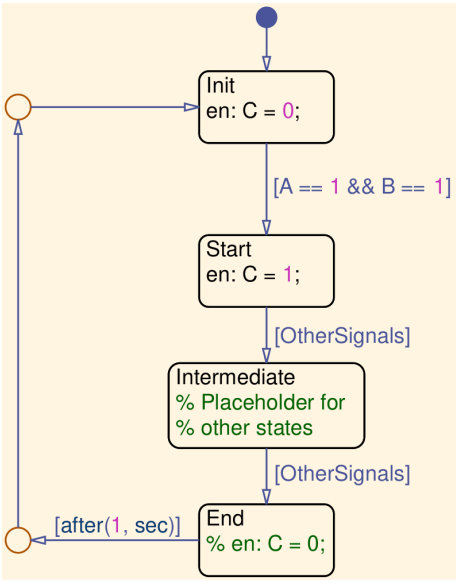


Figure 6.18: Altered Stateflow chart from Figure 6.14, where setting C to *false* in the End state is removed.

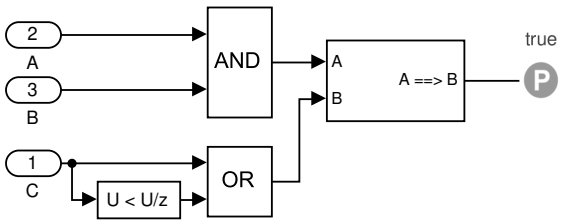


Figure 6.19: Model of Requirement 2 with the infinite Extender block removed and with an exemption for negative steps for signal C.

6.3.3 Remodeling the requirement

With the removed infinite Extender block, there is also the possibility that the requirement is incompletely specified. If the purpose of the requirement is to ensure that signal C is set to *true* when transitioning from the Init state to the Start state using inputs A and B. Thus, the requirement can be extended to only check for this behavior, for instance by adding a variable to track the current state, and add a check in the requirement to ensure the requirement looks at the transition from state Init to Start.

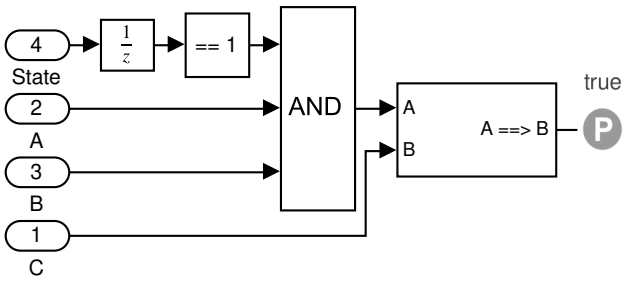
This can be done in several ways, as shown in Figure 6.20. To test the transition from Init to Start specifically, a check can be added to find transitions from the Init state. This is done by adding a condition that the previous state was the Init state, as shown in Figure 6.20a. Note the Unit Delay block, which ensures the Stateflow chart was in the Init state before the transition was made. This extended requirement is verified by SLDV in approximately 30 seconds.

An issue with this additional condition in the requirement is that it can be too restrictive. It can be useful in testing a certain part of the software, but it also stops SLDV from finding other possible errors in the software model. Hence, the requirement can instead be remodeled to ignore a specific error, to try and see if there are any more errors. Instead of checking specifically for the Init state, it is possible to add a check for the End state, and ignore the requirement in that state. This allows for detection of other errors in the software, or verification that the only errors are related to the End state. The remodeled requirement to check for this is found in Figure 6.20b. Note that both the current and previous state must be observed in this case, so that both transitions to and from the state are ignored, in addition to the time-steps where the chart remains in the End state. This requirement is also verified in approximately 30 seconds.

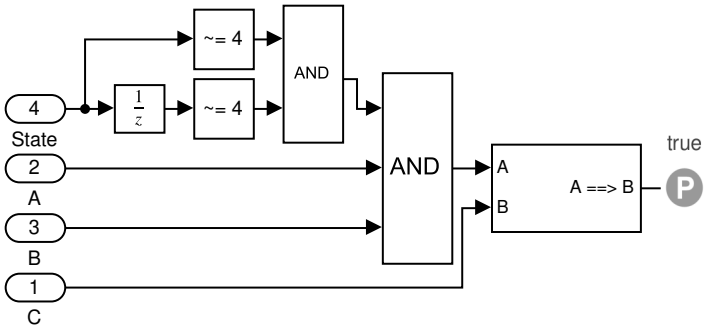
Finally, it is possible to add exemptions in the requirement for specific transitions. In this case, the counterexample given was related to the transition between the End and Init states. Therefore, a specific check for when the chart has entered the End state and has executed the transition to the Init state can be added. This can be done in multiple ways, but one of them is shown in Figure 6.21. The check for the transition is done by extending a negative step in signal C for the amount of time-steps the transition in the Stateflow chart waits for.

Unfortunately, the temporal logic in this transition creates a problem for SLDV. Even the very small structure shown in Figure 6.14 is challenging to verify by SLDV, and the temporal logic is therefore scaled down. For the simple model structure seen in Figure 6.14, a verification is reached in 30 seconds for detector lengths of 20 time-steps, 8 minutes for 40 time-steps, 11 minutes for 50 time-steps and 14 minutes for 60 time-steps. After 2 hours, no conclusion is made for this requirement with a detector length of 100 time-steps, *i.e.* the full length of the transition.

For the original software model, results are only retrieved for times less than 20 time-steps; for all these detector lengths, the requirement is verified by SLDV. The



(a) Extended model of Requirement 2 which only checks transitions from the Init state.



(b) Extended model of Requirement 2 which ignores time-steps when the chart is in or moves from the End state.

Figure 6.20: Different extensions to the Requirement 2 model, with various approaches to isolating the falsifications found earlier.

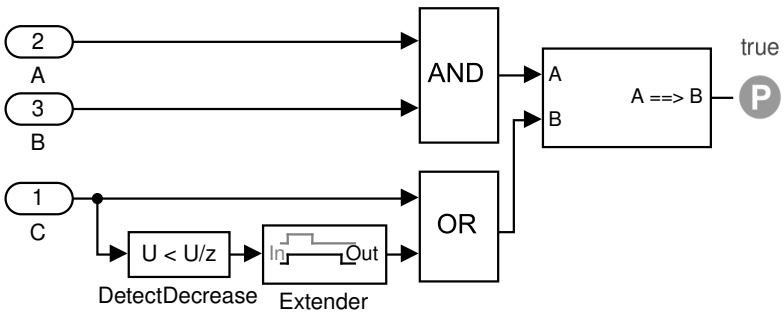
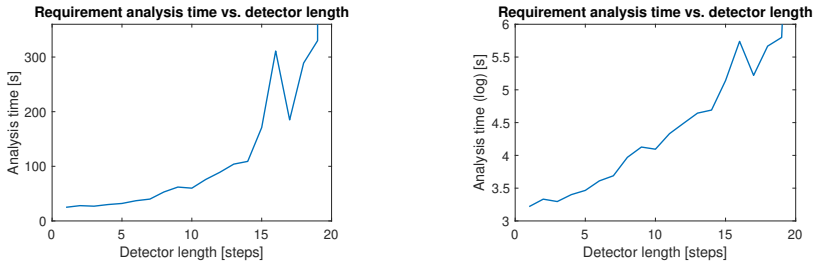


Figure 6.21: Extension to the Requirement 2 model, to track and ignore the transition from the End state to the Init state.

SLDV analysis time is plotted for detector lengths from 1 to 20 time-steps. For time-steps less than 20, a verification is reached within 6 minutes or less. However, 20 time-steps causes SLDV to compute for at least 14 hours without any conclusion. It is possible that loops in the Stateflow chart create a threshold for when SLDV must consider a more complex of varied behavior in the chart, but there can also be other reasons behind the sudden increase in analysis time. As the requirement is proven valid for the scaled-down temporal logic, it is likely also valid for the correct temporal transition, although this cannot be proven by SLDV in the current configuration.



(a) Linear analysis time **(b)** Log. analysis time

Figure 6.22: Analysis times plotted vs. detector length, to show the rapidly increasing computation times for increasing temporal logic lengths.

To remove the temporal logic drawback, the requirement can be remodeled to use a *TemporalTransition* input instead, where the temporal transition is changed to a *TemporalTransition* input. The new requirement model can be seen in Figure 6.23. Using this new model, the template structure in Figure 6.14 is proven valid in 6–7 seconds, and the original model with the full software is proven valid in 30 seconds.

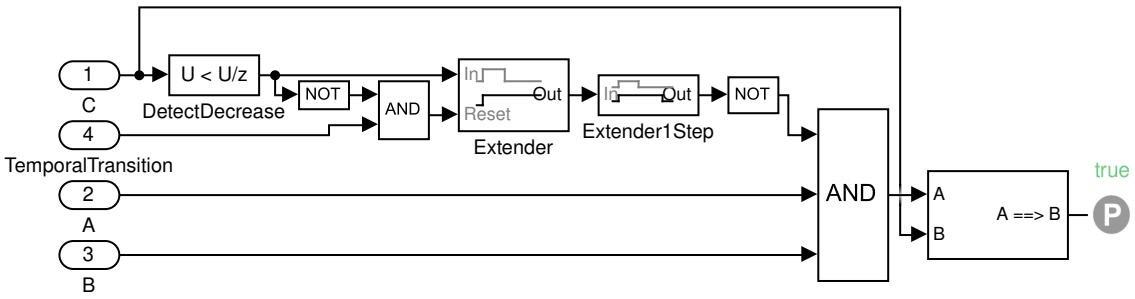


Figure 6.23: Requirement 2 model to track and ignore the *TemporalTransition* input triggered transition from the End to the Init state.

Lastly, the original model without the *TemporalTransition* inputs, *i.e.* the software model without the temporal transitions replaced by a *TemporalTransition* input transition, was tested against the requirement model seen in Figure 6.20b. This model was also verified in 30 seconds.

6.3.4 Conclusion

In this section, the process for verifying and falsifying Requirement 2 has been shown. The cause of the falsification is found, and several methods to find the cause of the falsification are shown. The temporal transitions seem to cause issues when combined with temporal logic in the requirement, and should be avoided if possible. When trying to find more counterexamples and to limit the error in the software model, it is suggested to add a state (and possible substate) output with unique state identifiers to track the current state in the chart. Then, a state exemption similar to that in Figure 6.20b should be implemented to try and find other counterexamples to the requirement. As shown, Unit Delay blocks are a simple yet effective way to allow the Stateflow charts to make their default transition, so that the requirement is not falsified in the very first time-step.

6.4 Requirement 3 – Fault detection

Much like Requirement 1, Requirement 3 deals with fault detection. The purpose of Requirement 3 is to ensure detection of a subset of errors from software inputs.

Requirement 3

Signal C shall be set to true whenever at least one of the following conditions apply:

- *signal A is set to enumeration value 3.*
- *Signal B is set to true for at least 4 seconds.*

Signal A is an enumeration signal, and signals B and C are boolean signals. Signals A and B are software inputs, while C is a software output. Signal A is propagated from the input into three different internal signals of the same enum type: A_a, A_b and A_c. A fourth enum signal D is used to "activate" at most one of these internal signals. Only signals A_a, A_b and A_c are used in the software, hence signal A is not used directly in the software. Signal C is output from an SWC which takes both A_c and B as inputs. It contains some temporal logic and Boolean operators, but consists mainly of a large Stateflow chart. However, the behavior of the SWC the requirement intends to test is outside the Stateflow chart. Hence, the temporal transitions in the Stateflow chart is replaced by *TemporalTransition* inputs unless stated otherwise. The custom temporal detector blocks in the software used to detect consecutive *true* inputs are replaced by equivalent SLDV Detector blocks set to synchronized and the appropriate number of steps to detect.

6.4.1 SingleSWC – Testing environment with one SWC

The SWC which outputs signal C, denoted SWC1 in this requirement, takes A_c and B as inputs. Note that signal A is the input to the software, and A_c is an internal signal which can be considered a "quasi-input": A_c is *activated* by another signal D, such that A_c is set to A whenever signal D has enumeration value 2, as described above. Another SWC, denoted SWC2 in this requirement, is used to evaluate A_c using signals A and D. In this environment, however, A_c will be used as an input in favor of A, to produce a single SWC environment. This means that signal A in the requirement must be replaced by A_c as well, which is an incorrect representation of the requirement. Nonetheless, using small test environments like SingleSWC can be a good way to perform small tests on the software to find errors or SLDV bottlenecks more easily.

The original requirement model used at Volvo Cars today to test Requirement 3 is shown in Figure 6.24. Note that the requirement model in this figure is not strictly the same due to the change from signal A to signal A_c, as described above. Hence, the results from this requirement model does not mean the original requirement is falsified or verified, but can still be useful as a correctness indication. The resulting environment and requirement model are shown in Figure 6.25. The requirement in this environment is proven valid by SLDV in approximately 30 seconds.

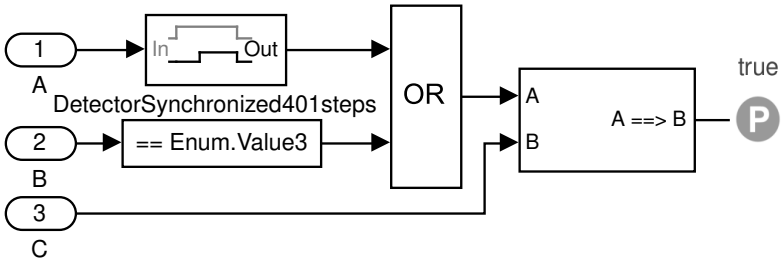
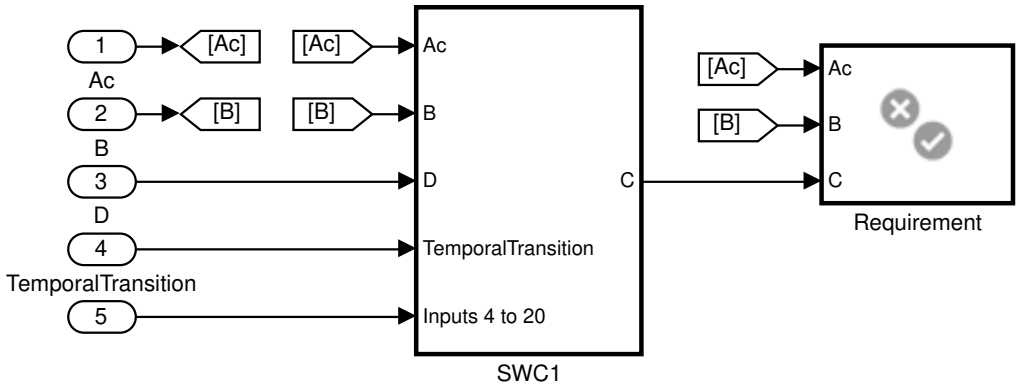
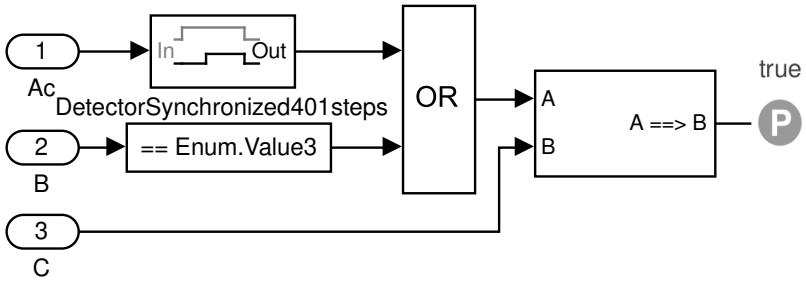


Figure 6.24: Original requirement used to test Requirement 3, optimized for SLDV.



(a) Requirement 3 SingleSWC test environment.



(b) Requirement 3 SingleSWC requirement model.

Figure 6.25: Models used for verification of Requirement 3 in the SingleSWC test environment.

6.4.2 Removing the Stateflow chart

As stated earlier, SWC1 contains a large Stateflow chart which is not actively tested in this requirement, although it can set signal C to *true* through its outputs. However, removing the Stateflow chart will still verify the requirement, as its logic is not tested for this requirement. This can be shown through a simple implication, as visualized in Figure 6.26. This implication is proven valid by SLDV in a few seconds. Adding more inputs to the `OR` gate will only make the output *true* more often. Since the output signal C is the output of an `OR` gate internally in SWC1, limiting the `OR` gate will cause the requirement to be verified less often. Removing the Stateflow chart causes the requirement to be verified in approximately 30 seconds; a little

shorter than when using the chart, but still within the margin of error.

