Aalto-yliopisto
Insinööritieteiden korkeakoulu

Kaarle Patomäki

Simulation Tool Based Structural Text Generation for Programmable Logic Controllers

Thesis submitted for examination for the degree of Master of Science in Technology

Turku 23.7.2020
Supervisor: Professori Kari Tammi
Advisor: Juuso Kelkka

Aalto University
School of Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi

Abstract of master's thesis

Author Kaarle Patomäki

Title of thesis Simulation Tool Based Structural Text Generation for Programmable Logic Controllers

Master programme Mechanical Engineering **Code** ENG25

Thesis supervisor Associate Professor Kari Tammi

Thesis advisor(s) Juuso Kelkka, M. Sc. (Tech.)

Date 23.7.2020 **Number of pages** 50+8 **Language** English

Abstract

Model-based design is a relatively new technique of developing software for embedded systems. It aims to reduce the cost of the software development process by generating the code from a simulation model. The code is generated automatically using a tool that is developed for this purpose. This way the errors in the system can be found and eliminated early in the development process compared to traditional software development project for embedded systems. As mentioned, the tools are at the time of this study still relatively new, and especially when considering code that has to comply with functional safety standards, the code has to fulfill certain requirements and it has to be clear enough so that it can be traced back to each function of the model. This study aims to determine how well these methods can be used with software development for embedded systems in mind. More precisely, this thesis focuses on MathWorks' Simulink as the modelling software, and CODESYS as the coding language of the programmable logic controller and ultimately the compatibility of these with each other. The workflow of a model-based design software generation process is determined and presented as the result of this study. That process includes building, testing and verifying the model, preparing it for code generation, configuring and using the code generation tool and finally verifying the generated code. An example model of a battery cell balancing system for the code generation process is built, and thus that area is also studied. In the end of this study, some different possible uses of this technique are briefly discussed as well as further possible areas of study regarding this topic.

Keywords Model-based design, CODESYS, Simulink, PLC Coder, IEC 61508, Functional safety, IEC 61131, Programmable logic controller

Aalto University
School of Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of master's thesis

Tekijä Kaarle Patomäki

Työn nimi Laiteohjaimien Structured Text -kielisten ohjelmien luonti käyttäen simulointityökaluja

Maisteriohjelma Mechanical Engineering		**Koodi** ENG25
Työn valvoja Professori (Assoc.) Kari Tammi		
Työn ohjaaja(t) Juuso Kelkka, DI		
Päivämäärä 23.7.2020	**Sivumäärä** 50+8	**Kieli** englanti

Tiivistelmä

Mallipohjainen ohjelmistosuunnittelu on melko uusi tekniikka sulautettujen järjestelmien ohjelmistosuunnittelussa. Sillä tähdätään pienempiin kehityskustannuksiin luomalla järjestelmien koodi suoraan simulointiin tehdystä systeemin mallista. Koodi luodaan hyödyntäen automatisoituja työkaluja, jotka ovat kehitetty tähän tarkoitukseen. Näin toimien mahdolliset kehitysvaiheessa tulevat virheet voidaan huomata ja poistaa paljolti jo hyvin aikaisessa vaiheessa kehitystyötä verrattuna perinteiseen sulautettujen järjestelmien ohjelmistokehitykseen. Kuten mainittu, tähän tarvittavat työkalut ovat tämän työn kirjoittamisen aikaan vielä melko uusia, ja erityisesti turvallisuuskriittistä koodia ajatellen koodin on täytettävä tietyt vaatimukset ja sen on oltava riittävän selkeää, jotta tietyt osat koodista voidaan jäljittää vastaaviin osiin mallista. Tämän työn tarkoituksena on selvittää, onko nämä menetelmät käyttökelpoisia sulautettujen järjestelmien ohjelmistokehitystä varten. Erityisesti tämä työ keskittyy MathWorks:n simulointiohjelmistoon Simulink, sekä ohjelmoitavan logiikan yhteydessä käytettyyn ohjelmointikieleen CODESYS sekä näiden yhteensopivuutta tätä prosessia ajatellen. Mallipohjaisen ohjelmistosuunnitteluprosessin suositeltu työnkulku mainittuja työkaluja hyödyntäen määritellään ja esitetään työn tuloksena. Tähän prosessiin kuuluu mallin rakentaminen, sen testaaminen ja toiminnallisuuden todentaminen, sen valmistelu koodin luontia varten, koodin luontiohjelmiston määritys ja käyttö sekä lopulta luodun koodin testaaminen ja toiminnallisuuden todentaminen. Esimerkkinä rakennetaan malli, joka tasapainottaa akkukennojen jännitteitä, jonka vuoksi myös tätä aihetta tutkitaan hieman. Työn lopussa käsitellään lyhyesti mahdollisia erilaisia tätä tekniikkaa hyödyntäviä sovelluksia sekä pohditaan millä tavoin tätä aihetta voisi tutkia edelleen.

Avainsanat Mallipohjainen suunnittelu, CODESYS, Simulink, PLC Coder, IEC 61508, toiminnallinen turvallisuus, IEC 61131, ohjelmoitava logiikka

Preface

First, I would like to give special thanks to my advisor M. Sc. Juuso Kelkka and my supervisor Associate Professor Kari Tammi. In addition, I would like to thank my co-workers in the Valmet Automotive Product Development -team for finding time for my questions. I would also like to thank Valmet Automotive for funding this thesis. I have found it very interesting to work with model-based design, since it meets my personal areas of interests and studies in Aalto University very well. Yet it has offered me the opportunity to be challenged with new working environment and areas of work. Also, working in the automotive industry has been a long-time dream for me, and I am very thankful of having got the change to write my thesis within this industry. I would like to thank my late father Lauri for always supporting my interests in mathematics, physics and especially technology by explaining and teaching things that I showed interest in. Also, I want to thank my mother Marjukka for giving opinions and advices regarding formal writing, and finally, my partner Maria for being extremely understanding and supportive through the writing process.

Kaarle Patomäki

Kaarle Patomäki

Contents

Nomenclature

BMS	Battery management system
DC	Direct current
FB	Function block
FBD	Function block diagram
HIL	Hardware in loop
IEC	International Electrotechnical Commission
IL	Instruction list
LD	Ladder diagram
NRMM	Non-road moving machinery
PLC	Programmable logic controller
POU	Program organization unit
SFC	Sequential function chart
SIL	Safety integrity level *or* Software in loop
SOC	State of charge
ST	Structured text

1 Introduction

When developing software for commercial vehicles and non-road moving machinery (later referred to as NRMM) the programs can be quite complicated and include sophisticated algorithms, for example calculating the state of charge or balancing battery cell voltages. In this thesis, programming software like this for programmable logic controllers, or PLCs, is the final target. However, manually handwriting software can be relatively laborious.

Today there are optional ways of creating software for PLCs, fortunately. One of these ways is to develop the software by using model-based design methods, which means to build the system as a simulation model and then generate the code based on that model using some tools. In this thesis, MathWorks' Simulink is used as the modelling tool, and then use additional tools to generate IEC 61131 -compatible code from them. The ease of building models in Simulink is mainly thanks to the wide range of built-in algorithms of MATLAB and Simulink.

1.1 Goal

This paper studies the possibility and workflow of translating Simulink models into CODESYS 3.5 structured text coding language, that is one of the IEC 61131-3 standard based languages that are widely associated with PLCs. The translation is done with using MathWorks' Simulink PLC Coder software. The workflow also takes into account the possible requirements of the functional safety (IEC 61508) standard. In this thesis, the things to consider regarding this code generation process while meeting the functional safety requirements are explained using only one model as an example, although in reality a battery cell balancing algorithm model like this would not need to meet the functional safety requirements. However, the discussion is more about if the code generated is sufficient for both scenarios since the safety standards are fulfilled more on the project level than within this process. It is still discussed that what kind of tools would be preferable to use when dealing with safety-related code. For demonstration purposes of the process, a Simulink model for determining the need for battery cell voltage balancing is created and then translated into CODESYS version 3.5 structured text.

1.2 Scope

In the next chapter, the software to be used is studied with this translation and testing process in mind. MathWorks' Simulink and CODESYS are presented briefly, as well as the Simulink PLC Coder, along with their restrictions within this process. The IEC 61131 standard, especially the software part (part three) is studied more thoroughly, and the programming languages included in that are explained. Some general code building guidelines are discussed, as well as the IEC 61508 standard for safety critical code testing. Finally, in the last section of the second chapter the process of balancing the battery cell voltages inside the battery pack is studied for building the example model.

After discussing the methods of the study, the third chapter presents the workflow of building the model for such translation, and to determine what has to be taken into account in each phase of the process. Those are presented as findings of this study, which are based on the information studied in chapter two as well as examination of the software while building the example model. Also, it is presented what kind of restrictions are there in that process and what are the preferred format and structure for the models. After building the model, all the preparations required for the translation process are presented, as well as the workflow of translating the code. The phases of testing of the code are listed with this specific process in mind. The whole process is presented in general in the beginning of this chapter as a flowchart figure (figure 3.1).

In the last chapter, the benefits, drawbacks, and problems related to using model-based design within this context in software development is concluded. In addition to this specific translation process, the diversity of this method of producing CODESYS structured text is shortly presented regarding other possible applications regarding this area. Also, the possible further objects of study regarding this topic are discussed, as well as the possible short comings or things to consider when reading the results of this study.

2 Methods

Developing software for programmable logic controllers used in the commercial vehicle applications and NRMM is a very exhaustive and laborious process compared to some other software developing. To meet all the industry related standards, the software engineer must have a good knowledge in the standards and in system development in general. Especially the testing and verification processes require this knowledge, and those can be extensively laborious. In order to ease and speed-up the process where it can be done, some additional tools for the software development process can be used.

In this chapter some possible tools to ease the development process are studied. This thesis focuses on the possibility to use MathWorks' Simulink and Simulink PLC Coder to be able to develop some functions for the software as model-based design in Simulink environment, and then translate the models using the PLC Coder to CODESYS structured text format. Structured text is a coding language based strictly on IEC 61131-3 standard, that is widely used with programmable logic controllers. The focus is on what the tools are, what they offer, and especially how the tools affect the efficiency of the whole development and testing process. Since the testing and verification phase is such a big phase of the whole process, it is examined if the tools make the software more or less laborious to test and verify. Even if the code writing phase becomes easier to do, the tools can still make the whole process harder, if the code from the PLC Coder is difficult to test or verify compared to handwritten code for any reason. Also, the theory behind the battery balancing is studied briefly in the end of this chapter in order to create a model for demonstration purposes for the process.

2.1 MathWorks' Simulink and Model Based Design

This section briefly describes MATLAB and Simulink and discusses the reasoning for using this software for programming software for logic controllers.

Model-based design has become more and more attractive in designing software for programmable logic controllers. When combined with automated code generation tools, such as the Simulink PLC Coder used in this study, model-based design can offer excellent tools for designing complex event-driven software for logic controllers (Conrad, 2009). From verification aspect, which is more studied later on, model-based design can decrease the amount of errors caused by mistakes in hand-written software, or *systematic errors*. This, however, creates even more importance in making the models in Simulink to function correctly. (He, Oke and Allen, 2016)

MathWorks' software Simulink offers an intuitive way of producing models of dynamic systems for digital signal processing using block diagrams. Simulink is integrated with MATLAB environment, and thus with Simulink models all MATLAB's built-in algorithms can be exploited. As Zamboni (2013) states in his book, Simulink is an excellent tool for "large companies manufacturing safety-critical products". These kinds of products usually require a specific software development workflow, which includes specification phase, development phase and testing phase. (Zamboni, 2013)

The functionality specification phase is common to all engineering: the better the requirements for the software is known, the easier the development phase usually is. While the documentation for requirements need to be highly specific and cover as much possibilities as possible, the testing of the requirements and the logic to fulfil the requirements are usually laborious to do for single specifications before the development phase. Finding out an error in the specified logic during the development process, or worse still, during the testing process, can create excessive amounts of extra work compared to making the specifications right in the first place. (Zamboni, 2013)

As mentioned in the introduction, an example of battery balancing program is used throughout this thesis. That topic is more thoroughly discussed in section 2.7.

We can define a requirement, that states:

Get the battery cell voltages (21 cells) in the module and the minimum required voltage to activate balancing as inputs. For each cell if the voltage is higher than the required minimum voltage and if the voltage is at least 0.1V higher than the lowest voltage in the module, then balancing should be set as ON. As an output the information for balancing for each cell is returned.

It is easy to see someone writing a description for a function like above. A description like this makes it possible to implement the function in multiple different ways as long as it meets the requirements mentioned in the description regarding inputs, outputs and functionality. However, this unfortunately also makes it possible to implement the function with different functionality than what is intended. This specification does not for example specify in what format the input and outputs are to be expected and returned: should they be in a container or are they 21 different variables, and as what data type the balancing information is returned. This is always quite hazardous, since some other programmer doing some other system on the same project, could handle the information returned by this function in a way that it is not supposed to be handled.

Creating requirements as Simulink models combined with written explanation leaves no room for errors like above, since these questions cannot be left open with the models. As seen from figure 2.1, all the logical requirements are unambiguously presented. Other developers can also easily and without the possibility of an error glance through the model and gather all the information from it that they need. In addition, as mentioned before, the Simulink model can be easily tested with for example Simulink's built-in signal builders.

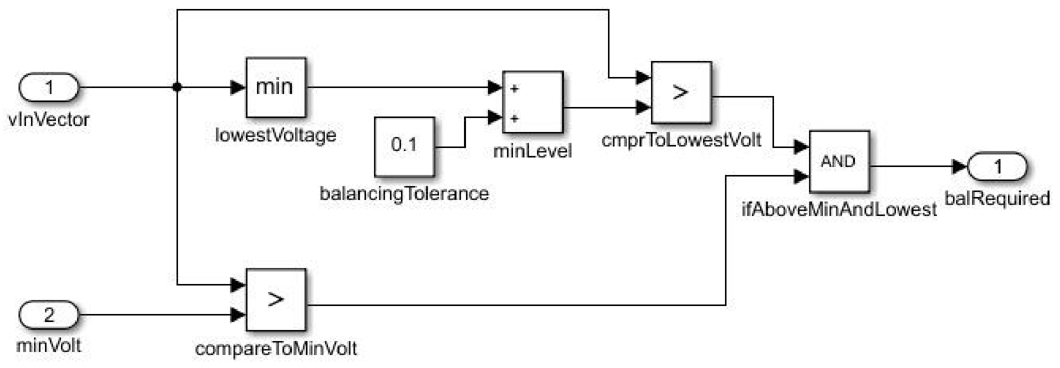


Figure 2.1 Example requirement specification as a Simulink model.

2.2 Programmable Logic Controllers and IEC 61131-3

This section presents programmable logic controllers in general and has a focus for the programming phase. The standard IEC 61131-3 that includes the standards for the software of the controllers is presented for the part that is relevant to this study.

Programmable logic controllers, or PLCs, are basically used to control a machine or device that has some sort of electro-mechanical functionalities as a bridge between the different parts of the machine. It is microprocessor based, and it stores instructions for handling the inputs and outputs into its reprogrammable memory. The instructions are usually written using a computer that is connected to the PLC device. PLCs are intended to be used also by engineers who may not have extended knowledge of programming, but maybe still want to develop an automation system for their product. The ways of programming a PLC device somewhat varies between different PLC manufacturers, but usually the principles are the same regardless of the device. The reconfigurability of PLCs also gives the opportunity to use the same controller and wiring for different functionalities and applications: only the programming must be changed. (Bolton, 2009)

Therefore, it could be said that PLCs are basically small computers dedicated into industrial use. They are usually built with their possibly rough working environment in mind, including varying and possibly extreme temperatures, increased vibration and high humidity. As the hardware of a PLC goes, it consists of a processor, that processes all the data, a memory to store the instructions written by the user, a memory to store internal data, a power supply, the input and output interfaces and also a communication interface for user communication (for example for saving the instructions to the PLC's memory). (Bolton, 2009)

As mentioned, programming of different PLCs can be done usually relatively similarly, thanks to IEC 61131-3 standard. The standard was published in 1993 after manufacturers started making PLC devices, and all of them building their own environments and programming languages that could differ quite a lot from one to another. As a result, today virtually all major PLC manufacturers comply with this standard with their products. After all, the IEC 61131-3 is not an absolute standard, but a guideline. Thus, the manufacturers are still able to do things that are not considered in the standard. (Hanssen, 2015)

This standard defines various programming languages designed for PLC devices, and they are:

- Structured Text (ST)
- Function Block Diagram (FBD)
- Ladder Diagram (LD)
- Instruction List (IL)
- Sequential Function Chart (SFC)

These programming languages can be split into two categories: graphical and text based. FBD, LD and SFC are graphical while ST and IL are text-based. (Hanssen, 2015)

Regarding the text-based languages, ST is a so called "high-level language", and it is much more advanced and more complex language compared to instruction list. It is also the most comparable language to some common programming languages like C or Pascal amongst the IEC 61131. This way, compared to IL, ST code can usually be written in a smaller space meaning that there can be more than one instruction per line of text. However, this also means that the translation to machine code is more complicated, and the final result can be less efficient when the code to be translated is complicated. A ST program consists of *statements,* which are separated from each other with a semicolon, like in C. *Comments* in code are framed by parentheses and asterixis, like following: (* comment *). They can be placed in wherever a space can be placed, which means that they can be placed within statements, and not only between them. Also, like in C, jump-statements are not used in ST, but instead IF structures are. The contents of a ST-program is explained in the next section, and an example of one can also be seen there. (John and Tiegelkamp, 2010)

IL on the other hand is a simpler language with more similar approach to what the translated machine code in the end will look like (execution-wise). Every *instruction*, which is a single executable command, is written in one single line. Like for example in BASIC, all instructions can be optionally labelled in order to reach them with a jump-statement within the program. Should a label be used, it has to be followed by a colon. After the label, the operator or the function of the instruction is stated, after which the operand(s) is stated. Comments can be used the same way as in ST. The IL is not one of the supported languages by the Simulink PLC Coder. As far as developing software with IL goes, the code tends to be much longer in terms of lines of code, and writing and reading it can be a little unintuitive to someone who is used to develop software in more traditional coding languages. (John and Tiegelkamp, 2010)

Regarding the graphical IEC 61131-3 languages, FBD and LD are relatively similar in the way they work. FBD is a traditional tool used with signal processing, but today it is also one of the most used languages amongst industrial controller devices. Basically, FBD and LD consist of the same parts as ST and IL, which are a leading and ending part of the program, the declaration part and the code part. Although FBD and LD are graphical languages, the declaration part can be either textual or graphical depending on the development software used. The code part, on the other hand, consist of *networks*, which include a label, a comment and a graphic. The label is an identifier for the network, and it can include both letters and numbers. The comment can then be inserted to explain the network before the graphical part again with the same delimiters as with ST and IL. The use of comments depends somewhat on the development software used, since comments for the graphical languages are actually not described by the IEC 61131-3, but this is the case most of the time amongst these languages. The network graphic then includes the graphical elements and connections between them, in which the information travels from one element to another. The combination between the information of different elements can be done with *crossings* in the connections. Whereas FBD is originally used mainly with integers and floating point numbers, LD is targeted to mainly being used with signals carrying Boolean values. The basic layout of an LD system is similar to the one in FBD, but the graphical part is different. In LD, the layout of the program is clearly based on logical conditions more than elements including values. LD is the other IEC 61131 -language supported by the Simulink PLC Coder, but the support is some what limited compared to ST. (John and Tiegelkamp, 2010)

Finally, there is the SFC, which has been built for being able to break down complex programs into smaller pieces. The main parts of SFC are steps and transitions. SFC programs are able to run different processes either sequentially or parallelly. The processes in the SFC program can include other programs from different IEC 61131-3 languages. (John and Tiegelkamp, 2010)

In this thesis we focus on Structured Text, and the Simulink models are translated into CODESYS 3.5 ST specifically. From the two languages supported by the Simulink PLC Coder, ST is the one with more supported functionalities. As a programming language it also tends to be more intuitive to especially software engineers compared to LD. (Hanssen, 2015)

In IEC 61131-3 a structure of a program is described to have a specific hierarchy (figure 2.2). The upmost level is the *configuration,* which includes the description of the hardware, memory addresses for the input and output channels and system capabilities. In one configuration, there can exist one or more *resources*, which is described in the standard as "consisting of a signal processor unit with its user interface and functions for sensor and actuator interfaces." So basically, a resource is a processing facility capable of running *programs*, and there are one or more of them in a configuration. Programs are built of *function* and *function blocks*. (Hanssen, 2015)

Within resources there can also be *tasks*, which are used to control how the resource uses its programs basically. The tasks can determine the order of running the programs for example by ordering them by their importance or having them running in a chronological order. Also, tasks can interrupt the running programs if some other program is required to run instead. (Hanssen, 2015)

Variables are data objects that can be accessed to read and write. They can be either associated with inputs or outputs, or they can be stored in the PLCs memory. All variables must be declared and the data type to be used with the variable defined. They can be global variables meaning that they can be accessed from every program and function within all the resources in a configuration, or variables can be declared within a program, function or function block and thus available only within that same level where they are declared to. (Hanssen, 2015)

In IEC 61131-3 all the programs, functions and function blocks are considered as Program Organization Units, or POUs. Functions and function blocks differ mainly in memory handling. Functions can, too, have internal variables, but they are reset every time the function is called. Basically, a function takes an input vector, does some logical and mathematical operations, and outputs the result. With the same input vector, the output should be the same every time. In contrast, a function block can have variables that are stored to internal memory even while the function is not called, so it is possible to output different values with the same input vector when dealing with function blocks. (PLCopen, 2016b)

In the next chapter, a language based on this standard called CODESYS, and more specifically its version 3.5 and structured text are presented.

