Tampereen yliopisto

Tapio Rytinki

MODEL-BASED RE-ENGINEERING OF CONTROL APPLICATION
Code generation and verification

Master of Science Thesis
Faculty of Engineering and Natural sciences
Examiners: Reza Ghabcheloo
Examiners: Niko Siltala
April 2022

TIIVISTELMÄ

Tapio Rytinki: Mallipohjaisen suunnittelun käyttö ohjaussovelluksen uudelleensuunnittelussa.
Diplomityö
Tampereen Yliopisto
Automaatiotekniikka
Huhtikuu 2022

Tämä opinnäytetyö esittelee tavan siirtyä perinteisestä ohjelmistokehityksestä mallipohjaiseen suunnitteluun ohjaussovellusalueella, erityisesti PLC-pohjaisissa ohjausjärjestelmissä. Perinteisellä ohjelmistokehityksellä tarkoitetaan prosessia, jossa ohjelmisto kirjoitetaan suoraan järjestelmä- ja moduulisuunnittelun perusteella järjestelmävaatimuksien määrittelyn jälkeen. Mallipohjaisella uudelleensuunnittelulla tarkoitetaan prosessia, jossa vanha ohjelmisto muunnetaan uudeksi toteutukseksi mallipohjaisen suunnittelun menetelmiä hyödyntäen. Mallipohjainen suunnittelu on matemaattinen ja visuaalinen menetelmä monimutkaisten ohjausjärjestelmän ongelmien ratkaisemiseksi, ja se keskittyy kehitysprosessin suunnitteluvaiheeseen.
Kiinnostus mallipohjaiseen suunnitteluun perinteisen kehitystyön sijaan on lisääntynyt useista syistä, mutta siirtyminen tähän on ongelmallista. Vanha ohjelmisto on yleensä käsin kirjoitettu ja muuntaminen mallipohjaisiksi ja esimerkiksi siirtyminen uudelle alustalle voi olla monimutkaista. Tämä opinnäytetyö vastaa esitettyyn ongelmaan kehittämällä systemaattisen tavan, miten voidaan siirtyä käyttämään mallipohjaista suunnittelua ohjelmistokehityksessä. Tämä saavutetaan yhdistämällä uudelleensuunnitteluun prosessi ja ohjelmistokehityksessä paljon käytetty V-malli toisiinsa. Lisäksi työssä esitellään esimerkkitapaus, jossa käytetään esiteltyä prosessia viiteohjelmiston muuntamiseksi uudeksi toteutukseksi mallipohjaisen suunnittelun avulla. Tämä esimerkkitapaus keskittyy siihen, miten ohjelmiston verifikaatio tapahtuu esitetyssä uudelleensuunnittelun mallissa. Tämä esimerkkitapaus on tehty käyttämällä mallipohjaiseen suunnitteluprosessiin tarkoitettua ohjelmistoa, joka tunnetaan nimellä MathWorks 'Simulink, ja tämän lisäosaa Simulink PLC-coder. Esimerkkitapauksessa onnistuttiin luomaan mallipohjaista suunnittelua hyödyntäen uudelleensuunniteltu ohjelmisto. Esimerkkitapauksen tuloksena arvioidaan uudelleensuunnitteluprosessin ongelmia, itse tapaustutkimusta ja esitetään idea jatkotutkimukselle.

Avainsanat: Model-based design, CODESYS, Simulink, code generation, IEC 61131, Programmable logic controller, Re-engineering, Verification

ABSTRACT

Tapio Rytinki: Model-based re-engineering of control application
Master of Science thesis
Tampere University
Automation Engineering
April 2022

This thesis introduces a way to transform from traditional software development to model-based design in the control application domain, specifically PLC-based control systems. Traditional software development refers to a process where code is written directly based on system and module design after system requirements are defined. Model-based re-engineering refers to a process where old software is converted to new implementation using a model-based design methodology. Model-based design is a mathematical and visual method to address complex control system problems and is focused on the design phase of the development process.

There are multiple reasons why there is a rising interest to use model-based design instead of traditional development but changing to this development model is problematic. The old codebase is usually done by handwritten code and transforming to model-based and new platforms can be complex. This thesis answers the presented problem by developing a systematic way how this transformation can be done. This is achieved by combining re-engineering with V-model. Furthermore, a case study is performed which uses the introduced process for transforming the reference code into a new implementation using model-based design. This case study focuses on how the verification process evolves when re-engineering is part of the V-model. This case study is completed using proprietary software for the model-based design process known as MathWorks' Simulink and Simulink PLC-coder. Case study showed that it was possible to create a redesigned software using introduced re-engineering model. As a result of this case study, problems in the re-engineering process and the case study itself are explored and an idea for further study is presented.

Keywords: Model-based design, CODESYS, Simulink, code generation, IEC 61131, Programmable logic controller, Re-engineering, Verification

PREFACE

This thesis was made in partnership with Cargotec Finland Oy, Kalmar. Special thanks to Development Manager Janne Kuosmanen from Cargotec for providing an interesting and relevant problem to study and pushing along the way. This topic was well suited for my characteristics and the topic is something that should receive more and more interest in the future. Also big thanks to Reza Ghabcheloo and Niko Siltala for good feedback and positive attitude and most of all, for his patience related to this thesis.

Special appreciation to my family for understanding my absence during these long workdays.

Tampere, 15 April 2022

Tapio Rytinki

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

MBD	model-based design
SW	software
SoM	StartingOfMotion -subfunction
TUT	Tampere university of technology
ST	Structured text
FB	Function block
PLC	programmable control logic
V-MODEL	refers to a specific software development process
RE	Re-engineering
HIL	Hardware-in-the-loop
SIL	Software-in-the-loop
MIL	Model-in-the-loop
SDLC	Software development life cycle

.

1. INTRODUCTION

Highly developed and complex control systems are hard to understand and increasingly hard to further improve and develop. Most control systems today are built on specific hardware/software platforms which require platform-specific know-how as well as codebase. A control system is usually set up from a software perspective and control-related aspects are underrated, which can cause an unorganized code base when new features are introduced. This raises the question, what happens when requirements exceed platform limitations? Codebase must be created or transformed into a new platform, which causes several problems: how to ensure that the new software is working as the old one and how to gain expertise for a new platform, and many others related to this transformation. This thesis explores this problem space and how *model-based re-engineering* could solve these issues. Model-based re-engineering in the context of this thesis means software development process transformation to model-based while retaining validated implementation from the old software.

Interest in model-based design has risen from its ability to decouple developing, testing, and verification of complex system models from coding practicalities. A traditional method for creating embedded code is to use text-based documents for design documentation and creating code based on requirements. This leads to a time-consuming process of interpretation of documents and there is a disconnect between engineering requirements and actual working software. Using model-based design (MBD) instead of the traditional method allows auto-generation of IEC-61131 compliant code which can eliminate human coding errors and make implementation more understandable without platform-specific expertise. In the long run, MBD can also lead to easier testing using Model-in-the-loop without needing actual hardware to test implementation.

Model-based design with code generation should lead to error-free translated programs but as a result the correctness of design models i.e., Simulink diagrams, is significantly important. And when this method is combined with re-engineering, models should also conform to old implementation.

The objective of this thesis is to study how to re-engineer control software using model-based design and create an example process model for it. To test this new process, a case study is performed for a real-life example controller. The case study aims to model and generate code that should not modify the functionality of the original controller. MathWorks' MATLAB/Simulink is used as software as there are no other viable options currently to generate IEC-61131 compliant code from a model. This thesis should provide a basic understanding how model-based design can be used together with Simulink PLC-coder to re-engineer software and how to verify results against the original implementation.

The following questions should be answered in the thesis:

- Describe model-based re-engineering process for PLC control software.
- What are the advantages and disadvantages of re-engineering control software with the proposed process model?
- How to verify generated PLC code with model-based design?
- Evaluate how applicable model-based design is for re-engineering control software in different scenarios.

Research questions are complex and arguable by nature and this thesis does not provide strict answers on how re-engineering should be executed and what are the best options for every case. Instead, options are explored and one case study is done with an actual controller which gives first-hand experience to answer qualitative thesis questions.

Individual tasks shall be divided as follows to answer thesis questions:

- Research traditional PLC software development and model-based design methodology.
- Research software re-engineering and related topics.
- Study and determine code verification and testing procedures.
- Propose a process on how to re-engineer PLC software using MBD.
- Testing the proposed process with an actual controller
- Analyse proposed process and results for case study

Background research was done before starting this thesis which showed that there is very little background study on this subject that combines MBD with re-engineering. Therefore, this combination must be done using individual studies of each subject and the process must be created based on these.

This thesis contains a total of 7 chapters. The first chapter introduces the reader to the theoretical aspect of the whole thesis and the problem statement. This is split into 6 subchapters which are essential to justify the created process. The first subchapter introduces control systems in general and PLCs as well as code-standard used in PLC programming. This subchapter is presented to give the reader understanding of the actual code that is presented in later parts. The second subchapter introduces generally used development models in SW development. The third subchapter introduces model-based design which could be thought of as a subset of the second chapter. The fourth subchapter introduces what verification and validation mean in a broader sense and how these are related to model-based design. The fifth subchapter show what does re-engineering generally means.

The third chapter, after the theoretical part of the thesis, combines the second chapter's topics with a general re-engineering process which is used in the fourth chapter as a base for a case study.

The fifth chapter works as a conclusion and presents results from the case study and analyses the findings. The last chapter discusses what has been archived in this thesis and gives future study proposals related to this topic.

2. KEY CONCEPTS

This thesis will approach the topic by introducing several key concepts which are related to this topic. The topic has multiple key concepts and these are shown in Figure 1. as a mind-map at the abstract level which should help the reader understand the relations between these knowledge domains.

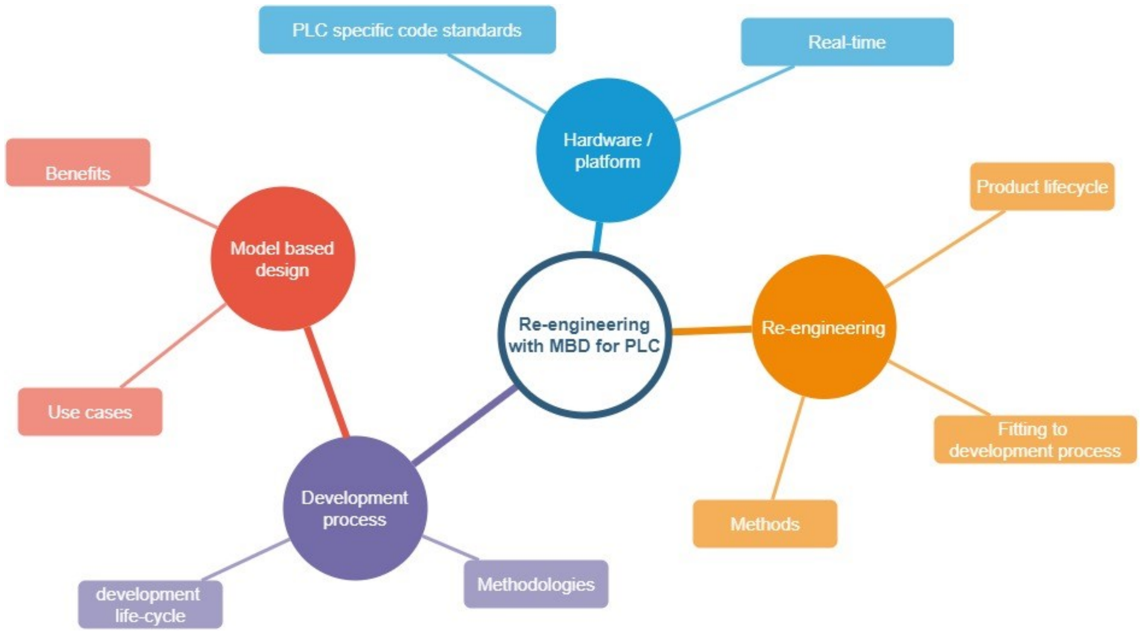


Figure 1, Key concepts as a mind map

As can be seen from the figure, this thesis will contain parts from multiple different domains. This was recognized as a risk at the start of this thesis as it could be hard to make a cohesive thesis. But as this kind of work has not been done before, the basis needs to be covered to some extent.

2.1 Control systems and controllers

This chapter introduces the definitions of systems and controllers. These definitions are simplified, and more detailed definitions are not needed for this thesis, because this thesis assumes that the reader has some basic knowledge of control systems.

A system can be considered as an arrangement of parts that form a system and has a boundary to its environment and the system changes its reaction based on some kind of manipulation of the environment and the system's changes can be observed. [1].

A control system can be defined as a system that can maintain or modify a system's state to desired result or value [1]. This means that each complex control system can be considered as many subsystems intertwined together to make a whole control system. In more practical terms, usually control system is a part of a more complex control system and ideally, one control system handles one kind of information and controls over part of the system based on requirements from other systems.

A controller is the part of the system that manages systems reactions and as a whole, these form a control system. Controllers can be implemented in multiple ways depending on the application requirements but usually, when a controller is mentioned, some kind of computational device is referred to. There are numerous different architectures on how control systems are rearranged and usually, computer applications are used in a higher level of a control system and programmable logic controllers (PLCs) are used when control is needed next to the field level. A controller software, in a broader sense, is a software program that manages and directs the flow of data between two entities.

2.1.1 Programmable logic controllers and code standard

A programmable logic controller (PLC) or programmable controller is an industrial digital computer that has been adapted for different environments and modified to be highly reliable. They consist of hardware, software, and environment like other computing systems. This special form of microprocessor-based controller stores programs as instructions and the programming language is designed to be more easily programmed than traditional computer programs. The operations of a PLC can be considered hard real-time operations because the control code is executed within a certain time limit. The term “logic” in PLC derives from the use case: PLCs are usually used for logic and switching operations [2] but are now increasingly used in mathematically complex controls with increased processing power.

Embedded Systems (and PLC systems) are characterized by the following attributes [3]:

- Usually, a control system is a subsystem providing function to a higher-level system.
- Control systems provide access and processing to other devices or subsystems
- Control systems perform physical control activities like reading sensors or commanding actuators.

- They are mostly not visible or directly accessible by the users. Users are not able to recognize them as computers

IEC 61131 -the standard covers the complete life cycle of PLCs from selecting a PLC to guidelines for implementation of PLC programs [2]. This thesis is focused on using structured text as a PLC programming language and its counterpart of the standard IEC-61131-3. It is a text-based programming language and can be generated from other programs rather easily because of its simple format, limited functionality, and syntax.

Programming with PLCs started with ladder logic programming which used a similar graphical language to electrical drawings [26]. This was widely adopted by many manufacturers with different kinds of nuances. As a result International standard International Electrotechnical Commission 1131-3 (now recognized as IEC 61131-3) has been established and adopted for ladder diagrams (LAD) and other PLC programming languages such as instruction list (IL), sequential function charts (SFC), structured text (ST) and function block diagram (FBD). The standard also defines a library of pre-programmed functions and function blocks [2] which are nowadays usually found in every PLC regardless of manufacturer.