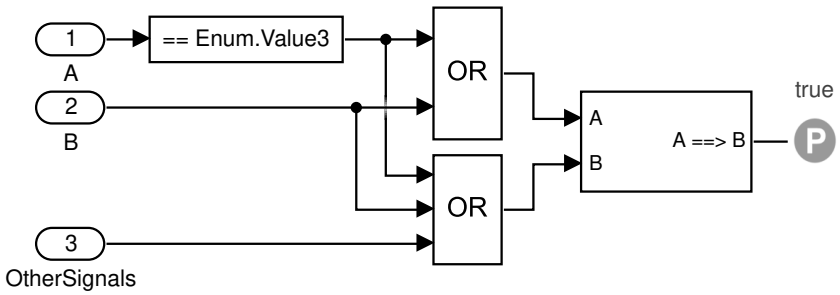


Figure 6.26: Implication showing verifications done for reduced `OR` gates are valid if the requirement is verified for reduced `OR` gates.

6.4.3 Adding original temporal Stateflow transitions

As mentioned above, the temporal transitions in the Stateflow charts were replaced by *TemporalTransition* input triggered transitions instead. To see the implication of temporal logic in the Stateflow chart on these, SWC1 can be reverted to its original Stateflow chart with explicit *after(x, tick)* transitions. Here, SLDV is still able to recognize that the Stateflow chart does not matter to the requirement and is still able to verify the requirement in under 30 seconds.

6.4.4 Testing different detector lengths

The requirement is verified for the single SWC test environment quickly, given that the Detector block length is 401 time-steps. For this test environment, when both the temporal logic in the requirement and the software are represented using SLDV Detector blocks, the internal representation used by SLDV probably ensures the analysis is invariant to the detection length of the Detector blocks in the requirement and the one tested in SWC1. The invariance can be shown by plotting the analysis times for a number of different detection times for both the Detector block in the requirement and the one tested in SWC1. The results are plotted in Figure 6.27. With the exception of a few spikes in analysis times, there is no clear dependence between analysis times and detector lengths, hence the temporal detection invariance has been shown for the SingleSWC test environment.

6.4.5 DoubleSWC – Test environment with two SWCs

As stated earlier, the SingleSWC test environment and requirement is not correct as signal A is replaced by signal A_c in these. To solve this and use a correct model of the software, SWC2 can be added to the software model. One of the purposes of SWC2 is to compute A_c from A and send A_c to SWC1. Note that both SWC1 and SWC2 take signal D as input, and SWC3 is used to evaluate signal D. Hence, SWC3 should be added to the software to achieve correct signal timings in the software, especially since SWC3 is executed in between SWC2 and SWC1. Note that in the DoubleSWC environment, SWC3 contains almost no logic and is only

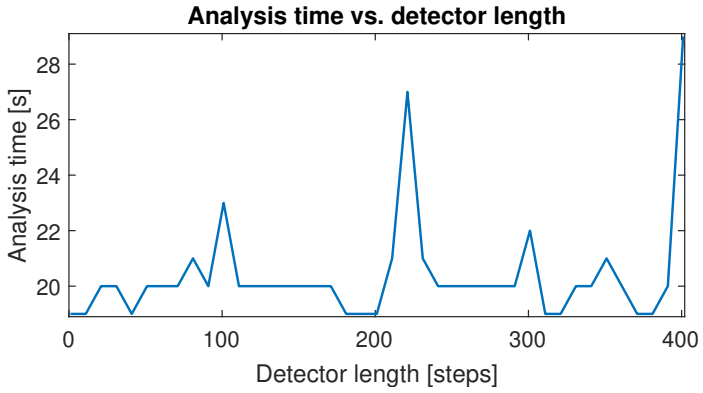


Figure 6.27: Analysis time vs. detector length for requirement 3.

used to synchronize signal D which is treated as an input in this case. The software model in this environment can be seen in Figure 6.28. The software components are executed in each time-step in the same order as the outputs from the Scheduler, *i.e.* SWC2 → SWC3 → SWC1.

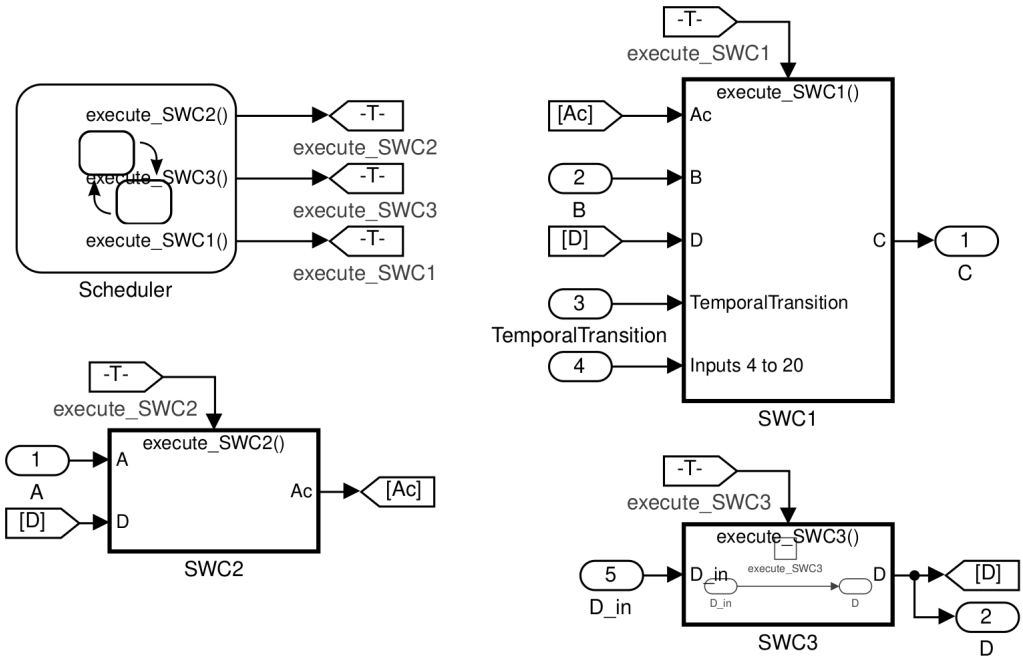


Figure 6.28: Double SWC test environment.

When trying to verify this software model using the original requirement model from Figure 6.24, an interesting issue with SLDV arises. A conclusion is never reached in 10 hours in the current test environment. To solve this, it is possible to remove the logic which uses signal D to activate signal A_c and simply set $A_c = A$. With this change, the requirement is verified. However, this is not the correct behavior of the software. As it turns out, SLDV has some problems with the initializations of the software. In the full software, the Scheduler chart will execute every SWC in order in every time-step, except the very first. Together with the logic to evaluate

A_c from A and D, this creates some problem for SLDV which causes the analysis time for this particular environment model to explode and never finish.

The Scheduler can be changed so that it will also execute all SWCs in the very first time-step as well. An altered Scheduler chart which solves the issue with execution in the first time-step can be seen in Figure 6.29. With this change to the Scheduler, the original requirement is falsified after 40 seconds. The issue is that for signal A to go straight through SWC2 and enter SWC1, signal D must be set to enum value 2. If D is not 2, enum value 3 will not be detected in SWC1 although it can be detected in the requirement, hence the requirement is falsified by such counterexamples.

Figure 6.29: Changed Scheduler chart from Figure 3.1b to allow execution of SWCs in the very first time-step of the simulation.

Solving the issue with signal D can be done in two ways. Either SWC2 is changed to no longer look at signal D and simply set $A_c = A$, or a check for D can be added in the requirement. Removing the check for signal D in SWC2 would result in the same test as in SingleSWC (although with different signal timings) which is proven valid by SLDV. Adding a check in the requirement can instead cause SLDV to verify the requirement for the current software. Note that in this case, a Unit Delay block must be added to D in the requirement, as SWC2 is executed before SWC3 and signal A_c will therefore have been computed using signal D from the previous time-step. The check with enum value 2 for signal D can be done in multiple ways. However, in order not to miss any counterexamples not found, the check should be added in the least restrictive way, *i.e.* in a way which changes the requirement model the least. An example for this is shown in Figure 6.30. With this requirement model and the software model as shown in Figure 6.28, Requirement 3 is proven valid in approximately one and a half minutes.

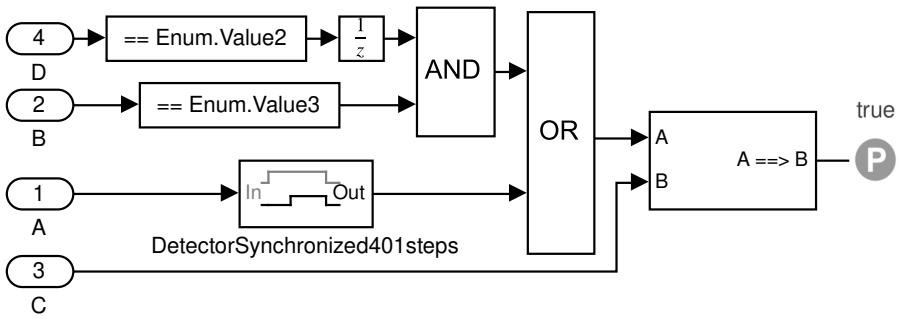


Figure 6.30: Changed Requirement 3 model, with an added check for signal D.

6.4.6 TripleSWC – Test environment with three SWCs

As done in Requirement 1, it is possible to extend the test environment to contain more of the software. The DoubleSWC results show that this is not necessary, but it is interesting to know how the analysis time scales with the model size. With the entire logic used to compute D added into SWC3, the requirement is verified in approximately a minute and a half using rational number approximation. However, the additional analysis to reduce rational number approximations remains unfinished after 15 minutes. The issue is related to a fixed-point input signal to SWC3 which is converted to a single-precision floating-point signal, and then compared to a parameter of the same type. Converting signals between data types with Data Type Conversion blocks in this test environment cause big issues. By removing the Data Type Conversions and adding a floating-point input instead, the model is verified under approximation in the same amount of time. The additional analysis takes a little over a minute to finish, for a total analysis time of approximately 2:45 to verify the requirement for the TripleSWC environment. This environment is visualized in Figure 6.31. The requirement model is the same as shown in Figure 6.30.

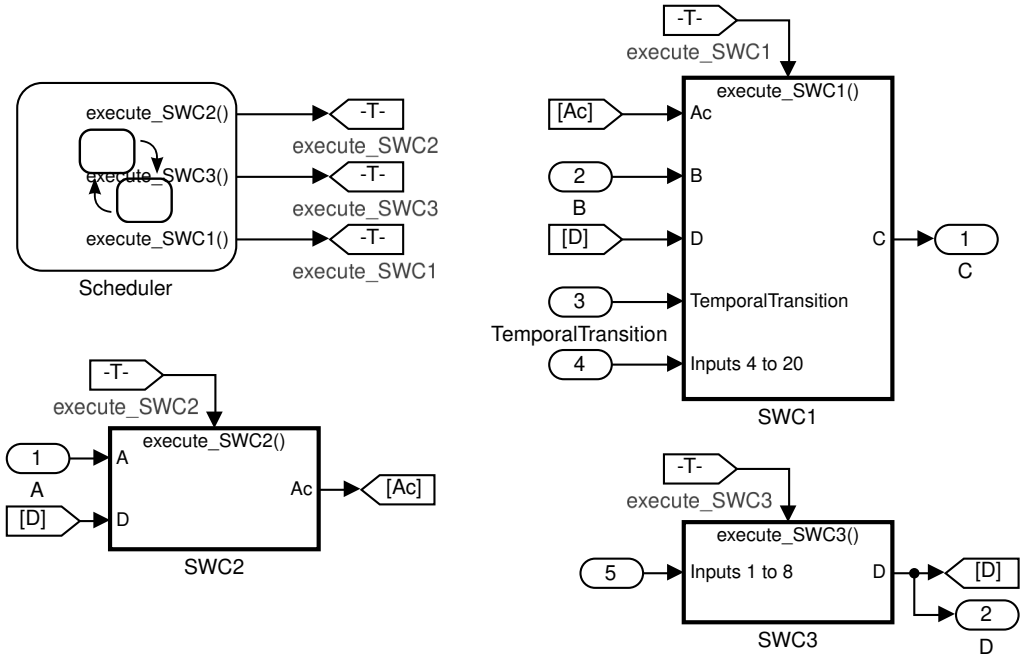


Figure 6.31: TripleSWC test environment structure.

6.4.7 Conclusion

The results from Requirement 3 show that the test environment is essential to achieve good performance in SLDV. The issues surrounding initializations show that the performance can benefit from allowing the software to initialize properly, before any inputs can be given to the system. The results also indicate that signals which are believed to be equivalent can be different, and for this purpose it is important that reductions of the software in the test environments are made correctly.

Analysis times and metrics for the different test environments for Requirement 3 are shown in Table 6.3 and Table 6.4.

Model	**Analysis time**
SingleSWC	0:26
- no Stateflow	0:25
- original Stateflow	0:29
DoubleSWC	
- original requirement	0:38
- augmented requirement	1:24
TripleSWC	2:46 (1:26)

Table 6.3: SLDV analysis times for different test environment models. Time in parentheses is the analysis time for the rational approximation result.

Model	**Number of blocks**
SingleSWC	221 (256)
DoubleSWC	421 (256)
TripleSWC	543 (256)

Table 6.4: Model metrics for the different test environment models. The numbers in parentheses are number of Stateflow objects.

6.5 Requirement 4 – Stateflow timeout

Much of the program flow in the software is dependent on the communication with the charging station. There are different protocols to be followed for different charging standards. The vehicle must be able to handle instances when the charging station does not follow the current protocol. For example, if an expected signal is not retrieved from the charging station, the software should not get stuck in an unwanted state. To handle such errors, the software uses timeouts to abort the charging process if the charging station has not responded within an appropriate amount of time. Requirement 4 ensures one of these timeouts is executed correctly.

Requirement 4

When signal A is set to true, the software should wait for signal B to turn to false. If signal B does not become false within 2.5 seconds, signal C should be set to true.

All signals A, B and C are Boolean signals. Signal B is an input to the system, whereas signals A and C are software outputs. The requirement tests a single SWC which mainly contains a large Stateflow chart. Therefore, test environments with only that SWC will be used for this requirement.

6.5.1 The Stateflow chart

The software component tested by Requirement 4 consists mainly of a large Stateflow chart with 256 Stateflow objects. There are many temporal state transitions, all of which have been replaced by *TemporalTransition* input triggered transitions except for one. The remaining transition is the one Requirement 4 is targeting. In the original software, it waits for 250 ticks, *i.e.* is formatted as *after(250, tick)*. However, for this test environment, its waiting time is reduced to 5 ticks, to reduce the analysis time. The effects of this will be discussed later in this requirement subsection. Since the requirement only tests a single SWC, the test environment only consists of the SWC set as an atomic subsystem (with the function- call trigger removed) connected to the requirement model.

A Stateflow chart with a similar structure to the software chart is shown in Figure 6.32. Note that this is heavily reduced compared to the original chart in the software and does not portray the exact same behavior, although it works perfectly for the discussions in this requirement. However, it may not suffice for further analysis of the software with extended temporal transitions.