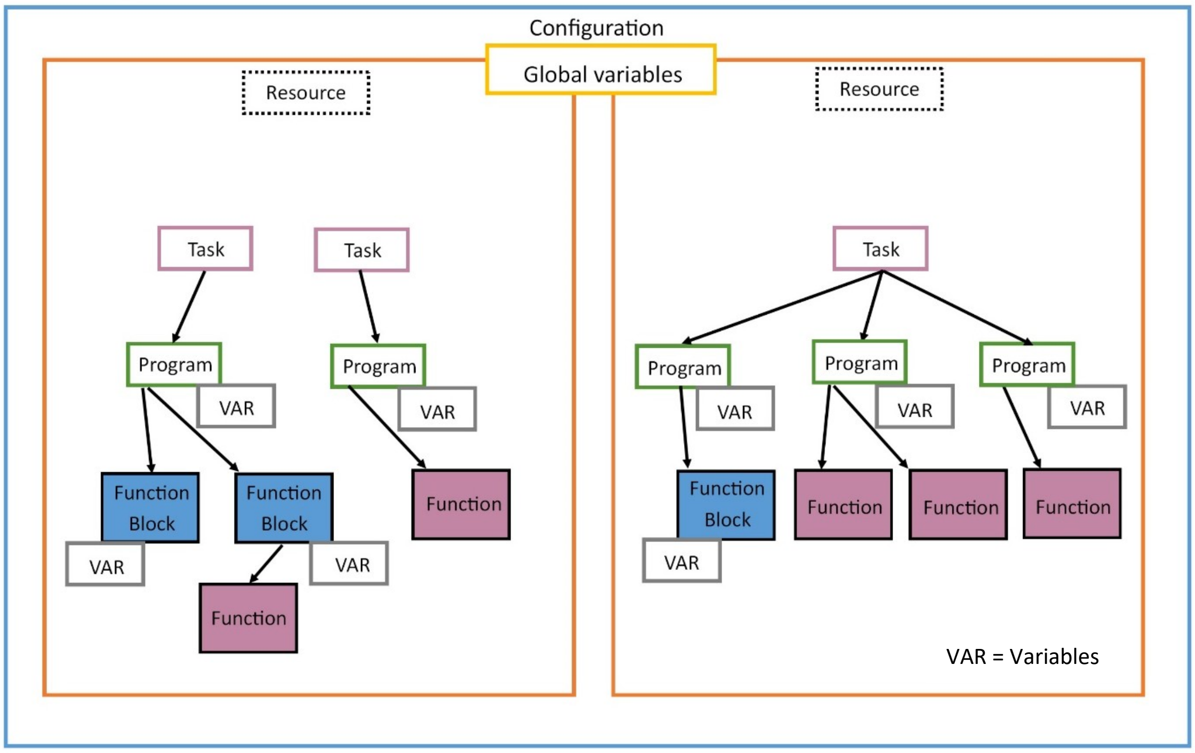


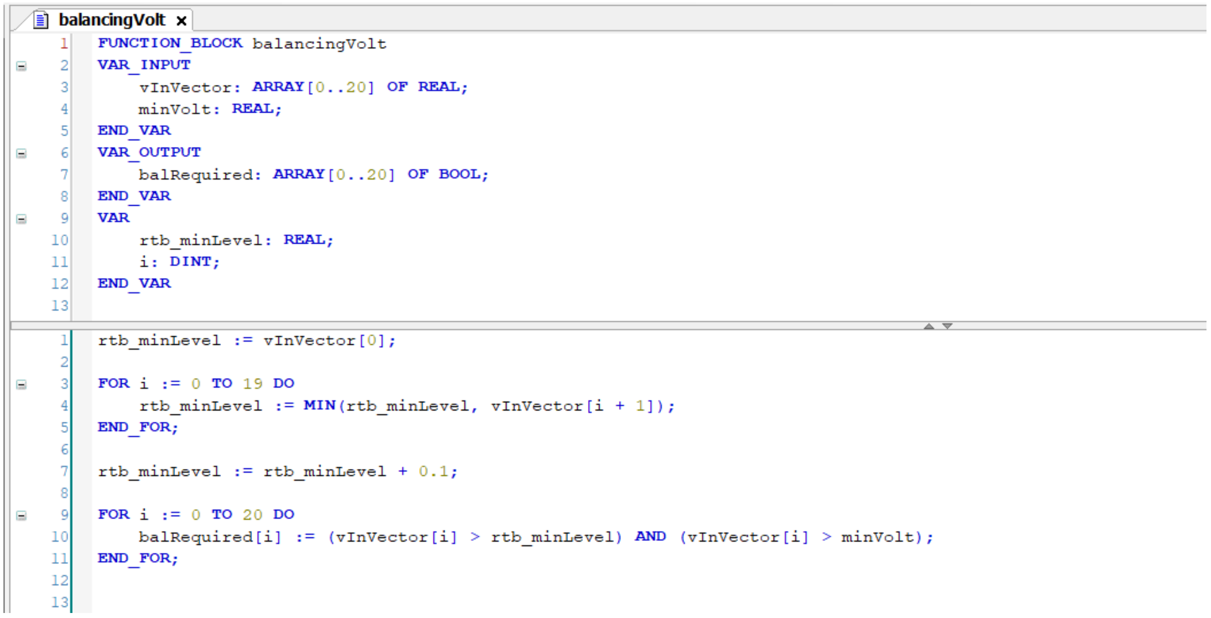
Figure 2.2. An example of the structure of a program defined by the IEC 61131-3 standard.

2.3 CODESYS and Structured Text

CODESYS is a programming tool implemented for PLCs or other programmable hardware that is very closely based on the IEC 61131-3 standard. As mentioned in the previous section, structured text (later referred as ST) is one of the languages defined by that standard, and it is also the one to be used in this thesis. CODESYS is not developed by any hardware manufacturer, but it is developed by a company that is focused especially in software development tools. The company is called Smart Software Solutions GmbH. CODESYS is not only supported, but also the only tool for over 400 hardware manufacturers. The software includes a device simulator so that the programs can be tested without actual hardware during the development process. CODESYS as a software is free to download and use. (CODESYS, 2020b)

CODESYS also offers a variety of tools that can be added in the basic programming environment to support the need of different companies and developers. For example, the Static Analysis and Test Manager tools can be used for the testing and verification process. As the name suggests, the Static Analysis tool is used for performing static analysis, or in other words, it is an automated tool to check that the code meets all the requirements set by certain rules. (CODESYS, 2020a)

From the IEC 61131-3 based languages, in this thesis structured text is used, as described in the previous section. Structured text is, as the name suggests, one of the text-based languages within IEC 61131-3. Basically, it shares the same structure as any other of the IEC 61131-3 languages (Hanssen, 2015). Since structured text is the language to be used in this thesis, the structure of a structured text -based program is explained briefly with an example. As an example, the cell voltage balancing algorithm is used, which was previously defined in Simulink in section 2.1.


balancingVolt

```
FUNCTION_BLOCK balancingVolt
VAR_INPUT
    vInVector: ARRAY[0..20] OF REAL;
    minVolt: REAL;
END_VAR
VAR_OUTPUT
    balRequired: ARRAY[0..20] OF BOOL;
END_VAR
VAR
    rtb_minLevel: REAL;
    i: DINT;
END_VAR

```

```
rtb_minLevel := vInVector[0];

FOR i := 0 TO 19 DO
    rtb_minLevel := MIN(rtb_minLevel, vInVector[i + 1]);
END_FOR;

rtb_minLevel := rtb_minLevel + 0.1;

FOR i := 0 TO 20 DO
    balRequired[i] := (vInVector[i] > rtb_minLevel) AND (vInVector[i] > minVolt);
END_FOR;

```

Figure 2.3. Example of a Structured Text Function Block shown in CODESYS development environment.

Figure 2.3 represents a typical, short example of a structured text function block. At the beginning, the function block is defined and given a name, which in this case is *balancingVolt*. When talking of the structure of the programs, they begin with the line **PROGRAM [program name]** and end with the line **END_PROGRAM** likewise. The controller device runs the program in a loop written between these two commands. Compared to C (the programming language), the **PROGRAM/END_PROGRAM** works the same way as a infinite loop would work in that. The END-line is not visible in the CODESYS development environment although it is in the actual file, which is good to acknowledge. This is, however, a function block, and so it begins with **FUNCTION_BLOCK [name],** and it ends respectively with **END_FUNCTION_BLOCK** statement, although not visible in the CODESYS environment.

After that, there are the definitions of the variables used in the function block. First, the input variables, then output variables and then local variables. As explained in the previous section, there can also be global variables that can be accessed from the function block without being declared separately. After defining the variables, the definition of the program/function starts.

2.4 Simulink PLC Coder

MathWorks also offers a software called Simulink PLC Coder (later referred to as PLC Coder) for generating "hardware-independent IEC 61131-3 Structured Text and Ladder Diagrams" from these models. This allows, according to MathWorks, the possibility to deploy the Simulink models to be used in different programmable logic controller (later referred to as PLC) and programmable automation controller (PAC) devices. (MathWorks, 2010)

MathWorks offers a complete, nearly 300-page long User's Guide for the PLC Coder, which is also widely referenced in this study. The PLC Coder has support for various IEC 61131 IDEs, including 3S CODESYS, B&R Automation Studio, Siemens SIMATIC and Open PLC (Yoon, 2013). However, the motivation for this chapter is to focus more on the specific data conversions from Simulink models to CODESYS 3.5 Structured Text (later referred to as ST) format.

The Simulink PLC Coder is a relatively new tool for Simulink, but it has gained a lot of interest in the business. In 2010, according to various magazines of this business area, the PLC Coder was actually approved by the TÜV SÜD to be used as a software developing tool for IEC 61508 applications. (Gomez, 2010).

2.5 Code testing and verification

Software development has become increasingly important in the commercial vehicle applications and NRMM in the last couple of decades. Regarding engineering within these industries, completely mechanically operating parts and functions are usually more reliable and less prone to unwanted functionality when they can function as designed. On the other hand, they usually are much less accurate, and they usually are worse in taking factors like wear or changing environment into account during operations without manual adjustment. While at the same time computers are getting cheaper and cheaper and simultaneously more and more powerful, it is easy to see why almost everything is controlled by a computer of some sort if by any means possible. (Conrad and Sandmann, 2009)

Increasing usage of software also means that software testing is in more and more critical part of commercial vehicle and NRMM engineering. Since software operates many safety critical functions, it is extremely important that the development process eliminates the possibility for bugs by following proper code building guidelines and testing standards. For this purpose, there has been developed standards to define the methods and workflow for software building for safety related systems, and when talking about commercial vehicle applications and NRMM engineering, IEC 61508-3 is the one to typically follow. However, in addition to the requirements in IEC 61508, it is good practice, and also required by the IEC 61508 standard itself, to have some predefined coding guidelines within the software development project. Regarding the context of IEC 61131, those coding guidelines can be based on for example on PLCopen Coding Guidelines that are made specifically for industrial environments (PLCopen, 2016a). (Conrad and Sandmann, 2009)

Generally speaking, we can mention a few particular targets of software testing. It has to fulfil the functionalities and requirements that were set for it during the design phase. An example of setting requirements for a certain program was given in chapter 2.1. The software should also be able to run and answer correctly regardless of the inputs. In practice this means that all functions should be tested with a sufficient amount of different input vectors, most importantly including the boundary values as well as some unexpected values. The software also needs to be able to run in its desired environment within a sufficient time range. (Hambling *et al.*, 2010)

Then again it needs to be remembered that it is nearly impossible to always cover every possible situation with tests. In reality, time is a limited factor with software projects as much as in any other project, and even relatively simple software testing implementation can take a bunch of time. When performing some initial dynamic testing for the software, the software engineer might easily think that “this seems to work well enough” or “I know what I’m doing, my software always works”, and move on to the next project that is still to be done. This is why it is important to always have some kind of guidelines for software testing and verification so that all software is guaranteed to be at least tested at the same level of detail. On the other hand, following the guideline software testing comes much more straight forward for the software engineer to do, and at the same time avoid extensive workload. (Hambling *et al.*, 2010)

Before talking more about software testing, a few terms should be defined. Testing and debugging should not be confused. Whilst testing in this context is something to be done ideally for an already finalized software, *debugging* is for testing a single function or other part of a program to identify what causes possible (obvious) issues with the software during development process. After the development process is finalized, the software has to be thoroughly *tested* in order to eliminate as many possible issues with the software as possible and it is a completely different phase from the development process. (Hambling *et al.*, 2010)

Testing can be divided into static testing and dynamic testing. The main difference between these two is that dynamic testing is done by running the program, whereas static testing is done without running it. Usually static testing, or *static analysis* is done by using some kind of automated tool. For example in CODESYS, the Static Analysis can be used in the code verification process.

In the next chapter, safety critical code is explained as well as the IEC 61508 standard and how and which parts of it are applied in this thesis.

2.6 IEC 61508 and Safety Related or Critical Code

IEC 61508 is a complete standard for safety related systems and their development, deployment and maintenance. Overall safety can be defined as being able to avoid directly or in-directly the risks of physical injury or of damage to the health of people or to the environment. Thus, *safety related* means something that has the ability to affect the safety of the system in-directly, or in other words not on its own. *Safety critical* is, on the other hand something that has direct effect on safety and can cause injuries or an accident should it fail. IEC 61508 has a general approach for safety related operations for electrical, electronic and programmable electronic (later referred to as E/E/PE) systems to perform safety related functionalities. Usually safety is achieved by applying many different technologies for the system so that if one technology fails, there is another technology to still prevent major accidents. For example, it is very common that all safety related hydraulic systems have some kind of mechanical safety function to prevent injuries or accidents in case of a hydraulic failure. (Smith and Simpson, 2016)

In this section we focus on chapter three of IEC 61508, which defines the “software requirements” for the “functional safety of electrical/electronic/programmable electronic safety-related systems”, as described in the international standard. Addressing safety related code on its own is a slightly difficult subject, because usually the safety of the systems is designed with the whole system in mind. For example, fatalities caused by errors in the code can be prevented with making an electrical design for the safety feature, for example a fuse. This thesis is not going to study the complete development process of a IEC 61508 compliant system, but in order to understand the requirements for a safety critical and non-safety critical code it is necessary to understand the whole concept at least to some extent. On the other hand, IEC 61508 only describes the requirements on a generic level and the requirements for each individual system are always different from one another, it might also be difficult to determine the requirements for any code in general. This is one of the reasons the example program of the battery balancing algorithm is useful in this study. (Conrad and Sandmann, 2009)

The targeted safety level can vary depending on the system to be assessed. Thus *Safety Integrity Levels,* or *SILs* have been developed within the standard. The safety level have to be addressed *quantitatively* and *qualitatively.* Summed up, quantitatively refers to hardware failures, which can be usually predicted using the history of experiences of the hardware to be used. Qualitatively refers to *systematic failures*, or in other words, software failures. Qualitative safety is the part of the assessment that this thesis focuses on regarding this standard. (Smith and Simpson, 2016)

Quantitative failure rate targets can be divided into two categories: high and low demand failure rates. In this, the demand refers to the predicted usage rate of the system. For example, in a passenger vehicle, headlights can be considered as a high demand safety related system since should they fail during use, the failure is prone to cause a fatality immediately after the failure occurs. On the other hand, airbags may not be used during the whole lifetime of a car, and if a fault occurs to the system, it does not necessarily mean that a fatality is followed. (Smith and Simpson, 2016)

Table 2.1. Safety Integrity Levels. (Smith and Simpson, 2016)

Safety integrity level	Continuous and high demand rate (dangerous failures/hr)	Low demand rate (probability of failure on demand)
4	$\geq 10^{-9} to < 10^{-8}$	$\geq 10^{-5} to < 10^{-4}$
3	$\geq 10^{-8} to < 10^{-7}$	$\geq 10^{-4} to < 10^{-3}$
2	$\geq 10^{-7} to < 10^{-6}$	$\geq 10^{-3} to < 10^{-2}$
1	$\geq 10^{-6} to < 10^{-5}$	$\geq 10^{-2} to < 10^{-1}$

The quantitative failure rates are presented in table 2.1. As mentioned before, quantitative failures are quite different to the qualitative ones, and meeting these numbers with the quantitative failure rates does not mean that the qualitative requirements are met, but they have to be assessed separately with different requirements. Whereas quantitative failure rates can be calculated and analysed assessing the rate of random hardware failures and their combinations, qualitative failures cannot be predicted with numbers, but they are to be limited with a variety of design disciplines that are according to the risk target profile. (Smith and Simpson, 2016)

There are a number of steps to be taken in the assessment process of IEC 61508 standard, and a brief presentation of those is following.