2.1.2 Structured text

Structured text (ST) is one of the standard defined programming languages for PLCs and it can be considered import part of this work. The structured text resembles Pascal-code and also has a similar structure as c-code. Usually, PLC manufacturers follow standards pretty well and code is usually transferrable between manufacturers with little changes using structured text. Even though all other PLC programming languages are just as well supported by manufacturers, interchangeability is not guaranteed as well as with structured text. Most of the IDEs that don’t support IEC-61131-3 based programming languages can sometimes handle structured text with plugins like Visual Code Studio. This is the reason why only this code format is used in this thesis. One short example of structured text is shown in Figure 2.

```
// PLC configuration
CONFIGURATION DefaultCfg
    VAR_GLOBAL
        b_Start_Stop  : BOOL;         // Global variable to represent a boolean.
        b_ON_OFF      : BOOL;         // Global variable to represent a boolean.
        Start_Stop AT %IX0.0:BOOL;    // Digital   input of the PLC (Address 0.0)
        ON_OFF     AT %QX0.0:BOOL;    // Digital output of the PLC (Address 0.0). (Coil)
    END_VAR

    // Schedule the main program to be executed every 20 ms
    TASK Tick(INTERVAL := t#20ms);

    PROGRAM Main WITH Tick : Monitor_Start_Stop;
END_CONFIGURATION

PROGRAM Monitor_Start_Stop          // Actual Program
    VAR_EXTERNAL
        Start_Stop  : BOOL;
        ON_OFF      : BOOL;
    END_VAR
    VAR                             // Temporary variables for logic handling
        ONS_Trig    : BOOL;
        Rising_ONS  : BOOL;
    END_VAR

    // Start of Logic
    // Catch the Rising Edge One Shot of the Start_Stop input
    ONS_Trig    := Start_Stop AND NOT Rising_ONS;

    // Main Logic for Run_Contact -- Toggle ON / Toggle OFF ---
    ON_OFF := (ONS_Trig AND NOT ON_OFF) OR (ON_OFF AND NOT ONS_Trig);

    // Rising One Shot logic
    Rising_ONS := Start_Stop;
END_PROGRAM
```

Figure 2, *Example code using Structured text*

The main difference between programming traditional programs and PLC programs is that the PLC program's main task is always run in an infinite loop and keeps repeating as long as PLC is in operational “RUN” mode.

2.2 Development of PLC control system

Developing embedded systems is well studied and mostly, the same principles and methods can be applied to developing PLC systems. Generally, developing control systems requires multidisciplinary skills and cooperation from many parties with different knowledge domains. This multidisciplinary nature of a PLC control system leads to the following challenges summarized for a PLC control system as follows, adapted from [3]:

- Flexibility: PLC control systems need high adaptability for new environments, requirements, and ease of integration to new services.
- Reliability: PLC systems need to operate under real-time constraints, resource-constrained, and in physically insecure environments.

- Control: PLC systems performing control functions need autonomy, reconfigurability, safety, fault-tolerant, and capability to work under missing data operation.

2.2.1 The traditional development model for control system

Traditionally, most embedded systems are developed using some variant of the classical waterfall model or its extension V-model [4]. The basic waterfall model for software development consists of several steps which follow each other :

- Requirement Analysis & Definition:

  Requirements are a set of functionalities and constraints that are defined by the end product and what is expected from the system. The end-product requirements are gathered from the end-user by consultation and in many cases, these requirements are presented in natural language. These are usually translated to software requirement specifications (SRS) and system requirements.

- System & Software Design:

  This stage plans the actual implementation by analyzing the requirements and creating technical specifications and system and module designs that meet the requirements. This is usually done by a system designer but in smaller projects software engineers.

- Implementation & Unit Testing:

  Using the system and module design from the previous stage, work is divided into the modules and actual programming begins. Each module usually presents one small program which is tested individually. Depending on the source, this contains also the system implementation part where each module is combined to form a system.

- Integration & System Testing:

  In the integration phase, the whole system is integrated and tested. This phase tests the whole system implementation per requirements and after successful testing, implementation is complete and ready to be delivered. It’s usually normal that this phase has failed at some level, and a complicated guessing game must be started on the reason. And usually, testing is done separately from actual implementation so the issue could be within the tests themselves.

- Operations & Maintenance:

This phase is a continuous phase where problems are corrected after initial delivery. Most likely all problems are not found in the previous stage so this phase could continue for a very long time and usually ends when the product is considered “end-of-life”

2.2.2 V-model

V-model is similar to the waterfall model and can be thought of as an extension of it. Its also known as the “Verification and Validation” model. The difference to the traditional waterfall model is that instead of moving linearly down a “waterfall”, the waterfall is bent upwards to form V-shape (Figure 3). The arrow in between refers to a verification and validation phase where each part of the down step has a test plan that verifies the correct functionality. The vertical axis represents the time the horizontal axis level of detail from abstract to specific.

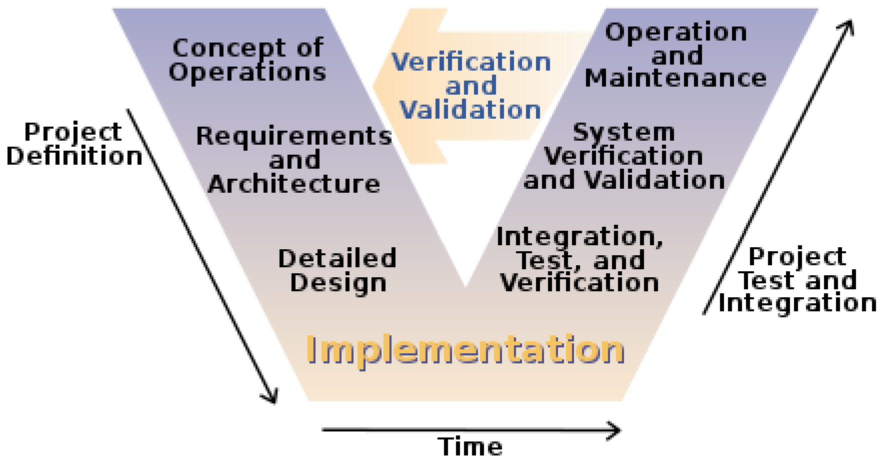
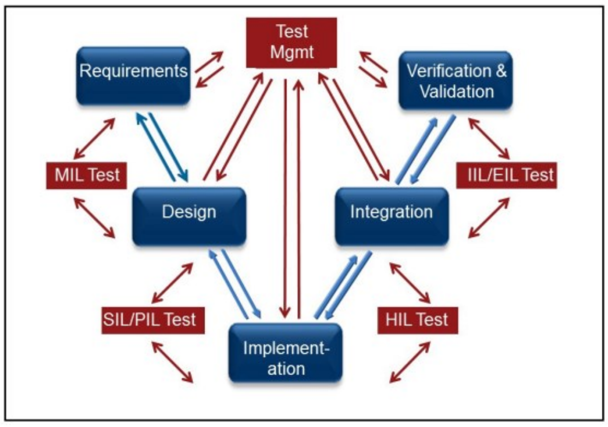

Concept of Operations
Verification and Validation
Operation and Maintenance
Project Definition
Requirements and Architecture
System Verification and Validation
Detailed Design
Integration, Test, and Verification
Project Test and Integration
Implementation
Time

Figure 3: *Generic V-model example [5, p.20, Fig 5]*

This presented V-model is a generic one and usually, implementations of this vary depending on the project. Figure 4 shows an example of how model-based design adopts to V-model.



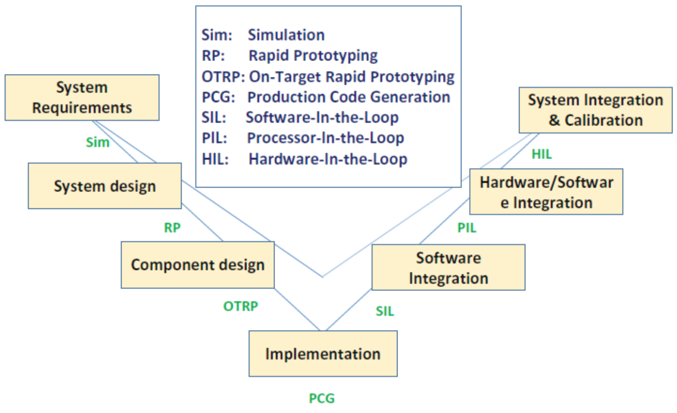

Sim: Simulation
RP: Rapid Prototyping
OTRP: On-Target Rapid Prototyping
PCG: Production Code Generation
SIL: Software-In-the-Loop
PIL: Processor-In-the-Loop
HIL: Hardware-In-the-Loop
System Requirements
Sim
System design
RP
Component design
OTRP
Implementation
PCG
SIL
Software Integration
PIL
Hardware/Software Integration
HIL
System Integration & Calibration

Figure 4: *V-model with MBD including verification, two different approaches. Top figure from V. Socci [6, p.2, Fig 1] and the figure below from Naijun Zhan et al [4, p.8, Fig 1.3]*

Model-based design is introduced further in the next subchapter but as can be seen, from Figure 4, different implementations can be used using V-model as long as the ideology of verification remains in each step.

2.3 Traditional software development drawbacks

Because of multidisciplinary development for control systems, there are multiple boundaries between different teams. Although every team has the same goal that the product works as wanted, these translations through boundaries may lead to errors in the final product [4]:

- As product requirements are created usually by a mix of engineering disciplines, these are transferred to natural language. These are hard to manage and are usually interpreted incorrectly.

- System engineer creates system- and module design. These are written using natural language which has the same interpretation problems and testing the fully integrated design cannot be done at this stage.
- In the implementation phase, the software engineer programs by specifications done by the system engineer. This phase is error-prone as there are interpretation errors and usually the module/system design does not directly transfer to the actual code.

The main problem here is that there is no straight path to validate and redefine requirements in early-stage if these are seen unfit for the system. The same issue occurs again and again as we go through the chain. Another problem is that there is a disconnect from the design phase to the testing phase as implementation usually provides no tracing method back to requirements [4].

In comparison to software development for computer applications, embedded controllers have usually more resilient requirements, and requirement changes can be harder to implement and validate and require much more specialized knowledge about the system. This problem is emphasized in the traditional development model.

2.4 Model-based design

To overcome the challenges mentioned in the previous chapter, a model-based design was proposed. Model-based design, referred to as MBD, uses visual and mathematical models in the development process and these models can be thought of as more generally understood without platform/code specific knowledge. The model-based design paradigm differs a lot from traditional design methodology because instead of using complex application/component/manufacturer specific coding to fulfill requirements, more general functional blocks can be used to describe needed functionality. Using the model-based design, the system model is the center of the development process, and the model is developed in the first stage of development and allows for early detection of system errors. Every discipline can use the same easily understandable model to test/validate changes and requirements to a controller with a simulated “plant” model. The model-based design addresses the problems related to the whole software cycle from the requirement to the maintenance phase by unifying and visualizing the development process and allowing traceability between each step. “Experiences from real-world industry indicate that the efficiency of code generation by tools based on MBD can be improved up to 50% compared to hand-written code in the conventional manner” [4].

As mentioned before, designing a control system requires a holistic approach that requires integration from multiple disciplines. Because most of the embedded systems are somehow connected to the real world and constraints are real-world-based, mathematic concepts are essential in developing control systems. Model-based development used in control engineering is founded on these mathematic concepts.

Model-based design workflow differs from traditional workflow because requirement analysis and high-level design can be quickly designed with simple models and used as a base for further development [7]. When code generation is available, also the actual implementation can be done automatically. As for now, MATLAB/Simulink supports code generations for multiple languages.

Simple model-based design process:

- Define the system
- Identify system components
- Model the system with equations
- Build the model
- Run the simulation
- Verify the simulation results

In a summary, the following benefits can be acquired using MBD and code generation [4]:

- Requirements can be modeled, and design can be tested against these requirements early on.
- The model plays an essential role and can reduce miscommunication between different teams.
- Fast evaluation of different designs.
- Automatic code generation can improve system reliability by reducing variability in code quality.
- Traceability between implementation and code.

There are some drawbacks to MBD, like increased complexity of the development process and using MBD as a blanket solution for all problems. This could lead to unnecessary development work without any benefits.

2.5 Verification and validation

Re-engineering will require some method to ensure that implemented software is running as the original software. Designing complex software systems leads to errors and errors should be found as early as possible in the development process. Verification and validation are processes that try to find errors in development before the actual development is completed. “Verification and validation are independent procedures that are used together for checking that a product, service, or system meets requirements and specifications and that it fulfills its intended purpose.”[8]. Although these terms seem to be used as synonyms, there is a clear boundary between them.

Validation: “The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.” [9]

Verification: “The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase” [9]

Traditional software development and MBD define verification in software development differently: Singh [2011], [8] and other sources define verification as a pure document review process. In contrast to this, Mathworks has defined a verification process to include an actual code testing process as shown in Figure 5, Verification and validation process. The differences stem from the different development models. In the traditional model, it is impossible to verify design early in the development phase.

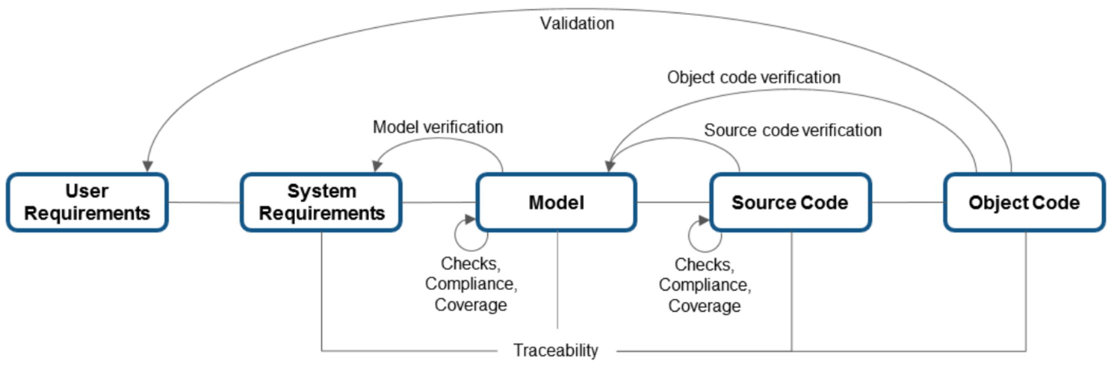


Figure 5, *Verification and validation process for a generic programming language, includes separate verification for object code and source code [adapted from 10]*

In traditional software development, a verification process is done only when translating software requirements to software specifications and validation is only done when the whole software is complete. A more detailed explanation for verification is presented by

Lettinin & Winterholer [11]: verification is divided into dynamic and static verification. The main difference is that static verification is done without running the software and dynamic vice versa, and as Abran & Moore defined "Software testing consists of the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain." [12]

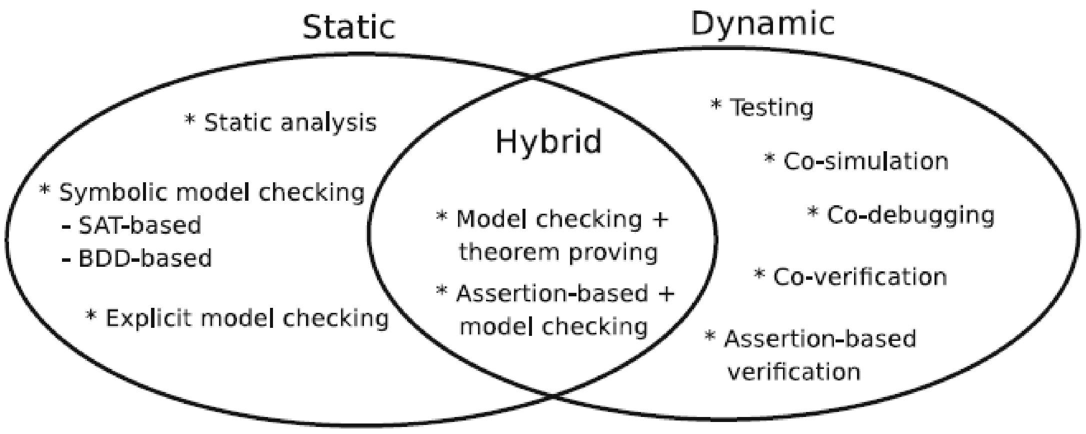

Static
Dynamic
Hybrid
* Static analysis
* Symbolic model checking
- SAT-based
- BDD-based
* Explicit model checking
* Model checking + theorem proving
* Assertion-based + model checking
* Testing
* Co-simulation
* Co-debugging
* Co-verification
* Assertion-based verification

Figure 6, *Verification approaches [11, p.10, Fig 1.5]*

As for the scope of this thesis, only dynamic verification methods are studied. Following verification methods are found from embedded software/hardware development and all of these verification methods are not directly applicable for PLC software development but are still found useful in case-study.

Dynamic verification methods are shown in Figure 6:

- Testing is a traditional empirical method to execute software to find design errors. This includes writing test cases and scripts to test different scenarios.
- Co-simulation means using different sub-simulations all in one model with synchronization. Different simulations are coupled with input/outputs at intervals.
- Co-debugging and co-verification mean debugging/verifying multiple systems at the same time. As for embedded systems, this means debugging/verifying firmware and hardware at the same time. This is not applicable on the PLC application side as these are handled by device manufacturers directly.
- Assertion-based verification means improving observability of the original design by injecting internal temporal properties to design and monitoring these for intended functionality during simulation.