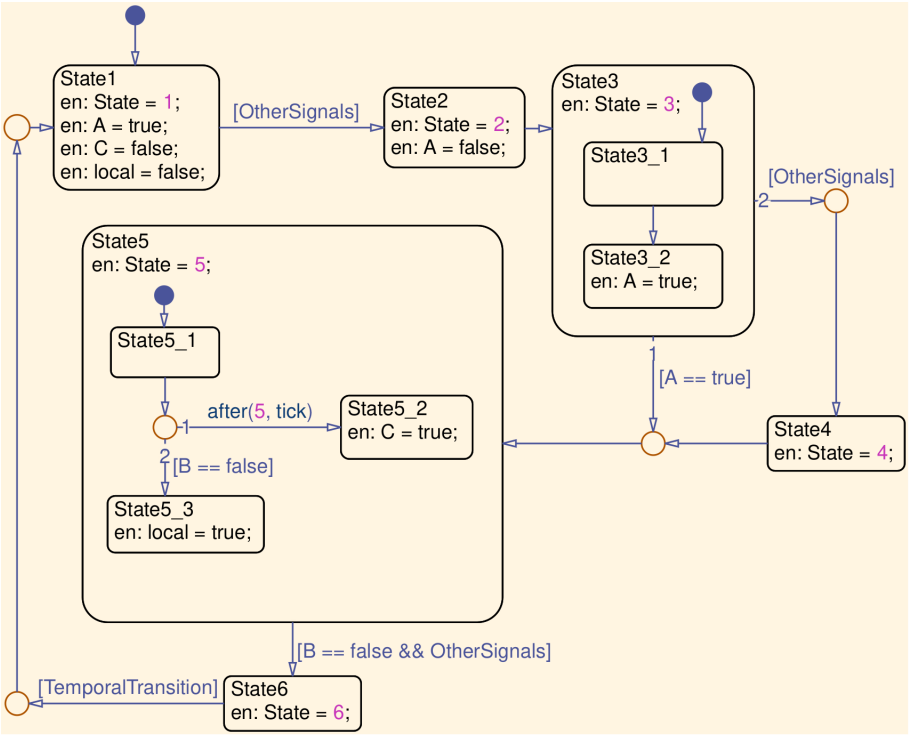


Figure 6.32: Equivalent Stateflow chart structure to that in the SWC mainly tested by Requirement 4. The *after(5, tick)* transition has an original full length of 250 ticks.

6.5.2 The requirement model

Requirements can be modeled in many different ways. The model used for Requirement 4, as seen in Figure 6.33, looks a bit different compared to the other

models. However, this is a simple way to model requirements of the same type as Requirement 4, where a timeout should be triggered if an appropriate response is not retrieved.

An infinite Extender block is used to trigger the check for a timeout. Interpreting the requirement specification can be difficult. Together with the supervisors, it was determined that the requirement refers to a positive edge detection for signal A. The infinite Extender starts the timeout check by outputting *true* into the *Synchronized* Detector block. The appropriate response from the charging station in this case is to set signal B to *false*. If this is not done within 2.5 seconds, the software should set C to *true*. If either signal B is set to *false* or signal C is set to *true* before the Detector block starts outputting *true*, the infinite Extender block will reset and therefore also the Detector block.

The Detector block is set to *Synchronized* to detect consecutive *true* inputs and react instantaneously when the requirement is reset. The Detector is set to detect 5 consecutive *true* inputs (when the temporal transition in the Stateflow chart is set to 5 ticks). If 5 ticks have passed and B is not set to *false* or C is not set to *true*, the Detector block will start outputting *true*, thus falsifying the requirement. The Delay block for two time-steps is added to wait for one time-step after the signal A step is triggered, and to wait for another time-step to allow the Stateflow chart to move into the correct state where the checks for signals B and C are done; remember, signal A is an output and a step in signal A can therefore not be both detected and generated in the next time-step.

There is an issue with this requirement model. It is entirely possible that C is set to *true* immediately and not after 2.5 seconds. This behavior will pass the verification. However, Requirement 4 does not specify that signal C shall not be set to *true* until 2.5 seconds have passed. Assuming the requirement is proven valid, the Detector length can simply be reduced until the requirement is falsified; at that point, it is possible to determine how long the software will wait for before setting C to *true*.

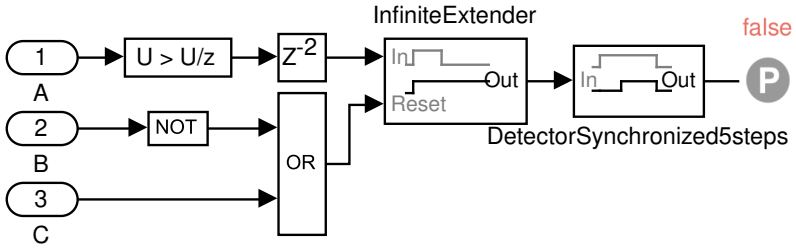


Figure 6.33: Requirement 4 model.

6.5.3 Falsifying the original requirement

With the test environment as explained above, the temporal transitions removed apart from the one set to 5 ticks, and the requirement shown in Figure 6.33, the requirement is falsified after just over two minutes. This test environment contains a total of 317 blocks.

The counterexample given by SLDV, with a total length of 20 time-steps, shows a way to bypass signal A being set to *true* in the equivalence of State3_2. The Stateflow chart will simply go through State2 and enter State3. before State3_2 is entered, however, the input signal *in* is set to *true* by SLDV so that A is kept false. The chart will then go through State5 and finally enter State1 again. This will trigger the positive edge detector in the requirement. At this point, no inputs are really needed; just following the transitions which are executed automatically without specific inputs will cause the Detector block in the requirement to eventually output *true*, falsifying the requirement.

In the limited Stateflow chart shown in Figure 6.32, the requirement is only falsified for a few time-steps before either B is *false* or C is set to *true* after the temporal transition. However, the original Stateflow chart in the software is much bigger. Hence, the counterexample was ported from the SLDV optimized chart with *TemporalTransition* input triggered transition and the shorter 5 tick transition, to the original chart with the correct temporal transitions. With the changed timings to account for the temporal transitions, the counterexample falsifies the requirement for several seconds, showing that for the given software and requirement models, the software does not fulfill the specified requirement.

6.5.4 Verifying the requirement through a state check

The reason behind the falsification can be linked to bypassing State3_2 by setting *in* to *true* when in State3_1. Later, a positive edge will be discovered when the chart enters State1 again. State1 is meant to initialize the local and output variables of the chart, and the transition into State2 is only triggered by chart inputs. Since there is no timeout detection in State1, the positive edge encountered in State1 is likely unintentional.

Due to the size of the Stateflow chart, the equivalence of the bypass of State3_2 in the original software is much more complicated and in-depth than the one shown here. Therefore, it can prove difficult to solve this issue as there may be many other requirements that need to be verified at the same time. However, it is possible to isolate the issue and try and find all the errors in the software model.

As explained in Requirement 2, it is possible to add a State check to exempt certain behavior from the requirement. In this case, the issue was caused by a positive edge in State1. Therefore, the requirement can be augmented by a State check which ensures the requirement condition is not activated if the chart is in State1. The augmented requirement model is shown in Figure 6.34. This requirement will ignore the positive edge when entering State1.

With the added State check, the augmented requirement is verified by SLDV in a little less than four minutes. Note that this verification is for the environment with the chart with *TemporalTransition* inputs and the five tick temporal transition. Nonetheless, it shows that, with the exception of the counterexample in which a positive edge is detected at State1 through the bypass of the equivalence of State3_2,

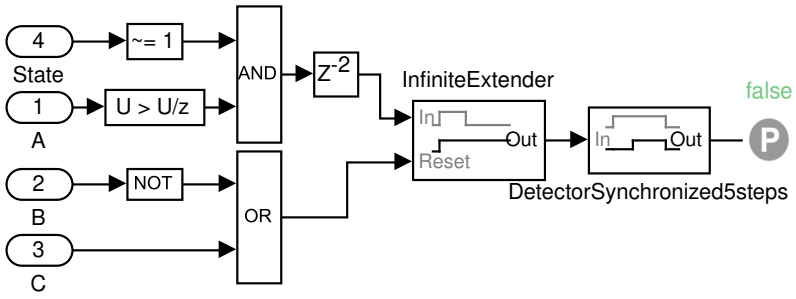


Figure 6.34: Requirement 4 model augmented with a state check.

the requirement is fulfilled in direct relation to signal A being changed from *false* to *true* in the Stateflow chart.

Despite the verification, it is still not known if the requirement is fulfilled for the full temporal transition duration of 250 ticks. Unfortunately, analysis times quickly become large when increasing the temporal logic length, as shown for Requirement 2 in Figure 6.22. For this test environment, increasing the Detector and temporal transition lengths by five ticks to ten, the requirement is instead proven valid after seven minutes. Adding ten additional ticks to the temporal lengths causes the analysis time to increase to 28 minutes. These results indicate that trying to verify the requirement for the full duration of 250 time-steps would require a very large amount of time. Additionally, the size of the Stateflow size can cause other issues as well. It is possible that as the temporal transition times increase, cyclic paths in the chart become available so that a counterexample is found by SLDV by cycling through the charging process multiple times, before 250 time-steps have passed. Other bypasses could possibly be found which would end up falsifying the requirement for the full temporal lengths.

6.5.5 Showing absence of cyclic counterexamples

The previous results indicate that the requirement is fulfilled, but due to temporal scaling it is indeed possible that for the full duration of the temporal transition, there are counterexamples which step through multiple cycles of the charging process, falsifying the requirement in an unintuitive manner. However, it can be shown that this is not the case. A simple test can be set up to show that whenever signal A steps from *false* to *true* and it does not occur in State1, then the chart must enter State5 in the next time-step. The test to prove this is shown in Figure 6.35. This test is verified in 30 seconds, showing that whenever a step of signal A occurs and the chart is not in State1, the chart will always be in State5 in the next time-step.

The next step in providing a proof to the absence of cyclic counterexamples is to show that the requirement cannot remain active as the chart leaves State5. This can be done in a multitude of ways. One way is to change the requirement model in Figure 6.34, to ensure the infinite Extender is never active once the chart leaves State5, which can only be done into State6. Note that the Detector block has been removed; this is done to show the absence of cyclic counterexamples, as for these

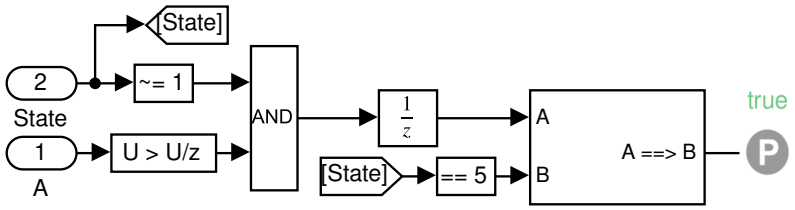


Figure 6.35: Model used to test entry into State5.

the Detector might not be active as the chart leaves State5 but the Detector can be activated through the infinite Extender later. The model to test this behavior is shown in Figure 6.36.

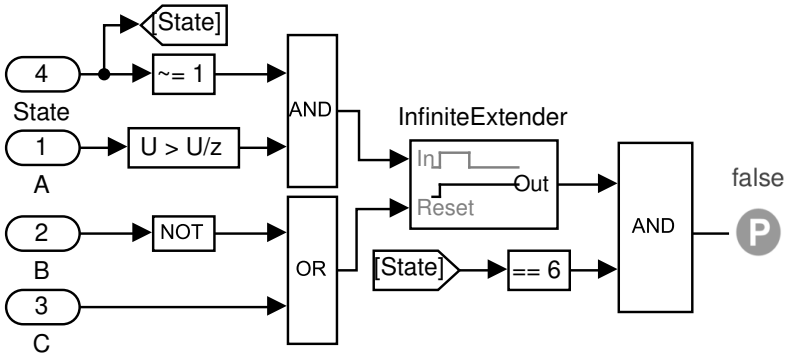


Figure 6.36: First model used to verify that there are no cyclic counterexamples for Requirement 4.

Unfortunately, this test model creates some issues for SLDV and remains undecided after four hours of analysis. Instead, the test proven valid in Figure 6.35, showing positive signal A steps not in State1 are always followed by entering State5, can be utilized. A stronger proof can be provided here, which instead states that once entering State5 (which is not necessarily preceded by a signal A step), signal B must be *false* or signal C must be *true* before going into State6. The model for this is shown in Figure 6.37. This test is much easier for SLDV to handle, and is verified in two minutes by SLDV. All of these tests combined show that given proper scaling in the temporal logic in State5, the requirement cannot be falsified as State5 will not be exited before either signal B is set to *false* or C is set to *true*. Note, however, that the counterexample with the bypass of State3_2 followed by a step of signal A in State1 still remains.

6.5.6 Summary and conclusion

The initial requirement model was falsified by SLDV, where a counterexample was given. This counterexample can be very valuable to developers, as it shows every step of how the unintentional behavior was achieved. By adding delays between signals, the counterexample can be ported from the SLDV optimized model to the real software model with full temporal transition lengths. If the reductions done to optimize the model for SLDV, this should always be possible.

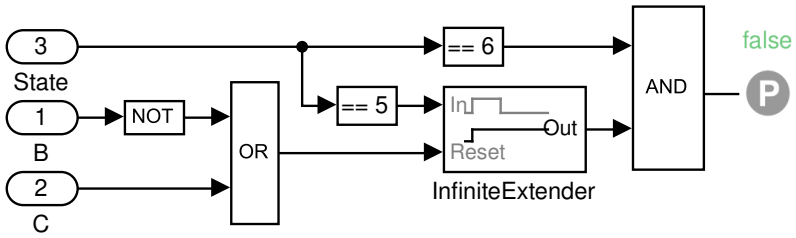


Figure 6.37: Improved model used to verify that there are no cyclic counterexamples for Requirement 4.

More tests were done on the model to further investigate the issues, and it was found that the only falsifying source in the software was State1 setting signal A to *true*. This simple extension to the requirement test shows that if the developers can ensure signal A is never *false* when entering State1, that error goes away. However, changes added to the Stateflow chart can cause falsifications to appear in other states instead.

Scaling up the temporal logic shows a large increase in analysis time, and is not a good strategy. However, it was shown that the requirement will never be active as the Stateflow chart leaves State5. It was also shown that if signal A goes from *false* to *true* and it is not done by State1, then the chart will always enter State5 in the next time-step. Since A is never set in State5, the only way to activate the requirement is in State1 and State3_2. Therefore, if the requirement is activated it must be deactivated in or when leaving State5. The temporal timings have been shown to work properly for scaled down temporal logic, indicating that signal C will be set in the correct time-step. Thus, the only thing remaining is to show that the requirement is deactivated if the temporal transition is never allowed to finish. The only possibilities for this is if the chart transitions into State5_3, or into State6. Both of these transitions require B to be *false*, which also resets/deactivates the requirement. Hence, a few "help tests" proven by SLDV has allowed a proof to be made for the requirement, if the error for State1 is to be exempted.

6.6 Requirement 5

Requirements 1 and 3 are designed to ensure faults are detected by the software. However, faults should not only be detected but also handled. The purpose of Requirement 5 is to ensure the charging process is aborted in a specific state, if faults are detected by the software in that state.

Requirement 5

If the Stateflow chart in SWC1 is in State2, then the software should abort the charging process by setting signals E and F to false if at least one of signals A, B, D become true, or if signal C has enum value 3.

Signals A, B and C are inputs to the software. Signals D, E and F are software outputs. All signals are Boolean except C which is an enum signal.

6.6.1 The software component

Requirement 5 only tests a single software component; signals A, B and C are all inputs to the SWC in question (denoted SWC1 in this requirement), and signals D, E and F are outputs from the same software component. Thus, a test environment is set up for a single software component: SWC1. The software component mainly consists of a large Stateflow chart but also includes some temporal logic and closed-loops, which will come into play for the sake of verifying this requirement. An equivalent structure of SWC1 and its Stateflow chart can be seen in Figure 6.38.

Since all inputs to the requirement are either also inputs to or outputs from SWC1, a test environment is set up with only SWC1 and the requirement model. All synchronizations of common signals are done in the software before SWC1 is executed. Consequently, no signal synchronization is needed and the Stateflow Scheduler chart in the software can be discarded.

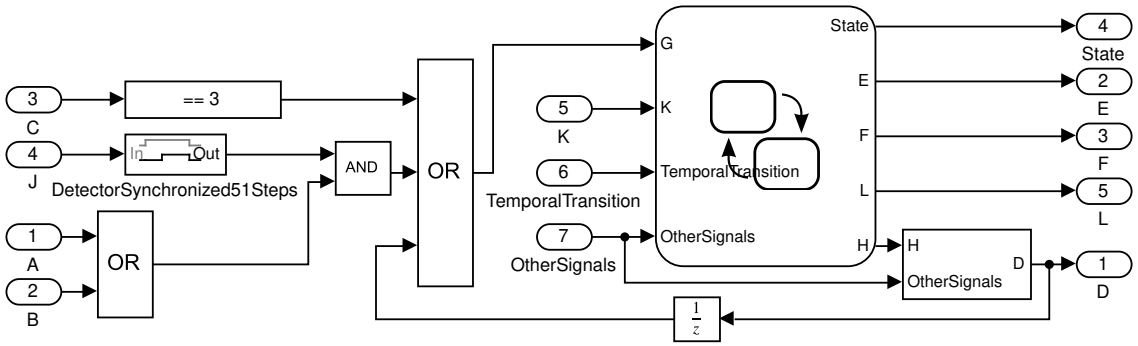
6.6.2 The original requirement model

The original requirement model used to test Requirement 5 at Volvo Cars today is shown in Figure 6.39, although with a few changes. A Unit Delay block has been added to fix signal timings due to the feedback of signal D. Since the requirements are designed for software level testing, they should only specify signals which are either inputs or outputs, but not internal signals. Therefore, the State output from the Stateflow chart is not used in the existing model. Instead, other inputs and outputs are used to try and track the current state. This has been done with signal K. However, falsifications can be given when in State1. Thus, another check for signal L is added to ensure the chart has moved from State1. A Unit Delay block is added since transitions from State2 cannot be done while entering State2. This results in the requirement model in Figure 6.39.