The **1st step** is to establish functional safety capability. This especially refers to the party that is responsible for the design of the safety related system that is being assessed. For this step it is important to be able to present the capability and competence for designing, producing, operating and maintaining the system that is functional safety related. This means the assessment of management procedures using the given check lists by the standard and of course the implementation of these procedures. (Smith and Simpson, 2016)

The **2nd step** is to establish a risk target. This step involves performing a full failure and hazard analysis for the system to come up with an outcome of all the possible hazardous events that can lead to serious injuries or worse. After that the maximum tolerable failure rates has to be set by performing a quantified risk assessment with the maximum tolerable fatality rates in mind. (Smith and Simpson, 2016)

The **3rd step** is to identify the safety related functions, which means to identify all the possible events leading to fatalities and furthermore, to identify the safety protection system for which a SIL is required. (Smith and Simpson, 2016)

The **4th step** is to establish the safety integrity targets for the safety-related elements. There are multiple ways to carry out this step, but basically it is a structured risk analysis. The possible ways of performing this include the quantitative approach, the layer of protection analysis and the risk graph method. Again, this thesis does not focus on this part of the standard and thus these methods are not discussed any further. The outcome however should be the target SIL. (Smith and Simpson, 2016)

The **5th step** is the quantitative assessment of the safety-related system. This step targets to determine the probability or rate of failure of the safety-related elements in the system, which is then to be compared with the target values. (Smith and Simpson, 2016)

The **6th step** is the qualitative assessment compared to the target safety integrity levels. This is one of the more relevant step within the area of study in this thesis in mind. This step targets to set the actions and requirements for limiting the systematic failures to comply with the target SIL. This step is discussed more later on in this chapter. (Smith and Simpson, 2016)

The **7th step** is to establish the principle "as low as reasonably possible", or as often referred, *ALARP*. Establishing merely the maximum tolerable risk target is not sufficient according to the standard, but it is necessary to examine possible further improvements and discuss whether they are reasonable to carry out or not. (Smith and Simpson, 2016)

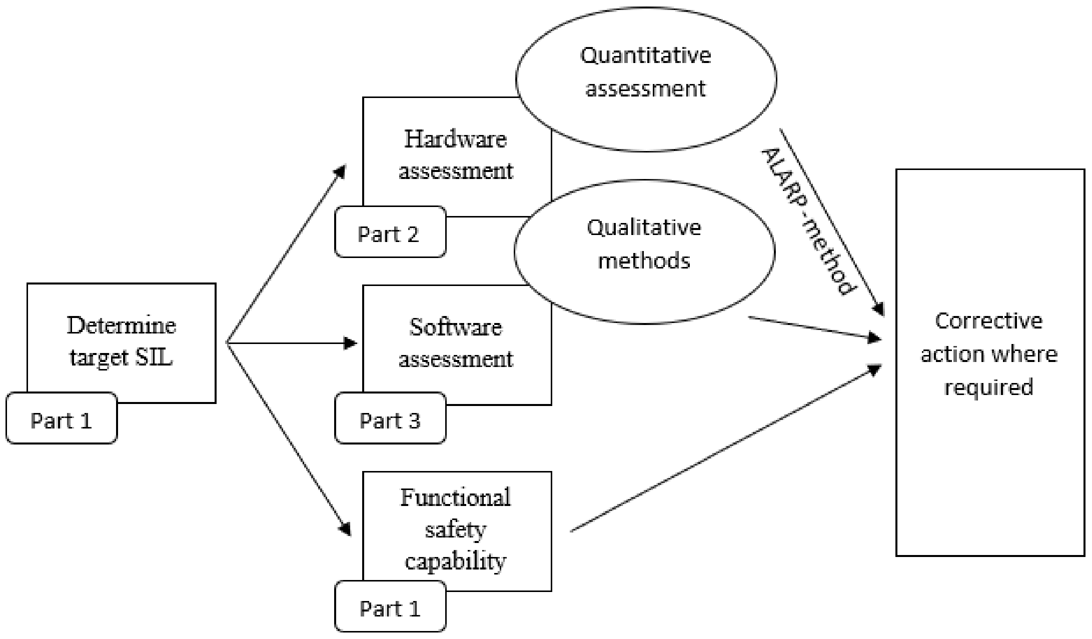


Figure 2.4. The main parts of IEC 61508. (Smith and Simpson, 2016)

There are three main parts in IEC 61508 (1-3) and four supplementary parts (4-7). The main parts are presented in figure 2.4 with the assessment process in mind.

The first part is called "General Requirements". It covers general functional safety management definition that covers the required actions and skills to perform the risk assessment and design to meet the standard. Also, it covers the life-cycle definition and requirements for each phase, and it explains what are SILs and the necessary procedures in order to determine the safety integrity target level. In addition, the requirements for people working with safety-related systems are presented, and the independence requirements for each SIL. Finally, the first part includes an example model for the document structure for such an assessment. (Smith and Simpson, 2016)

The second part is called "Requirements for E/E/PES safety-related systems", which means in practice to define all the requirements for the hardware and hardware related work in relation to functional safety. More precisely this means the life-cycle activities with the hardware, the quantitative reliability requirement as per the SIL level in question, the definition of systematic hardware failures and means to defend the system against them and finally architectural constraints, which mean the further safety measures to the safety-related hardware systems in addition to the required as per the SIL.

The third part is called "Software requirements", and as the title suggests, it consists the requirements and information required for the software design and assessment. As mentioned before, it is not relevant to think of any quantitative failure rates with software, so this part of the standard only focuses on minimizing the systematic failures with the safety-related software within the system. Again, various techniques and requirements are presented for each SIL.

The other four parts include further definitions for the terms used within the standard, some examples for the determination of the SILs, some informative annexes about calculating failure rates for the hardware, common cause failure, diagnostic coverage and applying the software requirements tables, and finally a reference guide to the techniques and measures of the standard. (Smith and Simpson, 2016)

In the next section, part 3 of IEC 61508 is studied more thoroughly in order to determine what is required from the source code gathered from Simulink PLC Coder software to be able to be used within the standard.

2.6.1 IEC 61508 Part 3 – The Software Requirements

The development of a software for a safety-related system includes first of all the requirement for organizing and managing the life cycle. This is relevant to all parts of IEC 61508, but is not relevant regarding the code converting and verification process as such, and thus it is not described more deeply. There is, however, a certain number of mandatory requirements for the standard regarding the organizing and management of the party responsible of the product to be certified. Some of the disciplines might be familiar from ISO 9001 standard that usually is applied before even thinking of producing a product that complies with the functional safety standard. While most of the requirements of this area apply to the whole standard, some of them apply only when a certain SIL is exceeded. (Smith and Simpson, 2016)

Most important single requirement is documentation. Without it proving the compliancy with the standard is impossible. Also, a project management structure to divide the responsibilities of the project to different people is important, as is to be sure of their competency. A quality and safety plan is also mandatory. That is a document that is supposed to determine the whole structure for documentation as well as the overall functional safety targets and plans. Also, a V-model should be used with the software design. An example of what could be such graphical figure of a V-model for a software design process is presented in figure 2.5. The process, according to that model, starts from the up-left corner of the model, which includes the safety requirements specification. After completing the specification and design steps, the actual coding can be done. For each step in the design phase, a testing phase should occur after the coding in reverse order. For example, the completion of module design should be verified with module tests, the software system design should be verified with module integration testing (that tests how the software works with the system) and so on. However, the life cycle should, as mentioned before, be documented as writing in detail. The graphical figure is mostly for presenting and clarity purposes. (Smith and Simpson, 2016)

Whatever the chosen method(s) for the design, the standard lists a set of requirements that they need to fulfil. For the system software it states that a diagnostic software is required for diagnosing faults in the system hardware and communication links as well as live testing of the application software modules. Also, should a fault occur, the system should be able to continue working if it is possible with the faulty part of the system disabled and isolated. For SIL1/2 and SIL3/4 there are different requirements for fault detection. In a system, where there are someway or other linked together non-safety and safety functions, the standard lists things to consider regarding possible faults. The list includes mostly remarks for using shared hardware for example memory, processors, and the communication interfaces. (Smith and Simpson, 2016)

Regarding the code itself, the standard states that it should be completed in small and manageable software modules. Software development process for all SILs should include design and coding standards as well as semiformal methods. Systems with SIL 2 or above should always, when possible, use trusted software modules only that has previously been in similar use. Dynamic objects that depend on the current state of the system when allocated should not be used with any of the SILs. For SIL 2 and up dynamic objects should not be used at all. For SIL 3 and 4 there are a few more requirements for the coding that make the code more "robust", including defensive programming and avoiding pointers and recursion in the code. (Smith and Simpson, 2016)

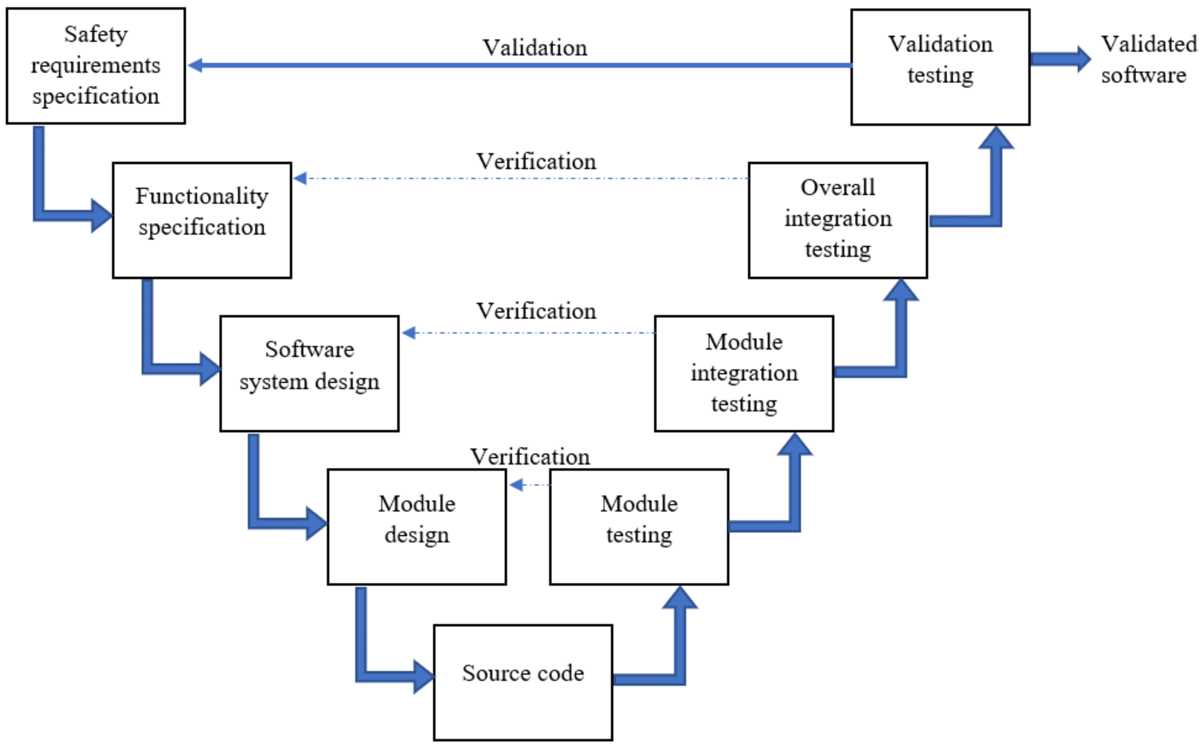


Figure 2.5. Development process for software within IEC 61508 presented in a V model. (Smith and Simpson, 2016)

The coding language used should be completely defined with no indistinction for all SILs. The support tools, like code compliers and static analysis software, should be at least proven in use or certified. With SIL3/4, the standard strongly recommends using certified tools only. (Smith and Simpson, 2016)

The IEC 61508 standard states the verification and validation of the software as follows. All of the software modules should be reviewed individually, and tests should be made to prove that they do not have any unwanted or unnecessary functionality as well as having the previously specified functionality. For this operation, a limited set of test data has to be chosen with the SIL in mind so that the higher the SIL is, the larger the test dataset should be. For SIL 1 and 2, tests with boundary values and partitioning testing should be sufficient, whereas with higher SILs additional tests should also be performed. (Smith and Simpson, 2016)

As figure 2.5 suggests, after testing the operation and functionality, the module integration should be verified according to the cases defined in the software system design phase. These tests should determine aspects like functionality within the system as well as performance in the system. All of the tests must be documented chronologically. Should faults occur during verification, they should be documented with the suggested corrective actions. The modules and test instructions should have version numbers that are included in the documentation. Also, the effects of any modifications made during the development process to the software should be analysed to determine the required tests to be done to the system. (Smith and Simpson, 2016)

After verifying the software integration, the software can finally be validated. This means that at this stage the software can be fully accepted to work as a whole meeting all the requirements and design procedures set for it. The validation plan must prove that all of the safety requirements have been fulfilled with a pass/fail checklist. From SIL 2, the validation should also include the test coverage metric, and from SIL 3 even more details from the coding process relevant to meeting the standard are required. (Smith and Simpson, 2016)

2.6.2 Model Based Design with IEC 61508

As this thesis focuses on the process of designing software with Simulink, and then translating that to structured text, it needs to be discussed how the IEC 61508 standard can be applied to this process. Fortunately, some research of this topic has been done. In this section, previous research of this topic is studied, and the most relevant findings for this thesis are presented.

Theoretically all code produced by the Simulink PLC Coder could be validated as any other code as if it were originally coded by hand. This process is presented in the previous section. However, performing the code as model-based design means performing part of the verification and validation for the model built for the translation. MathWorks offers a verification software for Simulink called IEC Certification Kit for ISO 26262 and IEC 61508, which makes the process of code verification quite efficient, and allows most of the errors to be determined early in the design phase.

Generally speaking of the process, two basic questions regarding Model-Based Design according to IEC 61508-3 are:

- Does the model correctly implement its (textual) requirements?

- Does the object code that will be deployed in the ECU correctly implement the model’s behaviour? (Conrad and Sandmann, 2009)

This means that in order to complete the verification and validation activities necessary for IEC 61508-3 compliance, it has to be proved that the model is correct and meets all the requirements set for it, and to be shown that the generated code functionally matches the model’s operation. (Conrad and Sandmann, 2009)

The goal for the *design verification* is to have a “golden model”, which means it has all the required functions, and on the other hand does not have any unwanted functionality. The workflow for this verification includes reviews, static analyses, and diverse functional testing, and it is all done for the model before generating the code. After the modules (subsystems) have been tested thoroughly, the module integration should be validated as well. This means that all modules should interact with each other as intended with a variety of predefined input parameters (test vectors), and once again, nothing unintended should occur. (Conrad and Sandmann, 2009)

The model subsystems are to be reviewed in Simulink with model testing. Also, in the process of developing this software, some modelling guidelines should be made and exploited to make the reviewing process more efficient. In addition to the guidelines, the models should be constructed following the rules and restrictions set by the code generation tool, which is Simulink PLC Coder in this case. (Boulanger, 2012)

The test cases for the model should be created according to the software requirements specification. As mentioned before, module testing should be able to determine that the software has its specified functionality but does not have any unintended functionalities. After the modules have been thoroughly tested individually for correct functionality, the integration of the modules must be tested. The result from that should be again that no unwanted functionality occurs, and the software performs as specified in the requirements. (Boulanger, 2012)

Firstly, the *code verification* includes numerical equivalence testing, the comparison being the model of which the code is generated from. The exact same test vectors must be used to test the generated code and compare its outputs with the model results. The environment for the code testing should correspond to the environment that the code is intended to run on, or alternatively the differences between the testing environment and the target environment should be analysed sufficiently. Exact or even close equality is difficult to achieve for example due to “limited precision of floating-point numbers, quantization effects when using fixed-point math and compiler differences”. After the equivalence testing, the code must be tested for unintended function. This can be for example by comparing the coverage of the model and the generated code, or alternatively by performing a complete traceability review. (Conrad and Sandmann, 2009)

2.7 Battery Module Cell Voltage Balancing

As an example in building a Simulink model following the guidelines, translating it to CODESYS structured text and testing it’s functionality as per IEC 61508 standard, a model of battery module cell voltage balancing is designed and developed for this thesis. This topic has been widely researched before, and battery balancing is not within the main research questions in this thesis. Thus, the chosen method for battery balancing is relatively simple, well known and widely used with large Lithium based battery packs so that the code building, conversion and testing process itself can be relatively easily examined and presented without requiring too much knowledge on the principles of the example program functionality. That being said, this battery balancing program built for this thesis is ultimately supposed to be a part of a final working battery management system, and for this reason this field is also studied to some extent in this thesis.

This chapter discusses the reasons for battery inconsistencies, the effects of batteries not being in balance when being in the same configuration and also some possible solutions for making keeping the batteries in as much balance as possible regardless of the inconsistencies, and how to avoid further inconsistency.

2.7.1 Battery Inconsistencies

Even batteries from the same manufacturing batch can be inconsistent. The inconsistencies in the manufacturing process are mostly caused due to reproduce the same product exactly. These kinds of inconsistencies would be the initial capacity, DC resistance, charging efficiency and self-discharge. Also, accepting only batteries within a tiny consistency margin increases the price of the batteries since fewer battery cells can be accepted from production into use. (Jiang and Zhang, 2015)

Basically a Li-Ion battery contains positive and negative electrode, a separator, an electrolyte, positive and negative terminals, an insulator, a safety valve, positive temperature coefficient, metal lid, gasket and a container of some sort (Wu, 2015). The packaging is usually one of the following three types: cylindrical, pouch and prismatic (Wu, 2015). When talking of automotive industry, all kinds of cells are being used. Tesla, for example, is famous for using cylindrical cells (Lambert, 2019), whereas Nissan used pouch cells for the 1st generation Leaf (Qnovo, 2015). Volkswagen, on the other hand, has made a decision to use both pouch and prismatic cells in their battery modules (Halvorson, 2018).

There are a number of benefits and drawbacks for each type of cells. Cylindrical cells are quite straightforward to produce, but temperature management and packaging requires a lot of careful design and consideration in automotive (large) battery pack applications. For these reasons the size of the cylindrical cell is always a bit of a trade-off: for larger cells the temperature issues become more apparent due to inconsistent temperature distribution (the core is always warmer than the shell) and also the wasted space in packaging gets larger due to the cylindrical shape. Then again, with thinner cells the problem will be the larger total amount of the cells: the volume per kWh gets larger when the cell gets thinner, and also the increased number of cells creates the need for more management electronics. Prismatic cells can be wound or stacked type. The casing is the strong point of prismatic cells: it is usually made of aluminium or relatively hard plastics, which makes it possible to package the cells very densely. This, however, can create a thermal management problem if they are put into a space that is too tight. Prismatic cells do have much better thermal distribution compared to cylindrical cell. Also, the packaging is also beneficial due to the terminals being on the same side opposed to being on different sides with the cylindrical cells. Pouch cells are in many ways like prismatic cells, but with soft packaging, usually polymer-laminated aluminium foil. The electrodes inside are usually stacked or Z-folded. The thermal distribution is much like with prismatic cell: quite an even distribution, but cooling is obviously still required. Needless to say, with soft packaging also becomes the requirement for much more robust supporting in the module compared to prismatic cells. These qualities are very general, and the do vary with different manufacturers. (Berg, 2015)

The cells used in the automotive industry are getting larger and larger. Producing larger cells, however, is not just a question of scaling everything up. It brings a set of completely new requirements with issues that might be neglectable with smaller cells. The production of the cells can be divided into *electrode manufacturing, cell production* and *formation.* (Berg, 2015). Regarding the causes of the manufacturing process for battery cell inconsistency is mainly due to the electrode manufacturing process. More precisely, it can be said that it is inevitable that there are inconsistencies in the thickness, porosity and composition in the electrodes which leads to difference in internal resistance and the capacity of the cell. (Lebel *et al.*, 2019)