Software testing is often used as synonym to validation and software testing is done to minimize failures in the final products. According to Yogesh Singh [13], testing as a whole can be thought of as verification and validation combined. The most essential part of software testing related to this thesis is determining the process of how the verification/validation is executed at the component level. Testing with MBD differs from traditional software testing which is presented in Figure 7. From the figure can be seen, that traditional testing only tests implementation in the "up-ward" part of the V-model and only static reviews are done in the most essential part of software development.

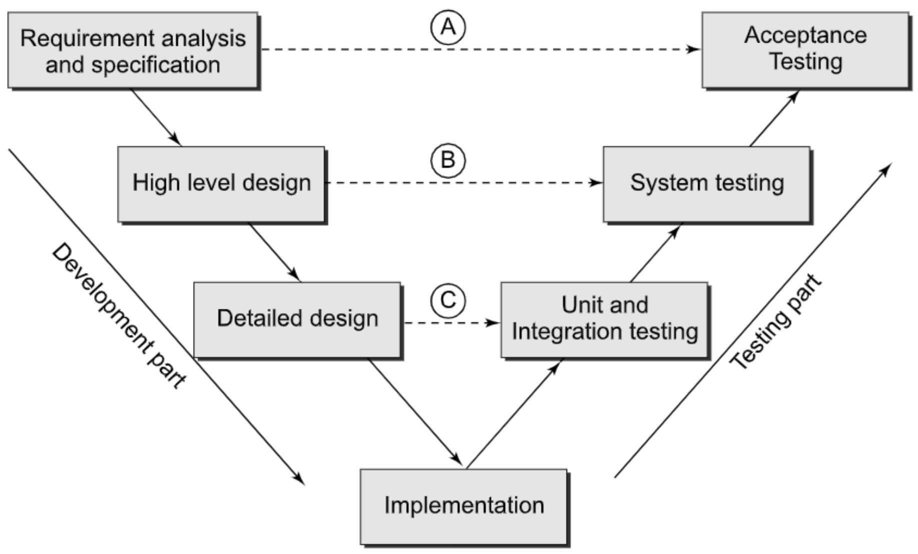


Figure 7, *Testing within V-model in standard software development**.** [13, p.27, Fig 1.10]*

Basic terminology used in software testing and this thesis:

- Bug: Informal name of defects.
- Error: Refers to the difference between actual output and expected output.
- Fault: a condition that causes the software to fail to perform its required function
- Failure: Failure is an instance of fault. Testing is usually done to find faults before they lead to failures.
- Test: a Test is an act of subjecting software to test cases. A test has two goals, find failures or demonstrate correct reactions.
- Test Case: A Set of defined inputs and expected outputs. Easy to do for static stateless systems, but complex for anything else.

- Test object or SuT: The individual element to be tested. Knowns also as "system-under-stress" There usually is one test object and many test items.

Although testing is a large subject on its own and could be explored in several ways, handling this subject is limited to how to define test cases. In a stateless system this is mostly easy: Create a set of test cases against which software is tested. Test cases are usually derived from product requirements and contain inputs from the software interface and some expected results from the software. The result portion is usually the part that is neglected because, depending on the test case level, the actual results could be hard to interpret. In this part usually domain-specific expertise is required. Changing the function from stateless to stateful increases testing complexity by manifold. This problem is usually called "State-space-explosion" which means, in simple terms, adding one state to a system increases all possible states exponentially, which causes problems when defining test cases. This is the normal case with testing as several modules interact with each other and each contributes to this "State-space-explosion". There are several research studies conducted to address this problem and it is not further studied in this thesis. In our simple case study, modules are simple and do not have internal states which could affect testing complexity.

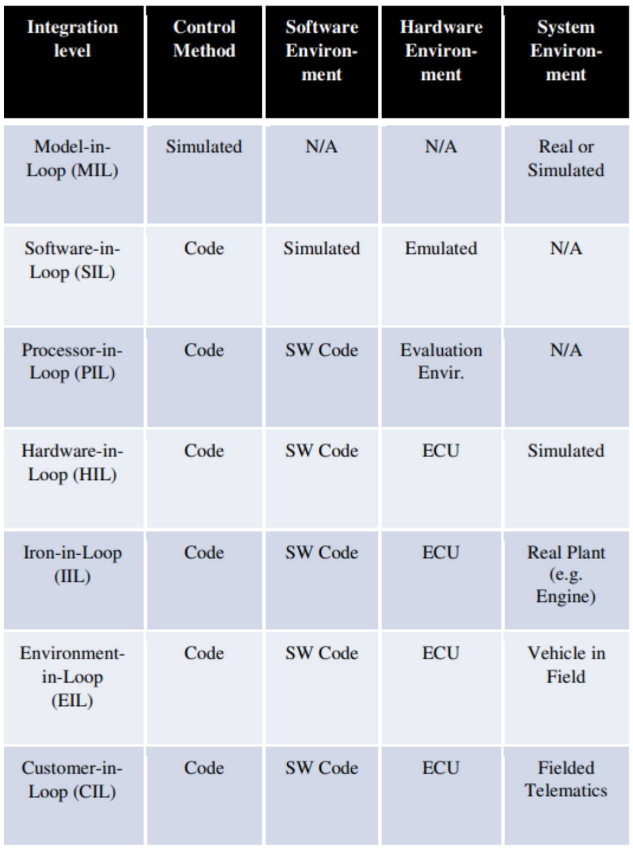
In control design, where usually dynamic properties are more interesting than static response, test case definition, test data generation, and analysing the actual results are harder than in combinatorial logic. That is why in-loop testing is mostly used when testing these kinds of functions. PLC software is usually a mix of both kinds of control problems (combinatorial and complex mathematical algorithms), and that is why both testing methods are explored.

The following test methods are interesting in the scope of this thesis: Specification-based testing, Code-based testing. In-loop testing is also presented, although is not directly a testing method, but rather an environment where tests are performed. In this thesis, all tests are performed and analysed manually so only an overview of these testing methods is provided and automatic testing is not covered.

Specification-based testing is used to test combinatorial logic. Specification-based testing is also known as "functional testing" or "black-box-testing". This relies on the notion that to test software throughout, nothing is known from the actual software being tested and it presents itself as a "black box". Only the expected behavior is needed to test implementation. There are benefits to this kind of approach. Test cases are independent of the software that is implemented and changing the software doesn't render test cases useless. Test cases can be also be made even if the implementation is not ready.

Code-based testing, in contrast to specification-based testing, code-based testing needs full access to the software being tested. This is also called white-box testing. This is beneficial because a person who generates the test cases can also take into account those scenarios which are implemented but not specified. This could happen for example when specifications do not cover every possible situation or specifications are too generally written.

In-loop testing refers to a testing methodology specific for complex real-time controllers whereas loop parts refer to feedback loop from a simulated environment model (sometimes called plant-model). Traditional software testing is not sufficient in PLC testing cause PLCs are connected to actual physical objects and must abide by some fundamental natural laws that are not arbitrary. This means that PLC software is usually tested with actual hardware and real or simulated environments which is summarized in Figure 8: Different integration levels [6, p.2, Table II]. Depending on the situation, black-box testing or white-box testing can be applied. Generally, black-box testing is usually applied in the in-loop environment in later stages (HIL, IIL, EIL) and white-box in earlier parts (MIL, SIL, PIL). This means that testing a program with knowledge of internal structures is beneficial in the early stage where the developer is usually involved. In later stages, black-box testing could be executed by the test engineer.



Integration level	Control Method	Software Environment	Hardware Environment	System Environment
Model-in-Loop (MIL)	Simulated	N/A	N/A	Real or Simulated
Software-in-Loop (SIL)	Code	Simulated	Emulated	N/A
Processor-in-Loop (PIL)	Code	SW Code	Evaluation Envir.	N/A
Hardware-in-Loop (HIL)	Code	SW Code	ECU	Simulated
Iron-in-Loop (IIL)	Code	SW Code	ECU	Real Plant (e.g. Engine)
Environment-in-Loop (EIL)	Code	SW Code	ECU	Vehicle in Field
Customer-in-Loop (CIL)	Code	SW Code	ECU	Fielded Telematics

Figure 8: *Different integration levels [6, p.2, Table II]*

Model-in-the-loop (MIL) testing is usually done early in the development loop to ensure that the model meets the requirements. In controller design, this means that the controller is modeled and tested against a controllable model (plant model) without any actual hardware components.

In Software-in-the-loop and Processor-in-the-loop (SIL/PIL) testing, code is generated for the controller, but code is run without actual hardware. In a PLC case, this means SIL is forgotten since running PLC code requires virtualized PLC or softPLC and cannot be run straight from modeling software. This testing method means that the hard real-time limits may not be tested but the code can be verified functionally against the plant model. According to V.Socci, shown in Figure 8, SIL/PIL testing doesn’t involve a system environment, but this is contradicted in many sources and the actual definition of “in-loop” refers to testing against some system environment.

Hardware-in-the-loop testing involves actual hardware in the loop where controller inputs/outputs can be from the actual environment or simulated one (plant model). This testing phase verifies that the actual hardware is performing adequately in real-time.

Iron-in-loop, Environment-in-loop (EIL), and Customer-in-loop (CIL) refer to the actual tests done with real environments and hardware. These are the most expensive tests to run and failures could cause even unsafe situations. That is why these tests are usually done after all other tests are passed.

As shown in Figure 4, the previous V-model section, there are differences in how these in-loop tests could be executed. At the bottom of the figure, rapid prototyping is introduced to the V-model and it’s suited for embedded controller design. As rapid prototyping is not used or in the scope of this thesis, the bottom model in the figure is chosen as a baseline for creating the re-engineering model.

2.6 Re-engineering

Software development life cycle (SDLC) is a well-known overview model for how a software product is made. This model usually describes 5 basic steps of software development: planning, analysis, design, implementation, and maintenance. Re-engineering is part of the most expensive[14] maintenance phase where products lifetime is extended by making necessary changes to adjust to new requirements. The required change is usually aimed at correcting software, adding functionality, transferring to new environment/hardware, or just restructuring. Although re-engineering is necessary to increase software service life, it is still an overlooked phase of the product life cycle as it is not nearly as researched as other phases of development.

Software re-engineering is a process where software is redone without affecting its functionality. The re-engineering process usually involves organizing and restructuring, re-documenting, and recoding system using modern programming languages [15]. Software re-engineering is usually required when the programming language or platform comes outdated, the architecture of the product is changed thoroughly, or new requirements are not possible to be fulfilled. Rosenberg, L states “The difficulty lies in the understanding of the existing system. Usually, requirements, design, and code documentation is no longer available or is very out of date, so it is unclear what functions are to be moved. Often the system contains functions that are no longer needed, and those should not be moved to the new system”.

There are four general objectives in re-engineering and at least one of the following is the main motivation for re-engineering activity [15]:

- Improving maintainability
- Migration
- Improving reliability
- Preparation for functional enhancements.

In the general model, presented in Figure 9, re-engineering starts with the implementation of the current system and ends with a new implementation of the system. This can be a simple process if the only driver for re-engineering is to adopt for new coding language. But, when the system needs re-organizing at the same time or for some other reason it is not enough to just translate the language, the process can be very complex. This means that old requirements and specifications must be carefully examined and studied and the whole implementation needs to be redesigned-

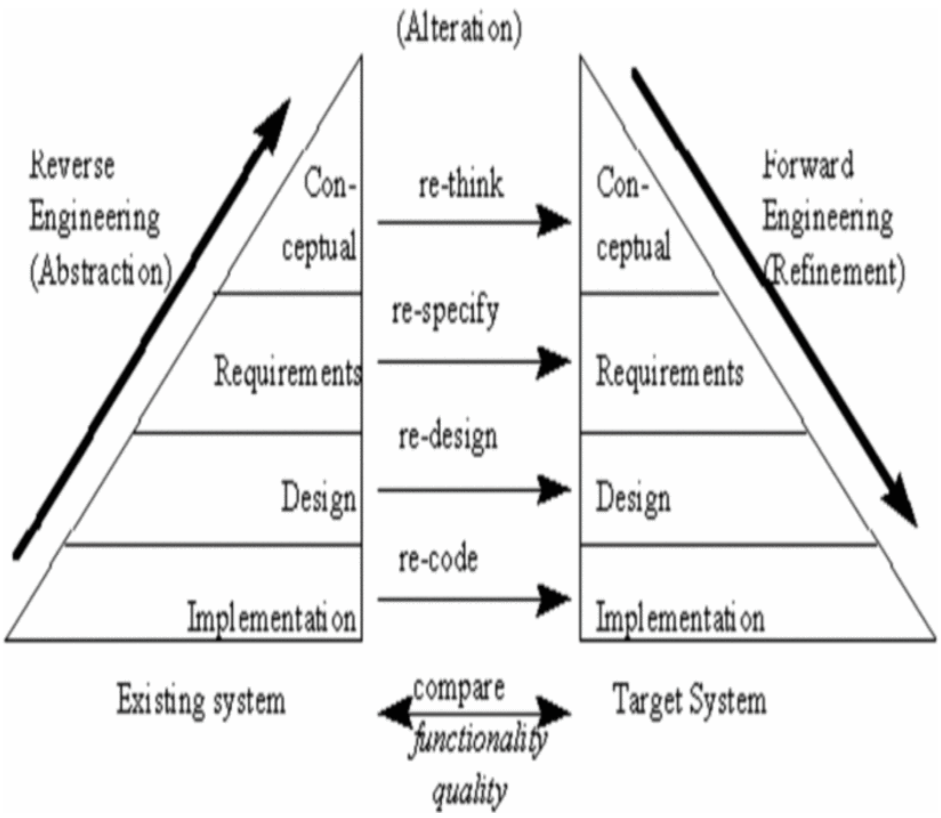


Figure 9: *The general model of re-engineering software [15, p.4, Fig 1]*

The general model In Figure 9, the re-engineering model is split into two major parts, reverse engineering, and forward engineering. Reverse engineering is described in detail in the following subchapter.

There are different approaches to facilitating the re-engineering process. Usually, there are some limitations on how re-engineering can be applied for specific software, for example, the structure of the program or the time limit for the re-engineering process. These different approaches are determined in the following list, each approach has its benefits and drawbacks [15].

1. Lump-Sum method: This approach replaces the whole system at once. This has the advantage of being straightforward, and each interface can be rewritten. But this is not applicable for a large system, not only for the recourses it takes to complete in a reasonable time but also usually new requirements must be in place before this re-engineering is complete so two systems must be maintained at the same time.
2. Incremental method: This approach replaces the original system part by part and more specifically component by component. There is a lower risk of failing with this approach because only small increments should be made at one step. This approach has the disadvantage of taking a long time before being completed and the old interfaces must be reused or at least supported.

3. Evolutionary method: As in the incremental method, this approach also replaces the system part by part. The difference to the incremental method is that whole individual functionality is replaced which can mean that at least some interfaces must be adjusted. The disadvantage is that their parts can be quite large and old implementation may also need to be adjusted when applying re-engineered parts.

2.7 Reverse engineering

Reverse engineering is a part of the re-engineering process, and the objective of reverse engineering is to evaluate code behaviour to duplicate it. Although the re-engineering process could be started without reverse engineering, the most reliable source of code behaviour is the source code. Reverse engineering is a process starting from source code and translating it back to specifications and requirements. This allows the re-use of the knowledge already implemented in programs, and specifically, the knowledge that has not been documented. Some automatic translation programs have been researched for reverse engineering. One method, which translates PLC program to UML diagram, is presented in “UML based approach for re-engineering PLC programs” [16],

When developing new software, requirement engineering is started when objectives for software are clear and these need to be formalized. The requirement engineering process is mostly overlooked and the main reason for failed projects is insufficient requirement engineering [17]. Requirement engineering is not just one step in software development but a process that overlaps with different software development phases.