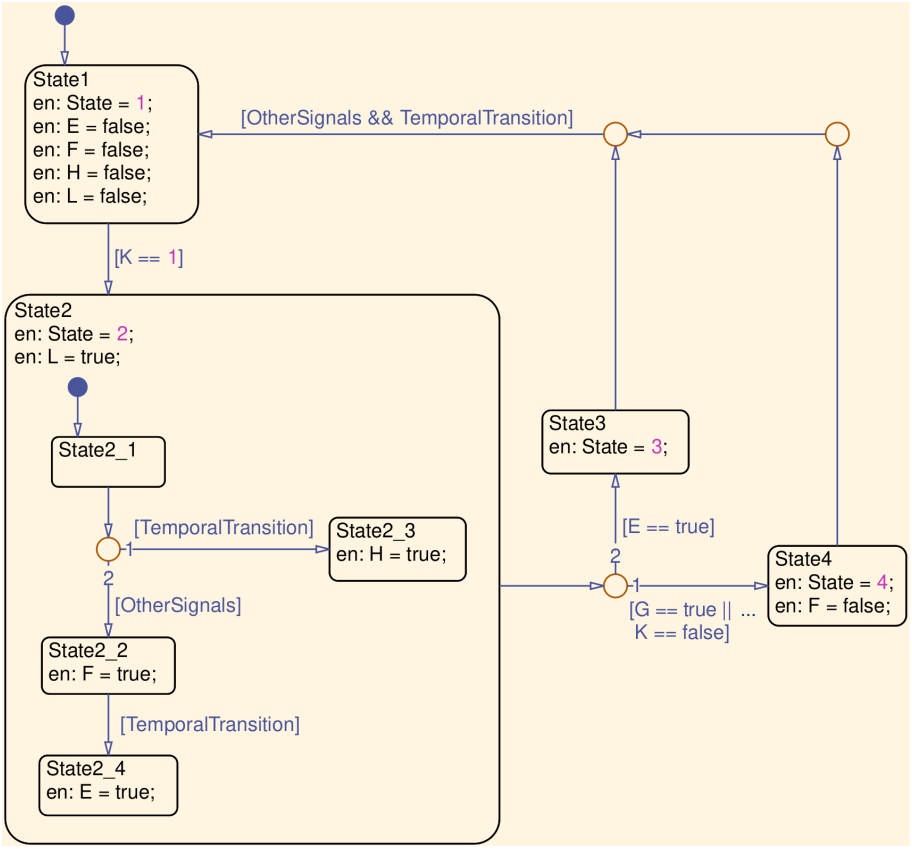
For the original requirement model in Figure 6.39, SLDV generates a counterexample in approximately 1 minute. The counterexample shows how signals A and B are unable to trigger a transition from State2 to State4; this is due to the Detector block at signal J in the SWC. Two simple solutions are to re-specify the requirement, or to remove signal J and the Detector of signal J.

6.6.3 Falsifying the requirement using violation detection

Using the latter of the two above-mentioned strategies, SLDV is unable to produce results within 10 hours. A possible cause of this is the difficulty of tracking States using other signals than the *State* output. However, the *Proving Strategy* setting can be changed from *Prove* to *FindViolation*; this is a much smaller problem as SLDV will only have to show that no counterexamples exist for x time-steps, rather than proving complete absence of counterexamples. With the default maximum number of 20 time-steps, SLDV finds a counterexample in 40 seconds. The generated counterexample requires many details of the software to explain. Hence, in order to protect the intellectual property of Volvo Cars, the explicit counterexample is omitted. However, the smaller software in Figure 6.38 with equivalent structure to



(a) Equivalent, reduced structure of SWC1.



(b) Equivalent, reduced Stateflow chart inside SWC1.

Figure 6.38: Equivalent software structure of SWC1, reduced for simpler visualizations.

the charging control software, albeit reduced, is also falsifiable by SLDV. The details of the counterexample are not strictly relevant, as the error occurs when the chart is in another state than State2, whereas the requirement specifies what should happen in State2. In short, the requirement model does not work correctly, but tries to detect counterexamples for incorrect states.

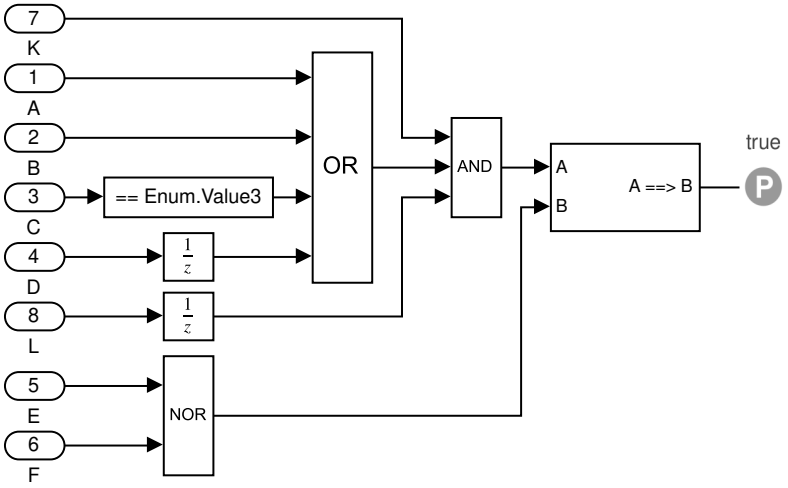


Figure 6.39: Altered version of the original Requirement 5 model, with added signals to track Stateflow states without using internal signals.

6.6.4 Updating the requirement model

Optimally, the requirement would be specified only in terms of software inputs and outputs, as the intent is to have an observer which cannot see internal signals in the software, only inputs and outputs. However, this is not the case for Requirement 5. As such, the requirement model can be reworked to instead look at the *State* output specifically. The reworked requirement model is shown in Figure 6.40. Note that for a complete proof for a "real" software requirement which only specifies input and output signals, it can be easier to use this reworked requirement model, and show an equivalence between $State == 2$ and the signals to track State2. This was done in a similar fashion (although not completely equivalently) for Requirement 4.

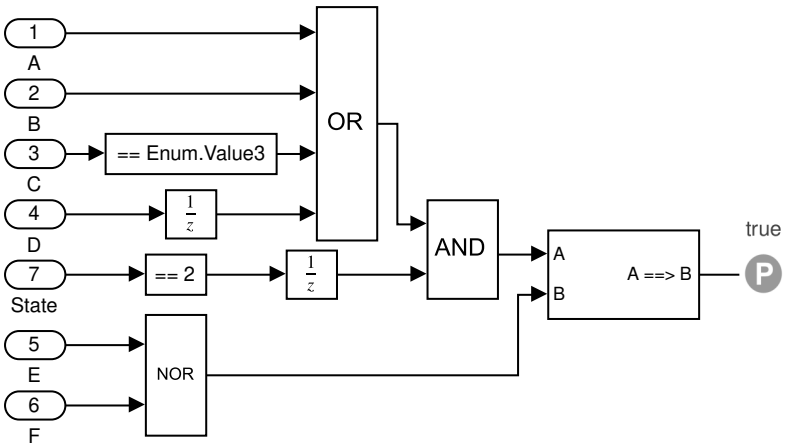


Figure 6.40: Requirement 5 model, using internal *State* signals to ensure the requirement is activated in the correct state.

6.6.5 Finding and fixing the Stateflow error

Using the reworked requirement model, an error related to State2 in the chart is found by SLDV in approximately one and a half minutes (after setting the *ProvingStrategy* back to *Prove*). Assume the chart is in State2_4. If signal G is *false*, then the requirement is not triggered and the chart moves into State3. However, if G becomes *true* in the same time-step as the chart would otherwise have entered State3, it will instead go into State4. Signal E is never set to *false* again, hence the requirement is falsified.

The solution to this is very simple: adding a line that sets E to *false* on entry to State4 will ensure E is never *true* when the chart exits State2 due to G being *true*. The updated SWC structure and chart with the mentioned fixes are shown in Figure 6.41. The requirement is valid under approximation for this updated SWC in 40 seconds, and the additional analysis to reduce rational number approximations completes after an additional 15 minutes; note that floating-point and fixed-point signals are baked into the *OtherSignals* input, hence the approximations.

6.6.6 Conclusion

For Requirement 5, a test environment is created which causes issues for SLDV to handle when using the *Prove* strategy. The differences in computation times show that *FindViolation* can be a good proving strategy in some instances, before the model is proven valid within the specified bound.

A systematic approach to isolate and remove errors in the software is shown. By performing similar steps to what is shown for this requirement, the cause of the error can be pinpointed and hence also resolved. The tests carried out indicate that SLDV can have difficulties using signals to track states, especially if these signals are used extensively in the chart. It is often more efficient to use unique state identifiers instead.

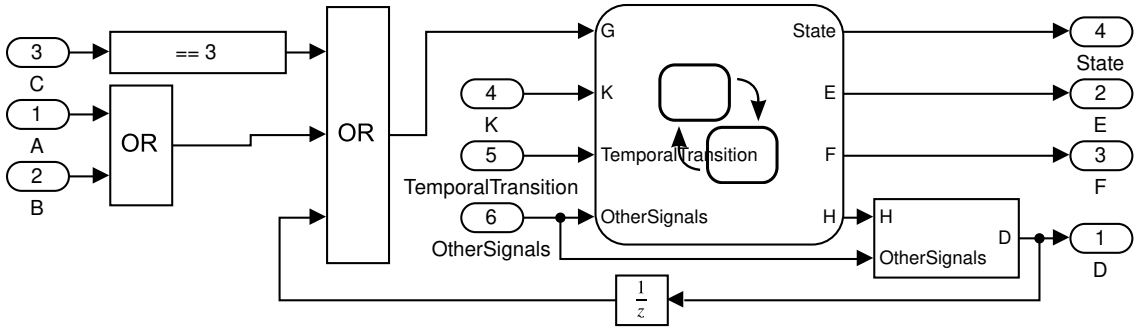
6.7 Requirement 6 – Continuous reference signal

So far, only requirements for discrete behavior have been dealt with. However, there are continuous signals in the charging control software as well, related to *e.g.* charging currents and voltages. The purpose of this requirement is to check if a reference signal is disabled for specific inputs, or if the rate is within a specified bound for other specific inputs.

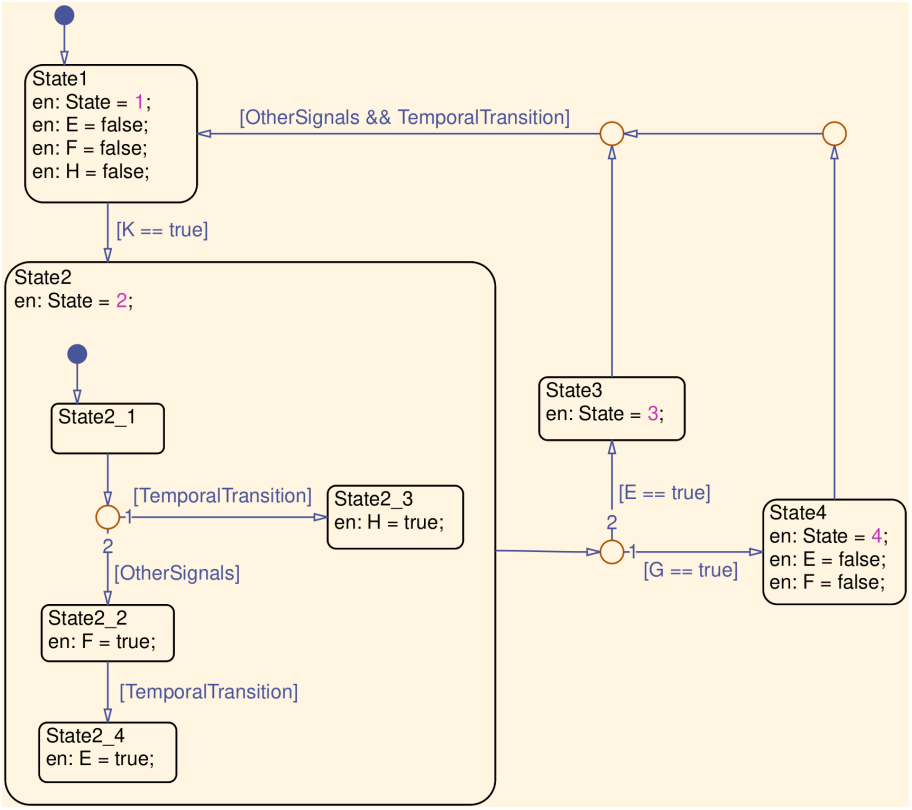
Requirement 6

Whenever signal A has enumeration value 2, signal C shall:

- *be set to 0 if signal B is false*
- *change with at most 20 units/s if signal B is true.*



(a) Updated SWC1 structure surrounding the Stateflow chart.



(b) Updated SWC1 Stateflow chart.

Figure 6.41: Updated, reduced SWC1 structure to comply with Requirement 5.

All signals A, B and C are outputs from the software, although A and B are also inputs to the SWC that computes signal C. Signal A is an enumeration signal, signal B is Boolean and signal C is a uint8.

6.7.1 The software component

With the logic to compute signal C isolated in the SWC (denoted SWC1 for this requirement), there are 10 inputs to the SWC: one enum (signal A), 5 Booleans (one of which is signal B), and 4 uint8 signals. The difference in this SWC and requirement from the previous ones is that signal C is computed using continuous signals. The computations are done internally with single-precision floating-point

values; the uint8 inputs are converted to single-precision signals for this purpose, and the result is converted back into a uint8 for output signal C.

Since condition signals A and B are inputs to SWC1 which computes signal C, a single SWC test environment is created to test Requirement 6. SWC1 in this test environment is shown in Figure 6.42. Note the "single" specifier for inputs 3–6. Since the Data Type Conversion blocks were found to be a big issue in Requirement 1, these are removed and the continuous signals originally represented as uint8 are instead replaced by single-precision signals, *i.e.* the same signal type which is used internally to compute signal C. Running property proving on the original model with all Data Type Conversion blocks in place can be done for more than 10 hours without any conclusion from SLDV. Note that for correct input ranges, the minimum and maximum values for the single-precision inputs must be specified as 0 and 255 respectively. Thus, the uint8 signals are discarded in favor of single-precision floating-points, and the single-precision inputs to the test environments are limited to the intervals $[0,\ 255]$.

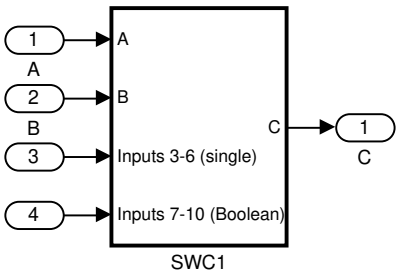


Figure 6.42: Software component tested in the SingleSWC test environment.

SWC1 contains a total of 504 blocks. The operations performed in the SWC are typically Boolean algebra, linear operations with continuous signals, signal routing with Switch blocks, filtering and saturation (clamping) of internal signals.

6.7.2 The requirement model

The original requirement model currently used to simulate tests for Requirement 6 at Volvo Cars today can be seen in Figure 6.43. However, in order to simplify for SLDV and to make the the model more explicit and readable, it can be remodeled into two different sub-requirements, as seen in Figure 6.44a. An important change from the original requirement is also made in the CheckRate subsystem. In the original requirement model, the signal C input is of type uint8, whereas its corresponding signal in the latter requirement model is a single-precision signal. As such, computing the derivative in the original requirement model is much more complicated, as the smallest integer change from one time-step to the next (apart from no change) is equivalent to a derivative of 100. However, with the removal of the Data Type Conversion blocks, all continuous signals are instead represented by single-precision values which makes the derivative computation much easier. The updated derivative check is shown in Figure 6.44b. Note that this representation changes the

behavior of the software somewhat, but it can be useful to have this representation anyway since it allows for SLDV to analyze the model much faster.