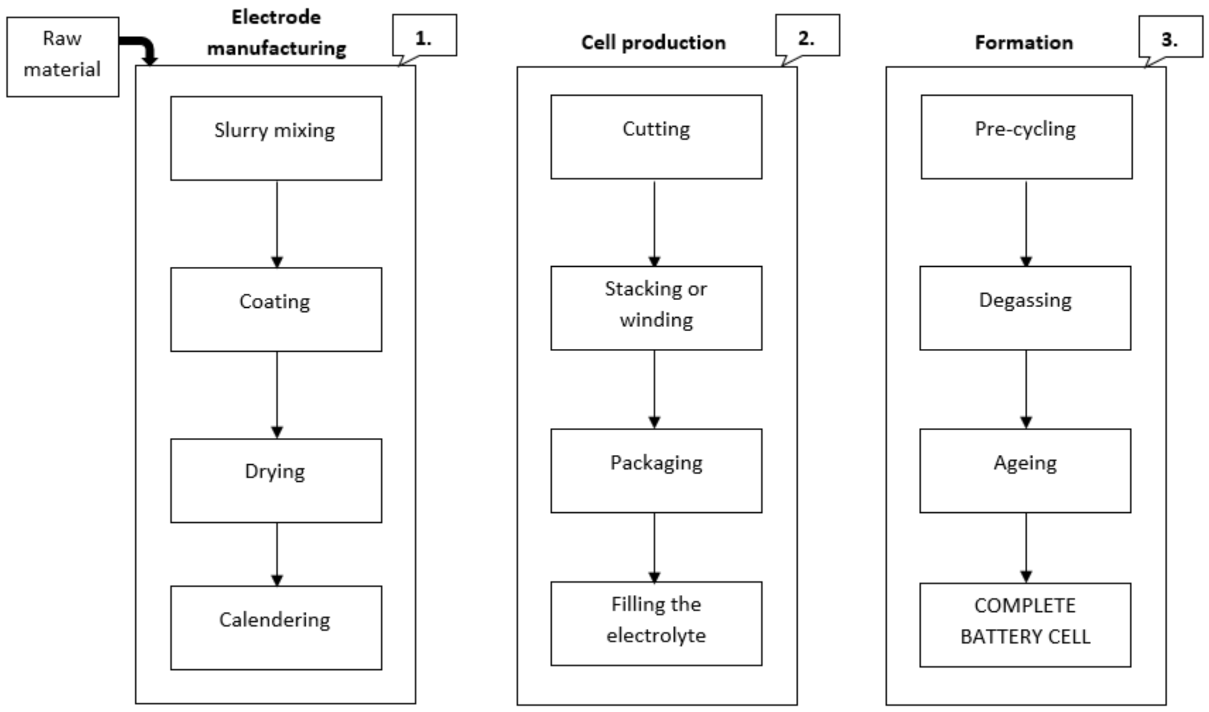


Figure 2.6. The manufacturing process of battery cells.(Berg, 2015)

These inconsistencies, while usually small initially, are gradually getting larger when used. The charging and discharging rates have to be adjusted for the worst performing cell. The fact, that the charging and discharging currents are the same for all of the cells means that those cells with worst initial capacities, largest internal resistances and high self-discharge rates are all the time on more stress than the ones with better qualities. This becomes a positive feedback loop, which means in this case the worse the initial condition is, the faster it will deteriorate compared to cells in better condition. Therefore, it is especially important to use as consistent cells in a battery pack as possible. Also, the heat distribution and uneven cooling of the battery pack also effects on both the short term and long-term performance differences between the cells. (Jiang and Zhang, 2015)

The consequences that the inconsistencies have in practice are shown in capacity, power output and estimation of battery state. As mentioned, the battery pack can only be charged or discharged as much as the worst cell allows, which means that the capacity of the battery pack is limited to whatever the worst cell has as its maximum capacity. This capacity is obviously smaller than what the theoretical combined capacity would be, if the maximum capacity of each individual cell could be used. Regarding the power output, the charging or discharging current is decreased towards both ends of any given cells state of charge (later referred to as SOC). Hence the available power band in relation to SOC is only as wide as it is for the cell with the smallest capacity. Also, the estimation of the battery state becomes increasingly difficult to determine when there are large inconsistencies between individual cells. (Jiang and Zhang, 2015)

2.7.2 Battery Pack Equalization

This study will not examine all the possibilities of balancing the battery pack, since that is not the are of study in this thesis. Instead, the most commonly used technics are presented and examined in order to build the model for battery voltage balancing.

The problems described above results in inefficient operation of the battery and reduced capacity. There are basically two ways of doing the balancing: passive and active balancing. The difference between the two can be described shortly as follows: passive balancing is easier, and usually cheaper, to build, but it is less efficient, since with passive balancing the overcharging of the cells reaching their full SOC first is just permitted with discharging them, which means the extra energy goes into heat. In active balancing, the excess charge is moved between battery cells so that a cell with a lower SOC receives charge from one with higher SOC. This way the loss of energy with active balancing is close to none, and thus much smaller than with passive balancing, especially when the cells get more and more out of balance over time, as explained above. Usually, however, the benefits of active balancing do not offer enough savings for it to be economically better choice. This division between passive and active balancing, although not maybe the most intuitive way, seems to have become somewhat standard way of speaking. (Warner, 2015)

Passive balancing is usually implemented with a shunt resistor. This is a cheap and reliable balancing method, and the balancing control algorithm is easy to implement. As mentioned, no energy from the balancing process is recovered, but all goes to waste via the resistor. This also means that the balancing should only be done during charging. According to Cao et. Al. (2008), passive balancing should not be applied to Lithium based batteries, since passive balancing increases cell temperatures, which should be kept as stable as possible with Lithium based batteries. In this article Cao et. al. clearly refers with passive balancing only to the fixed shunt resistor setup, where the resistors are connected all the time parallel to the battery cell, and nothing regarding the balancing is controlled. Balancing with a shunt resistor can, however, be done in a way that they are actively controlled, and that usually is the case, especially when there are more than just a few cells, the voltages are high and when the battery chemistry limits the safe SOC area. (Daowd *et al.*, 2011; Xiong and Shen, 2019)

As with fixed shunt balancing, active shunt balancing also removes the excess energy with a shunt resistor, and that energy goes to waste, but the resistors are activated using switches by the balancing algorithm. Thus, the discharging of the higher energy cells can be done before the cell reaches overcharge or near overcharge state while still allowing the cells to eventually reach maximum SOC. This is the simplest form of active balancing. This setup can operate in two different modes: the switches can be controlled with the same on/off signal, or the switches can be operated individually. With the individual operation, the algorithm detects when the batteries go out of balance and chooses the correct shunt or shunts to turn on. This method can be used with Lithium based batteries, too. (Jiang and Zhang, 2015)

The judgement criteria of a battery cell's SOC used in this study's model for balancing is the external voltage of single battery cells. In practice, every battery cell's voltage should be within a certain threshold of the average voltage in the battery pack. Also, the balancing is limited to be done only in voltage area, where SOC is about 70% or more. In practice, this means that balancing is done during the end of charging. This is quite a common way to do cell balancing with Lithium based batteries.

3 Workflow of Converting Simulink Models to Structured Text

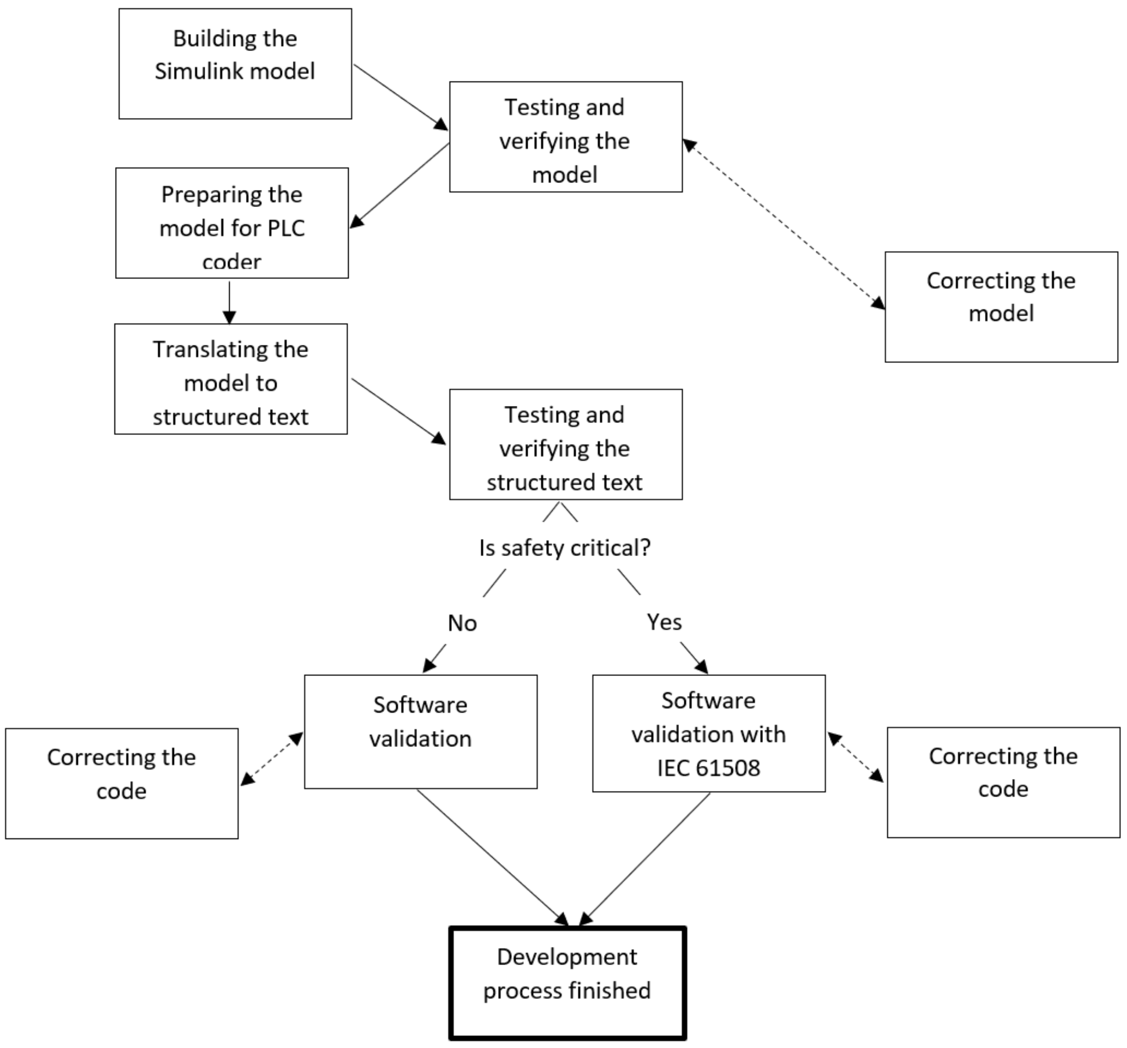

Building the Simulink model
Testing and verifying the model
Preparing the model for PLC coder
Correcting the model
Translating the model to structured text
Testing and verifying the structured text
Is safety critical?
No
Yes
Correcting the code
Software validation
Software validation with IEC 61508
Correcting the code
Development process finished

Figure 3.1. The development process flowchart.

In this chapter, the workflow of the structured text code development process based on the research done in the previous chapter is presented. As an example, a program for balancing battery cell voltages in a battery pack is developed and tested, and the phases of that process are shown.

The code generation process ended up being a relatively quick and straight forward process, but there are things to consider. For example, variable data types are not usually necessary to consider in Simulink since they are determined automatically in the models. However, the choice of correct data types become a big part of the model building process when building the model for code generation purposes. Also, things to consider when generating safety-related code are discussed shortly.

3.1 Building the Simulink Model

It is assumed in this thesis that the reader has some basic knowledge of using Simulink. With this assumption, this chapter only discusses the process in general, and presents the results of building the model with the methods in chapter 2 taken into account.

The example program that is expected as an outcome from this process is a battery balancing algorithm. This was briefly presented and used as an example throughout the second chapter as well. However, for clarity of the whole process it is now presented in this chapter, too.

First, the requirements for the system are needed to be defined. As an input, it is supposed to take the current voltages of the battery cells connected in series in a battery module, and the minimum voltage for balancing. As an output, the algorithm gives a list of balancing requirements for each cell as Boolean values. True as in balancing is required and false as in balancing is not required. The basis and reasoning for the algorithm is explained more thoroughly in chapter 2.7, but basically balancing is required if two conditions are met. Firstly, the voltage is exceeding the lowest voltage in the module by more than a pre-determined margin, which for example could be 0.1 volts, like in this case. That, however, much depends on the battery type and application where it is used. Also, the voltage has to be higher than the lowest voltage, where balancing can be activated, which the program gets as an input parameter.

After knowing the requirements, they can be simply implemented in Simulink as a model. When building complex algorithms as software with model-based design, it is usually recommended to define the algorithm as a mathematical model first, and then implement it into a graphical simulation model, like Simulink. However, in this case the algorithm is simple enough to skip that phase, since it is basically just checking fulfilment of two conditions and outputting a simple Boolean value instead of a numerical result.

When building the model, it is important to name every block according to the guidelines defined for the software project. Even if most of the names are not used in the generated code, they are visible in the code as comments for traceability purposes if the properties of the coder are set to include comments in the code, as shown later on.

The Simulink model presented in figure 3.2 is the whole Simulink model with the model testing and verification parts. On the left, there is a *testingSystem*-subsystem, that includes the voltages for all the cells in the battery module, and a voltage figure, which represents the value of about 70% of state of charge (SOC), which is used as a minimum voltage for balancing in this case. There is a total of 21 battery cells in this model. The voltages are sent as a vector to the balancing subsystem, the minimum voltage is just a constant (that would vary over a longer period of time on a real system) and the output is a vector of Boolean values. This is what the balancing function would get in the real device as inputs and outputs as well, so it is a ready function to be translated to structured text and used in the logic controller in that regard.

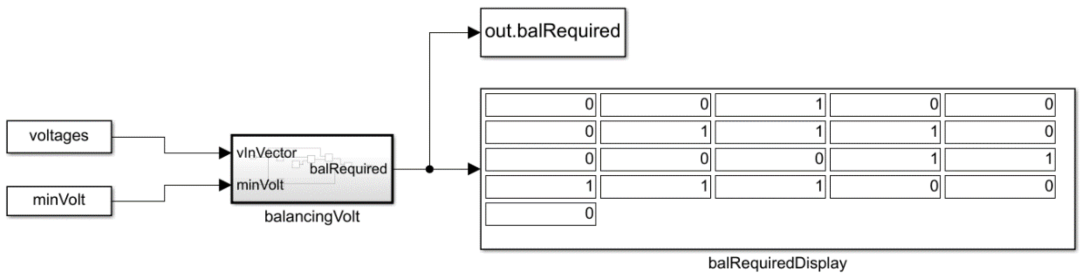


Figure 3.2. The Simulink model with testing.

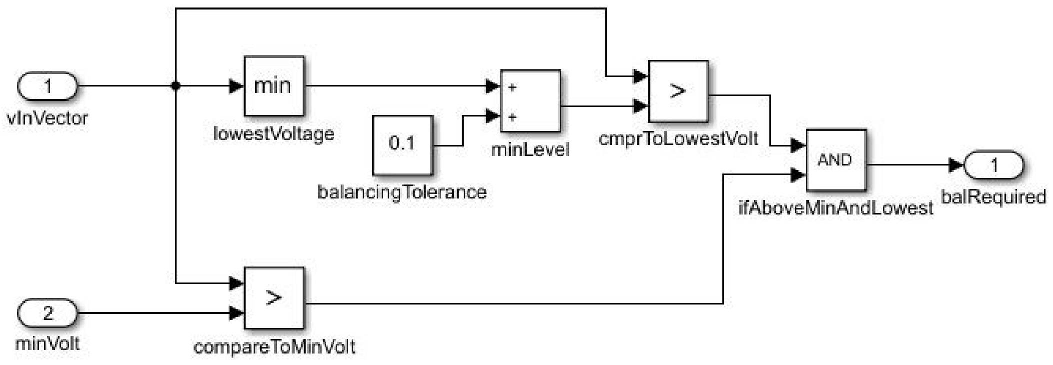


Figure 3.3. The Simulink subsystem to be converted to structured text.

When building the subsystem to be translated using the PLC Coder, it is important to remember the limitations that the coder requires, as well as CODESYS itself. First of all, the blocks used in the model need to be compatible with the coder. These blocks can be checked by typing *plclib* into the Matlab command window. Simulink then opens a library that includes all of the blocks that are compatible with the coder, as seen in figure 3.4. Generally speaking, this doesn't limit the model building too much, but it is always good practice to keep the list of compatible blocks open when designing the model and check for compatibility when adding some block that hasn't been checked to be in the list before.

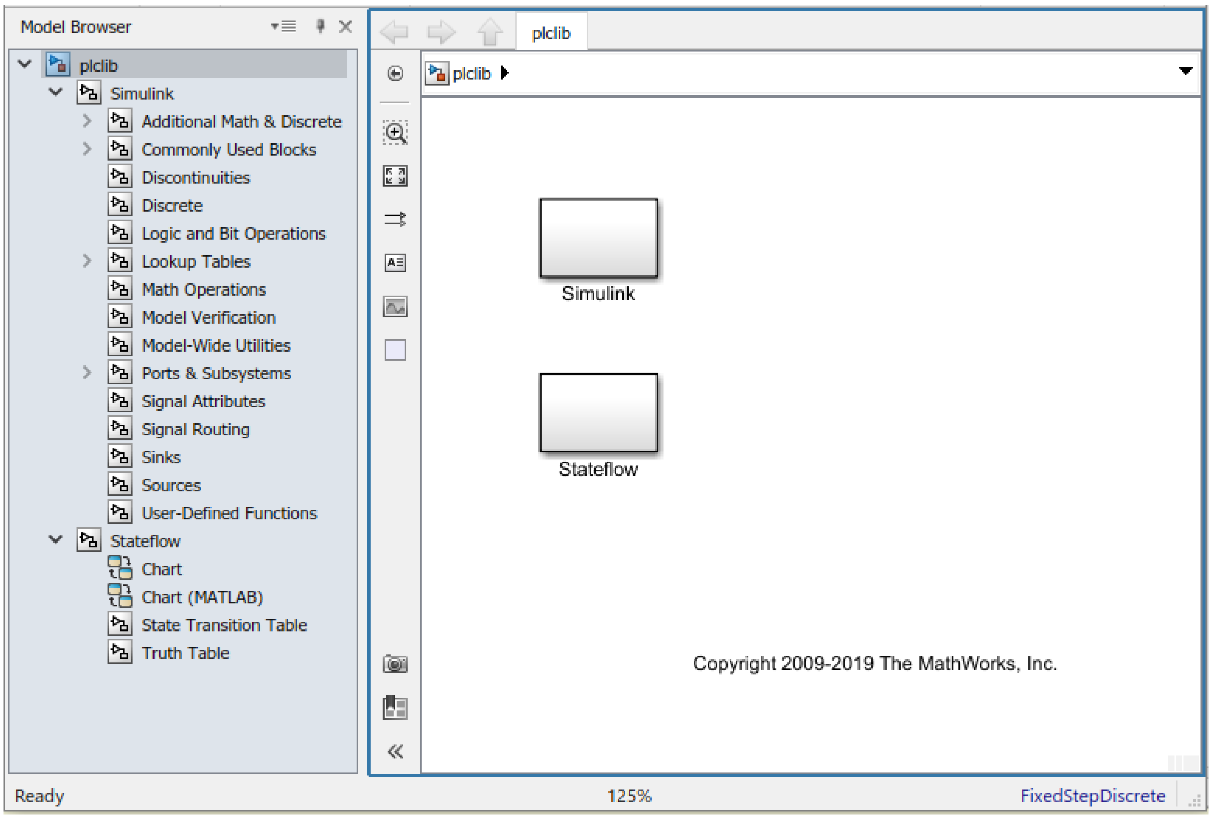
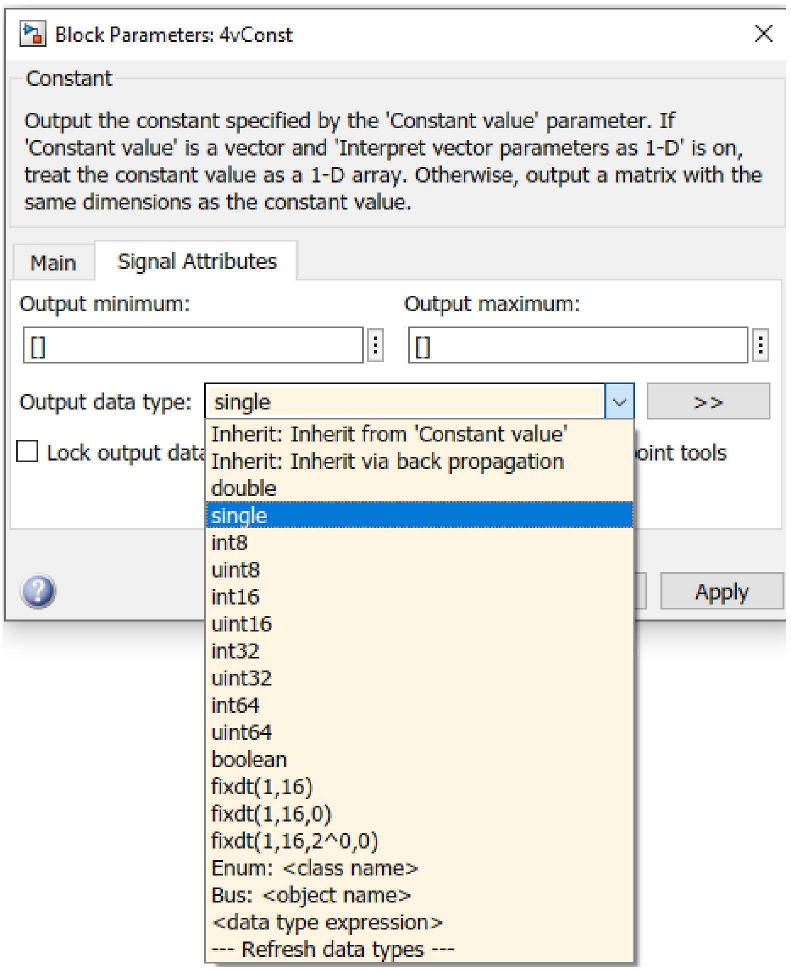


Figure 3.4. The plclib-library for checking block compatibility.