In the planning phase, requirements are gathered from clients and stakeholders. In the analysis phase, requirements are checked for conflicts between other requirements. The design phase translates these requirements to specifications which are then used to create needed software. In the testing and validation phase, requirements are compared to actual software functionality. In the management phase of the software development, requirements are maintained as new software requirements are discovered. The maintenance phase is usually overlooked in the requirement process and new requirements are not properly handled in requirement engineering. This causes requirement conflict and misalignment between code and requirements.

Traditionally, requirements are usually neglected in reverse engineering and reverse engineering is handled as a separate process from requirement engineering [18]. Reverse engineering design/code back to requirements is a difficult process and requires human intervention when requirements are presented in natural language. In Figure 10, Syed Ahsan Fahmi and Ho-Jin Choi propose a modified process to take requirements into account in the reverse-engineering process.

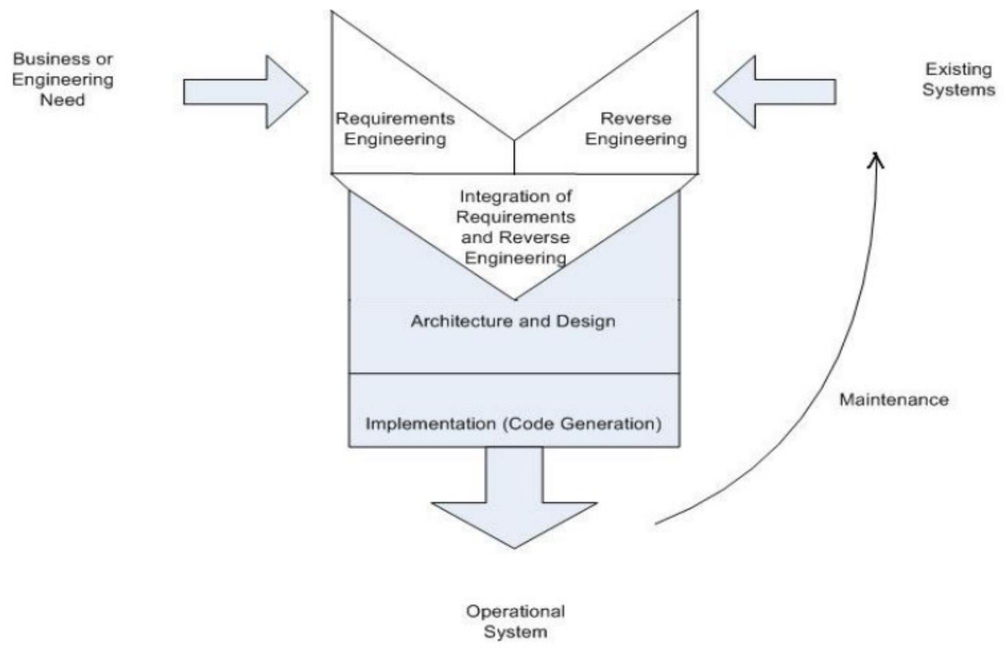

Business or Engineering Need
Requirements Engineering
Reverse Engineering
Existing Systems
Integration of Requirements and Reverse Engineering
Architecture and Design
Implementation (Code Generation)
Maintenance
Operational System

Figure 10: *Modified requirement handling in re-engineering [18, p.5, Fig 5]*

Although this process does not provide any help in creating these requirements, there are several benefits to complying with this:

- The number of new requirements can be found
- Outdated requirements can be traced out
- Original requirements can be revised

3. RE-ENGINEERING MODEL SYNTHESIS

In this chapter, all the previously introduced topics are analysed and the first two thesis questions are answered: 1. How to describe model-based re-engineering process for PLC controller software. 2. What are the benefits of re-engineering with the proposed process model.

Although there are several more modern development models in addition to the V-model that was introduced, V-model is used as a base for this thesis. V-model is still a frequently used process model for software development and even in more modern models like example agile, the underlying best practices are still founded on the v-model. Most of the agile methods have sprung from the v-model and both of these models have similar properties as stated in the presentation “Agile V-model” [19]. And from the practical point of view, the used development model is most likely mixed from many models and modified as seen fit.

The proposed model, in Figure 11, is created by combining different aspects in re-engineering, model-based design, and V-model. This model is a combination of different models introduced in the previous chapter and has the following properties:

- The model extends V-model
- The model can account for several different re-engineering problems meaning the model is generic.
- The model includes the re-engineering phase.
- The model conforms to model-based design

The model is read from left to right to timewise and the “V”-shape is now an “N”-shape where the model can be thought to have two validation loops. One validation loop is presented in the re-engineering phase where new implementation is compared to the original implementation and original requirements and one actual validation where new requirements are compared to the new implementation. But for clarity, the first validation loop is called the verification phase as it is still just a step before the actual validation. In theory, the model could be thought like a flipped “V”-shape if a testing part were omitted and only re-engineering considered. But in practice, these kinds of changes, at least in controller development, require new integration and validation testing.

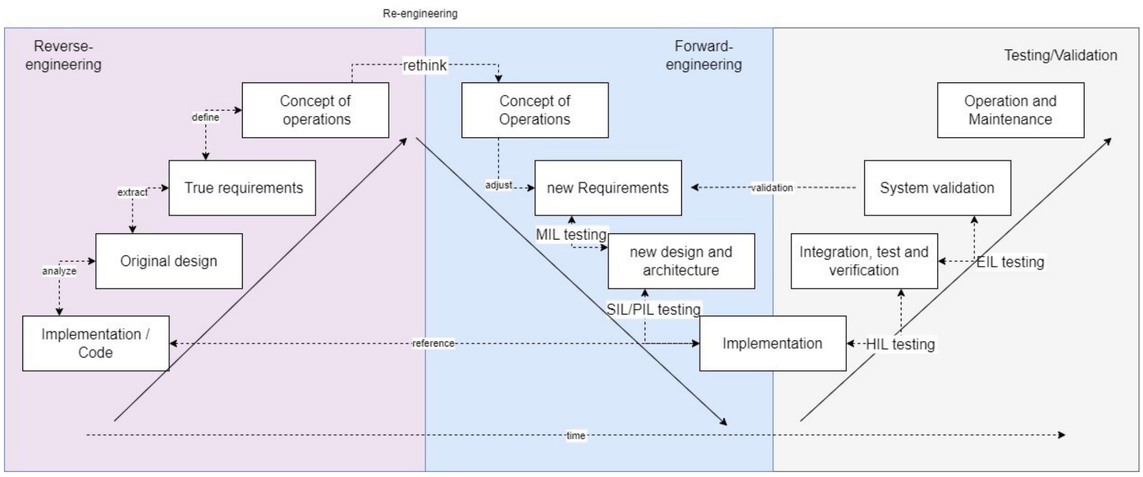


Figure 11: *Proposed model to re-engineer control software with MBD*

A generic model presents all necessary parts in re-engineering old software to model-based design. Starting from the left, the original implementation is analysed for design and then requirements are extracted. These extracted requirements can be used to re-thinking the actual operation and redefine original requirements. After reverse engineering, part of the normal V-model is used as a development process except that the re-engineering artifacts are used as a reference in verification. The model also shows verification steps in each step which indicates that this is specific to the model-based development process.

There are several decisions in implementing the model in practice:

1. What re-engineered method should be used?
2. How to use new requirements and how can these be tested in the modeling phase?
3. How to verify that the new implementation matches the original?

All of these decisions are individual to a specific re-engineering activity, questions need to be answered case by case as the conditions vary in many ways. The model may need to be adjusted in some ways in these individual cases. For example, there could be some difficulties using the original implementation as a reference to the new implementation in SIL/PIL testing as the specifications could have changed. This would require re-thinking i.e could there be another way to verify correct functionality after re-engineering meaning code reviews or other verification methods. And depending on the situation, it may be beneficial to skip some of the parts as the process as a whole is quite demanding.

Benefits of using this process model as re-engineering with model-based design:

- It uses familiar processes for the development
- Holds the same benefits as V-model and model-based design, discussed in the previous chapter.
- Updates the requirements from the original implementation.
- Original implementation can be in any language and hardware, as long as it can be linked to a Simulink model or otherwise verified against the new implementation
- The model is complete in the sense that it links from the original implementation to the new maintenance phase.

This model, like all models, has its drawbacks:

- The process is complex and requires the involvement of different teams. Most likely the benefits from moving to model-based design do not justify these resource requirements alone and the process needs to be adjusted and broken up into smaller and possibly incomplete parts. This problem should be looked into and automating these processes should be considered if possible.
- A second problem with the proposed model is that if the system requirements are modified heavily, the redesigned controller may not be directly comparable to the original controller. Usually in re-engineering as discussed in previous parts, new requirements should not be brought in this stage, but still adjusting the requirements could modify the compatibility of the new controller and original controller to some extent and makes verification harder.
- The third problem relates to using the original controller as a reference in verification. Depending on how the re-engineering is done, this could be impossible or at least require manual adjustments, but this problem can be mitigated with careful planning.

The presented model is partially studied in practice in the next chapter.

4. CASE-STUDY

In this case study, a real functioning controller is used as a reference (in further text parts “reference” refers to this controller) and the same controller is recreated with a model-based design. The controller is currently in a maintenance phase and was provided to this thesis for examination. The purpose of this case study is to experience the actual workflow of the proposed re-engineering model and present detailed information on each step and what kind of tools can be used and what issues are found. As the re-engineering is quite extensive in work effort, It was decided to focus on the re-engineering part (first half of the “N”-model) and specifically to forward engineering.

Case-study starts by introducing the reference controller. Afterward proposed model re-engineering model is applied within limitations. Limitations are mostly defined by chosen functions' simplicity and the need to focus on verification. The re-engineering model is referenced at the start of each chapter to help the reader to understand what part of the model is applied.

4.1 Introduction to the controller under examination

This section gives a small introduction to the controller to be examined. As the whole controller is too complex and large to be studied in the timeframe of this case study, only an overview of the complete controller is introduced to give some background knowledge and a possibility to understand the terms used.

The complete controller is a motion controller for the crane position. The crane itself moves on rails and has three degrees of freedom (Figure 12).

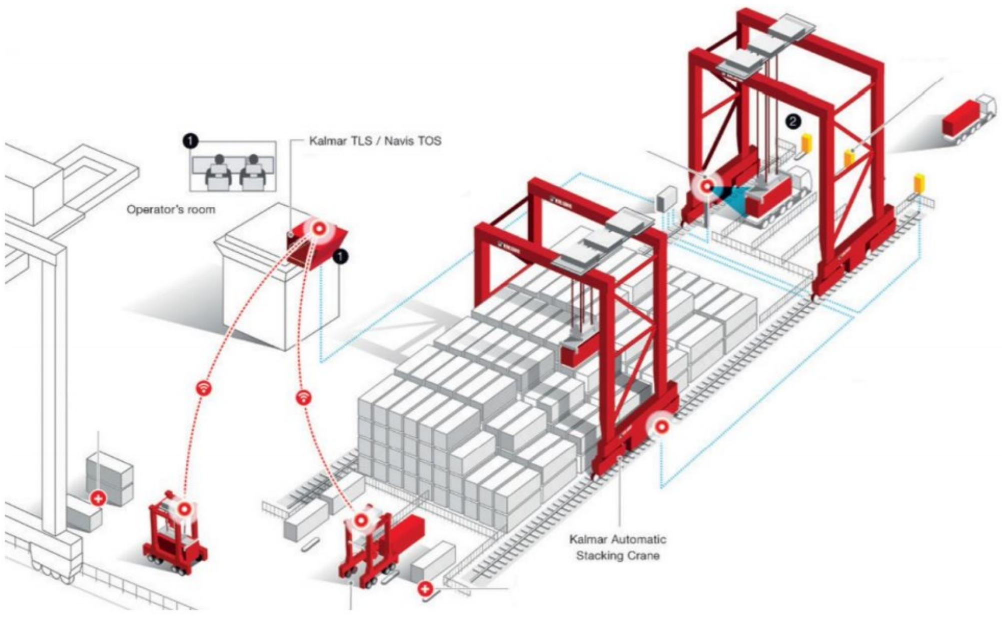


***Figure 12** Automated stacking crane system (from Kalmar website: ASC application infographic)*

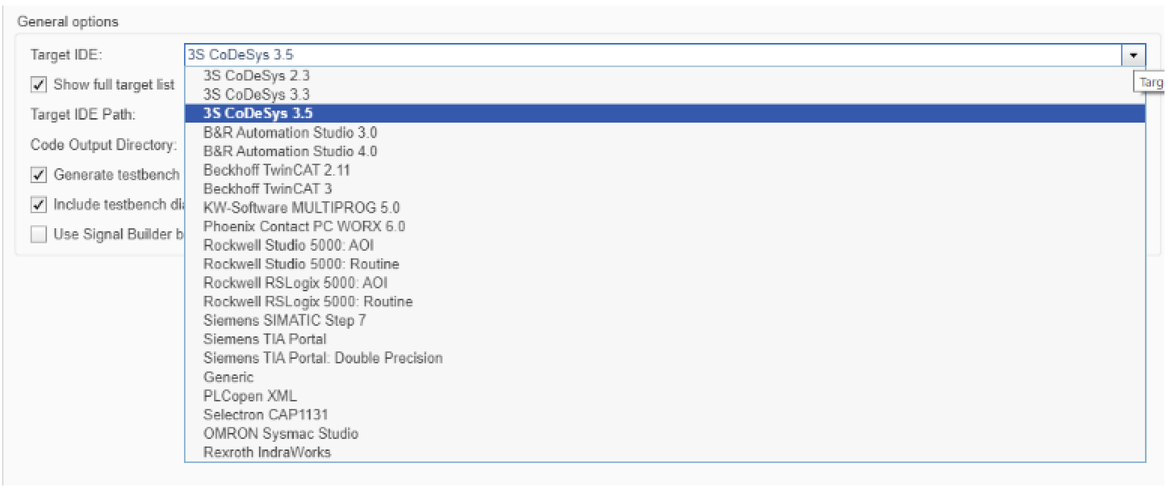
Initially, the goal was to get the complete motion controller part re-engineered but because of time constraints, only part of the controller is examined. These subfunctions were chosen as they present basic functionality types that controllers generally use. The first one is the generic PI-controller which is used in multiple parts of the motion controller. The second one is a simple fault detection function that executes combinatorial logic to detect faults between commanded and executed actions and controls if the command execution can be continued. In this work, it's referred to as StartingOfMotion (SoM) as it is originally named.

In real environments, the program is executed using SoftPLC running Codesys control runtime environment on Industrial PC. Codesys is a runtime system that includes an integrated development environment (IDE) and runtime environment (RTE).

4.1.1 Tools used in case-study

Mathworks MATLAB is a platform for numeric calculations and programming developed by MathWorks. When using Matlab with model-based design, Simulink is a popular toolbox for this kind of work. Simulink is even so popular that it is usually referred to as individual software and this toolbox now has its own separate toolboxes. Simulink is designed for modeling, simulating, and analyzing complex systems. The most important toolbox within Simulink, related to this case study and

MBD in general, is the PLC coder. This allows transforming the model directly to PLC code, within certain limits. The full list of supported PLC IDEs is shown in **Virhe. Viitteen lähdettä ei löytynyt.**. There also exist other code generation toolboxes and multiple other languages are supported, for example, C and C++. One other important toolbox within Simulink is Stateflow which can be used to model reactive


General options

Target IDE: 3S CoDeSys 3.5

☑ Show full target list

Target IDE Path:

Code Output Directory:

☑ Generate testbench

☑ Include testbench di

☐ Use Signal Builder b

3S CoDeSys 2.3
3S CoDeSys 3.3
3S CoDeSys 3.5
B&R Automation Studio 3.0
B&R Automation Studio 4.0
Beckhoff TwinCAT 2.11
Beckhoff TwinCAT 3
KW-Software MULTIPROG 5.0
Phoenix Contact PC WORX 6.0
Rockwell Studio 5000: AOI
Rockwell Studio 5000: Routine
Rockwell RSLogix 5000: AOI
Rockwell RSLogix 5000: Routine
Siemens SIMATIC Step 7
Siemens TIA Portal
Siemens TIA Portal: Double Precision
Generic
PLCopen XML
Selectron CAP1131
OMRON Sysmac Studio
Rexroth IndraWorks

***Figure 13** All supported IDEs for PLC coder*

logic systems using state systems. Matlab R2021b was used in this thesis and at least the PLC coder -toolbox evolves rapidly so some of the things mentioned here could become outdated quickly.