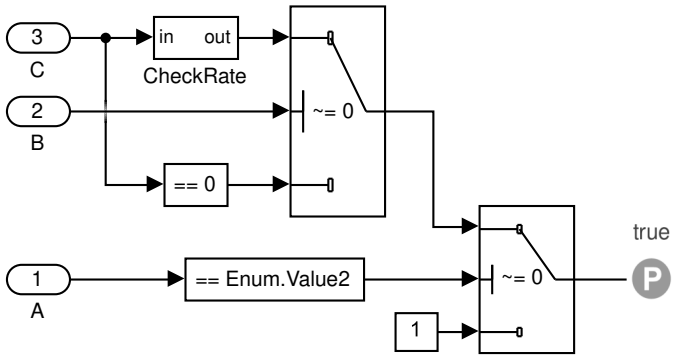
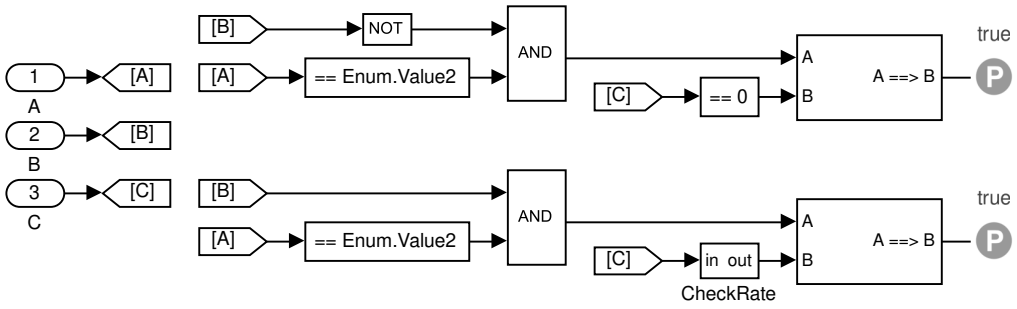
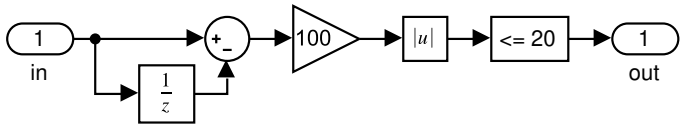


Figure 6.43: Original Volvo Cars model for Requirement 6.



(a) Updated Requirement 6 model, split into two sub-requirements.



(b) CheckRate subsystem.

Figure 6.44: Model for Requirement 6, updated for two sub-requirements and for the use of single-precision signals.

6.7.3 Violation detection without input filtering

In the full software model, 2 of the 4 continuous input signals are passed through low-pass filters, to reduce the noise in the inputs. To allow SLDV to reach a conclusion faster, these LP-filters are removed for this subsection. Proofs obtained for the non-filtered inputs are stronger than using filtered inputs, as the LP-filters limit the inputs but the same behavior can be reflected by non-filtered inputs.

Running Property Proving on the test environment in Figure 6.42 with the optimized requirement in Figure 6.44 takes a long time to finish. The Proof Objective for $B == \mathit{false}$ is valid under approximation after less than a minute. However, the additional analysis to reduce the rational approximation takes a full hour to complete (although this includes the analysis for the second sub-requirement). The other Proof Objective is falsified after 37 minutes. Because the first Proof Objective (for

$B == false$) is proven valid by SLDV, that sub-requirement is verified and does not need to be further checked.

An apparent issue here is the long analysis times for the second sub-requirement, due to the continuous signals. To simplify for SLDV, the Proving Strategy can instead of *Prove* be set to *FindViolation*. The counterexample given by SLDV earlier is 4 time-steps long, so the maximum violation length is set to 5. For the same model, a counterexample is instead returned after 10–15 seconds of computation. To see how much the derivative limit can be exceeded, it can be increased in the requirement to 40 instead of 20. For this relaxed requirement, SLDV finds a violation in 30 seconds.

6.7.4 Adding a requirement tolerance

It is possible that the requirement is only falsified for single time-steps at a time. Hence, an Extender block can be added to the output to the Implies block, which extends *true* signals for one second. This will cause the Proof Objective to ignore *false* signals for single time-steps, whereas consecutive *false* values will falsify the Proof Objective. When using the Extender as a tolerance block, the requirement is falsified in 40 seconds for both derivative limits 20 and 40.

6.7.5 Using constant values for signal A

In all the given counterexamples, signal A has changed to and from enumeration value 2. Hence, it is interesting to see what happens when signal A is kept constant. In order for the requirement to activate, it is here kept to a constant value of 2. With a constant signal A and without the requirement tolerance, a counterexample is found by SLDV in 10 seconds for a limit of 20, and 30 seconds for a limit of 40. Adding in the requirement tolerance makes it trickier to find counterexamples, but SLDV is able to find a counterexample after 2 minutes and 20 seconds for a derivative limit of 20. For a limit of 40, the requirement is undecided with counterexample after 2:15, and is falsified after the additional rational number approximation analysis is complete after another 16 minutes. Hence, there are still errors even when the requirement is always enabled. Note that the same test for signal B is not carried out, as some of the counterexamples returned by SLDV have constant B inputs.

6.7.6 Adding input filtering

As explained above, 2 of the 4 continuous input signals are LP-filtered in the software. To see if the filters can cause the requirement to turn valid, these LP-filters are added back in. Thus, the test environment is slightly altered. The software in the updated test environment is shown in Figure 6.45. The Scheduler executes SWCs in the same order as the outputs. With tolerance disabled and without a constant signal A, the requirement is falsified after 50 seconds for a derivative limit of 20. With enabled tolerance, the sub-requirement is undecided due to nonlinearities after 6 minutes. The additional analysis to reduce rational number approximations remains unfinished after 1 hour. With constant signal A set to enumeration value 2, the requirement is falsified after 1:31 without the Extender tolerance, but is valid

within the bound of 5 time-steps after approximately 9 and a half minutes with the Extender block enabled. Increasing the violation limit to 6 time-steps causes the requirement to be valid within bound after 25 minutes.

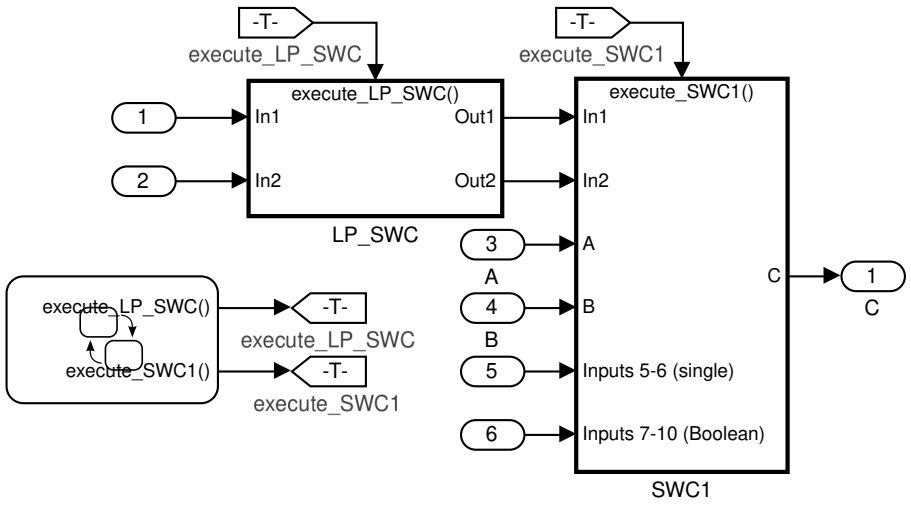


Figure 6.45: Test environment for the added input LP-filtering. SWCs are executed in the same order as the outputs from the Scheduler chart.

6.7.7 Conclusion

The results for Requirement 6 show that the software does not fulfill the second sub-requirement, although the first sub-requirement is proven relatively easily. The *FindViolation* strategy can be very effective in finding falsifications to continuous requirements when first investigating these. The results also show that the maximum violation steps setting can greatly affect the analysis times for SLDV. Strategies for actions to take to test and isolate possible causes for the error in the code are shown. These strategies show that a flickering signal A can make it difficult for SLDV to reach a conclusion regarding verification or falsification. Adding the Extender block as a tolerance is necessary for the Proof Objective to be verified by SLDV for the given proving strategy. The tests with and without LP-filters on two of the input signals show that the inputs must be filtered if the requirement is to be fulfilled.

A verification is finally obtained within the violation bounds when tolerance and a constant signal A input are enabled. The results suggest that flickering of signal A makes it easier for SLDV to falsify requirements but more difficult to verify requirement, although it is not shown that it matters to the final conclusion by SLDV. The results suggest flickering of signal A can cause robustness issues for the software, and that an LP-filter must be applied as done in Figure 6.45, and an Extender block must be added to represent a tolerance of falsifications for single time-steps at a time. Note that these results do not prove that the requirement is verified for these settings, but information is gathered for developers to look at, and can together with the counterexample be very valuable in the process of fixing the errors found.

A summary of the results obtained for Requirement 6 are shown in Table 6.5.

Filtered inputs	**Proving Strategy**	**Limit**	**Constant signal A**	**Enabled tolerance**	**Result**
✗	P	20	✗	✗	Falsified (37:00)
✗	FV (5)	20	✗	✗	Falsified (0:10–0:15)
✗	FV (5)	40	✗	✗	Falsified (0:30)
✗	FV (5)	20	✗	✓	Falsified (0:40)
✗	FV (5)	40	✗	✓	Falsified (0:40)
✗	FV (5)	20	✓	✓	Falsified (2:20)
✗	FV (5)	40	✓	✓	Falsified (2:15, 18:30)
✓	FV (5)	20	✗	✗	Falsified (0:52)
✓	FV (5)	20	✗	✓	Undecided due to nonlinearities (6:06, >1:00:00)
✓	FV (5)	20	✓	✗	Falsified (1:31)
✓	FV (5)	20	✓	✓	Valid within bound (9:22)
✓	FV (6)	20	✓	✓	Valid within bound (25:08)

Table 6.5: Summary of the SLDV analysis results for Requirement 6. Numbers in parentheses in the Result column are the analysis times. If there are two times reported in parentheses, the first time shows the analysis time for an approximation result, and the second time shows the analysis time after the additional analysis to reduce rational approximations is complete (total analysis time). The entry with a time >1:00:00 is unfinished after 1 hour.

6.8 Implementation requirements

The intent of the previously presented requirements is to define borderline behavior of the software, *i.e.* expected outputs from the software given specific inputs and other outputs of the software. This was not strictly the case for all requirements, but none of the requirements specify what should happen internally in the software on a detailed level. Requirements for this purpose can be labelled Implementation Requirements, and can be used to verify that the implementation contains the desired behavior. The requirements can be simple and deal with very small parts of the software, yet be very important. For instance, bugs are found in the software for certain custom library blocks designed to deal with temporal logic; this is explained in Section 6.1. This can be detected relatively easily with implementation requirements for those specific blocks.

Five simple requirements will be shown here. These were generated by looking at the implementation and did not exist beforehand; their sole purpose is to show a Proof-of-Concept of implementation requirements in the charging control software. These were generated with respect to a Stateflow chart in an SWC, hence the test

environment will only contain that SWC and the requirement models. The requirements are made for states early, in the middle of and in the end of the Stateflow chart program flow.

6.8.1 Early requirements

For the early states, two requirements are set up. Their purpose is to show that the correct transitions are made for different inputs. A reduced software component can be seen in Figure 6.46, used to visualize the behavior the requirements are verifying. The requirements are modeled so that if the previous chart state was State2 and one of the outgoing transitions are triggered (either through chart outputs or inputs), the chart should move into the correct state.

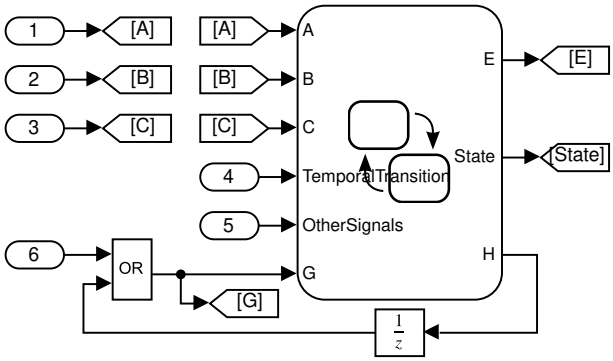
The first requirement, modeled as in Figure 6.47a, checks the left transition from State2 to State3. Since this transition is the first transition in the outgoing execution order, no additional checks for other transitions have to be made. The requirement states that if either signals A or B are *false*, or signals C or G are *true*, the chart will make the transition into State3 (if coming from State2). This requirement is proven valid by SLDV after 30 seconds.

The seconds requirement, modeled in Figure 6.47b, checks the transition into State4. Because this transition is the second in the execution order from State2, the left transition must also be checked. If the left transition is not triggered but the right one is, the requirement will be activated. The requirement is fulfilled if the chart moves into State4 in this case. This requirement model is verified by SLDV after 30 seconds.

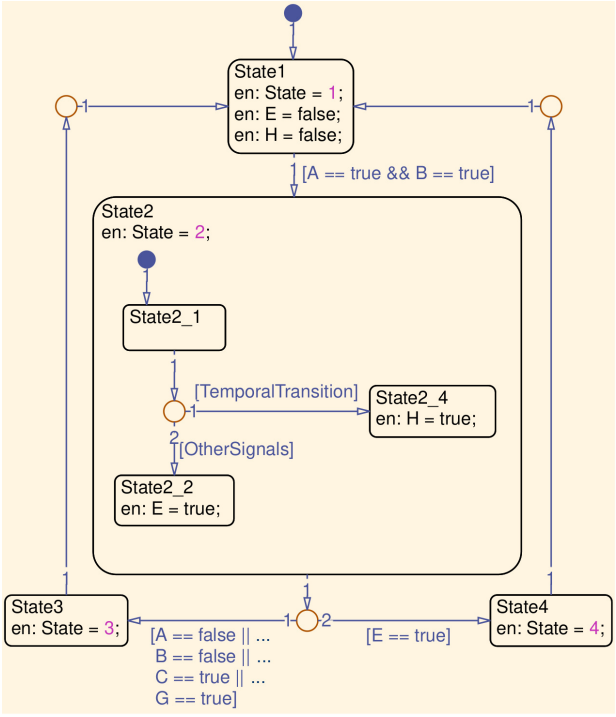
Note that the transition into State4 is falsified if the original software model is used. Input signal C is not submitted directly into the software but is fed through an SR latch instead. Consequently, a positive signal C can be stored by the SR latch and the transition into State3 is executed in favor of into State4. Since the input signal C can be set to *false* in this case but is overridden by the stored *true* value in the SR latch, the requirement is falsified. Hence, the SR latch is omitted in the model shown in Figure 6.46a and the Stateflow chart input is used in the requirement model instead, causing it to be verified by SLDV. However, these simple "errors" can be very useful to find and should be noted and tested for. The reduction was done to showcase the ability of SLDV to verify transitions which are shadowed by other transitions, as in Figure 6.46b.

6.8.2 Middle requirements

Three requirements are set up to test the behavior in the middle of the Stateflow cycle. The requirements are used to detect unique signal patterns, and from those determine the current state of the chart. These requirements are modeled in Figure 6.48. Note that the signal names and the *STATE_ID* in the requirement models represent different signals and states in the actual software. As the behavior of the signals can be quite complex, the software is omitted as a detailed explanation would



(a) Model showing the closed loop for the Stateflow chart.



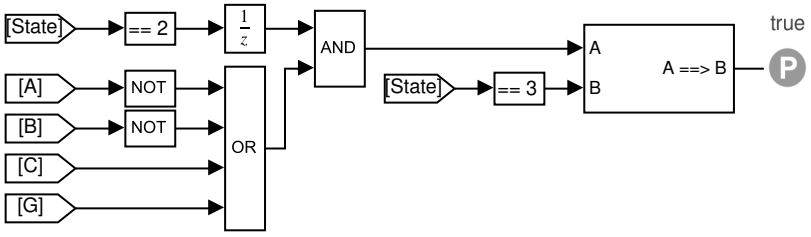
(b) Stateflow chart for implementation requirements early in the Stateflow cycle.