3.1.1 Data Types in Simulink

All variables – inputs, outputs and local variables – has to be individually modified to have the correct data type, as seen in figure 3.5. These datatypes are specified in the requirements of the model. Otherwise Simulink will use automatically the “largest” data type. For example, if a constant is created, Simulink will automatically create it as a *double,* which translates to *LREAL* in structured text. Keeping in mind that these programs are meant to run on PLC devices, the memory consumption should be kept at minimum. For this reason, it is not optimal to use data types that consume a lot of memory and choosing data types manually should be done when building the model just like when writing regular code. Even more importantly, choosing the data types for each input and output variable is crucial in order to avoid data type conflicts. This, on the other hand, gives yet another reason for also choosing data types for local variables manually. Even if an input variable is a *single*-data type, if there is a local variable that is set to *inherit data type,* and then they are compared with each other, Simulink will keep the local variable as *double* and convert the input variable as *double.* This obviously is not only consuming excess memory, but also prolongs the running time unnecessarily.



Block Parameters: 4vConst

Constant

Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.

Main | Signal Attributes

Output minimum: []

Output maximum: []

Output data type: single

Lock output data ... oint tools

Inherit: Inherit from 'Constant value'
Inherit: Inherit via back propagation
double
single
int8
uint8
int16
uint16
int32
uint32
int64
uint64
boolean
fixdt(1,16)
fixdt(1,16,0)
fixdt(1,16,2^0,0)
Enum: <class name>
Bus: <object name>
<data type expression>
--- Refresh data types ---

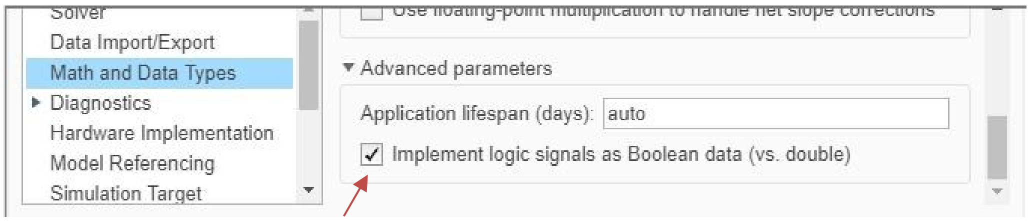
Figure 3.5. Choosing the data type in Simulink.

Table 3.1 shows how different datatypes translate from Simulink to structured text.

Table 3.1. Data Types from Simulink to Structured Text

Simulink	Structured Text
double	LREAL
single	REAL
int8	SINT
uint8	USINT
int16	INT
uint16	UINT
int32	DINT
uint32	UDINT
boolean	BOOL

When handling logic signals, it is important to tick the “Implement logic signals as Boolean data” setting from the model configuration parameters in Simulink. The option in question can be found under Math and Data Types -> Advanced parameters, as seen in figure 3.6.



Use floating-point multiplication to handle net slope corrections

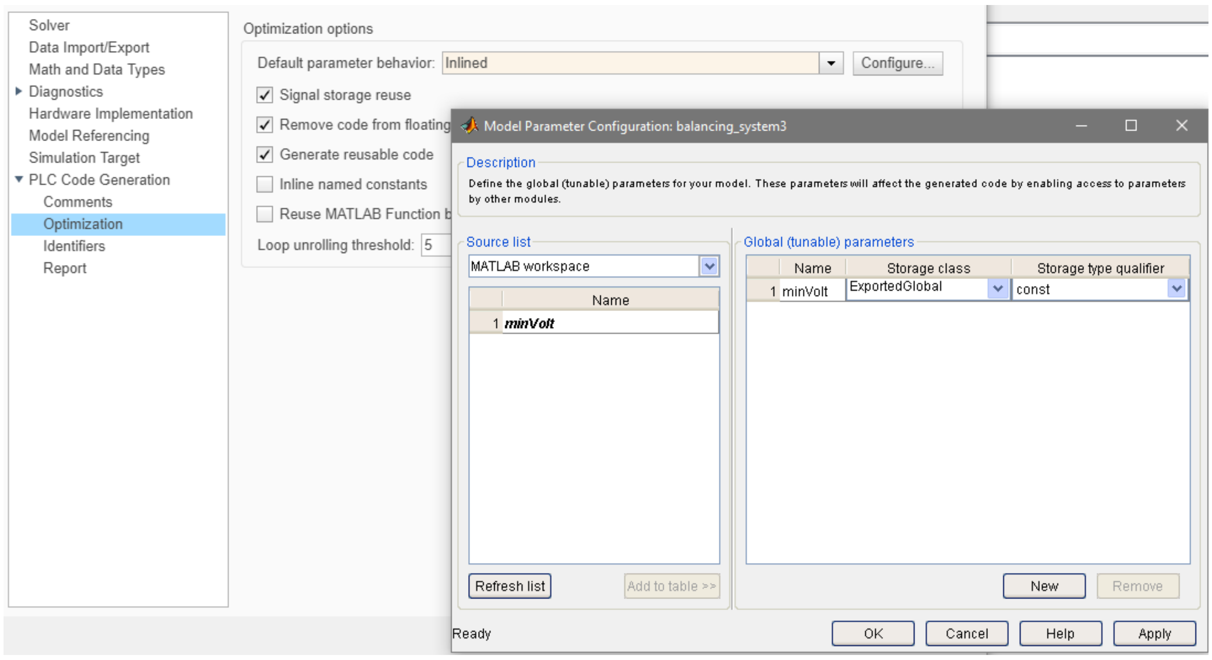
Advanced parameters

Application lifespan (days): auto

☑ Implement logic signals as Boolean data (vs. double)

Figure 3.6 Logic signals as Boolean option.

Global variables or constants can also be defined in the process. Parameters that are in the MATLAB workspace can be set as “tunable parameters” in the PLC Coder properties, as seen in figure 3.7.



Optimization options

Default parameter behavior: Inlined

☑ Signal storage reuse
☑ Remove code from floating
☑ Generate reusable code
☐ Inline named constants
☐ Reuse MATLAB Function b

Loop unrolling threshold: 5

Model Parameter Configuration: balancing_system3

Description

Define the global (tunable) parameters for your model. These parameters will affect the generated code by enabling access to parameters by other modules.

Source list: MATLAB workspace

	Name
1	***minVolt***

Global (tunable) parameters

	Name	Storage class	Storage type qualifier
1	minVolt	ExportedGlobal	const

Ready

Figure 3.7. Set parameters as tunable to be defined as global parameters in the generated code.

In CODESYS, Boolean values are actually 8-bit data types, and therefore when trying to save memory of the logic controller, it would be better to use the BIT-type, which is only 1-bit. It is not, however, a data type of the IEC 61131 -standard, although it is a standard data type in CODESYS, nor is it a data-type in Simulink. For these reasons, it is not surprising that the BIT-type is not supported by the PLC Coder. It is possible to use always user-specified fixed data type in Simulink with logical values and define that data type to be a 1-bit value. Unfortunately, this is handled by Simulink as an unsigned short integer, and that is how the PLC Coder translates it to CODESYS structured text as well.

That being said, it is always possible to modify the code, and change the generated short integers to BIT-values. After that the code cannot be anymore verified as automatically generated but has to be verified and validated as any other hand-written code, since changing data types of any variable in the code creates a significant difference compared to the model, and the code might not be able to work as it is supposed to.

3.1.2 Other PLC Coder Properties

Before generating the code, it is good practice to go through the PLC Coder properties. In general options the correct target IDE should be chosen. In this case it is *3S CoDeSys 3.5*. Also, the file path for accessing the target IDE software must be specified. The name of the folder where the PLC Coder places the generated code can be chosen by the user. The folder is placed in whatever is the current folder in MATLAB. In the general options it is also chosen whether the testbench is created for the generated code. The testbench-function was not used in this case, since the code generated for testing was not sufficient for the code verification process.

In the comments-section of the properties the user can specify if the PLC Coder generates comments in the code as well. This has proven to be quite useful, as it generates comments that provide clear links between the generated code and the Simulink model for traceability purposes. This has been presented more thoroughly in section 3.3.2.

In the optimization options default parameter behaviour should be set as "Inlined". Should there be some global parameters, they have to be configured separately as explained in the previous section. Also, the "Generate reusable code" option should be checked when there are multiple identical subsystems in the model so that only one function block is generated instead of multiple identical ones. In the identifiers-section there are options about the names of the function block and variables for the generated code. These are mostly project-specific options for example to avoid some certain variable names or to determine the maximum length for an identifier. In the report-section the user can choose whether the traceability report is generated and if it is opened automatically after a successful code generation. Generating the traceability report is usually recommended, since it provides useful information that can be used in the code validation process.

3.2 Design Verification

As discussed in chapter 2, developing the code with model-based design technique eases the process of verifying the code. It also creates the need of verifying the model that is created.

For verifying the design of the model, a separate model for testing should be implemented. As explained in chapter 2.6, the model must be verified in Simulink, and the results must be documented. The model for the testing should be made so that the functionality of the program can be verified, but also so that the same exact input for the program can be used in the code verification phase to fulfil the testing requirements for IEC 61508. The PLC Coder testbench -tool can be used to create the testing for the code, but there are still a few remarks on using that wisely.

The values for the test vectors need to be chosen so that they provide sufficient range of different combination of input values regarding the operating range of the function. A table of different operating conditions can be created, after which it is easy to determine the input vectors. In addition, since there are multiple cells to be checked and operated, different variations of these conditions have to be created which then can be reproduced while testing the code in the coding environment as well as later on in the actual hardware in the integration phase of validation. The correct functionality of the program is always checked using the conditions for the balancing information of each cell in the module provided in the requirements, or more clearly for testing purposes in table 3.2.

Table 3.2. Operation of cell balancing.

	Voltage **below** the minimum voltage	Voltage **above** the minimum voltage
Voltage **less** than 0.1V higher than the lowest voltage in module	No balancing	No balancing
Voltage **more** than 0.1V higher than the lowest voltage in module	No balancing	*Balancing*

There are multiple ways of performing the tests itself. When working with a larger software development project, it is advisable to use tools that are made for testing purposes. As mentioned before, with the code testing part in CODESYS, there is add-ons like Test Manager to use in that. There is a similar tool in Simulink called Simulink Test, and combined with the IEC Certification kit, all of the testing can be performed for the model and the code to come out with IEC 61508 certified software. However, in this case the program is relatively simple, so the tests are made using default tools provided in MATLAB and Simulink, since all the testing and certification tools can get expensive. Doing the verification process this way can be a viable option when not every part of the software is made using Simulink, but some of the code is written manually straight to CODESYS. As mentioned several times in chapter 2, Simulink is a great tool for building models that include algorithm like this that can be laborious to manually develop as hand written code, and so model-based design as presented in this thesis can be used only for those parts of the software.

The testing vectors are easy enough to make in MATLAB, from which they can be sent to the Simulink model, and all the tests can be run automatically with the model. The results are sent back to MATLAB where they are stored. In this case, five different input vectors are made:

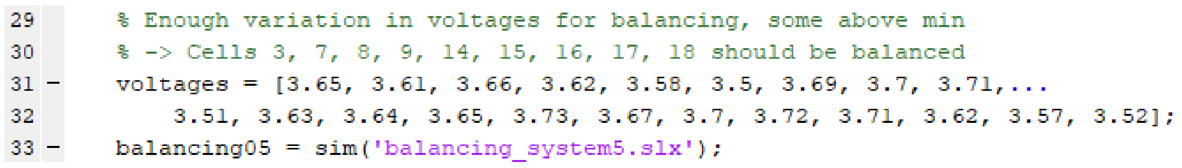
1st test) No variation in voltages and no voltage is above the minimum balancing voltage

2nd test) No variation in voltages and all voltages are above the minimum balancing voltage

3rd test) Some voltages are high enough to exceed the tolerance limit, some are higher but within the tolerance, but none are above the minimum balancing voltage

4th test) Some voltages are high enough to exceed the tolerance limit, some are higher but within the tolerance, all voltages are above the minimum balancing voltage

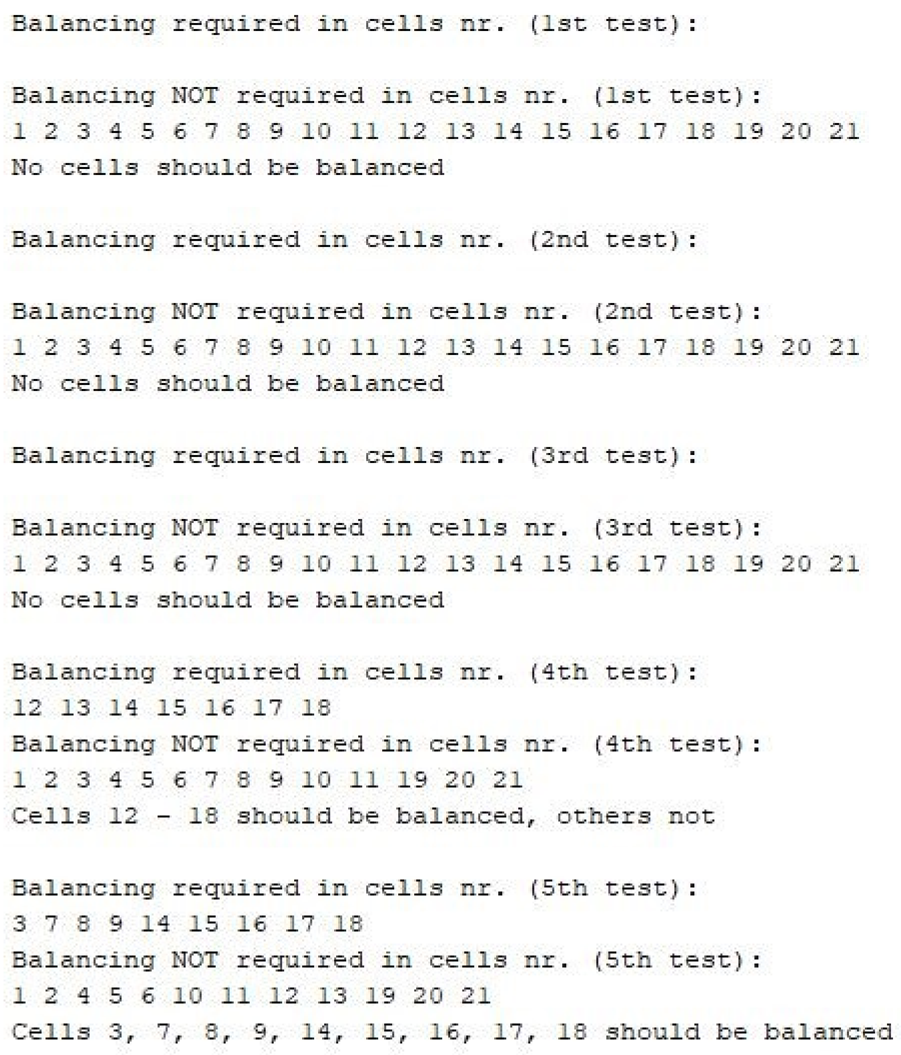
5th test) Some voltages are high enough to exceed the tolerance limit, some are higher but within the tolerance, some of the voltages are above the minimum balancing voltage



```
% Enough variation in voltages for balancing, some above min
% -> Cells 3, 7, 8, 9, 14, 15, 16, 17, 18 should be balanced
voltages = [3.65, 3.61, 3.66, 3.62, 3.58, 3.5, 3.69, 3.7, 3.71,...
    3.51, 3.63, 3.64, 3.65, 3.73, 3.67, 3.7, 3.72, 3.71, 3.62, 3.57, 3.52];
balancing05 = sim('balancing_system5.slx');
```

Figure 3.8. Example of test input definition written in MATLAB (the 5th test).

The results can be seen in figure 3.9, which is a snapshot of the testing results printed in MATLAB. The code for the tests as well as the results can be seen completely in appendix 1. In addition, some dynamic testing would usually be required that would show the transition phase from one state to another. In this thesis only the static part of the testing is presented since as mentioned before, the testing of the code can vary a lot depending on the project in question, and the testing principles are an area of study of its own. This thesis rather focuses on the automatic code generation process, and hence mainly the idea of model-based design testing is presented using this as an example.



```
Balancing required in cells nr. (1st test):

Balancing NOT required in cells nr. (1st test):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
No cells should be balanced

Balancing required in cells nr. (2nd test):

Balancing NOT required in cells nr. (2nd test):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
No cells should be balanced

Balancing required in cells nr. (3rd test):

Balancing NOT required in cells nr. (3rd test):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
No cells should be balanced

Balancing required in cells nr. (4th test):
12 13 14 15 16 17 18
Balancing NOT required in cells nr. (4th test):
1 2 3 4 5 6 7 8 9 10 11 19 20 21
Cells 12 - 18 should be balanced, others not

Balancing required in cells nr. (5th test):
3 7 8 9 14 15 16 17 18
Balancing NOT required in cells nr. (5th test):
1 2 4 5 6 10 11 12 13 19 20 21
Cells 3, 7, 8, 9, 14, 15, 16, 17, 18 should be balanced
```

Figure 3.9. Snapshot of the design verification results done in MATLAB and Simulink.