In theory, modelling and simulation software could be anything that meets certain requirements. The modelling software was decided beforehand and not given much thought as Simulink seemed like an obvious choice as one of the requirements was to use generate code for PLC and no alternative could be found.

4.1.2 Workflow for case-study

This section provides an overview of a model implementation used in the case study. The steps follow the proposed re-engineering model but are further detailed here. The original controller is used as a reference in new implementation verification.

In Figure 14, the used workflow is mapped to the proposed re-engineering model.

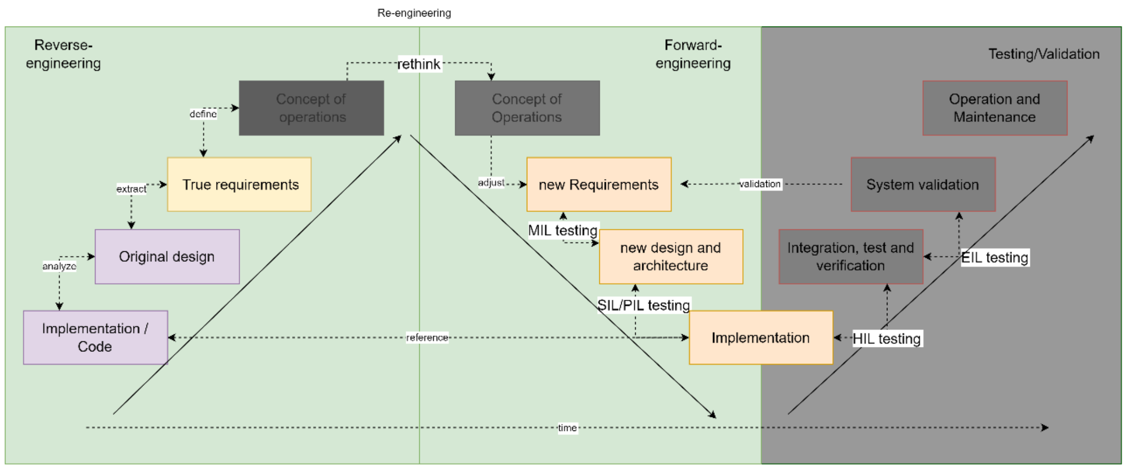


Figure 14, *Workflow mapped to re-engineering model, the dark grey colour representing parts that are not part of this case study. Focus areas are coloured in orange.*

As we can see, the case study does not involve the testing/validation phase at all and reverse engineering is only applied lightly. The case study focuses is in forward engineering and particularly on verification. Light grey colored areas in Figure 14 represent steps that in some sense were done partly when deciding how this case study was to be completed and not further described in this thesis as the reference implementation is Kalmar’s intellectual property.
Re-engineering steps for both controllers:

1. Generate requirements for the controller model. This means first analysing the original controller and extracting requirements.
2. Decide and create testing procedures for verification.
3. Create a model for a controller using Simulink / Stateflow
4. Transfer Model to actual PLC (In this case soft-PLC)
5. Verify generated code against a model.
6. Verify functionality by comparing reference and new controllers.

4.2 Requirement analysis

This part is related to both reverse engineering and forward engineering as shown in Figure 15

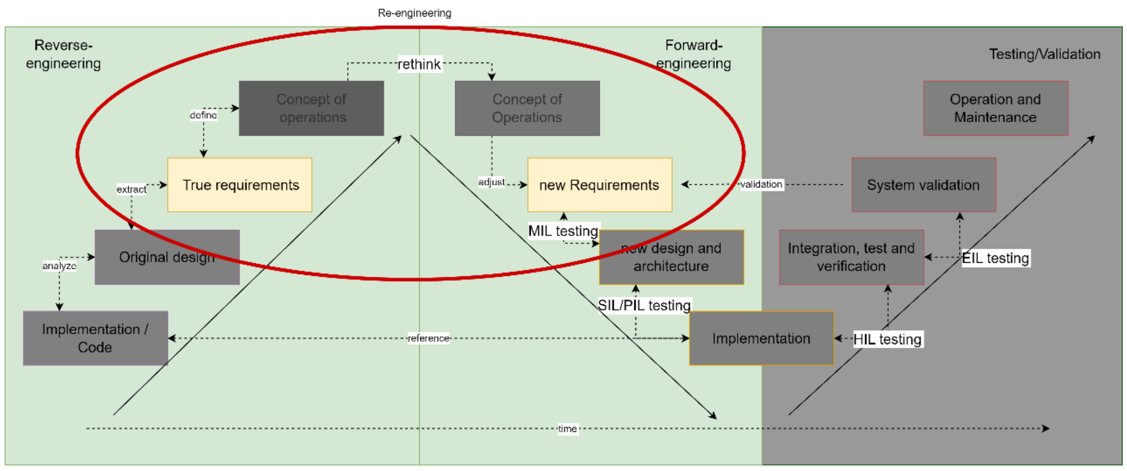


Figure 15, *Requirement analysis as part of re-engineering model*

In large software, each module or system represents some functionality and gives some specific requirements which can be traced back [17]. When talking about modules and functions, modules usually represent a larger part of a code and may contain many functions and each function's contribution may not be easily identified to generate any specific requirement. Because this case study only involves a small part of the whole motion control system, each property of these functions is handled as a requirement instead of a specification which would be an appropriate description.

Requirement analysis in reverse engineering is not as studied topic as other topics related to reverse engineering [18]. Methods used in this thesis to find out software requirements can be explained in these steps:

1. Analyse the current system with an outside (black box) perspective, meaning testing the function to see the outcome or just by analysing the interface.

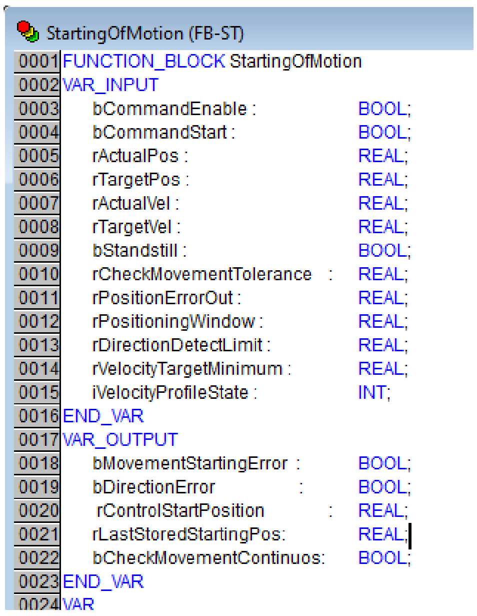
2. Analyse subsystems through the code for individual requirements (white-box). This usually helps to understand non-functional requirements.

3. Combine requirements between steps 1 and 2 and check for conflicts.

Because this re-engineering happens incrementally and is required to be compliant with the old implementation, the controller/module interface must be part of the requirement/specification. These derived requirements are used in the next step in modeling and creating test cases.

4.2.1 Analysis for StartingOfMotion-function.

An analysis is started by using the function interface as the starting point and continuing through the code line by line to see the differences between actual implementation and supposed requirements. The actual interface is shown in Figure 16. This sets the first requirement/specification as it needs to remain untouched for the verification part.



```
FUNCTION_BLOCK StartingOfMotion
VAR_INPUT
    bCommandEnable :              BOOL;
    bCommandStart :               BOOL;
    rActualPos :                  REAL;
    rTargetPos :                  REAL;
    rActualVel :                  REAL;
    rTargetVel :                  REAL;
    bStandstill :                 BOOL;
    rCheckMovementTolerance   :   REAL;
    rPositionErrorOut :           REAL;
    rPositioningWindow :          REAL;
    rDirectionDetectLimit :       REAL;
    rVelocityTargetMinimum :      REAL;
    iVelocityProfileState :       INT;
END_VAR
VAR_OUTPUT
    bMovementStartingError :      BOOL;
    bDirectionError           :   BOOL;
     rControlStartPosition    :   REAL;
    rLastStoredStartingPos:       REAL;
    bCheckMovementContinuos:      BOOL;
END_VAR
VAR
```

***Figure 16,** SoM interface*

As a “black box” analysis, from the interface perspective, three preliminary requirements can be derived.

1. The software has detected if a movement has not started after being commanded and generates a fault.

2. The movement has not started in the right direction in a configurable window after being commanded and the fault is generated.

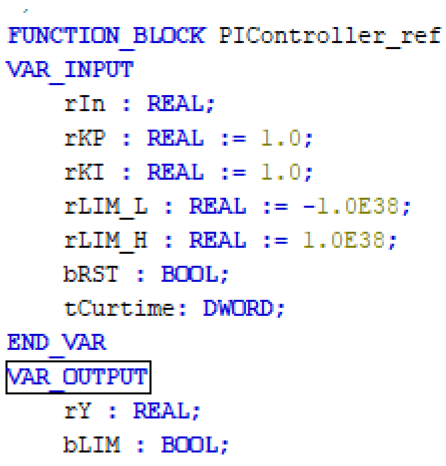
3. The last target and control target position must be stored when a new command is received.

These requirements can be seen linked to movement speed, position, commands, and parameters. Requirements should be kept simple and not defined code in detail.

In the white box analysis, requirements can be clarified by analysing the code in detail. Starting the movement should be checked only after a configurable minimum distance has been reached and monitoring conditions are set. Conditions for monitoring movement start: Automatic movement and job execution are enabled, movement direction is right, and traveled distance is large enough. The error should be generated if the movement hasn’t started in a configurable time. If the movement has started, the correct direction should be verified separately. Direction is checked once when the velocity command is given and the distance to be traveled is long enough. Direction error is set if the absolute moved distance exceeds a configurable limit and the detected movement is in the wrong direction.

4.2.2 Analysis for PI-controller

PI-controller is a generic controller with anti-windup. The interface for this controller can be seen in Figure 17. One requirement can be seen from the interface. The Control signal value is limited by parameters and if the control signal is saturated, this is expressed in the interface with a separate signal. Requirements are not further analysed for this controller because these are more qualitative and not in the scope of this thesis.



```
FUNCTION_BLOCK PIController_ref
VAR_INPUT
    rIn : REAL;
    rKP : REAL := 1.0;
    rKI : REAL := 1.0;
    rLIM_L : REAL := -1.0E38;
    rLIM_H : REAL := 1.0E38;
    bRST : BOOL;
    tCurtime: DWORD;
END_VAR
VAR_OUTPUT
    rY : REAL;
    bLIM : BOOL;
```

Figure 17, *PI-controller interface*

4.3 Modeling controller with Simulink

The purpose of this section is to provide information on how the modelling was done and offer a small guide. Modelling part is part of “new design and architecture” as shown in Figure 18.

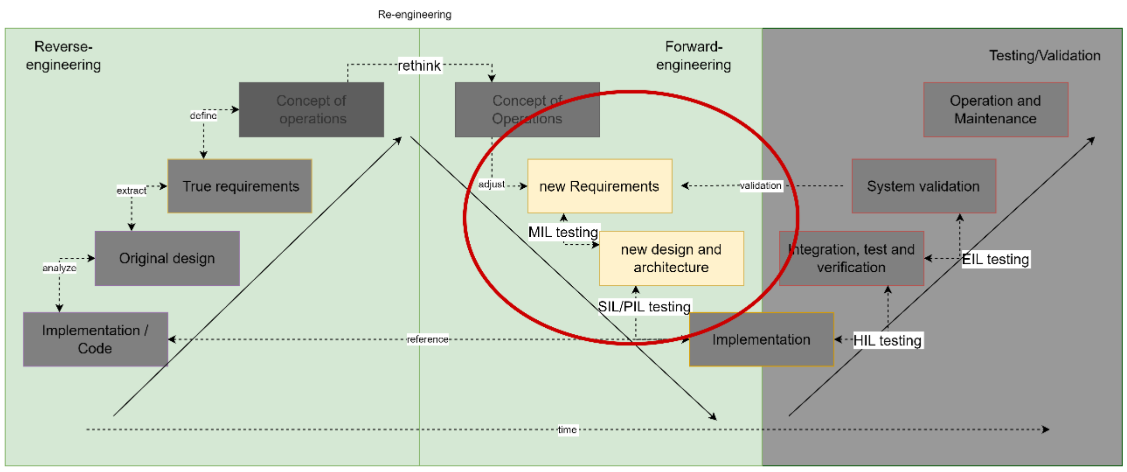


Figure 18, *Modelling part of the re-engineering model*

In the scope of this thesis, the basic principles of modelling with Simulink are not described and the reader is expected to have basic knowledge of these. Actual modelling is done differently for two controllers: SoM is modelled using basic building blocks found on Simulink and PI-controller as Matlab System-block. Matlab System-block allows implementing functions as Matlab scripts. This decision was to detect limitations of the PLC coder and as PI-controller is a mathematical algorithm, it's more straightforward to implement. This option was not available at least in the previous version of the PLC coder.

4.3.1 Interfaces

The modelling process starts from the interfaces because interfaces need to remain unchanged to allow generated code to be compatible with the original software. Creating interfaces to the Simulink model is done by bus creator [20]. Bus creator allows the creation of a bus object which has predefined signals as inputs. Bus object combines signals graphically which makes reading diagrams more readable. Using bus creator also enables generating interface programmatically. The bus can be either virtual or non-virtual. The virtual bus is used just to make diagrams clearer and the non-virtual

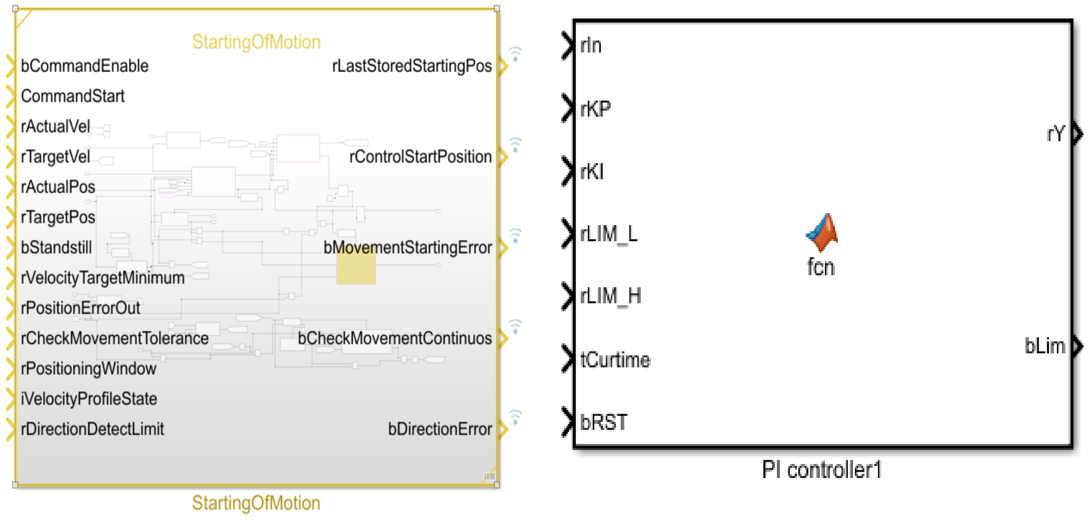


Figure 19, Interfaces in Simulink-model. Differences in visual presentation is because PI-controller is System-block and SoM is standard block

packs the signals to a single structured data type with only one pointer. This applies to generated code, meaning non-virtual bus will generate a struct-datatype which is essential in re-engineering. At least when the original interface contains signals structs and needs to remain unchanged. Although interfaces were not implemented to either model with busses, it is the recommended way of doing this and initially, these were used. Busses were removed in the testing part as they made testing unnecessarily more complex. Figure 19 shows the final implemented interfaces.

These are almost identical to the original software to help with the testing part. One difference was made regarding the simulation time, and this was probably an unnecessary change if other paths were thought out. In the original PI-controller, cycle time was read directly inside the function but in the model, it is brought through the interface. In general, copying the interfaces could be considered a good practice when implementing the incremental method in re-engineering. One of the most important parts of copying these interfaces is the implemented datatype. This is not usually crucial in the modelling phase, but correct datatypes should be checked from documentation or actual code generation that Simulink datatypes are translated properly to the target. An example of this datatype translation is given in Table 1

***Table 1**, Simulink - Codesys datatype transformation*

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

If datatypes are not defined for the model, Simulink will use automatically the “largest” datatype, double which translates to “LREAL” in Codesys. This datatype checking should be also done inside the model to avoid unnecessary memory usage.