Figure 6.46: Software model used to test implementation requirements early in the Stateflow cycle.

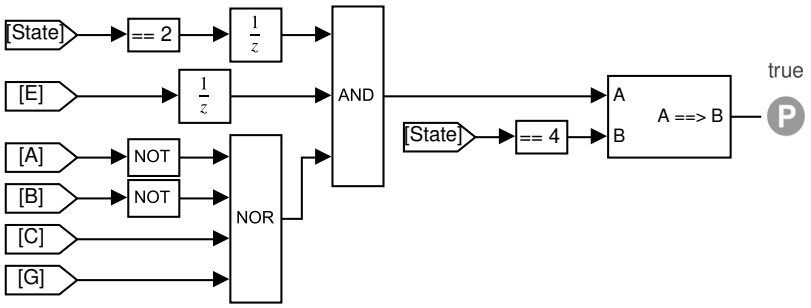
reveal much of the confidential software. Proving all these requirements at the same time takes SLDV about 30 seconds.

6.8.3 Late requirement

This requirement is intended to determine the state by looking at signal A. Signal A is set three times in the Stateflow chart, but only once to *true*. Hence, if a positive edge is found for signal A the current state can be determined, and similarly for a negative edge. A heavily reduced Stateflow chart showing this behavior is shown in Figure 6.49a. A requirement set up to check if a negative edge on signal A implies

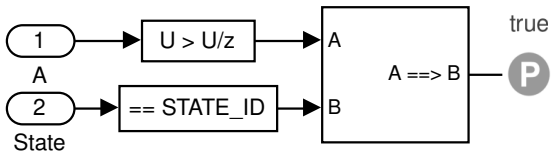


(a) Requirement model for the left Stateflow transition.

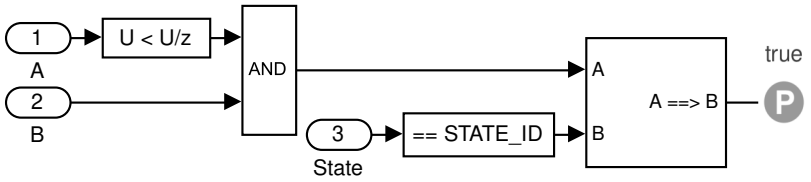


(b) Requirement model for the right Stateflow transition.

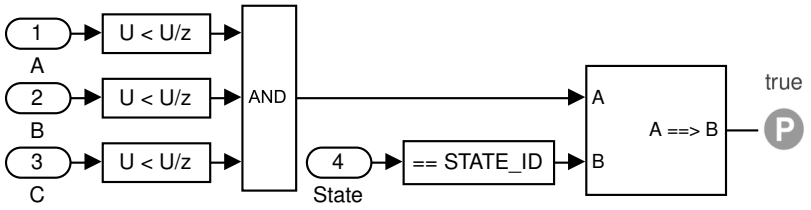
Figure 6.47: Implementation requirement models for the states early in the Stateflow cycle.



(a) Middle requirement 1 model.



(b) Middle requirement 2 model.

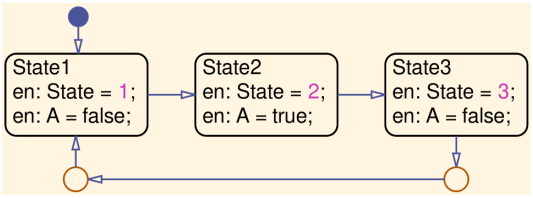


(c) Middle requirement 3 model.

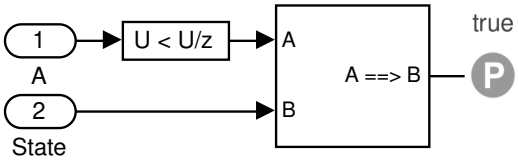
Figure 6.48: Implementation requirement models for states in the middle of the Stateflow cycle.

the current state is State3 is shown in Figure 6.49b. The equivalent requirement for the original SWC is verified in 30 seconds. Although the requirement is small, it shows that it is possible to verify requirements for states very late in the Stateflow

cycle, such as State3, in a short amount of time.



(a) Reduced Stateflow chart showing the basic behavior tested in the late implementation requirement.



(b) Implementation requirement model used to test states late in the Stateflow cycle.

Figure 6.49: Stateflow chart and requirement model used to test states late in the Stateflow cycle.

6.8.4 Conclusion

The made-up requirements show that it is possible to verify requirements in different stages in the Stateflow chart. However, issues are still encounterable; setting up requirements to track four or more signals can cause SLDV to analyze for several hours without coming to a conclusion, if the signals are often set and reset which can be difficult for SLDV to handle. The results suggest that a variety of requirements for large Stateflow charts can be handled by SLDV. The issues with tracking multiple signals can possibly be mitigated by alternative representations and sub-requirements, as shown in Section 6.5.

6.9 Summary

In this chapter, the results of the case study have been presented. The case study consists of falsification and verification of a subset of the requirements used to test the charging control software at Volvo Cars. Processes for applying SLDV to these requirements and the software are presented. Although there are common denominators in the approaches, much of the work done to optimize the models for SLDV are very specific, rendering it difficult to give clear step-by-step instructions for how to introduce SLDV in the verification process in a general case. However, the systematic approach can serve as a guidance to readers interested in the process of implementing an SLDV workflow in their development process.

7

Designing requirement models

This chapter aims to answer and discuss the first two research questions from Section 1.4. The basis for the discussions of these questions is the case study and the conclusions from it. The purpose of this chapter is to use the case study in an attempt to analyze how requirement modeling should be handled in a more general case. It is important to note that while the case study is a good basis for this discussion, it is still a case study consisting of a few requirement samples, and the conclusions drawn from it and the answers to these research questions are therefore not solid truths but rather guidance to how requirement modeling should be approached in SLDV.

After research questions 1 and 2 are discussed, a note on modeling requirements will be presented. Ensuring falsifiability and satisfiability is a simple way to detect potential modeling errors in the requirements, which is discussed in Section 7.3.

7.1 Requirement modeling for SLDV optimization

This section will discuss and attempt to answer the first research question:

1. *How should requirements be modeled in Simulink to allow Simulink Design Verifier to come to conclusions faster?*

The purpose of this research question is to get a basic understanding of how requirements should be modeled to optimize for use in SLDV. Structures used for test case simulations might not be optimal for SLDV and must be remodeled. Different approaches for optimizing the requirement models will be discussed in this section.

7.1.1 Dividing requirements into different sub-requirements

Some requirements encountered at Volvo Cars have different conditions which specify what the desired behavior of the software is when each of the conditions is fulfilled. This is often modeled in branches using Switch blocks. This can be quite difficult for SLDV to handle in some instances. Branching through Switches is not found to be a bottleneck in all cases, and the cause behind the slower analysis times in SLDV can possibly be traced back to other sources than the Switch blocks.

What is shown to be effective in most instances was dividing requirements with multiple conditions into sub-requirements. In this case, smaller requirements are created which are much easier to handle. Possibly, the internal representations in SLDV can be simplified by explicitly dividing the requirements into multiple parts. This could potentially be used together with Switch blocks if done correctly. However, an effective strategy found was to model this behavior using Implies blocks for the separate conditions. An example of this is shown in Figure 7.1. The requirement in this example is *"If A then B, otherwise C"*. Using this approach, two smaller sub-requirements have been formed which seems to work better in the general case than attempting to model these sub-requirements using single Proof Objective blocks.

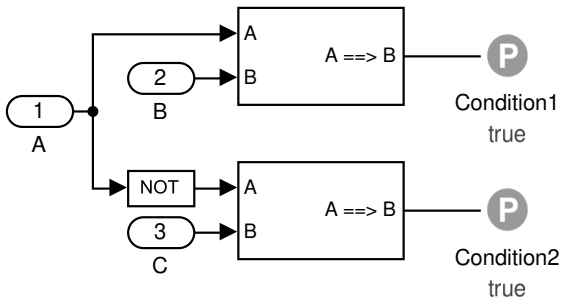


Figure 7.1: Model showing how requirements can be split into sub-requirements.

It is not always possible to divide requirements into different conditions. However, a possible approach for these cases is to subdivide the condition. This is done in Subsection 6.5. For two implications, it follows that

$$(a \to b) \wedge (b \to c) \Rightarrow a \to c \tag{7.1}$$

Hence, if $a \to c$ is difficult for SLDV to handle, the requirement can be split up into two parts $a \to b$ and $b \to c$. If both these implications are proven valid by SLDV, then it follows that $a \to c$ is also verified.

If analysis times are long in SLDV, the requirement should be reduced to smaller parts if possible. Not only can this improve analysis times, but it can also help debugging. If it is found that only one condition can be falsified, then debugging is instantly reduced to investigating the specific falsified condition rather than the requirement in its entirety. Finding appropriate sub-requirements can be difficult, but it is possible that the tested software contains patterns that make the division process much simpler.

7.1.1.1 Number of signals

As discussed in Subsection 6.48, requirements with many signals can be difficult to handle. Thus, if a requirement contains many different signals, it could potentially be more efficient to split the requirement into different sub-requirements, or try and rewrite the requirement in terms of other, equivalent signals to try and reduce the signal complexity. This is especially important if this leads to a lower complexity in the test environment, as discussed in Chapter 8. Requirements with many signals

are not necessarily difficult to handle, as shown for Section 6.2, but a rule of thumb is to reduce the number of signals in a requirement through dividing the requirement into sub-requirements, or reformulating the requirement in terms of other, fewer signals, if possible.

7.1.2 Temporal logic

One of the largest bottlenecks for SLDV in the case study is temporal logic. This applies to both the test environment and requirement models. The requirement models used to simulate test cases at Volvo Cars use custom library blocks for temporal logic. In their original form, there are some bugs in the library blocks which are found using SLDV. However, even after fixing the bugs in the custom temporal library blocks SLDV struggles to handle the temporal logic. The solution is to replace the temporal logic blocks with SLDV temporal operators, such as the Detector block. This does not always solve the issues of temporal logic, as shown in the case study in many of the test environments with Stateflow charts. However, the analysis times in other requirements are independent of the temporal logic time, implying supported internal representation in SLDV for temporal logic. The case study shows that modeling temporal logic using the SLDV temporal operator blocks often leads to shorter analysis times.

7.1.2.1 Tolerance vs signal synchronization

Many of the different existing requirement models for the charging control software at Volvo Cars contain tolerance blocks, to allow the requirement to be falsified in single time-steps at a time. This allows for simpler requirement modeling, as delicate signal timings are not as strict, and if the requirement will only be falsified for a single time-step in very specific instances, the requirement could be considered verified. However, this can cause issues for SLDV as (possibly more) temporal logic is introduced, which can increase the analysis times of Property Proving. Additionally, tolerance blocks may not handle cases in which the requirement is falsified very often, although only for one time-step at a time. It is advised to solve signal timings and other synchronization explicitly, in favor of adding tolerance to the requirement for simpler modeling.

7.1.3 Proving strategies

In SLDV, there are three proving strategies. SLDV is able to try and find an actual proof or counterexample to a requirement, or try and find a violation within a specific amount of time-steps. The third proving strategy is a combination of the earlier two, where a violation detection is first performed, and if the requirement passes the violation detection an attempt at a proof is performed.

Using the *FindViolation* proving strategy is typically much faster than the *Prove* strategy. As a result, when developing requirement model tests, using *FindViolation* can be very effective to get an indication of if the requirement is verified, although the *Prove* strategy is required to get a definite answer to this question. When a

test environment and requirement model has been set up and the *FindViolation* strategy is shown to be much faster than the *Prove* strategy, SLDV can be set to the *ProveWithViolationDetection* for the environment and requirement. With the *ProveWithViolationDetection* strategy, SLDV will first perform a *FindViolation* search, and will then perform attempt to *Prove* the Proof Objective blocks valid if the *FindViolation* is passed. This means that testing the requirement for all future changes made to the software model, trivial counterexamples can often be found quickly in the *ViolationDetection*, while the overall analysis time for an actual proof is not affected much.

7.1.4 Continuous signals

Requirement 6, which mainly deals with continuous signals, is shown be difficult for SLDV to handle in the test environments presented in Section 6.7. In order to get some results, the Data Type Conversions must be removed from the test environment. Note that this change results in an incorrect environment as the conversions to integers, for which rounding or flooring of the floating-point signals must be performed, are removed. However, a good estimate of how the software performs with respect to Requirement 6 can still be obtained. The results indicate that the data types of the input signals to the requirement can impact the analysis speed of SLDV, and appropriate data types should be used to optimize for performance.

7.2 Isolating errors through requirement models

This section aims to discuss and answer the second research question:

2. *How should Simulink requirement models be altered to isolate errors and show absence of other errors, with the use of Simulink Design Verifier?*

The testing process is not necessarily done when a result is produced. In the case of a falsification, the error must be analyzed to figure out if the error is in the software or in the requirement or test environment model. The purpose of this research question is to describe a process which follows a falsification by SLDV, in order to isolate the error and find other requirement violations in the software.

7.2.1 Sub-requirements

As discussed for research question 1 in Subsection 7.1.1, it is often possible to split the requirements into different sub-requirements. The details will not be repeated here. However, by *e.g.* splitting the requirement into different sub-requirements based on activation conditions, it is possible to determine for which conditions the requirement is falsifiable. The conversion to sub-requirements is not necessarily an optimization in terms of analysis speed by SLDV, but it can be helpful for debugging. In the case of splitting up implications into sub-implications using transitivity ($(a \rightarrow b) \wedge (b \rightarrow c) \Rightarrow (a \rightarrow c)$), the error can potentially be isolated to a certain part of the transitivity chain. If it is not clear from the counterexample given by SLDV what the error in the model is, these approaches for sub-requirements are recommended.

7.2.2 Requirement exemptions

When a requirement is falsified and the cause of the error is clear, it can be useful to look into if the error appears somewhere else in the software model. To do this, an exemption can be added to the test requirement. This allows SLDV to ignore the error found to instead try and find other errors. By doing this, it is possible to learn the extent of the error. If the error is found in lots of places in the software, it is typically better to create a larger, robust fix for the error rather than adding small fixes and iteratively finding new errors, as this type of code will likely be less structured, robust and more difficult to maintain.

7.2.2.1 Test single states and behavior

Two approaches for exemptions in the software were used in Section 6.3: isolating and testing a single state in a Stateflow chart, versus isolating the state in which the error was detected. In a more general case, this can also include other behavior, beside Stateflow states.

Requirements can be specialized to only consider specific states and behavior of the software. If the requirement is only supposed to test behavior in one location of the software, this can be an effective strategy to verify that the desired behavior works properly in the intended location. Using this strategy can allow verification of the behavior, potentially indicating that the behavior is working properly, and the condition for it must be avoided in unintended locations and states of the software.

7.2.2.2 Isolate and exempt specific states and behavior

A downside to this approach is that very much of the software is discarded, meaning there could be many errors missed in the rest of the software. A way to mitigate this issue is to add an exemption in the requirement to the specific state and behavior which produced the error in the first place. If the requirement is verified with the exemption, the error has been isolated. If another error is found, more information on the faulty behavior in the model is acquired; this reduces the risk of iteratively adding small fixes.

7.3 Ensuring satisfiability and falsifiability in requirements

In order for a requirement to be relevant, it must be falsifiable. This means that there must exist some set of inputs for which the requirement is falsified. Assume a model with a requirement with open inputs, *i.e.* the inputs are inputs to the model itself and are not connected to an SWC inside the model. If the requirement is always fulfilled regardless of input, then the requirement will not impose any restrictions on the behavior of the corresponding SWCs. In order for a requirement to be meaningful, it must be falsifiable when disconnected from other subsystems. Similarly, it is important to ensure the requirement is satisfiable for some input.

A requirement which is never true regardless of input cannot be used to verify the behavior of a component.