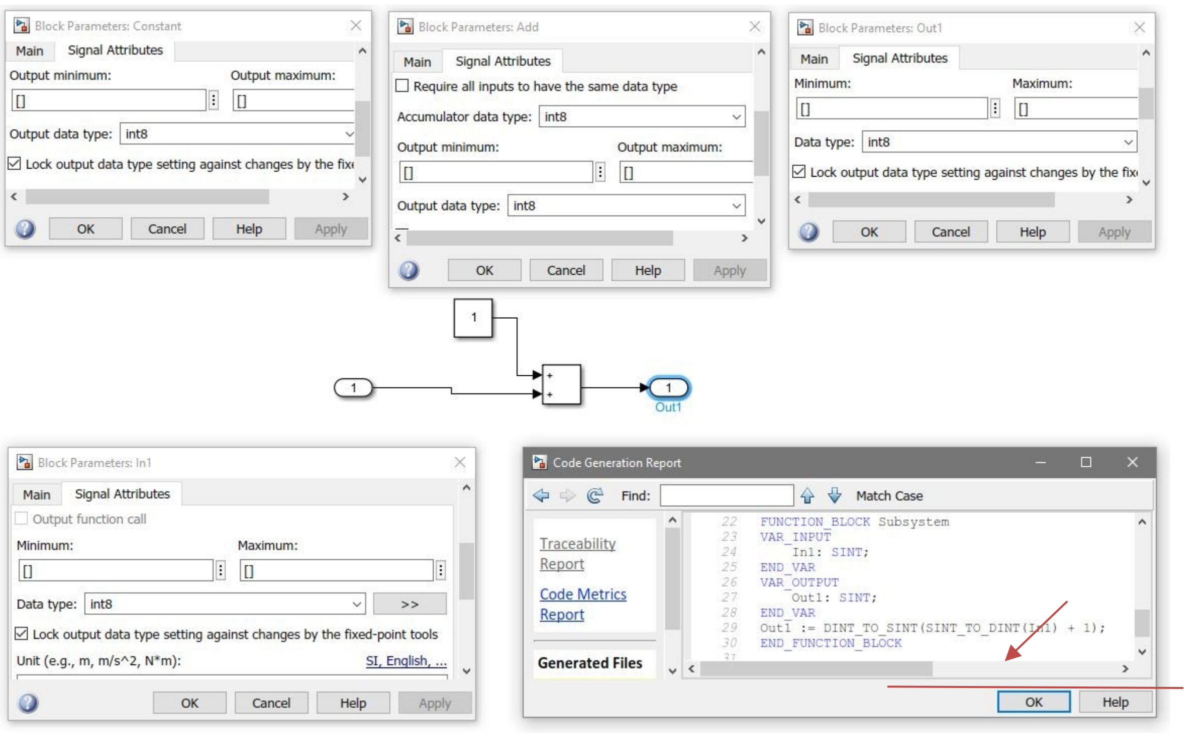
3.3 Translated Model as Structured Text

Even before importing the structured text to the CODESYS environment, it is always good practice to do certain checks for the code after a successful code generation already in Simulink. This way it is much easier to correct any mistakes that are obvious in the code without having to delete the function blocks and possible global variables from the CODESYS environment and then importing them again. Simulink also provides some static analysis data for the documentation of the code validation process.

3.3.1 Verifying the Variables and Data Types

Immediately after a successful translation process, it is necessary to review the code manually. First, it is to be checked that the variable names are what they are supposed to be. This is especially the case with input and output variables to ensure compatibility with other parts of the software. All variable data types need to be checked as well to correspond with the requirements defined for the program. If some of the data types are wrong, the Simulink model needs to be rechecked to have correct data types.

In addition, the translated code needs to be checked for unnecessary data type conversions. Some of the blocks in Simulink, while compatible with the PLC Coder, do not work properly with some low-level data types. For example, if a simple math operation, like add, with two variables is done with an int8 (or SINT in structured text, 8-bit integer) and a constant, PLC Coder for some reason first converts the variable to int32 (or DINT accordingly), does the calculation with that data-type, and then converts it back to int8. That is even if the constant is defined as int8 in Simulink. The parameters for the blocks can be seen in figure 3.10. This is obviously unnecessarily consuming valuable resources of the logic controller. The unnecessary conversion can be manually deleted from the structured text code, after which the functionality of that part of the code must be checked. The situation must be documented with the development and validation process.


Block Parameters: Constant — Signal Attributes: Output minimum: [] Output maximum: [] Output data type: int8 ☑ Lock output data type setting against changes by the fixe

Block Parameters: Add — Signal Attributes: ☐ Require all inputs to have the same data type; Accumulator data type: int8; Output minimum: [] Output maximum: [] Output data type: int8

Block Parameters: Out1 — Signal Attributes: Minimum: [] Maximum: [] Data type: int8 ☑ Lock output data type setting against changes by the fixe

Block Parameters: In1 — Signal Attributes: ☐ Output function call; Minimum: [] Maximum: [] Data type: int8 ☑ Lock output data type setting against changes by the fixed-point tools; Unit (e.g., m, m/s^2, N*m): SI, English, ...

Code Generation Report:

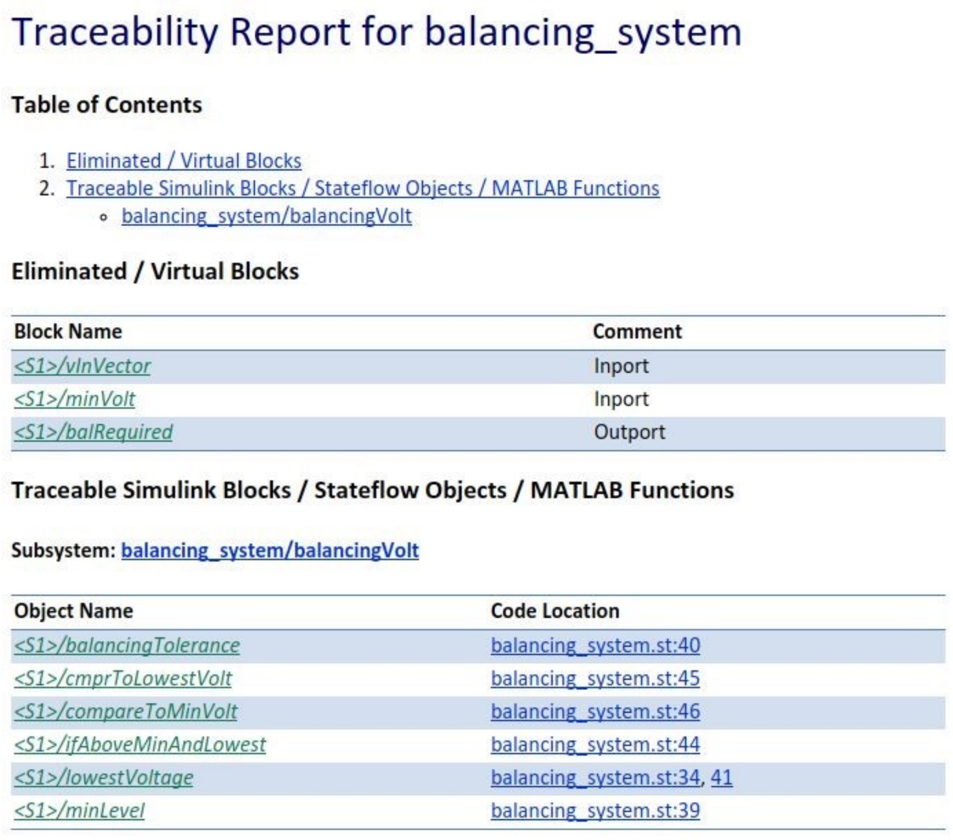
```
FUNCTION_BLOCK Subsystem
VAR_INPUT
    In1: SINT;
END_VAR
VAR_OUTPUT
    Out1: SINT;
END_VAR
Out1 := DINT_TO_SINT(SINT_TO_DINT(In1) + 1);
END_FUNCTION_BLOCK
```

Figure 3.10. The unnecessary data-type conversion by PLC Coder.

3.3.2 Code Generation Report

As mentioned in section 3.1, it is important with PLC programming to optimize the resources since they can be somewhat limited in use, and so it is necessary to verify the generated code in this regard after it has been generated. When the code generation report is opened in Simulink it is important to check that all variables are as expected by previewing the code as seen in figure 3.12. The code generation report also offers a traceability report and a code metrics report.

The traceability report, as seen in figure 3.11 helps to understand the links between the code and the model. The traceability report is only created when "Generate traceability report" is checked in the PLC Coder parameters under "Report". From the traceability report it is easy to navigate to the block or line of code where that block is used by clicking the links in the report. Besides of the traceability report, the lines are also linked in the code preview itself, and by clicking them in the report, the corresponding block in the Simulink model is highlighted. It also works the other way around: in the Simulink model, by right-clicking a block and then choosing PLC Code -> Navigate to Code, Simulink opens the Code Generation Report and highlights the line(s) of code where it is used. (MathWorks, 2019)



Traceability Report for balancing_system

Table of Contents

1. Eliminated / Virtual Blocks
2. Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions
   - balancing_system/balancingVolt

Eliminated / Virtual Blocks

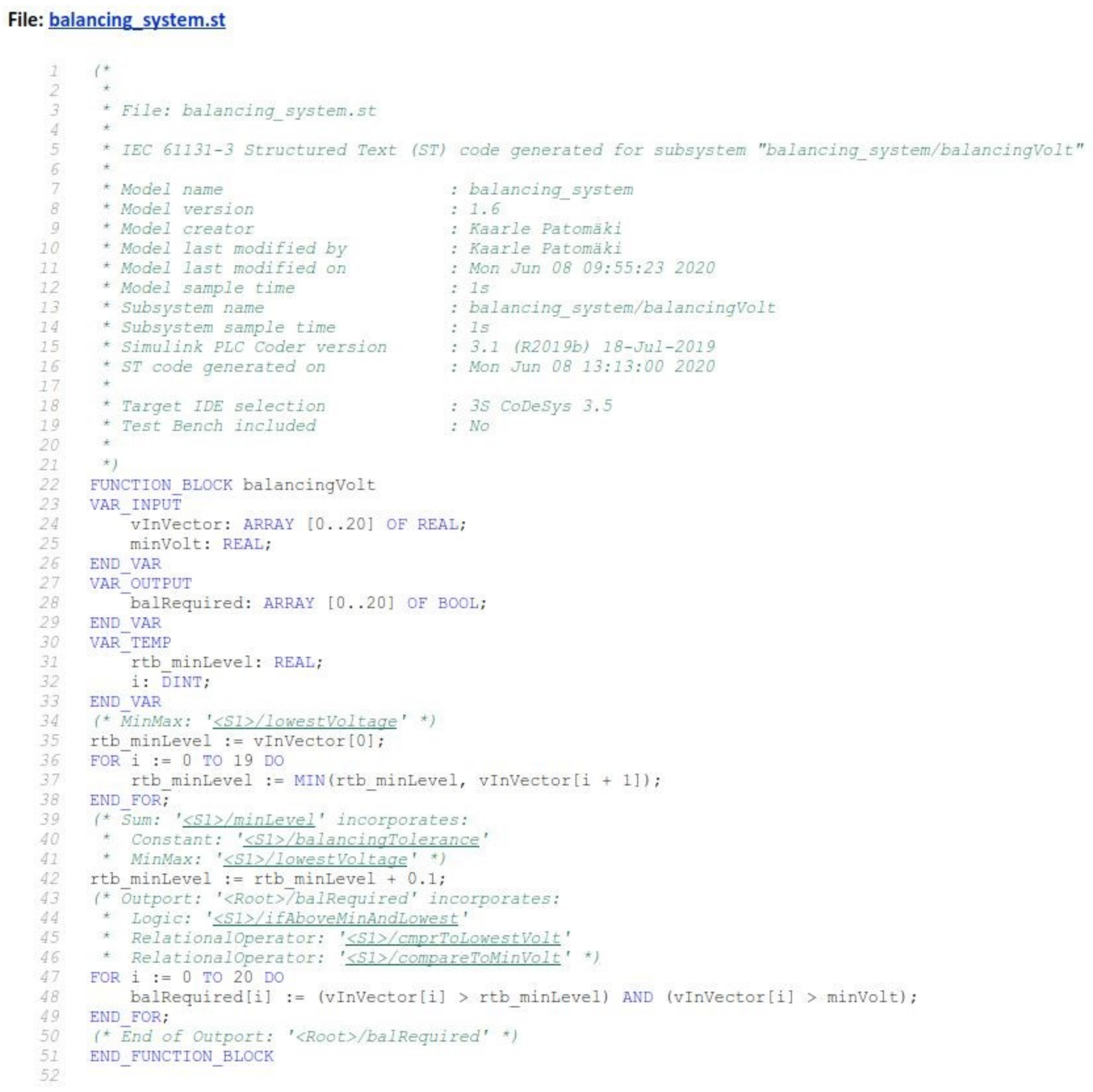
Block Name	Comment
<S1>/vInVector	Inport
<S1>/minVolt	Inport
<S1>/balRequired	Outport

Traceable Simulink Blocks / Stateflow Objects / MATLAB Functions

Subsystem: balancing_system/balancingVolt

Object Name	Code Location
<S1>/balancingTolerance	balancing_system.st:40
<S1>/cmprToLowestVolt	balancing_system.st:45
<S1>/compareToMinVolt	balancing_system.st:46
<S1>/ifAboveMinAndLowest	balancing_system.st:44
<S1>/lowestVoltage	balancing_system.st:34, 41
<S1>/minLevel	balancing_system.st:39

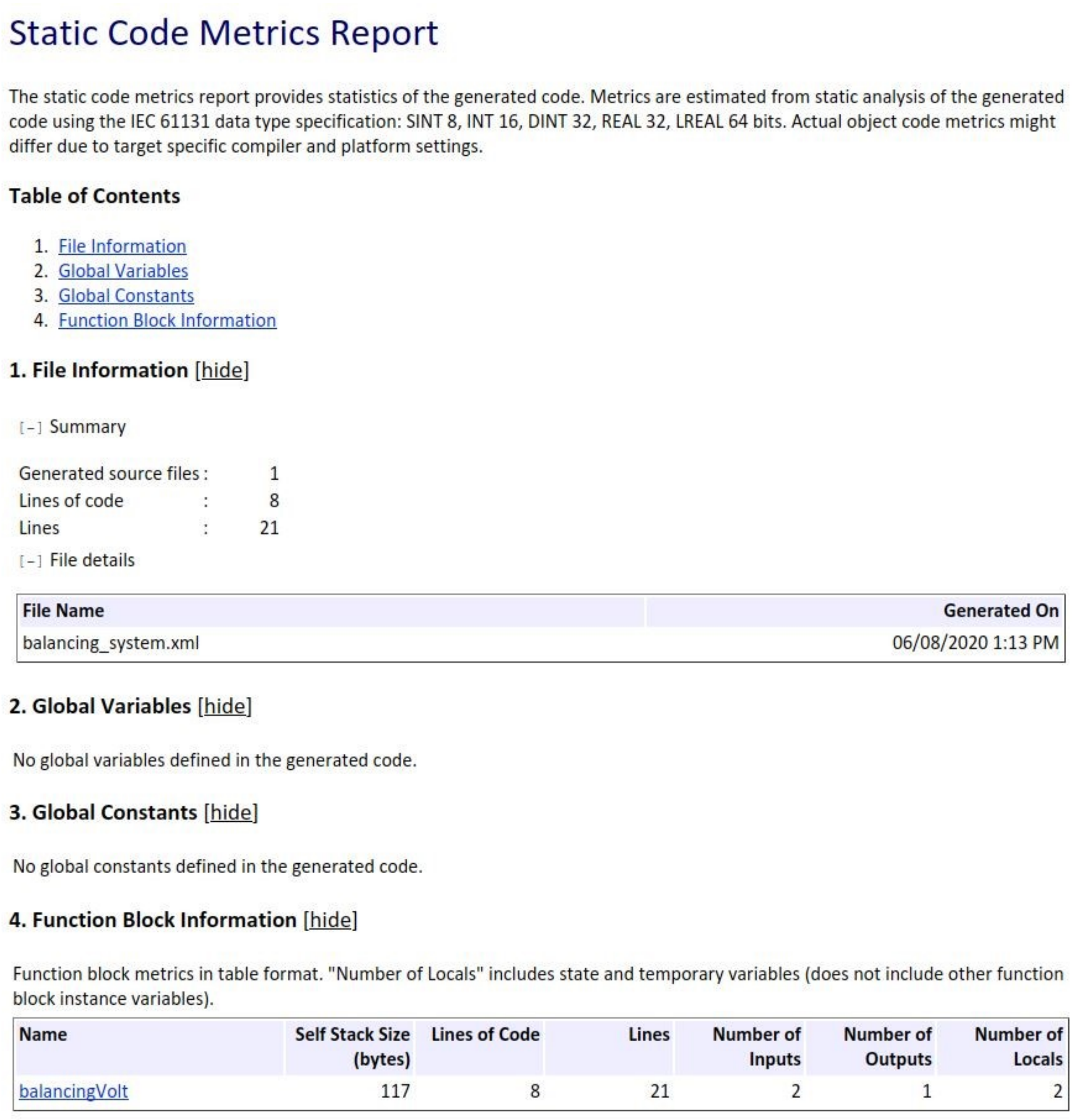
Figure 3.11. Traceability Report in Simulink's Code Generation Report.


File: balancing_system.st

```
(*
 *
 * File: balancing_system.st
 *
 * IEC 61131-3 Structured Text (ST) code generated for subsystem "balancing_system/balancingVolt"
 *
 * Model name                       : balancing_system
 * Model version                    : 1.6
 * Model creator                    : Kaarle Patomäki
 * Model last modified by           : Kaarle Patomäki
 * Model last modified on           : Mon Jun 08 09:55:23 2020
 * Model sample time                : 1s
 * Subsystem name                   : balancing_system/balancingVolt
 * Subsystem sample time            : 1s
 * Simulink PLC Coder version       : 3.1 (R2019b) 18-Jul-2019
 * ST code generated on             : Mon Jun 08 13:13:00 2020
 *
 * Target IDE selection             : 3S CoDeSys 3.5
 * Test Bench included              : No
 *
 *)
FUNCTION_BLOCK balancingVolt
VAR_INPUT
    vInVector: ARRAY [0..20] OF REAL;
    minVolt: REAL;
END_VAR
VAR_OUTPUT
    balRequired: ARRAY [0..20] OF BOOL;
END_VAR
VAR_TEMP
    rtb_minLevel: REAL;
    i: DINT;
END_VAR
(* MinMax: '<S1>/lowestVoltage' *)
rtb_minLevel := vInVector[0];
FOR i := 0 TO 19 DO
    rtb_minLevel := MIN(rtb_minLevel, vInVector[i + 1]);
END_FOR;
(* Sum: '<S1>/minLevel' incorporates:
 *  Constant: '<S1>/balancingTolerance'
 *  MinMax: '<S1>/lowestVoltage' *)
rtb_minLevel := rtb_minLevel + 0.1;
(* Outport: '<Root>/balRequired' incorporates:
 *  Logic: '<S1>/ifAboveMinAndLowest'
 *  RelationalOperator: '<S1>/cmprToLowestVolt'
 *  RelationalOperator: '<S1>/compareToMinVolt' *)
FOR i := 0 TO 20 DO
    balRequired[i] := (vInVector[i] > rtb_minLevel) AND (vInVector[i] > minVolt);
END_FOR;
(* End of Outport: '<Root>/balRequired' *)
END_FUNCTION_BLOCK
```

Figure 3.12. The generated structured text in Simulink's Code Generation Report.

The Static Code Metrics Report generated by the Simulink PLC Coder provides statistical information of the code generated. It is also created when the traceability report is generated. The report is shown below in figure 3.13. It helps to estimate the memory allocation of the program. These metrics are also required to be documented within IEC 61508, but it is also good practice in any larger programming project. It also gives a list of any global variables or constants created when generating the code, should there be any.


Static Code Metrics Report

The static code metrics report provides statistics of the generated code. Metrics are estimated from static analysis of the generated code using the IEC 61131 data type specification: SINT 8, INT 16, DINT 32, REAL 32, LREAL 64 bits. Actual object code metrics might differ due to target specific compiler and platform settings.

Table of Contents

1. File Information
2. Global Variables
3. Global Constants
4. Function Block Information

1. File Information [hide]

[-] Summary

Generated source files	:	1
Lines of code	:	8
Lines	:	21

[-] File details

File Name	Generated On
balancing_system.xml	06/08/2020 1:13 PM

2. Global Variables [hide]

No global variables defined in the generated code.

3. Global Constants [hide]

No global constants defined in the generated code.

4. Function Block Information [hide]

Function block metrics in table format. "Number of Locals" includes state and temporary variables (does not include other function block instance variables).