4.3.2 Created model based on requirements and reference design

The next step in the modelling is to define the functions needed to create the actual model. PLC coder support referenced subsystem, and this was used to create models. Subsystem referencing saves subsystem as separate .slx file which can be referenced in models. This allows multiple advantages including using the same subsystem in multiple models and easy multiuser modifications e.g. modifications are reflected in every model/user project. As for SoM-modeling, most of the modelling copies the original design and it does not fully follow the defined proposed re-engineering model as the model should be fully recreated based on the requirements. But this decision was just to find out limitations in Simulink modelling, and there is nothing wrong with just copying the design if it is properly done already. This decision did affect the readability of the Simulink model, shown in Figure 20, and more consideration should be done when transferring to visual representation.

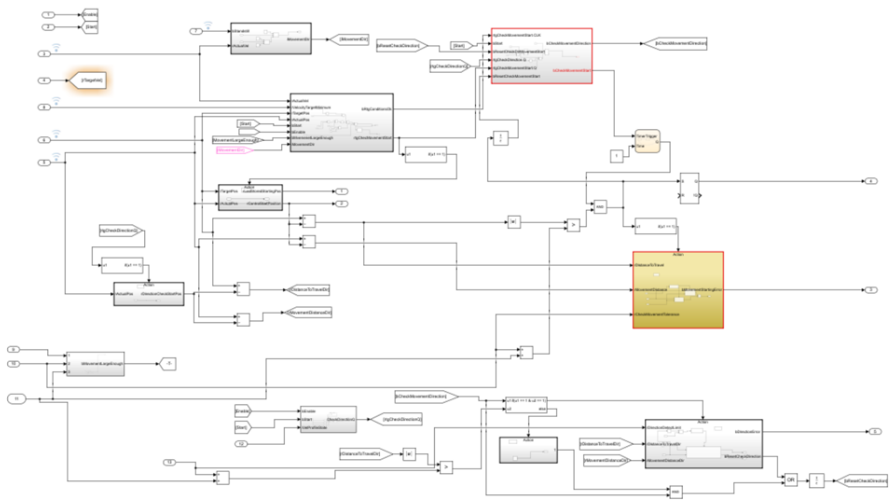
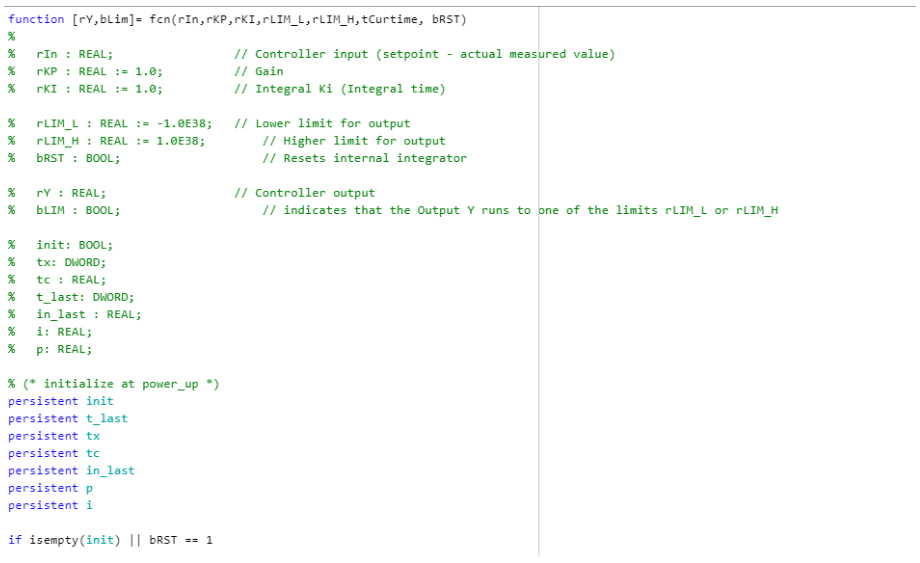


Figure 20, *Created SoM-function simulink model, full size image as appendix 4*

This kind of “signal-routing-mess” is not uncommon when subfunctions are not identified correctly and this could be cleared properly using in/out-ports but resolving this was not necessary to create a working model.

PI-controller modelling was straightforward as this is just an algorithm. Original code was translated to Matlab-script and used as a System-block in simulation. This model was extended by creating an arbitrary plant model that could be used for verification.



```
function [rY,bLim]= fcn(rIn,rKP,rKI,rLIM_L,rLIM_H,tCurtime, bRST)
%
%   rIn : REAL;                // Controller input (setpoint - actual measured value)
%   rKP : REAL := 1.0;         // Gain
%   rKI : REAL := 1.0;         // Integral Ki (Integral time)

%   rLIM_L : REAL := -1.0E38;  // Lower limit for output
%   rLIM_H : REAL := 1.0E38;       // Higher limit for output
%   bRST : BOOL;                   // Resets internal integrator

%   rY : REAL;                 // Controller output
%   bLIM : BOOL;                   // indicates that the Output Y runs to one of the limits rLIM_L or rLIM_H

%   init: BOOL;
%   tx: DWORD;
%   tc : REAL;
%   t_last: DWORD;
%   in_last : REAL;
%   i: REAL;
%   p: REAL;

% (* initialize at power_up *)
persistent init
persistent t_last
persistent tx
persistent tc
persistent in_last
persistent p
persistent i

if isempty(init) || bRST == 1
```

Figure 21, *Part of the matlab script that impelements new PI-controller*

A generic imaginary mass-spring-dampener system was chosen randomly to represent the plant. The actual function of the plant model is not important in the scope of this thesis. Figure 22 shows the extended model with the plant model in a feedback loop to the controller.

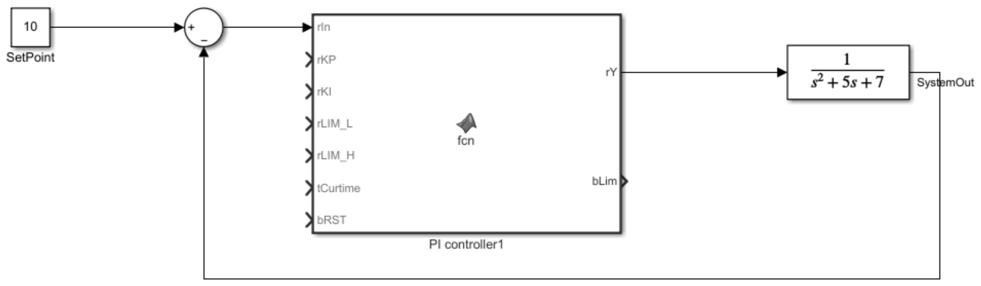


Figure 22, PI-controller extended with a simple plant model

Figure 23 represents the whole model with some arbitrary inputs in a test harness. PI-controller is simulated in a closed-loop with the plant model to allow dynamics to play part in a testing phase.

4.4 Verification

The verification part relates to small arrows shown in Figure 24 between each step which is further explained in detail in this chapter.

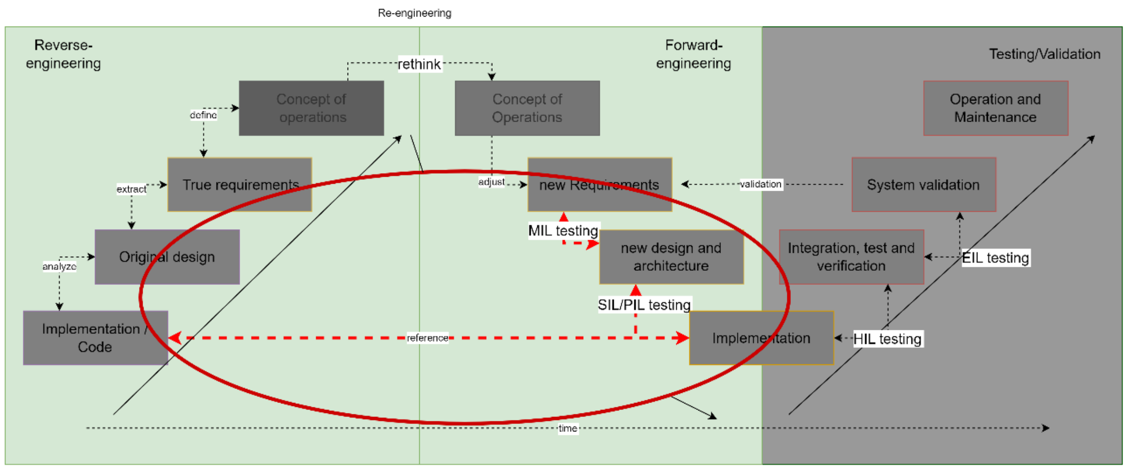


Figure 24, *Verification as shown on re-engineering model*

Deciding how to verify the model and implementation to match requirements is not generally an easy choice. Figure 25 shows each verification phase. 1st Verification is using specification-based testing, meaning that generating tests is done based on specifications (in this thesis requirements). More advanced verification methods, like

Model checking, are not needed for this simple case. 2nd and 3rd verification use numerical equivalence testing which means that new and reference implementations should have the same result compared to each other and the model in the 1st verification.

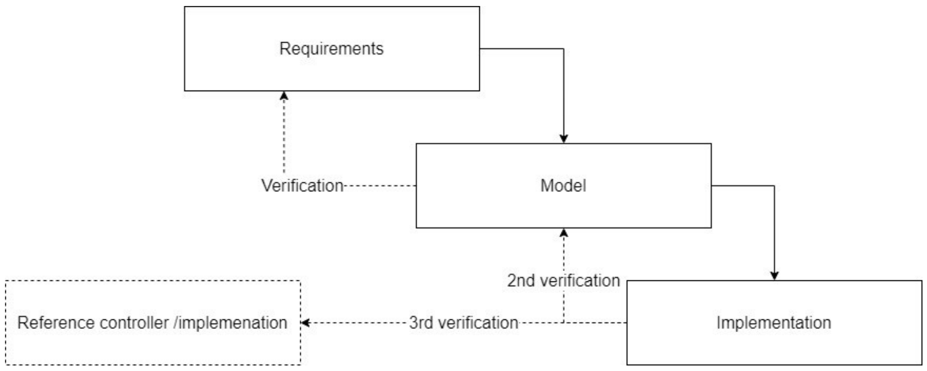


Figure 25: *Three verification phases*

This numerical equivalence testing uses the tests that are generated in the first verification, although it could be argued that code-specific testing would be more appropriate as it would allow more detailed testing and would take account of the inner workings of these test objects.

The limitation in PI-controller -verification is that plant (acting) model should be as close to real-life as possible to fully test against requirements. But as requirements were intentionally defined loosely, the arbitrary plant model was created which was shown in the previous section. As for verification, some arbitrary test vector can be created and numerical equivalence compared between each verification step. A discrete plant model could also have been used with SoM-model verification, as shown in the article " Increasing the efficiency of PLC Program Verification using a plant model” [21]. The study was using formal verification methods and concluded that adding this plant can increase engineers' capability to prove new properties but increases complexity in verification.

The verification process is described in detail in Figure 26. for both types of functions. As shown in the re-engineering process model, verification is not a step to be executed but a condition to move forward in design. Verification processes used in this thesis are quite demanding in effort and only examples in these two specific cases. Normally this process should be adapted to the complexity of the functions and presented verification methods are excessive relative to the functions.

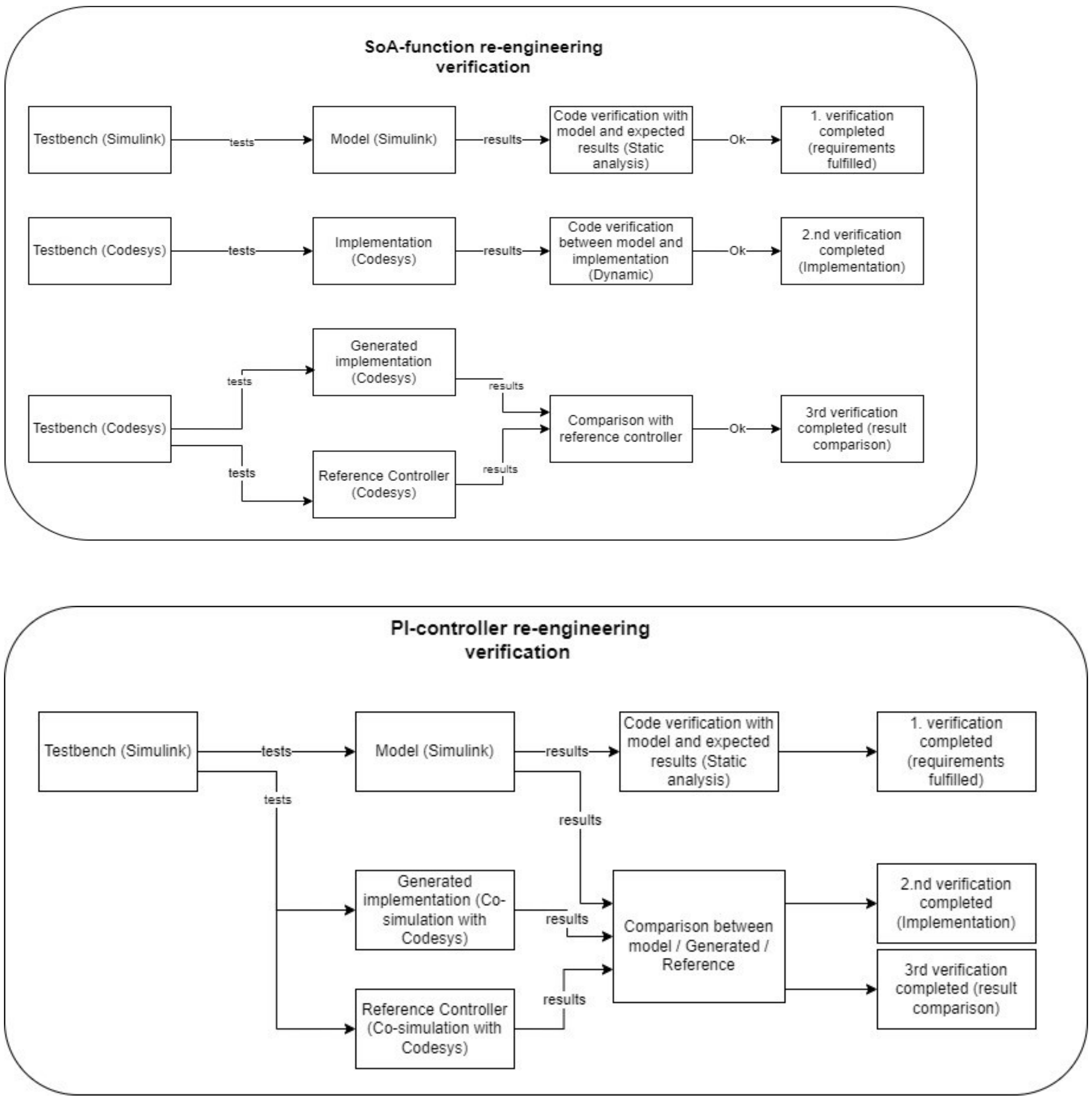

SoA-function re-engineering verification
Testbench (Simulink)
tests
Model (Simulink)
results
Code verification with model and expected results (Static analysis)
Ok
1. verification completed (requirements fulfilled)
Testbench (Codesys)
tests
Implementation (Codesys)
results
Code verification between model and implementation (Dynamic)
Ok
2.nd verification completed (Implementation)
Testbench (Codesys)
tests
Generated implementation (Codesys)
results
Comparison with reference controller
Ok
3rd verification completed (result comparison)
tests
Reference Controller (Codesys)
results
PI-controller re-engineering verification
Testbench (Simulink)
tests
Model (Simulink)
results
Code verification with model and expected results (Static analysis)
1. verification completed (requirements fulfilled)
tests
results
Generated implementation (Co-simulation with Codesys)
results
Comparison between model / Generated / Reference
2.nd verification completed (Implementation)
3rd verification completed (result comparison)
Reference Controller (Co-simulation with Codesys)
results

Figure 26, *Top: SoM-verification, Bottom: PI-controller verification*

As shown in the pictures, the verification process differs between the two controllers. Verification methods differ because SoM-function is using combinatorial and sequential decision logic and in practice doesn’t usually need complex verification methods. Usually, MBD is used to model complex control logic with algorithmic functions and the PI-controller’s workflow presents a more general and simpler approach to this type of control. In PI-controller verification, two verification steps are combined as this step is executed with co-simulation in software-in-the-loop testing. Co-simulation is not necessary, and the SoM-verification method could be used as well with PI-controller, but this was done to find a less demanding approach to workload.