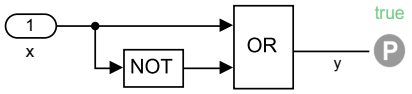
Two examples of missing falsifiability and satisfiability in requirements can be seen in Figure 7.2. The left model outputs

$$y = x \vee \neg x \tag{7.2}$$

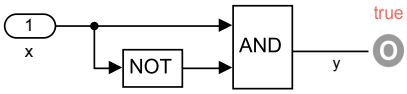
which is always 1, and the right model outputs

$$y = x \wedge \neg x \tag{7.3}$$

which is always 0. Both models represent requirements which are fulfilled when $y = 1$. The left model fulfills the satisfiability criteria, *i.e.* it should be satisfied for at least one input. However, it does not fulfill the falsifiability criteria; it cannot be falsified but is always true, regardless of input. The right model is the opposite: it fulfills the falsifiability criteria but not the satisfiability criteria.



(a) Requirement model which is always satisfied.



(b) Requirement model which is never satisfied.

Figure 7.2: Examples of requirements which do not fulfill the falsifiability or satisfiability criteria of meaningful requirements; the requirements are fulfilled when the output is 1.

To ensure satisfiability and falsifiability in requirements, the regular Proof Objective can be kept. A test objective should also be added to the same signal, with the same objective. Then, the requirement should be disconnected and evaluated in SLDV using both property proving and test case generation. For the requirement to satisfy both the falsifiability and satisfiability criteria, the test case generation should succeed whereas the property proving should fail. If that is the case, then the requirement is both falsifiable and satisfiable. The test objective can also be left in, as it allows for debugging whenever a requirement is not proven valid when connected to the software. A visualization can be seen in Figure 7.3. Note that the requirement must have connected inport blocks in this case; disconnected inputs will be treated as zeroes by SLDV.

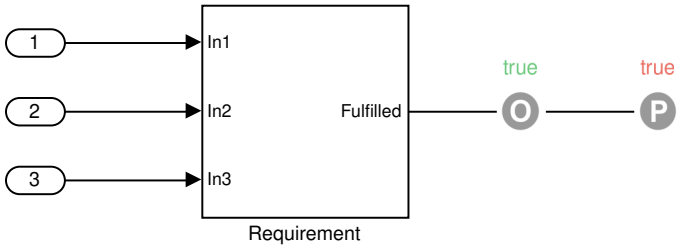


Figure 7.3: Example of unconnected requirement with unsatisfied proof objective and satisfied test objective.

8

Building a Test Environment

This chapter aims to discuss and answer the third and final research question:

3. *How should test environments be set up to allow for Simulink Design Verifier to easily formally verify or falsify the requirements, without altering the behavior of the software?*

The previous research questions regard the requirement models specifically. However, the software must also be taken into account. This section will discuss how test environments should be set up to allow SLDV to come to a conclusion as quickly as possible, and what the implications the test environment structure has on the results of SLDV.

8.1 Initializations

In the case study in Chapter 6, multiple counterexamples and issues causing SLDV to run for a very long time can be traced to the initialization of the software. Multiple ways of mitigating these issues are presented in the case study. A common denominator in the actions to resolve these issues was to ensure the software was properly initialized. This involved alterations to the SWC Scheduler Stateflow chart, but also adding Unit Delay blocks to the input signals to ensure no input could be given in the very first time-step. Although it is important to look at the possible errors in the initializations, these errors should not shadow the potential errors when the software is running "normally". As such, it is suggested to ensure the software and requirement models are properly initialized before any inputs can be given to the models, and initialization errors are tested separately.

8.2 Temporal logic

As shown in the case study, temporal logic can cause issues for SLDV. Even when it is modeled "correctly" using the built-in SLDV temporal operators, analysis times can be massive. On the other hand, the temporal logic length can sometimes be irrelevant, as shown in several of the requirements in the case study, if modeled correctly. To optimize for SLDV, its built-in temporal operators should be used whenever possible. Since the temporal operators require a license to use, the devel-

opers might still want to keep custom library detector blocks in the software; in this case, the custom library blocks can be replaced by the built-in temporal operators using the Block Replacement feature in SLDV, explained in Section 5.4. However, separate equivalence tests should be set up to ensure the original blocks are equivalent to the blocks that the original blocks are replaced with, unless the intention is to test behavior different from the original model.

8.2.1 Reductions of temporal logic

A simple way to reduce the computational complexity of the test environment is to reduce all temporal logic. For instance, if the software is to detect consecutive *true* inputs for a specific signal, the behavior is typically not removed from the software. If the detection is removed (*i.e.* inputs are passed straight through the Detector), then the model has become more general as inputs can still be given as one would expect from the output of the Detector. An issue this can cause is if the temporal logic is scaled down too far. For instance, there could be faulty behavior related to a temporal operator which could take x time-steps to appear. However, if the temporal logic is scaled down below x, then this faulty behavior is easily missed. Thus, reductions in temporal logic should be done together with sub-requirements to show that the reductions are correctly handled.

To reduce the temporal logic in Stateflow charts, a solution was presented in the case study where the temporal transitions were replaced by input triggered transitions. With the use of a Boolean *TemporalTransition* input, the complexity of the Stateflow chart can be massively reduced, while no behavior is removed; the *TemporalTransition* input can still be applied in the "correct" time-step. It is shown in the case study that this was not required in all instances, but should be done whenever possible to reduce any potential future problems due to temporal logic. The method for this reduction is shown in the case study in Subsection 6.1.2.

An inherent issue to note for temporal logic is that the *FindViolation* proving strategy in SLDV could be difficult to use. Due to the number of violation steps needed, violation detection can become a big problem for SLDV. Thus, if violation detection is to be used, reduction of temporal logic to reduce the complexity has been shown in the case study to give better performance for SLDV. Whenever a counterexample is obtained for test environments with reduced temporal logic, the counterexample should be extended to apply to the "real" software with full temporal logic. If the reductions are done correctly, this should not be too complicated to carry out.

8.3 Minimizing the test environment

Larger models typically take longer to be processed and evaluated in SLDV, as shown in Table 6.1. Thus, there is a need to optimize the testing environment to achieve better computational times. An effective way of doing this is to minimize the test environment by isolating software components.

8.3.1 Isolating Software Components

Ideally, the requirement should be tested with the entire software as a whole. For small software models, this is perfectly fine. However, the complexity in large models can easily make requirement proving infeasible. A solution to proving requirements on these large, complex software models is to isolate its components. The principle behind this is to reduce the complexity of the software to allow SLDV to falsify or prove a requirement much quicker. However, the correct parts of the software must be removed so that the input space to the requirement model does not become too large.

8.3.1.1 Input spaces

A requirement has a specific input space, which contains all possible combinations of inputs to the requirement. As explained in Section 7.3, a meaningful requirement is falsifiable in an open setting. However, if the requirement is to be provable, it must be satisfiable for a subset of inputs. If the requirement is to be verified, the software inputs and outputs used for the requirement must all be elements in the satisfiable requirement input subset.

In general, the requirement input space can be restricted by introducing more of the software. When larger portions of the software are included in the test environment, there are more "rules" to the signals which are passed into the requirement model, effectively limiting the input space to the requirement in the test environment.

Mathematically, the requirement input space for different models are related to one another as

$$\Omega_{es} \subseteq \Omega_{te} \subseteq \Omega_{req} \tag{8.1}$$

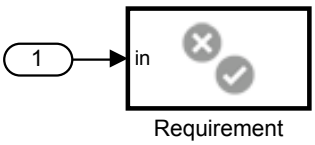
where Ω_{es} is the requirement input space when the requirement is connected to the entire software, Ω_{te} when connected to a certain test environment, and Ω_{req} when unconnected, *i.e.* connected to pure model inputs. The satisfiable input subset Ω_{sat} is the set of inputs for which the requirement is fulfilled. In a falsifiable requirement model (as explained in Section 7.3), this is a true subset to Ω_{sat}. The goal is that the test environment set should be a subset of Ω_{sat}, *i.e.*

$$\Omega_{te} \subseteq \Omega_{sat} \tag{8.2}$$

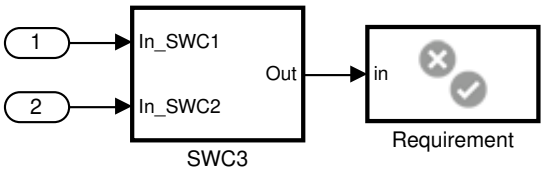
Isolating the components more, *i.e.* removing a larger portion of the software, will increase the size of Ω_{te}. Thus, this becomes an optimization problem in which Ω_{te} is increased by removing parts of the software in order to reduce test environment complexity, which in turn decreases the analysis times of SLDV.

Figure 8.1 shows how a varying degree of the software model can be put into the testing environment together with the requirement. Two cases of corresponding input spaces into the requirement can be seen in Figure 8.2. In the left case, the constraints on the requirement input space imposed by SWC3 is not enough for the input space to be a subset of the satisfiable input space. Introducing SWC1 and SWC2 puts further limitations on the input space to the requirement, so that it

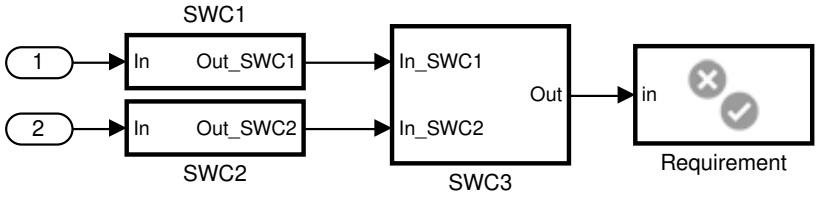
becomes a subset of the satisfiable space. In the right case, the satisfiable space is large enough to have a testing environment consisting of only the requirement block and SWC3 to satisfy the requirement. Adding more SWCs only add more limitations to the requirement input space. Thus, the requirement input space for a larger testing environment is going to be a subset of the input space for a smaller testing environment (where all components of the smaller environment also exist in the larger one).



(a) Isolated testing environment containing requirement only.

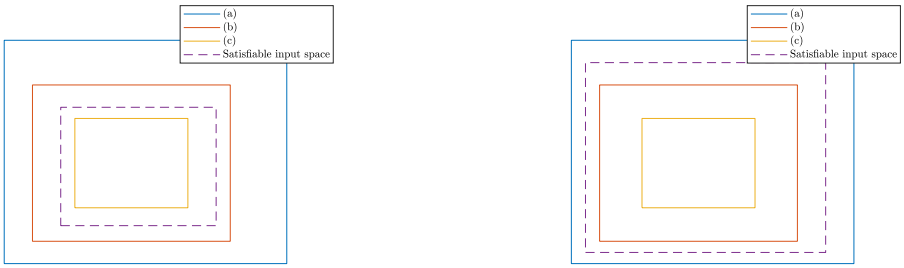


(b) Isolated testing environment containing requirement and directly connected SWC.



(c) Isolated testing environment containing requirement, directly connected SWC and its directly connected SWC inputs.

Figure 8.1: Visualization of how the requirement and software components can be isolated in a testing environment in different orders of magnitude; the full software model contains more than the three SWCs as seen in the figure.



(a) **(b)**

Figure 8.2: Examples of input domains together with satisfiable domain.

8.3.1.2 Standard actions for isolating software components

Knowing how to isolate software components in a test environment can be difficult. A simple strategy is to remove unused signals and logic. However, in a software

model with as many SWC interconnections as the charging control software, the end result can still be much too big for SLDV to handle.

A harsh but effective strategy is to remove the logic for software outputs which are only used as antecedents in the implications used in the requirements. For instance, in Requirement 1 in Section 6.2, signals A and B are software outputs. However, in several of the test environments they are treated as inputs. Since these signals are also inputs to SWCs which are the primary target of the requirement and there is no restriction on the behavior of signals A and B in the requirement, they can be considered input; in fact, the requirement would not be worded differently if signals A and B were indeed pure software inputs instead. These signal outputs (Which are treated as inputs in the test environment) are denoted *condition outputs*.

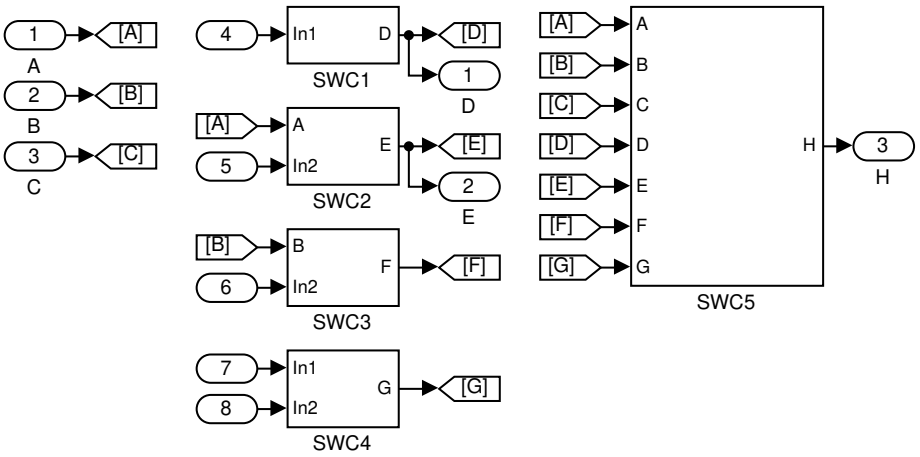
A simple rule of thumb can now be stated: the SWC "chains" starting from either the inputs shared by the software and the requirement or from condition outputs, ending in the non-condition-outputs, should be kept in the test environment. All other SWCs should be removed. This strategy is used for all requirements in the case study to a varying degree. Even with this strategy, there could be SWCs left in the test environment which are not strictly needed. In that case, an evaluation must be made regarding which SWCs are necessary for the test environment, but a simple approach is to try and remove the ones which are likely unnecessary, and these can be added back in if a falsification is traced back to the missing SWC.

An example of how this is applied in practice is shown in Figure 8.3. The signals used in the corresponding requirement are A, B, C, D, E and H. Signal D is a condition output. The software component isolation for this example reduced the software in the test environment from Figure 8.3a to Figure 8.3b. The reduction is done as follows:

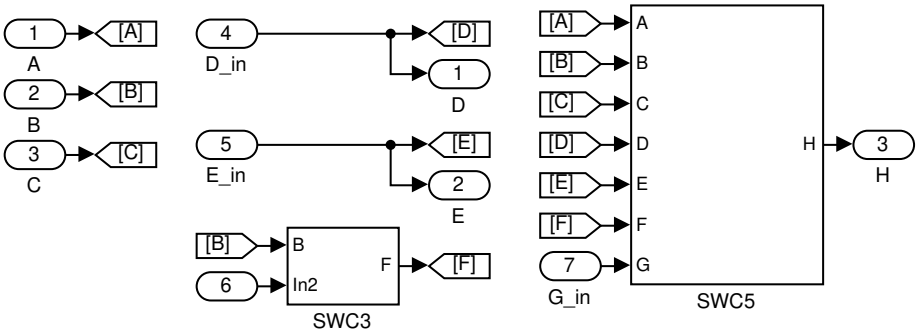
- **SWC1**: Produces a condition output, hence this SWC is removed.
- **SWC2**: Produces a condition output, hence this SWC is removed, despite depending on signal A (in case of a falsification, this SWC could be brought back in).
- **SWC3**: Produces input to SWC5 based on input signal B, hence this SWC remains.
- **SWC4**: Produces input to SWC5 based on non-tested inputs, hence this SWC is removed.
- **SWC5**: Produces non-condition output used in the requirement, hence this SWC remains.

By following the rule of thumb, the software has been heavily reduced in the test environment. This allows SLDV to come to conclusions much quicker.

If a falsification is returned by SLDV in a test environment with isolated SWCs, an evaluation must be performed if the test environment should be expanded to include more SWCs in order to try and restrict the input range, to have it fall into



(a) Software inside test environment before reduction.



(b) Software inside test environment after reduction.

Figure 8.3: Models showing how the SWCs in the software can be isolated to help SLDV come to conclusions faster by making the test environment smaller.

the satisfiable input subset. This might not work well, as the complexity of the test environment can become too large for SLDV to handle in a reasonable amount of time. In this case, attempts can be made to see if the given counterexample is possible in the full software model. This check can be done using manual or automatic test case generation, for instance through SLDV. If this is possible, then a counterexample for the full software is found and the requirement is indeed falsified.

However, generating test cases might not be possible in a short enough amount of time. A reasonable approach in this case would be to resolve the issue in the software for the given test environment. This can potentially lead to a larger and somewhat slower software due to possible redundant error checks, but it will also make the software more robust and less prone to errors.