Name	Self Stack Size (bytes)	Lines of Code	Lines	Number of Inputs	Number of Outputs	Number of Locals
balancingVolt	117	8	21	2	1	2

Figure 3.13. Code Metrics Report in Simulink's Code Generation Report.

.

3.4 Verifying the Structured Text Code

After verifying the model and successfully generating the code in Simulink, it is time to open CODESYS development environment. As with Simulink, it is assumed that the reader has basic know-how of how to use CODESYS, and thus only the code importing and integrating process is shown, as well as verifying it in CODESYS. It has to be noted that this verification, although done so that it can be part of a IEC 61508 validation process, is not sufficient on its own, but needs more verification. This is, however, very project dependent. This chapter tries to explain the things to be done and things to consider that are normally not part of the verification process when developing all the code manually.

First the generated code has to be imported to a CODESYS project. This is done by importing the generated *.xml* -file as shown in figure 3.14 below.

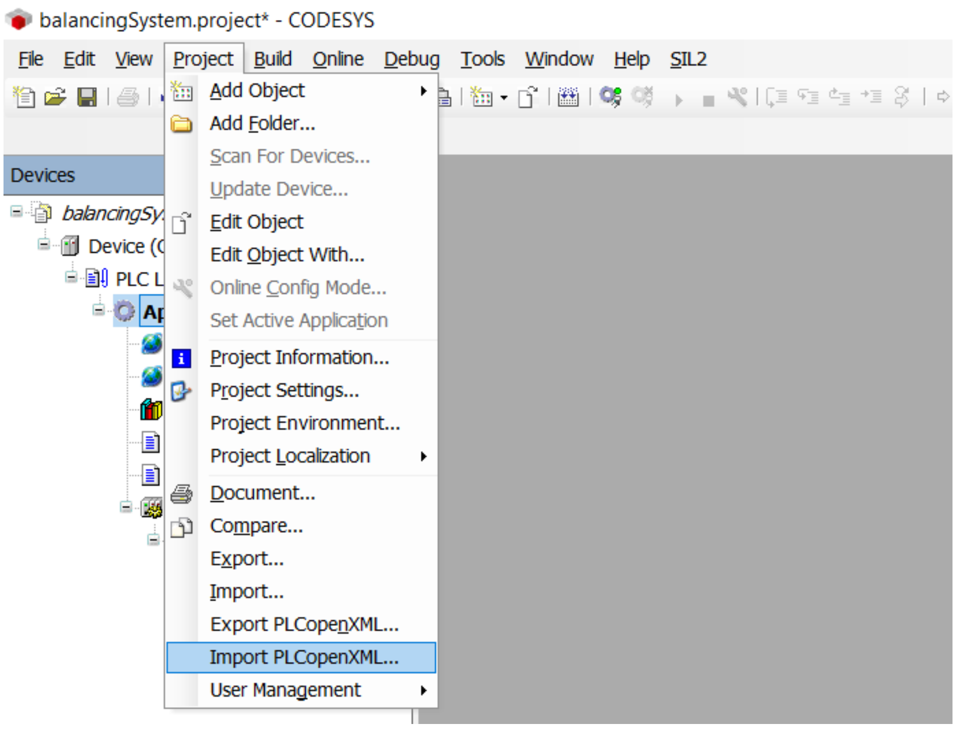
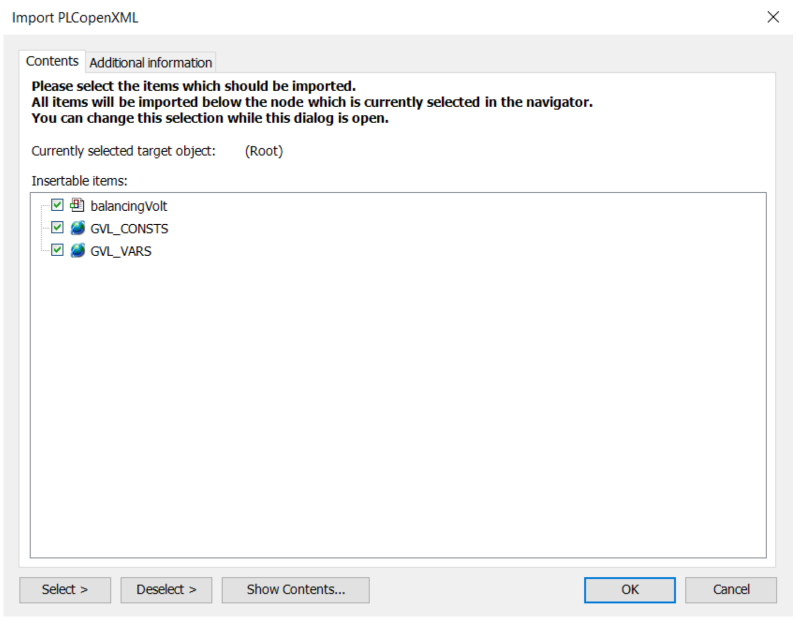


Figure 3.14. Importing the generated code in CODESYS environment.

After choosing the correct *.xml* -file, it has to be checked that it includes the correct files. At this point it has to be chosen whether the global constant and variables generated by the PLC Coder are imported to the project or not, as seen in figure 3.15. It has to be noted that files with the same name (the global variable lists) cannot be merged with the imported ones, but they have to be named differently. The importer in CODESYS will give an error when attempting the import, and gives a choice of either renaming the files, replacing the existing with the imported ones or to not import the files with the name conflict.


Import PLCopenXML

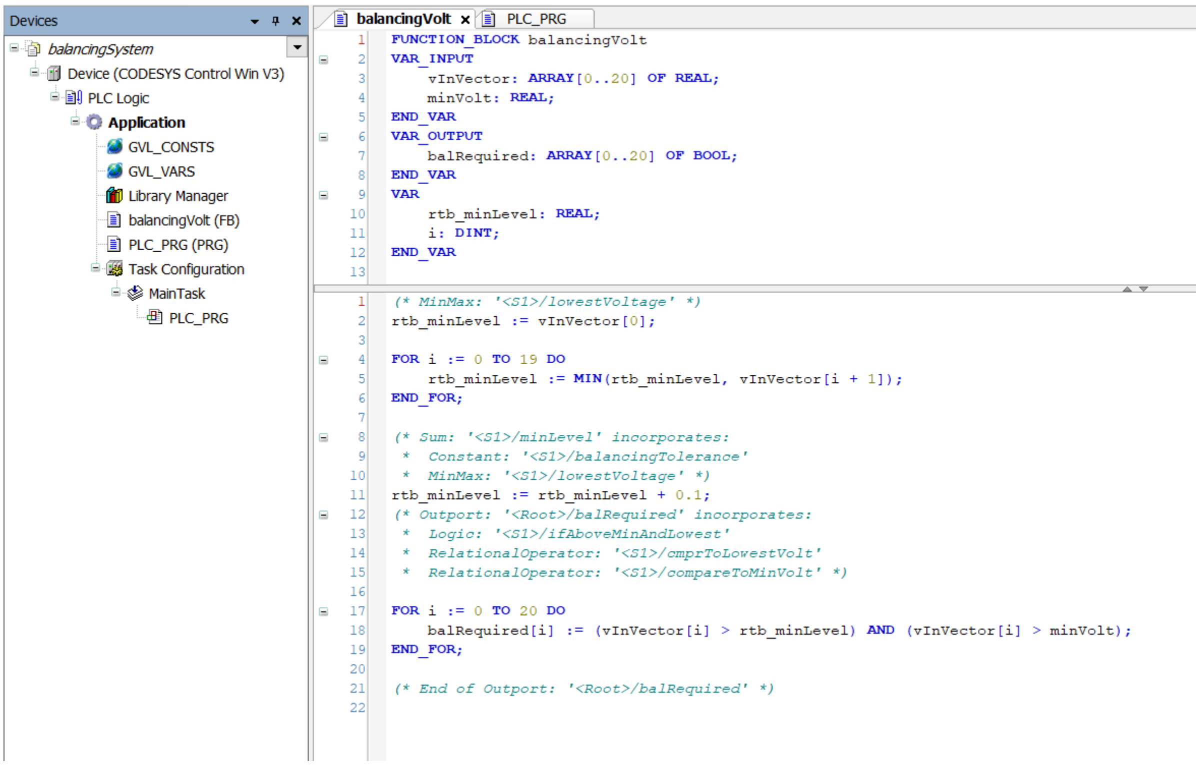
Please select the items which should be imported.
All items will be imported below the node which is currently selected in the navigator.
You can change this selection while this dialog is open.

Currently selected target object: (Root)

Insertable items:

- ☑ balancingVolt
- ☑ GVL_CONSTS
- ☑ GVL_VARS

Figure 3.15. The XML-file contains the ST-file as well as the global constants and variables.



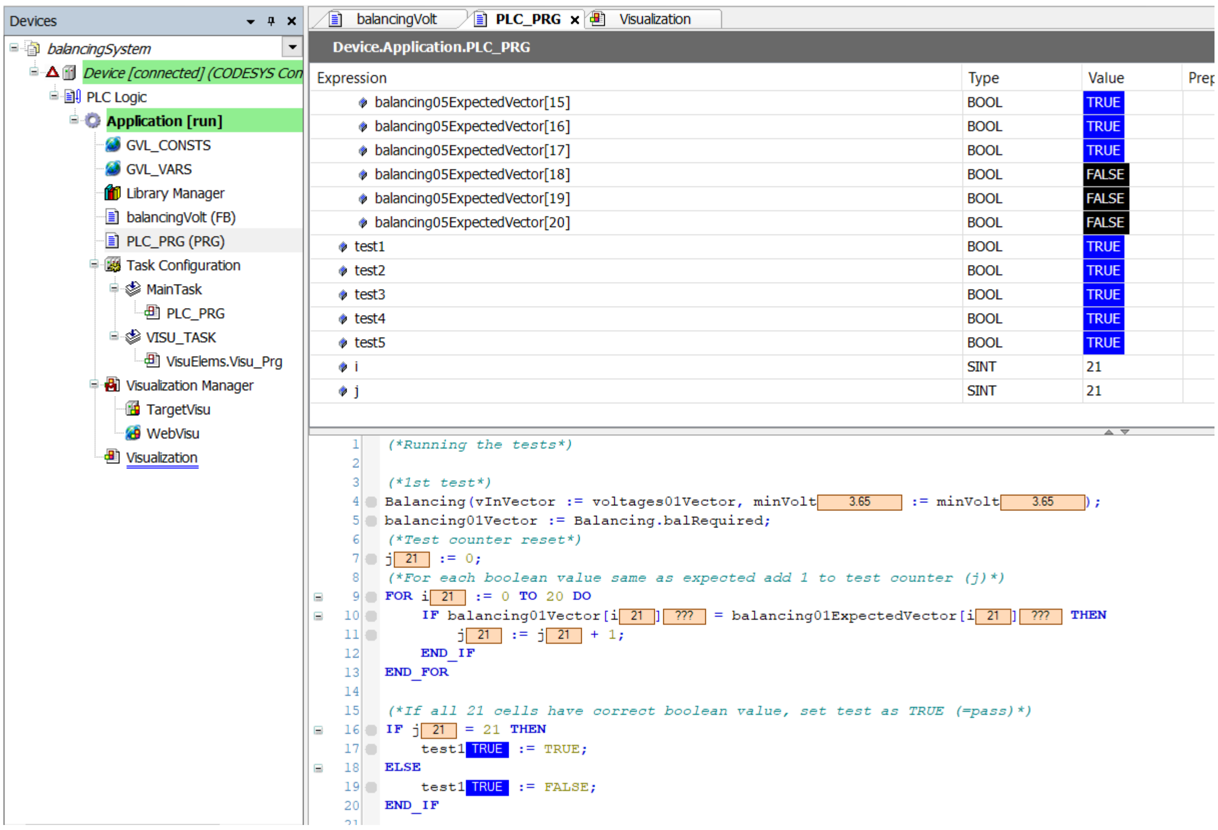
```
FUNCTION_BLOCK balancingVolt
VAR_INPUT
    vInVector: ARRAY[0..20] OF REAL;
    minVolt: REAL;
END_VAR
VAR_OUTPUT
    balRequired: ARRAY[0..20] OF BOOL;
END_VAR
VAR
    rtb_minLevel: REAL;
    i: DINT;
END_VAR
```

```
(* MinMax: '<S1>/lowestVoltage' *)
rtb_minLevel := vInVector[0];

FOR i := 0 TO 19 DO
    rtb_minLevel := MIN(rtb_minLevel, vInVector[i + 1]);
END_FOR;

(* Sum: '<S1>/minLevel' incorporates:
 *  Constant: '<S1>/balancingTolerance'
 *  MinMax: '<S1>/lowestVoltage' *)
rtb_minLevel := rtb_minLevel + 0.1;
(* Outport: '<Root>/balRequired' incorporates:
 *  Logic: '<S1>/ifAboveMinAndLowest'
 *  RelationalOperator: '<S1>/cmprToLowestVolt'
 *  RelationalOperator: '<S1>/compareToMinVolt' *)

FOR i := 0 TO 20 DO
    balRequired[i] := (vInVector[i] > rtb_minLevel) AND (vInVector[i] > minVolt);
END_FOR;

(* End of Outport: '<Root>/balRequired' *)
```

Figure 3.16. Imported function block in CODESYS environment.

After the import has been successful, the imported files become visible under the Application in CODESYS. At this point the imported files has to be checked to have the correct code. In this example, the commenting in the PLC Coder has been turned on, and the comments created by the coder are visible (the green text).

From this point on, the verification process can vary. In this case the imported code is verified using the same input vectors as when testing the model. If there are no testing tools to be used, the tests can be made manually with CODESYS like they were done in Simulink. In this case code verification is done by writing test code for the generated code using the exact same test vectors as input as was used when verifying the model. The results from the design verification were copied to CODESYS as arrays, and they were compared to the results from the generated function block with the test vectors and verified to be the same. Part of the code written for testing can be seen in figure 3.17, and the code is visible completely in appendix 3.



Devices
- balancingSystem
  - Device [connected] (CODESYS Con
    - PLC Logic
      - Application [run]
        - GVL_CONSTS
        - GVL_VARS
        - Library Manager
        - balancingVolt (FB)
        - PLC_PRG (PRG)
        - Task Configuration
          - MainTask
            - PLC_PRG
          - VISU_TASK
            - VisuElems.Visu_Prg
        - Visualization Manager
          - TargetVisu
          - WebVisu
        - Visualization

balancingVolt | PLC_PRG | Visualization

Device.Application.PLC_PRG

Expression	Type	Value	Prep
balancing05ExpectedVector[15]	BOOL	TRUE	
balancing05ExpectedVector[16]	BOOL	TRUE	
balancing05ExpectedVector[17]	BOOL	TRUE	
balancing05ExpectedVector[18]	BOOL	FALSE	
balancing05ExpectedVector[19]	BOOL	FALSE	
balancing05ExpectedVector[20]	BOOL	FALSE	
test1	BOOL	TRUE	
test2	BOOL	TRUE	
test3	BOOL	TRUE	
test4	BOOL	TRUE	
test5	BOOL	TRUE	
i	SINT	21	
j	SINT	21	

```
(*Running the tests*)

(*1st test*)
Balancing(vInVector := voltages01Vector, minVolt 3.65 := minVolt 3.65 );
balancing01Vector := Balancing.balRequired;
(*Test counter reset*)
j 21 := 0;
(*For each boolean value same as expected add 1 to test counter (j)*)
FOR i 21 := 0 TO 20 DO
    IF balancing01Vector[i 21 ] ??? = balancing01ExpectedVector[i 21 ] ??? THEN
        j 21 := j 21 + 1;
    END_IF
END_FOR

(*If all 21 cells have correct boolean value, set test as TRUE (=pass)*)
IF j 21 = 21 THEN
    test1 TRUE := TRUE;
ELSE
    test1 TRUE := FALSE;
END_IF
```

Figure 3.17. Snapshot of code verification with tests written in CODESYS.

4 Conclusions and Discussion

In this chapter, the results of this study are discussed. The benefits and drawbacks of using model-based design when developing software for programmable logic controllers are considered as well as different possible ways to use this method. It has to be remembered that this thesis is written based on a very specific way of using this method in mind. However, throughout the study, alternative uses for the method came up, and some of them are discussed here.

Furthermore, other related topics outside of the scope of this study are discussed briefly.

4.1 Benefits, Drawbacks and Problems of Model Based Software Design and Simulink PLC Coder

This thesis had a specific target to study: what are the possibilities of using partly or completely model-based design when developing software for programmable logic controllers, and more specifically those used on commercial vehicles and NRMM. The IEC 61131 -compatible language used for these controllers is CODESYS 3.5 structured text. Some of the software used in the controllers has to be IEC 61508 certified, but some parts of the code can be made without that requirement and thus only has to be made according to pre-determined project-specific quality requirements. That being said, in any case extensive testing is required, and the difference between testing code that has to be certified and code that has not, is mainly in documentation of the testing.

In this case, using model-based design only partly in the software development process means that model-based design is only used in parts of the software that does not require IEC 61508 certification. However, the differences and requirements for using model-based design in the whole software development process are also studied throughout the thesis. The idea is that with some functionalities, that include an algorithm of some sort, would be easier and less time consuming to develop using model-based design. As mentioned in chapter 2.1, Simulink has an extensive number of built-in algorithms, and using those is basically just a matter of adding blocks within a graphical model one after another following the functional requirements set for the function in question. The functionality is also easy to dynamically test throughout the development process since building a testing environment is quite a simple process compared to having to write manual tests for hand-written code.

Regarding the verification part in the case when software is only partly being developed with model-based design there are a couple of viable options. One option is just to use Simulink for creating the model and generating the code with the PLC Coder, and then just verify and validate the code manually or/and using automated tools just like if it was hand-written code. With this method there are still the benefits of using model-based design, but with minimal extra costs software-wise. In that case only the tools are required that would be used anyways throughout the software development process. It would not either interfere with the strict requirements of IEC 61508 certification since only functions without the need of certification would be done using this method. There would still be the cost of the PLC Coder that would not be otherwise required. Having Simulink, on the other hand, has much more benefits besides model-based software design. Usually these kinds of systems are simulated during the development process, and Simulink is a widely used tool for that purpose either in software in the loop (SIL) or hardware in the loop (HIL) testing. This also means that there is a high possibility that the software would in any case be largely modelled for the simulations in Simulink. With this in mind, model-based design makes even more sense in a process like this.

On the other hand, it can be said that with some additional tools for automated code certification with model-based design, as in this case the IEC Certification kit for Simulink would be, the software development process can be simplified (from the engineer's perspective) quite significantly. Engineers that are used to do software development by traditionally writing the code by hand most certainly would not have problems in developing software with Simulink and PLC Coder. Also, engineers with maybe little knowledge in coding, but that are familiar with the system, could then be part of the software development process, too. That is one of the main benefits of this method in addition to the time savings. That being said, as mentioned before, it is crucial in any case to have at least some one in the software development process who also understands the code-part well, and all engineers should be able to have some idea what the code means and how it works. As a conclusion, model-based design could definitely be used as the main software development method in a project that involves programmable logic controllers and the IEC 61508 safety standard.

4.2 Further Possible Areas of Study in Using the Simulink PLC Coder

As this thesis had a quite specific area of study, it is now discussed what could be studied next regarding this field. First of all, the usage of the automated tools mentioned throughout the thesis could be studied and presented more thoroughly. For example, regarding the IEC 61508 Certification kit for Simulink is not specifically designed for PLC Coder. Although it is claimed to produce automatically IEC 61508 certified code with the PLC Coder, it could be studied whether there are in reality some problems using that.