Test cases are created and used to verify functionality in between model and requirements but also in this case, between a reference controller and a new controller. These test cases are created based on requirements defined in the previous phase. Test cases do not test all possible outcomes but give confidence that at least the basic principles work. Test cases are missing a few relevant program paths identified from the code but are left out intentionally and defined as irrelevant in the scope of this thesis.

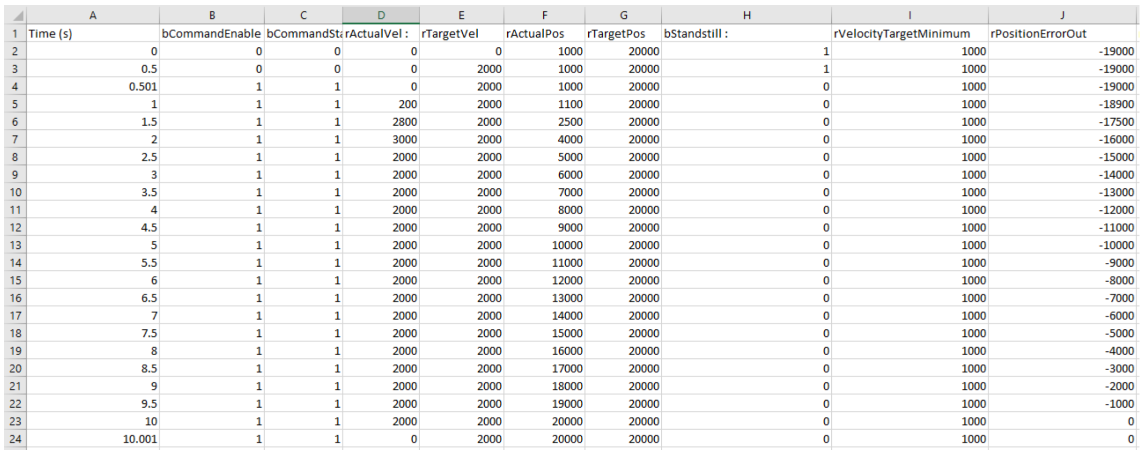
SoM has following properties/methods:

1. Determine movement starting error. The movement has not started.
2. Determine movement direction error when movement has started.
3. Store last starting position
4. Store control target position
5. Initiate continuous movement checking

To test these properties, three test cases were created:

```
   Test Case 1: Normal drive
   1. Store starting position for comparing when target position changes or
start command is turned on
   2. Start monitoring ("on fly" OR when velocity is low enough) AND movement
distance is long enough (Positioning window)
   3. Start moving at appropriate speed.
   4. When 1s has elapsed check that axis has moved at least Y mm

Expected result: No errors and last starting position and control start posi-
tion are stored.


   Test Case 2: Movement did not start fault
   -  1. Store starting position for comparing when target position changes
or start command is turned on
   -  2. Start monitoring ("on fly" OR when velocity is low enough) AND move-
ment distance is long enough (Positioning window)
   -  3. Vehicle is standstill

Expected result: When 1s is elapsed, Error "movement starting error" is visi-
ble and last starting position and control start position are stored.


   Test Case 3: Movement started to wrong direction
      1. Store starting position for comparing when target position changes
or start command is turned on
      2. Start monitoring ("on fly" OR when velocity is low enough) AND move-
ment distance is long enough (Positioning window)
      3. Start moving on the direction

Expected result: When DirectionDetectLimit is exceeded, Error "Direction er-
ror" is visible and last starting position and control start position are
stored.
```

These test cases were then translated to test vectors, meaning a set of inputs to test provided system. This was done using a simple spreadsheet (Figure 27) and importing it to Simulink signal builder (Figure 28). The headers in the spreadsheet had the same symbols as in the testable interface so these could be easily identified when imported. One notable thing about using this method is that the first column needs to define simulation time. There are likely more modern ways to introduce these test cases to simulation like using Simulink Test, but this was simple and effective.



	A	B	C	D	E	F	G	H	I	J
1	Time (s)	bCommandEnable	bCommandSt	rActualVel :	rTargetVel	rActualPos	rTargetPos	bStandstill :	rVelocityTargetMinimum	rPositionErrorOut
2	0	0	0	0	0	1000	20000	1	1000	-19000
3	0.5	0	0	0	2000	1000	20000	1	1000	-19000
4	0.501	1	1	0	2000	1000	20000	0	1000	-19000
5	1	1	1	200	2000	1100	20000	0	1000	-18900
6	1.5	1	1	2800	2000	2500	20000	0	1000	-17500
7	2	1	1	3000	2000	4000	20000	0	1000	-16000
8	2.5	1	1	2000	2000	5000	20000	0	1000	-15000
9	3	1	1	2000	2000	6000	20000	0	1000	-14000
10	3.5	1	1	2000	2000	7000	20000	0	1000	-13000
11	4	1	1	2000	2000	8000	20000	0	1000	-12000
12	4.5	1	1	2000	2000	9000	20000	0	1000	-11000
13	5	1	1	2000	2000	10000	20000	0	1000	-10000
14	5.5	1	1	2000	2000	11000	20000	0	1000	-9000
15	6	1	1	2000	2000	12000	20000	0	1000	-8000
16	6.5	1	1	2000	2000	13000	20000	0	1000	-7000
17	7	1	1	2000	2000	14000	20000	0	1000	-6000
18	7.5	1	1	2000	2000	15000	20000	0	1000	-5000
19	8	1	1	2000	2000	16000	20000	0	1000	-4000
20	8.5	1	1	2000	2000	17000	20000	0	1000	-3000
21	9	1	1	2000	2000	18000	20000	0	1000	-2000
22	9.5	1	1	2000	2000	19000	20000	0	1000	-1000
23	10	1	1	2000	2000	20000	20000	0	1000	0
24	10.001	1	1	0	2000	20000	20000	0	1000	0

Figure 27, *Testcase 1 as a spreadsheet*

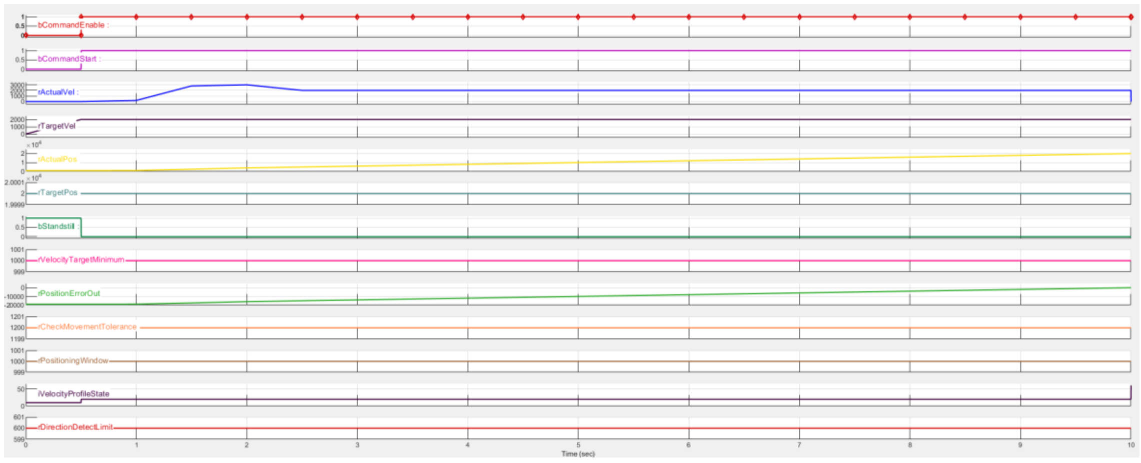


Figure 28, *SignalBuilder view with TestCase1*

For the PI-controller, the actual test cases were not created. Instead, an arbitrary test vector was used as the requirements are mostly qualitative, and testing controller limits seemed unnecessary. As an expected result could be defined that the controller can drive the mass-spring-system to a steady state with a certain time limit without overdamping in the transient phase but the actual values are not important.

For both controllers, test harnesses were used for isolating components under test so that the actual model would remain usable when these test signals were added. The general test harness setup can be seen in Figure 29.

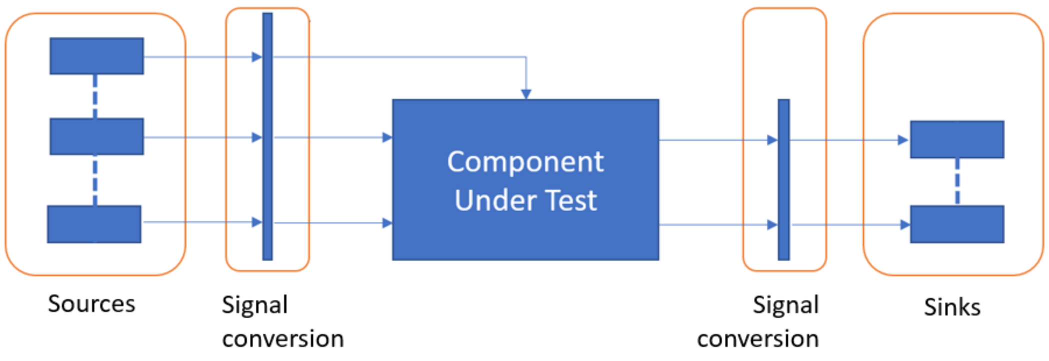


***Figure 29:** Simulink Test Harness [22]*

Also, when using a test harness, the PLC coder can export verification code (test-vectors) and required test benches with the actual code.

4.5 Model verification

Model-in-the-Loop (MiL) testing is done in the early stage of development. This will ensure that created model conforms to the re-created requirements. Model-in-Loop testing is related to unit testing in traditional software development, while the units under test are “model units”. Only PI-controller is simulated “in-loop” with plant-model. SoM-model is simulated and results analysed that the model meets the requirements but as this does not possess any dynamic properties and plant model is present, the “in-loop” definition is not directly accurate and that’s why referred to just as model verification although the goal is the same. Both models are tested using the test vectors (signal builder) created in the last section and the results are analysed. If the requirements are met in each case, the model is correct, and the next step (implementation) can be performed

4.5.1 SoM: Model verification

StartingOfMotion (SoM) 1. st verification test harness is shown in Figure 30, Model testing on SoM. As can be seen from the model, Signalbuilder generates all signals with double-datatype (this can’t be changed) and uses continuous signals. These need to be transformed to correct datatype and discrete-time for the model to be executed. These changes were done by “Convert” and “unit delay” -blocks and as these did not affect the tests, other options were not searched. SoM-model has five outputs which are observed in analysis phase: LastStoredStartingPosition, ControlStartPosition, CheckMovementContinous, MovementStartingError, DirectionError. LastStoredStartingPosition -naming is confusing as it saves the Target position (input) when movement is determined to start. ControlStartPosition saves the current position (input) when movement has been determined started. CheckMovementContinous is set to true when MovementStartingError is checked. MovementStartingError is set when movement is determined to have started but the crane has not moved within a certain time limit. DirectionError is set when Crane moves in the opposite direction against the target position for too far (set by input-parameter). ControlStartPosition, CheckMovementContinuous, LastStoredStartingPosition are used inside the calling function which is not covered in this thesis.

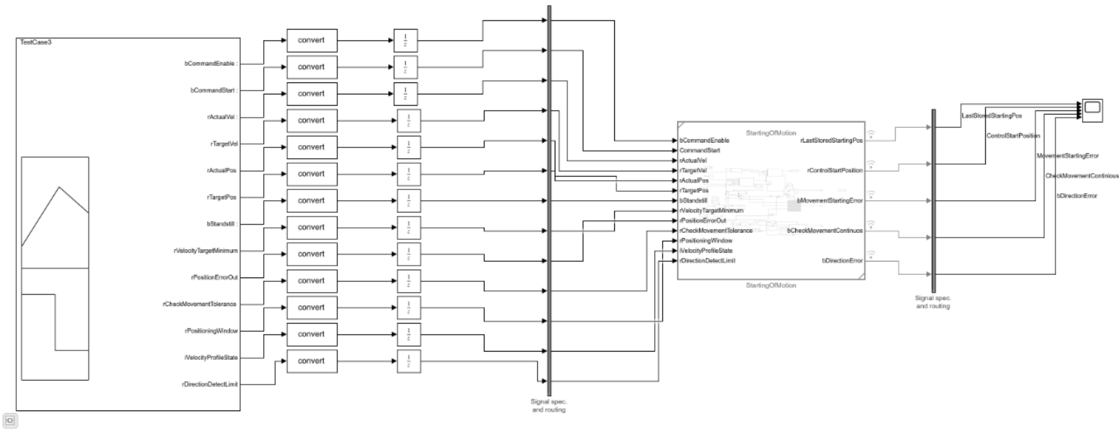


Figure 30, *Model testing on SoM- test harness, full image as appendix 5*

Results from running test vectors are shown in Figure 31, Figure 32, and Figure 33.