8.4 Reducing continuous signal complexity

The case study shows that continuous signals can be troublesome for SLDV. For instance, simply removing the Data Type Conversion blocks in the test environment in Requirement 1 improves the analysis speed of SLDV significantly. Additionally, the

Data Type Conversions must be removed for the test environment in Requirement 6 to get the analysis times reported there. To optimize for performance, the complexity of continuous signals should be reduced as much as possible. For instance, if Data Type Conversions are used on signals which are then only compared to a threshold, then the signals could possibly be replaced by Boolean signals instead, which represent if the threshold is met or not.

The continuous signal reduction is especially important for floating-point and fixed-point signals, as SLDV will have to perform additional analysis to reduce approximations for these types of signals. While the approximation of rational numbers can lead to correct results, the additional analysis can come to a different conclusion. The case study shows that the full additional analysis can take a long time compared to the approximation analysis time, and actions should be taken to reduce the amount of additional analysis SLDV has to make, in order to optimize for low computation times. One such action is to use single-precision instead of double-precision floating-point signals, as shown in Subsection 6.1.1.

9

Conclusion

This masters thesis has presented a case study in which formal verification with the use of SLDV was applied to requirements for the charging control software at Volvo Cars. A small subset of a relatively varied types of requirements for the software are investigated. The case study is qualitative and describes how problems encountered in the development of a formal verification process can be countered, and what steps can be taken to isolate errors, which can be useful in the process of bug fixing the software.

The research questions focus mainly on the requirement modeling, both in terms of optimization and error detection and isolation, and on building an optimized test environment. The results presented in the case study for these questions are relatively specific to the charging control software, especially due to the low number of requirements studied. As such, these results can be used as a guidance for a more general case but no conclusion on what applies to the general case can be drawn.

9.1 Manual work

The case study has shown that formal verification using SLDV can require much manual work. This drawback can make the tool unattractive for development processes in which tests need to be performed often. It is worth noting that once the work to set up test environments and requirement models, the testing can be repeated as long as the changes in the test environment apply to the software. However, if structural changes are made to the software which makes the test environment incorrect or unsuitable for use with SLDV, the manual work must be applied again to design a new test environment.

It is important to remember here that the case study is of a large software model. Inherently, large software will change slowly over time as a developer can only work on a very small piece of the software at a time. As such, whenever changes are made most of the software will likely remain the same, and as such most of the test environments built for the software will still be viable for testing with SLDV. In short, once the manual work with implementing test environments for use with SLDV, the amount of manual work needed to keep test environments and requirement models usable is not too high, as the software will change slowly over time due to its size.

9.1.1 Continuous integration

Continuous integration, described in Section 4.3, is commonly used in software development. The usage of formal verification in continuous integration requires effective test environments and requirement models to work properly, if verification for a large number of requirements is to be performed several times per day. Additionally, the case study shows that the analysis times are highly dependent on the test environments and requirement models, and usage for formal verification in CI requires robust and consistent computation times. It is important to take the possibility of SLDV not reaching a conclusion in a short enough amount of time into account, and derive a process for what to do in these cases. The varying analysis times in SLDV and the dependence of the test environment (hence also the software) for quick results, using SLDV in cases where consistent analysis times are important, such as CI, can potentially be difficult.

9.2 Using SLDV for different requirement levels

As explained in Section 4.1, testing can be applied for different levels. The case study shows that test environments for SLDV should be as small as possible to allow SLDV to analyze it fast. In *e.g.* Section 6.2, bringing "unnecessary" software components proved to cause significant increases to the analysis time for the environment. Therefore, SLDV might not be suitable for property proving on all levels. If the entire software or a large portion of it needs to be included in the test environment for a requirement, then the task could prove too difficult for SLDV. Hence, other testing methods or tools should be used for requirements for the whole software, if the software is big enough.

The qualitative approach of the case study makes it difficult to draw conclusions on what types of requirements work well for SLDV, and if there are other patterns in the software that are important to requirement and test environment modeling, or that can suggest if formal verification with SLDV can be used. The case study shows that continuous signals can be difficult for SLDV to handle, but an extensive study in how continuous signal requirements can be formally verified by SLDV for similar software structures could be very useful.

9.3 Future work

Many of the requirements for the charging control software deal with temporal logic, *e.g.* for defining timeout behavior in the communication between the EV and the EVSE. The temporal logic is shown difficult for SLDV to handle. Different software have different important issues to rule out for the use of SLDV, and one such issue for the charging control software is the temporal logic. Focusing on frequent issues can be an effective way of optimizing the usage for SLDV. Some strategies for how to mitigate the problems with temporal logic are presented in the case study (Chapter 6), but this should be further investigated, both for the use of Property Proving with SLDV but also for other testing methods as well.

9.3.1 Comparing SLDV to other testing methods

The usage of Property Proving in SLDV is very powerful, as it allows testers to prove or disprove correctness of the system. However, this assumes SLDV is able to handle the test environments and requirement models which will not always be true. For requirements which are typically difficult for SLDV, other methods should be considered. For instance, automatic test case generation using SLDV (as explained in Appendix C) can be used to generate test cases, and the software can then be simulated for the generated test cases. To make an informed decision about if and how SLDV should be included in the development and testing process of large, mostly discrete software, detailed comparisons between Property Proving and Test Case Generation using SLDV, as well as other testing methods, should be carried out.

Bibliography

[1] Jan Schroeder, Christian Berger, Alessia Knauss, Harri Preenja, Mohammad Ali, Miroslaw Staron, and Thomas Herpel. Predicting and evaluating software model growth in the automotive industry. 08 2017.

[2] UPPAAL. `http://www.uppaal.org/`. Accessed: 2020-03-27.

[3] About – Supremica. `https://supremica.org/about/`. Accessed: 2020-03-27.

[4] Simulink Design Verifier – MATLAB & Simulink. `https://se.mathworks.com/products/simulink-design-verifier.html`. Accessed: 2020-03-27.

[5] Brandon Dick, Bill Haskins, Gregory Moroney, Jonette M Stecklein, Jim Dabney, and Randy Lovell. Error cost escalation through the project life cycle. *14th Annual International Symposium*, 2004.

[6] Mattias Nyberg, Dilian Gurov, C. Lidström, A. Rasmusson, and Jonas Westman. Formal verification in automotive industry : Enablers and obstacles. In *8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2018 :*, volume 11247 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 139–158, 2018. QC20190405.

[7] Marcus Liliegård and Viktor Nilsson. Model-Based Testing with Simulink Design Verifier. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2014.

[8] Rohit Agrawal. Semi-Automated Formalization and Verification of Automotive Requirements using Simulink Design Verifier. Master's thesis, KTH Industrial Engineering and Management, Stockholm, Sweden, 2015.

[9] Frank Dordowsky. An experimental study using ACSL and frama-c to formulate and verify low-level requirements from a DO-178C compliant avionics project. In Catherine Dubois, Paolo Masci, and Dominique Méry, editors, *Proceedings Second International Workshop on Formal Integrated Development Environment, F-IDE 2015, Oslo, Norway, June 22, 2015*, volume 187 of *EPTCS*, pages 28–41, 2015.

[10] Nikolai Kosmatov and Julien Signoles. Frama-C, A Collaborative Framework for C Code Verification: Tutorial Synopsis. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 92–115, Cham, 2016. Springer International Publishing.

[11] Pat Blaabjerg Frede Dragičević Tomislav, Wheeler. 8.1 overview of ev and evse markets and trends, 2018.

[12] Axel Gandy and James Scott. Unit Testing for MCMC and other Monte Carlo Methods, 2020.

[13] H. K. N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990*, pages 290–301, 1990.

[14] Lionel Briand and Yvan Labiche. A UML-Based Approach to System Testing. *Software and Systems Modeling*, pages 10–42, September 2002.

[15] Vivek Jaikamal. Model-based ECU development – An Integrated MiL-SiL-HiL Approach. 2009. SAE Technical Paper 2009-01-0153.

[16] Jackson A. [Prado Lima] and Silvia R. Vergilio. Test Case Prioritization in Continuous Integration environments: A systematic mapping study. *Information and Software Technology*, 121:106268, 2020.

[17] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. [Jenny Li], and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978 – 2001, 2013.

[18] Unit testing. `https://www.cs.cmu.edu/~rjsimmon/15122-s13/rec/07.pdf`. Accessed: 2020-04-23.

[19] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarck's proof procedure for propositional logic. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design*, pages 82–99, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[20] The MathWorks. *Simulink Design Verifier User's Guide*. The MathWorks.

[21] The MathWorks. *Simulink Design Verifier Getting Started*. The MathWorks.

A

Boolean Algebra

The content of this thesis is heavily based on Boolean algebra, *i.e.* algebra with logical variables and operations. To denote the two Boolean values, *true* and 1 will be be used interchangeably, and *false* and 0 will be used interchangeably. The Boolean operators used in this thesis along with their explanations can be found in Table A.1.

Operator	**Name**	**Explanation**
$\neg$	`NOT`	Negation. Outputs the opposite Boolean value of the operand.
$\vee$	`OR`	Conjunction. Outputs 1 if any of the operands are 1, 0 otherwise.
$\wedge$	`AND`	Disjunction. Outputs 1 if both operands are 1, 0 otherwise.
$\oplus$	`XOR`	Exclusive `OR`. Outputs 1 if the operands are not equal, 0 if the operands are equal.
$\rightarrow$	Implies	Mathematical implication. $a \rightarrow b$ is equivalent to $\neg a \vee b$.

Table A.1: Boolean operators used in this thesis.

B

Linear Temporal Logic

Linear Temporal Logic (LTL) is modal temporal logic, where the modal part refers to time. LTL can be used to formulate formulas which describe the future of logical variables, hence describing temporal behavior of the variables.

Linear Temporal Logic contains operators similar to "regular" boolean logic, such as logical `AND`, `OR` etc. There are also temporal operators, which purpose is to define the behavior of variables. These can either be *global*, meaning they define the temporal behavior for all time, or *timed*, meaning they apply to a certain time interval. The LTL operators can be found in Table B.1 together with their explanations.

Operator	**Name**	**Explanation**
$\neg$	`NOT`	Negation. Same as in Table A.1.
$\vee$	`OR`	Conjunction. Same as in Table A.1.
$\wedge$	`AND`	Disjunction. Same as in Table A.1.
$\oplus$	`XOR`	Exclusive `OR`. Same as in Table A.1.
$\rightarrow$	Implies	Mathematical implication. Same as in Table A.1.
$\bigcirc$	Next	Unary operator. The operand (φ in $\bigcirc\varphi$) must be true in the next time-step.
$\square$	Global always	Unary operator. The operand (φ in $\square\varphi$) must always be true from the current time-step.
$\lozenge$	Global eventually	Unary operator. The operand (φ in $\lozenge\varphi$) will eventually be true from the current time-step.
$\mathcal{U}$	Global until	Binary operator. The first operand (φ in $\varphi\,\mathcal{U}\,\psi$) must be true until the second operand (b) is true.
$\square_{[a,b]}$	Timed always	Unary operator. The operand (φ in $\square\varphi$) must always be true in the time interval $[a, b]$.
$\lozenge_{[a,b]}$	Timed eventually	Unary operator. The operand (φ in $\lozenge\varphi$) will be true at least once in the time interval $[a, b]$.
$\mathcal{U}_{[a,b]}$	Timed until	Binary operator. The first operand (φ in $\varphi\,\mathcal{U}\,\psi$) must be true until the second operand (b) is true in the time interval $[a, b]$.

Table B.1: LTL operators used in this thesis.

C

Simulink Design Verifier – Complementary

This appendix is meant to be a complement to the introduction to SLDV given in Chapter 5.

C.1 Test Case Generation

SLDV also supports test case generation. This consists of generating sets of inputs which produce a desired behavior, state or output of the system model. This differs from property proving in that a Proof Objective must always be fulfilled for all possible inputs to the system, whereas the test case generation only aims to find one set of inputs which is able to fulfill the test objective.

Much like the Proof Objective block is used for property proving in Simulink, there is a corresponding block for test case generation called the *Test Objective* block, as seen in Figure C.1. A value or set of values or intervals can be specified, and SLDV will attempt to find inputs which fulfill each of the values or intervals in the objective set.

(a) Regular **(b)** Satisfied **(c)** Falsified **(d)** Undecided **(e)** Disabled

Figure C.1: SLDV Test Objective block.

An example of this can be seen in Figure C.2, where the Test Objective block has an objective set of three intervals or values: [1 2], 3 and [4 5]. Without constraints, SLDV is able to generate inputs to fulfill all three intervals or values in the objective set, as seen in the left figure. In the right figure, an input constraint has been set such that the input can never be larger than 3. Thus, the last interval in the objective set cannot be fulfilled, whereas inputs can be generated for the interval [1 2] and the value 3.

Test Objective blocks can be used in SLDV to debug models, or to generate inputs which can achieve certain behavior in the system. This can be especially

(a) Unlimited input. **(b)** Input limited to [1, 3].

Figure C.2: Examples showing when Test Objectives are satisfied or falsified in SLDV.

useful in later stages of testing large models such as Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL) testing, where test cases can be generated by SLDV to test certain behavior in SiL or HiL. Note that test case generation for the sake of testing and simulating certain behavior is beyond the scope of this thesis.

The Assumption block is only used for property proving in SLDV. However, the *Test Condition* block is the equivalence for test case generation. It can be found in Figure C.3.

(a) Enabled **(b)** Disabled

Figure C.3: SLDV Test Condition block.

C.2 Detecting Design Errors

The third major feature of Simulink Design Verifier is the ability to detect design errors. This involves errors such as integer overflow and division by 0, but also finding dead logic within the model. Much like the test case generation, detecting design errors on its own is beyond the scope of this thesis, and is merely used as a tool to allow for more efficient property proving; removing dead logic reduces the size of the model which can improve the computation times of the model, and can also help debugging the model when a requirement is falsified.

A simple example of using SLDV to detect dead logic is found in Figure C.4. SLDV will look at certain blocks, often logical operators or switches, and decide whether all values in the input and output ranges are possible. It will also find unreachable states in Stateflow charts. In the example, these blocks are the `NOT` block and the `OR` blocks. The `NOT` block is directly connected to the binary input meaning all inputs and outputs are covered, *i.e.* contains no dead logic. The upper `OR` gate is equivalently covered. The lower `OR` gate models the tautology $x \vee \neg x$ and can never output *false*, hence contains dead logic.

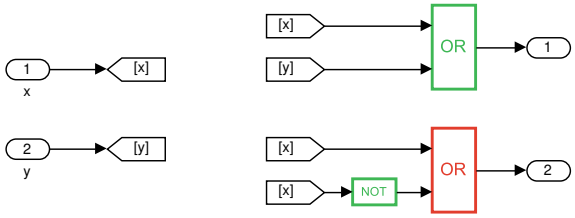


Figure C.4: SLDV usage example to detect dead logic.

D

Block replacement example

```
function rule = blkrep_rule

    rule = Sldv.xform.BlkRepRule;
    rule.FileName = mfilename;

    rule.BlockType = 'SubSystem';

    rule.ReplacementPath = sprintf('<Lib>/<Block>');

    rule.ReplacementMode = 'Normal';

    rule.IsReplaceableCallBack = @replaceableCallback;
    rule.PostReplacementCallBack = @postReplacementFunction;
end

function out = replaceableCallback(block_handle)
    out = false;
    if strcmp(get(block_handle, 'ReferenceBlock'), '<ReferenceBlockPath>')
        out = true;
    elseif strcmp(get(block_handle, 'MaskType'), '<MaskType>')
        out = true;
    elseif ~isempty(regexpi(get(block_handle, 'Name'), '<BlockName>'))
        out = true;
    end
end

function postReplacementFunction(block_handle, preReplacementCompiledInfo, ...
                                 sldvOptionsH, configSubsystemH)

    src_block = get(block_handle, 'PortHandles');
    src_block = src_block.Inport(2);
    src_block = get(src_block, 'Line');
    src_block = get(src_block, 'SrcBlockHandle');
    length = get(src_block, 'Value');
    length = str2double(length);
    if isnan(length)
        length = evalin('base', [get(src_block, 'Value') '.Value']);
    end
    assert(~ischar(length))
    set(find_system(block_handle, 'LookUnderMasks', 'all', 'Name', ...
        'Detector16'), 'in_hold', num2str(int16(length*100)+1))
end
```