Regarding the software developed as an example for this thesis it could be argued that it was a relatively simple model, and a more complex model with more blocks and multi-level subsystems would have caused more problems in the code generation process. There is, however, a couple of reasons why a model like this was used for this thesis. First of all, a model that is too complex would have made reading this thesis and thus understanding the process significantly more difficult, especially if the reader is somewhat unknowledgeable of the tools used. Secondly, especially in projects like this, but in coding in general, it is usually good practice to keep the functions simple enough. Should there come problems with a model that is more complex, it might be good idea to rethink the layout of the program or function in question. That is not, however, always the case, and sometimes ending up with a more complex model or function is just inevitable, and that is why it could be a good idea to extend this research to some more complex programs, maybe with more than one function. This would also make a good ground for the extended automated verification tool research.

5 References

Berg, H. (2015) *Batteries for electric vehicles: Materials and electrochemistry*, *Batteries for Electric Vehicles: Materials and Electrochemistry*. Cambridge University Press. doi: 10.1017/CBO9781316090978.

Bolton, W. (2009) *Programmable logic controllers*. 5th ed. Newnes.

Boulanger, J.-L. (2012) *Formal method : industrial use from model to the code*. 1st edn. ISTE ; Wiley (Industrial implementation of formal methods series.).

CODESYS (2020a) *CODESYS - THE COMPREHENSIVE SOFTWARE SUITE FOR AUTOMATION TECHNOLOGY*. Available at: https://www.codesys.com/the-system.html (Accessed: 4 May 2020).

CODESYS (2020b) *Data Sheet CODESYS Static Analysis*. Available at: https://store.codesys.com/codesys-static-analysis.html?___store=en (Accessed: 4 May 2020).

Conrad, M. (2009) 'Testing-based translation validation of generated code in the context of IEC 61508', *Formal Methods in System Design*, 35(3), pp. 389–401. doi: 10.1007/s10703-009-0082-0.

Conrad, M. and Sandmann, G. (2009) 'A Verification and Validation Workflow for IEC 61508 Applications', in *SAE Technical Papers*. SAE International. doi: 10.4271/2009-01-0271.

Daowd, M. *et al.* (2011) 'Passive and active battery balancing comparison based on MATLAB simulation', in *2011 IEEE Vehicle Power and Propulsion Conference, VPPC 2011*. doi: 10.1109/VPPC.2011.6043010.

Gomez, K. (2010) 'MathWorks Simulink PLC Coder certified by TÜV SÜD', *PACE: Process and Control Engineer; South Melbourne*. Available at: https://search.proquest.com/docview/940846596?accountid=27468 (Accessed: 25 March 2020).

Halvorson, B. (2018) *Here's the battery pack behind VW's global electric-vehicle push*, *Green Car Reports*. Available at: https://www.greencarreports.com/news/1118974_heres-the-battery-pack-behind-vws-global-electric-vehicle-push (Accessed: 25 May 2020).

Hambling, B. *et al.* (2010) *SOFTWARE TESTING*. 1st editio. British Informatics Society Limited.

Hanssen, D. H. (2015) *Programmable logic controllers : a practical approach to IEC 61131-3 using CODESYS*. First edit. Wiley.

He, N., Oke, V. and Allen, G. (2016) 'Model-based verification of PLC programs using Simulink design', in *IEEE International Conference on Electro Information Technology*. IEEE Computer Society, pp. 211–216. doi: 10.1109/EIT.2016.7535242.

Jiang, J. and Zhang, C. (2015) *Fundamentals and applications of lithium-ion batteries in* *electric drive vehicles*. John Wiley & Sons Inc.

John, K. H. and Tiegelkamp, M. (2010) *IEC 61131-3: Programming industrial automation systems: Concepts and programming languages, requirements for programming systems, decision-making aids*, *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-12015-2.

Lambert, F. (2019) *Tesla battery supplier Panasonic is considering switching 18650 cell production to 2170 cells - Electrek*, *electrek*. Available at: https://electrek.co/2019/04/25/tesla-battery-supplier-panasonic-18650-2170-cell-production/ (Accessed: 20 May 2020).

Lebel, F. A. *et al.* (2019) 'Implications of lithium-ion cell variations on multi-cell battery pack thermal runaway', in *2019 IEEE Vehicle Power and Propulsion Conference, VPPC 2019 - Proceedings*. Institute of Electrical and Electronics Engineers Inc. doi: 10.1109/VPPC46532.2019.8952282.

MathWorks (2010) *Simulink ® PLC Coder^{TM} User's Guide R2020a*. Available at: www.mathworks.com (Accessed: 3 March 2020).

PLCopen (2016a) *Coding Guidelines*. Available at: https://plcopen.org/system/files/downloads/plcopen_coding_guidelines_version_1.0.pdf (Accessed: 13 July 2020).

PLCopen (2016b) *IEC 61131-3: a standard programming resource*. Available at: https://plcopen.org/sites/default/files/downloads/intro_iec_oct2016.pdf (Accessed: 7 April 2020).

Qnovo (2015) *69. Inside the battery of a Nissan Leaf*. Available at: https://qnovo.com/inside-the-battery-of-a-nissan-leaf/ (Accessed: 25 May 2020).

Smith, D. and Simpson, K. (2016) *The Safety Critical Systems Handbook, 4th Edition*. 4th editio. Butterworth-Heinemann.

Warner, J. (2015) *Handbook of Lithium-Ion Battery Pack Design - Chemistry, Components, Types and Terminology - Knovel*. Elsevier. Available at: https://app.knovel.com/web/toc.v/cid:kpHLIBPDC3/viewerType:toc//root_slug:handbook-of-lithium?kpromoter=marc (Accessed: 13 July 2020).

Wu, Y. (2015) *Lithium-Ion Batteries*. 1st editio. CRC Press.

Xiong, R. and Shen, W. (2019) *Advanced Battery Management Technologies for Electric Vehicles*. 1st editio. Wiley.

Yoon, S.-H. (2013) *Automatic Code Generation*. Available at: https://it.mathworks.com/content/dam/mathworks/mathworks-dot-com/solutions/automotive/files/kr-expo-2013/Track_2_4.pdf (Accessed: 13 July 2020).

Zamboni, L. (2013) *Getting Started with Simulink*. 1st editio. Packt Publishing.

Appendix I The MATLAB/Simulink Test Code

```
% INPUT VECTORS FOR BATTERY CELL BALANCING TESTING
% Cells in array numbered 1...21

% No variation in voltages, no voltage above min
% -> No cells to be balanced
voltages = [3.6, 3.6, 3.6, 3.6, 3.6, 3.6, 3.6, 3.6, 3.6,...
    3.6, 3.6, 3.6, 3.6, 3.6, 3.6, 3.6, 3.6, 3.6, 3.6, 3.6, 3.6];
minVolt = 3.65;
balancing01 = sim('balancing_system5.slx');

% No variation in voltages, all voltages above min
% -> No cells to be balanced
voltages = [3.7, 3.7, 3.7, 3.7, 3.7, 3.7, 3.7, 3.7, 3.7,...
    3.7, 3.7, 3.7, 3.7, 3.7, 3.7, 3.7, 3.7, 3.7, 3.7, 3.7, 3.7];
balancing02 = sim('balancing_system5.slx');

% Enough variation in voltages for balancing, but none above min
% -> No cells to be balanced
voltages = [3.54, 3.65, 3.65, 3.65, 3.65, 3.65, 3.65, 3.65, 3.65,...
    3.65, 3.65, 3.65, 3.65, 3.65, 3.65, 3.65, 3.65, 3.65, 3.65, 3.65, 3.65];
balancing03 = sim('balancing_system5.slx');

% Enough variation in voltages for balancing, all above min
% -> Cells 12 - 18 should be balanced, others not
voltages = [3.66, 3.67, 3.68, 3.69, 3.7, 3.71, 3.72, 3.73, 3.74,...
    3.75, 3.76, 3.77, 3.78, 3.79, 3.8, 3.81, 3.82, 3.83, 3.67, 3.7, 3.75];
balancing04 = sim('balancing_system5.slx');

% Enough variation in voltages for balancing, some above min
% -> Cells 3, 7, 8, 9, 14, 15, 16, 17, 18 should be balanced
voltages = [3.65, 3.61, 3.66, 3.62, 3.58, 3.5, 3.69, 3.7, 3.71,...
    3.51, 3.63, 3.64, 3.65, 3.73, 3.67, 3.7, 3.72, 3.71, 3.62, 3.57, 3.52];
balancing05 = sim('balancing_system5.slx');

% Print out the results

% 1st test
fprintf('Balancing required in cells nr. (1st test): \n');
fprintf('%i ', find(balancing01.balRequired));
fprintf('\n');
fprintf('Balancing NOT required in cells nr. (1st test): \n');
fprintf('%i ', find(~(balancing01.balRequired)));
fprintf('\n');
fprintf('No cells should be balanced');
fprintf('\n\n');

% 2nd test
fprintf('Balancing required in cells nr. (2nd test): \n');
fprintf('%i ', find(balancing02.balRequired));
fprintf('\n');
fprintf('Balancing NOT required in cells nr. (2nd test): \n');
fprintf('%i ', find(~(balancing02.balRequired)));
fprintf('\n');
fprintf('No cells should be balanced');
fprintf('\n\n');

% 3rd test
fprintf('Balancing required in cells nr. (3rd test): \n');
fprintf('%i ', find(balancing03.balRequired));
fprintf('\n');
fprintf('Balancing NOT required in cells nr. (3rd test): \n');
fprintf('%i ', find(~(balancing03.balRequired)));
fprintf('\n');
fprintf('No cells should be balanced');
fprintf('\n\n');

% 4th test
fprintf('Balancing required in cells nr. (4th test): \n');
fprintf('%i ', find(balancing04.balRequired));
fprintf('\n');
fprintf('Balancing NOT required in cells nr. (4th test): \n');
fprintf('%i ', find(~(balancing04.balRequired)));
fprintf('\n');
fprintf('Cells 12 - 18 should be balanced, others not');
fprintf('\n\n');

% 5th test
fprintf('Balancing required in cells nr. (5th test): \n');
fprintf('%i ', find(balancing05.balRequired));
fprintf('\n');
fprintf('Balancing NOT required in cells nr. (5th test): \n');
fprintf('%i ', find(~(balancing05.balRequired)));
fprintf('\n');
fprintf('Cells 3, 7, 8, 9, 14, 15, 16, 17, 18 should be balanced');
fprintf('\n\n');
```

```
Balancing required in cells nr. (1st test):

Balancing NOT required in cells nr. (1st test):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
No cells should be balanced

Balancing required in cells nr. (2nd test):

Balancing NOT required in cells nr. (2nd test):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
No cells should be balanced

Balancing required in cells nr. (3rd test):

Balancing NOT required in cells nr. (3rd test):
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
No cells should be balanced

Balancing required in cells nr. (4th test):
12 13 14 15 16 17 18
Balancing NOT required in cells nr. (4th test):
1 2 3 4 5 6 7 8 9 10 11 19 20 21
Cells 12 - 18 should be balanced, others not
```

```
Balancing required in cells nr. (5th test):
3 7 8 9 14 15 16 17 18
Balancing NOT required in cells nr. (5th test):
1 2 4 5 6 10 11 12 13 19 20 21
Cells 3, 7, 8, 9, 14, 15, 16, 17, 18 should be balanced
```

Published with MATLAB® R2019b

Appendix II The Generated Structured Text Code

```
(*
 *
 * File: balancing_system5.st
 *
 * IEC 61131-3 Structured Text (ST) code generated for subsystem
"balancing_system5/balancingVolt"
 *
 * Model name                       : balancing_system5
 * Model version                    : 1.9
 * Model creator                    : Kaarle Patomäki
 * Model last modified by           : Kaarle Patomäki
 * Model last modified on           : Thu Jun 11 14:20:43 2020
 * Model sample time                : 1s
 * Subsystem name                  : balancing_system5/balancingVolt
 * Subsystem sample time            : 1s
 * Simulink PLC Coder version       : 3.1 (R2019b) 18-Jul-2019
 * ST code generated on             : Sun Jun 14 18:41:38 2020
 *
 * Target IDE selection             : 3S CoDeSys 3.5
 * Test Bench included              : No
 *
 *)
FUNCTION_BLOCK balancingVolt
VAR_INPUT
    vInVector: ARRAY [0..20] OF REAL;
    minVolt: REAL;
END_VAR
VAR_OUTPUT
    balRequired: ARRAY [0..20] OF BOOL;
END_VAR
VAR_TEMP
    rtb_minLevel: REAL;
    i: DINT;
END_VAR
(* MinMax: '<S1>/lowestVoltage' *)
rtb_minLevel := vInVector[0];
FOR i := 0 TO 19 DO
    rtb_minLevel := MIN(rtb_minLevel, vInVector[i + 1]);
END_FOR;
(* Sum: '<S1>/minLevel' incorporates:
 *  Constant: '<S1>/balancingTolerance'
 *  MinMax: '<S1>/lowestVoltage' *)
rtb_minLevel := rtb_minLevel + 0.1;
(* Outport: '<Root>/balRequired' incorporates:
 *  Logic: '<S1>/ifAboveMinAndLowest'
 *  RelationalOperator: '<S1>/cmprToLowestVolt'
 *  RelationalOperator: '<S1>/compareToMinVolt' *)
FOR i := 0 TO 20 DO
    balRequired[i]   :=   (vInVector[i]   >   rtb_minLevel)   AND
(vInVector[i] > minVolt);
END_FOR;
(* End of Outport: '<Root>/balRequired' *)
END_FUNCTION_BLOCK
```

Appendix III The CODESYS Test Code

```
PROGRAM PLC_PRG
VAR
Balancing : balancingVolt ;
voltages01Vector : ARRAY [ 0 .. 20 ] OF REAL
:= [ 3.6 , 3.6 , 3.6 , 3.6 , 3.6 , 3.6 , 3.6 , 3.6 , 3.6 ,
3.6 , 3.6, 3.6 , 3.6 , 3.6 , 3.6 , 3.6 , 3.6 , 3.6 , 3.6 ,
3.6 , 3.6 ] ;

voltages02Vector : ARRAY [ 0 .. 20 ] OF REAL
:= [ 3.7 , 3.7 , 3.7 , 3.7 , 3.7 , 3.7 , 3.7 , 3.7 , 3.7 ,
3.7 , 3.7 , 3.7 , 3.7 , 3.7 , 3.7 , 3.7 , 3.7 , 3.7 , 3.7 ,
3.7 , 3.7 ] ;

voltages03Vector : ARRAY [ 0 .. 20 ] OF REAL
:= [ 3.54 , 3.65 , 3.65 , 3.65 , 3.65 , 3.65 , 3.65 , 3.65 ,
3.65 , 3.65 , 3.65 , 3.65 , 3.65 , 3.65 , 3.65 , 3.65 , 3.65
, 3.65 , 3.65 , 3.65 , 3.65 ] ;

voltages04Vector : ARRAY [ 0 .. 20 ] OF REAL
:= [ 3.66 , 3.67 , 3.68 , 3.69 , 3.7 , 3.71 , 3.72 , 3.73 ,
3.74 , 3.75 , 3.76 , 3.77 , 3.78 , 3.79 , 3.8 , 3.81 , 3.82 ,
3.83 , 3.67 , 3.7 , 3.75 ] ;

voltages05Vector : ARRAY [ 0 .. 20 ] OF REAL
:= [ 3.65 , 3.61 , 3.66 , 3.62 , 3.58 , 3.5 , 3.69 , 3.7 ,
3.71 , 3.51 , 3.63 , 3.64 , 3.65 , 3.73 , 3.67 , 3.7 , 3.72 ,
3.71 , 3.62 , 3.57 , 3.52 ] ;

 minVolt : REAL := 3.65 ;

balancing01Vector : ARRAY [ 0 .. 20 ] OF BOOL ;
balancing02Vector : ARRAY [ 0 .. 20 ] OF BOOL ;
balancing03Vector : ARRAY [ 0 .. 20 ] OF BOOL ;
balancing04Vector : ARRAY [ 0 .. 20 ] OF BOOL ;
balancing05Vector : ARRAY [ 0 .. 20 ] OF BOOL ;

balancing01ExpectedVector : ARRAY [ 0 .. 20 ] OF BOOL
:= [ FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE ,
FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE
, FALSE , FALSE , FALSE , FALSE , FALSE , FALSE ] ;

balancing02ExpectedVector : ARRAY [ 0 .. 20 ] OF BOOL
:= [ FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE ,
FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE
, FALSE , FALSE , FALSE , FALSE , FALSE , FALSE ] ;

balancing03ExpectedVector : ARRAY [ 0 .. 20 ] OF BOOL
:= [ FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE ,
FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE
, FALSE , FALSE , FALSE , FALSE , FALSE , FALSE ] ;

balancing04ExpectedVector : ARRAY [ 0 .. 20 ] OF BOOL
```

```
:= [ FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , FALSE , TRUE , TRUE , TRUE , TRUE , TRUE , TRUE , TRUE , FALSE , FALSE , FALSE ] ;

balancing05ExpectedVector : ARRAY [ 0 .. 20 ] OF BOOL
:= [ TRUE , FALSE , TRUE , FALSE , FALSE , FALSE , TRUE , TRUE , TRUE , FALSE , FALSE , FALSE , FALSE , TRUE , TRUE , TRUE , TRUE , TRUE , FALSE , FALSE , FALSE ] ;

test1 : BOOL ;
test2 : BOOL ;
test3 : BOOL ;
test4 : BOOL ;
test5 : BOOL ;

i : SINT ;
j : SINT ;

END_VAR


(*Running the tests*)

(*1st test*)
Balancing ( vInVector := voltages01Vector , minVolt := minVolt ) ;
balancing01Vector := Balancing . balRequired ;
(*Test counter reset*)
j := 0 ;
(*For each boolean value same as expected add 1 to test counter (j)*)
FOR i := 0 TO 20 DO
      IF balancing01Vector [ i ] = balancing01ExpectedVector [ i ] THEN
            j := j + 1 ;
      END_IF
END_FOR

(*If all 21 cells have correct boolean value, set test as TRUE (=pass)*)
IF j = 21 THEN
      test1 := TRUE ;
ELSE
      test1 := FALSE ;
END_IF

(*2nd test*)
Balancing ( vInVector := voltages02Vector , minVolt := minVolt ) ;
balancing02Vector := Balancing . balRequired ;

j := 0 ;
FOR i := 0 TO 20 DO
      IF balancing02Vector [ i ] = balancing02ExpectedVector [ i ] THEN
```

```
            j := j + 1 ;
        END_IF
END_FOR

IF j = 21 THEN
        test2 := TRUE ;
ELSE
        test2 := FALSE ;
END_IF


(*3rd test*)
Balancing ( vInVector := voltages03Vector , minVolt := minVolt ) ;
balancing03Vector := Balancing . balRequired ;

j := 0 ;
FOR i := 0 TO 20 DO
        IF balancing03Vector [ i ] = balancing03ExpectedVector [ i ] THEN
            j := j + 1 ;
        END_IF
END_FOR

IF j = 21 THEN
        test3 := TRUE ;
ELSE
        test3 := FALSE ;
END_IF


(*4th test*)
Balancing ( vInVector := voltages04Vector , minVolt := minVolt ) ;
balancing04Vector := Balancing . balRequired ;

j := 0 ;
FOR i := 0 TO 20 DO
        IF balancing04Vector [ i ] = balancing04ExpectedVector [ i ] THEN
            j := j + 1 ;
        END_IF
END_FOR

IF j = 21 THEN
        test4 := TRUE ;
ELSE
        test4 := FALSE ;
END_IF


(*5th test*)
Balancing ( vInVector := voltages05Vector , minVolt := minVolt ) ;
balancing05Vector := Balancing . balRequired ;
```

```

j := 0 ;
FOR i := 0 TO 20 DO
	IF balancing05Vector [ i ] = balancing05ExpectedVector
	[ i ] THEN
		j := j + 1 ;
	END_IF
END_FOR

IF j = 21 THEN
	test5 := TRUE ;
ELSE
	test5 := FALSE ;
END_IF
```