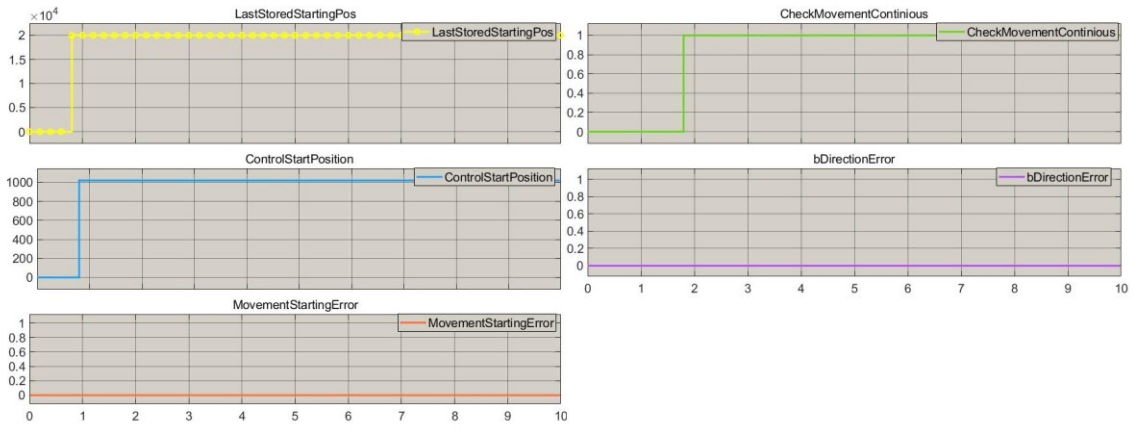


Figure 31, *Testcase 1 on SoM MIL*

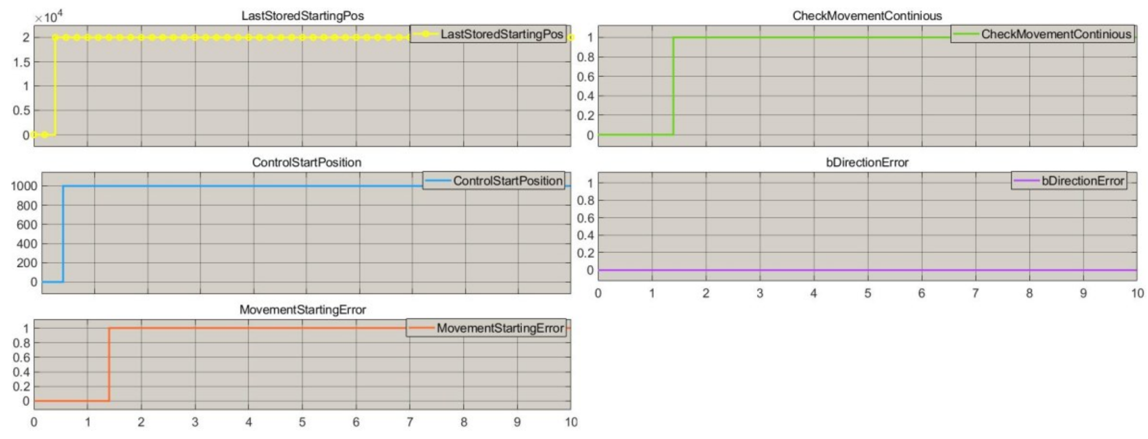


Figure 32, *TestCase2 on SoM MIL*

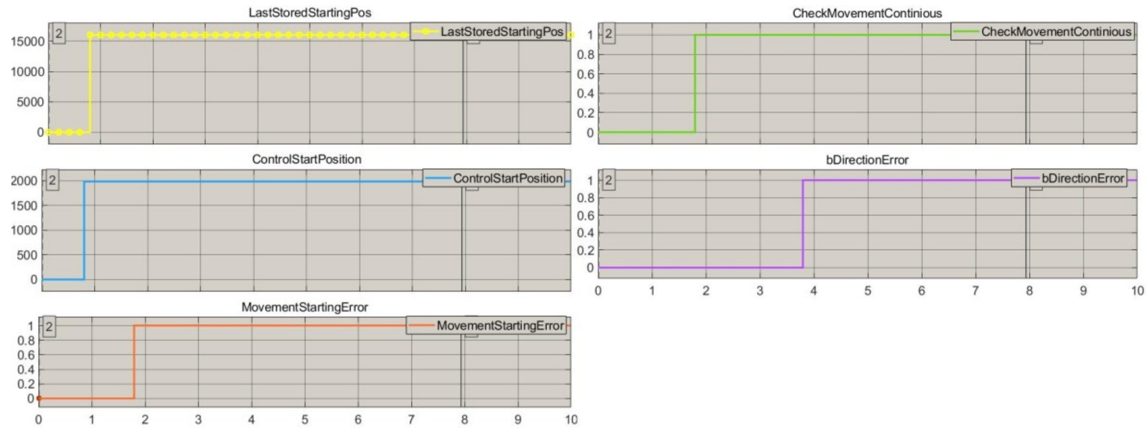


Figure 33, *TestCase3 on SoM MIL*

Three test cases were created and the expected results are as follows:

<u>TestCase 1 Expected result</u>: No errors and the last starting position and control start position are stored.

<u>TestCase 2 Expected result</u>: When 1s is elapsed, Error “movement starting error” is visible and the last starting position and control start position is stored.

<u>TestCase 3 Expected result</u>: When DirectionDetectLimit is exceeded, Error “Direction error” is visible and the last starting position and control start position is stored.

Comparing results to expected results shows that nothing suspicious is happening and all three test cases are passed as they meet the expected results. This indicates that the model is performing adequately against the requirements. No further analyzing needed at this point.

4.5.2 PI-controller: Model-in-the-loop verification

PI-controller’s MIL verification is straightforward because no strict requirements were imposed. The only requirement that was set was that System output is controlled and reach a stable state within a reasonable time window. Figure 34 shows the test harness used in testing, and in this case, only static values were used as test vectors for the PI-controller. PI-controller model control output (rY) and diagnostic output (bLIM), as well as Plant output (SystemOut), were observed and analyzed.

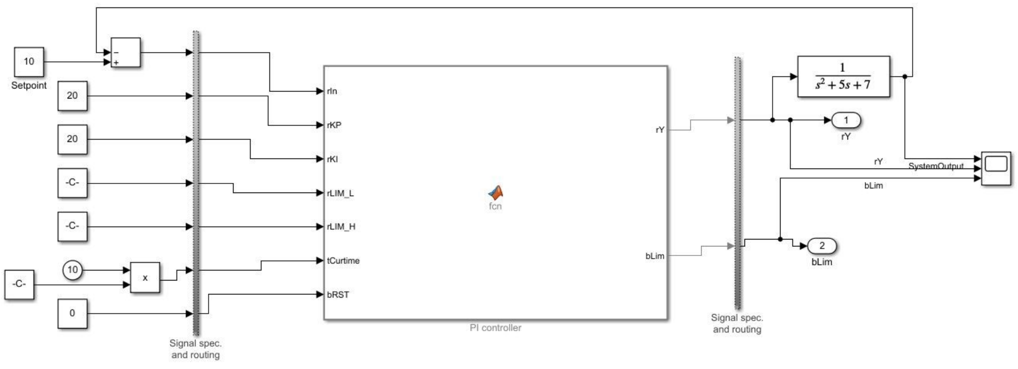


***Figure 34**: MIL testing on PI-controller test harness*

Running a simulated PI-controller with the static test vectors shown in Figure 34 produces results shown in Figure 35: MIL test results.

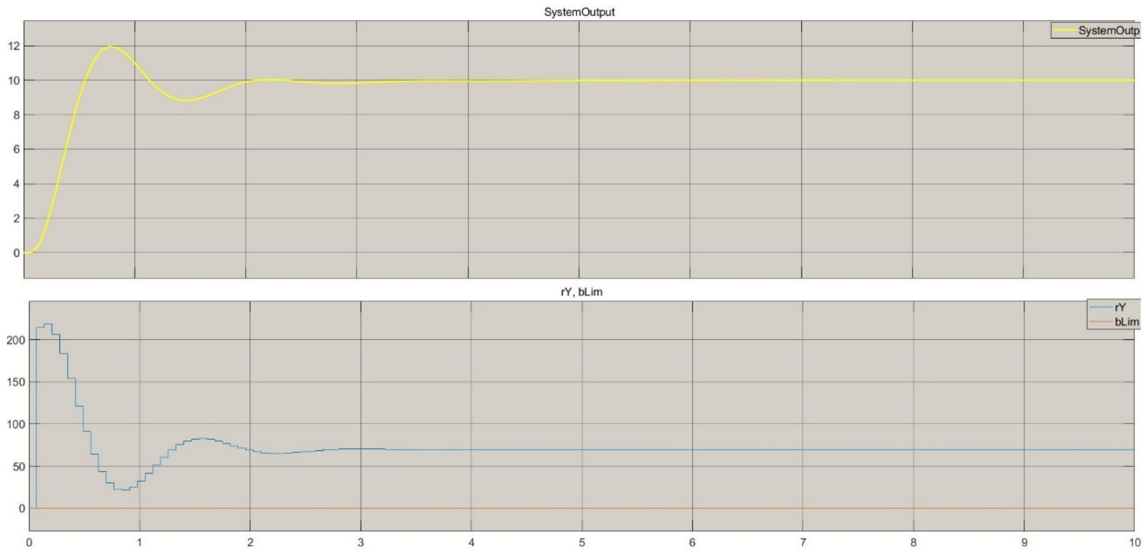


***Figure 35**: MIL test results, Top: Plant output, Bottom: Controller output*

From the MIL-test results, we can determine that the output of the system stabilizes in setpoint (10) as defined in requirements and the controller is not saturated at any point. The actual performance is not analysed as there are no requirements defined.

4.6 Implementation and Code-generation with Simulink PLC coder

This part is referenced as the implementation part in the re-engineering model.

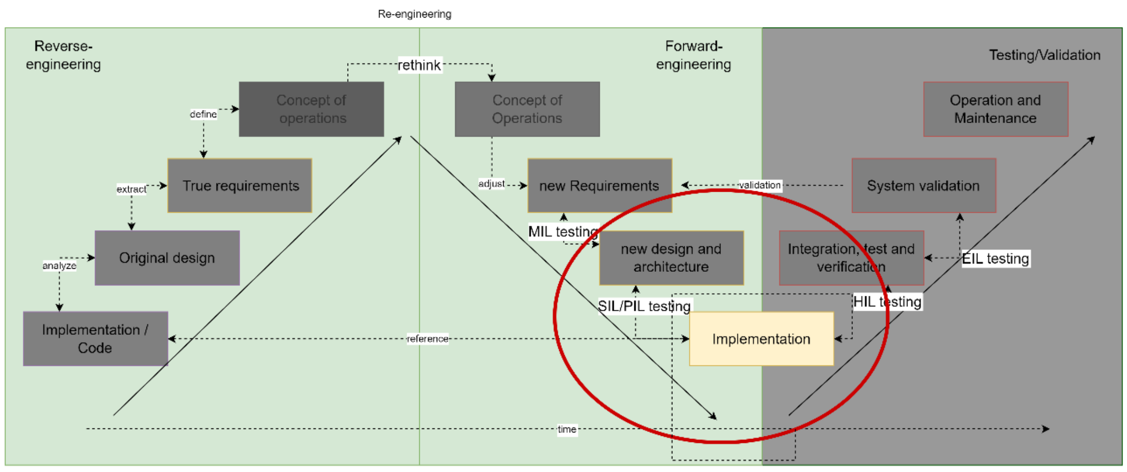


Figure 36. *Implementation in re-engineering model*

Implementation was done by generating PLC code straight from the model. This was done using Mathworks PLC coder which allows the generation of IEC 61131-3 compliant Structured text code. PLC coder doesn't seem to support the 3rd revision of IEC-61131-3 which extends the previous standard with new data types and conversion functions, references, namespaces, and the object-oriented features of classes and function blocks [IEC 61131-3], and also other limitations are present. A full list of limitations was available for the PLC coder [23] but this didn't seem to be updated, as some of the limitations mentioned were functioning regardless. This indicates some mismatch in the documentation versus the PLC coder version.

In code generation, one of the most noticeable limitations was the inability to generate code for a simple timer in Simulink. This was essential in the SoM-model because usually at least in high-level alarm generation, errors need to be active for a defined time before setting an alarm. These timers are usually attached to the error-generating function as a “low-pass filter” to reduce false positives. Timer generation was supported for PLC ladder logic and Stateflow, and in the end, the limitation was bypassed by using Stateflow for timer generation.

PLC coder's most useful feature in the thesis was the ability to generate test bench directly to generated code. Test bench means that the generated code has a test function that is isolated from the actual function, which automatically tests the model with wanted test vectors. Generated SoM-testbench can be found in appendix 1.

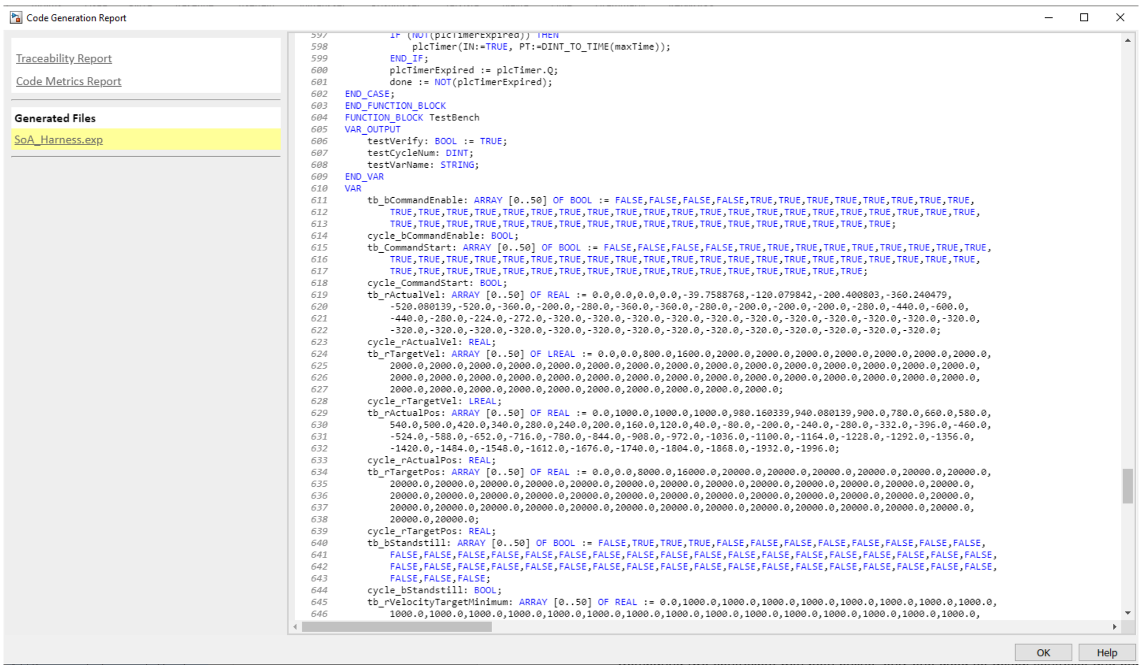
Another feature worthy of mention is that the PLC coder can generate a traceability report which links the model to generated code line so each can be compared manually when needed. This is required when using code generation for safety-related IEC 61508-3 code generation [24]. Also, custom comments can be linked from model to code which helps readability, although generated code is usually not meant for manual intervention.

The actual implementation is executed on Raspberry PI Zero and Codesys runtime On Linux after code generation with both controller types.

4.7 Implementation verification

Software-in-loop verification refers to testing using actual implementation. As mentioned before, this part is done differently from SoM-function compared to PI-controller. In SoM-verification, the model is not “in-the-loop” which means SIL-term is not fully accurate, but the purpose of the test remains the same.

As mentioned in the previous chapter, a PLC coder was used to generate the implementation and also test benches from the test harness. This will verify that the actual implementation runs exactly as the model in Simulink.



```
        IF (NOT(plcTimerExpired)) THEN
            plcTimer(IN:=TRUE, PT:=DINT_TO_TIME(maxTime));
        END_IF;
        plcTimerExpired := plcTimer.Q;
        done := NOT(plcTimerExpired);
END_CASE;
END_FUNCTION_BLOCK
FUNCTION_BLOCK TestBench
VAR_OUTPUT
    testVerify: BOOL := TRUE;
    testCycleNum: DINT;
    testVarName: STRING;
END_VAR
VAR
    tb_bCommandEnable: ARRAY [0..50] OF BOOL := FALSE,FALSE,FALSE,FALSE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,
        TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,
        TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE;
    cycle_bCommandEnable: BOOL;
    tb_CommandStart: ARRAY [0..50] OF BOOL := FALSE,FALSE,FALSE,FALSE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,
        TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,
        TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE;
    cycle_CommandStart: BOOL;
    tb_rActualVel: ARRAY [0..50] OF REAL := 0.0,0.0,0.0,0.0,-39.7588768,-120.079842,-200.400803,-360.240479,
        -520.080139,-520.0,-360.0,-200.0,-280.0,-360.0,-360.0,-280.0,-200.0,-200.0,-200.0,-280.0,-440.0,-600.0,
        -440.0,-280.0,-224.0,-272.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,
        -320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0,-320.0;
    cycle_rActualVel: REAL;
    tb_rTargetVel: ARRAY [0..50] OF LREAL := 0.0,0.0,800.0,1600.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,
        2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,
        2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,
        2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0,2000.0;
    cycle_rTargetVel: LREAL;
    tb_rActualPos: ARRAY [0..50] OF REAL := 0.0,1000.0,1000.0,1000.0,980.160339,940.080139,900.0,780.0,660.0,580.0,
        540.0,500.0,420.0,340.0,280.0,240.0,200.0,160.0,120.0,40.0,-80.0,-200.0,-240.0,-280.0,-332.0,-396.0,-460.0,
        -524.0,-588.0,-652.0,-716.0,-780.0,-844.0,-908.0,-972.0,-1036.0,-1100.0,-1164.0,-1228.0,-1292.0,-1356.0,
        -1420.0,-1484.0,-1548.0,-1612.0,-1676.0,-1740.0,-1804.0,-1868.0,-1932.0,-1996.0;
    cycle_rActualPos: REAL;
    tb_rTargetPos: ARRAY [0..50] OF REAL := 0.0,0.0,8000.0,16000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,
        20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,
        20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,
        20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,20000.0,
        20000.0,20000.0;
    cycle_rTargetPos: REAL;
    tb_bStandstill: ARRAY [0..50] OF BOOL := FALSE,TRUE,TRUE,TRUE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,
        FALSE,FALSE,FALSE;
    cycle_bStandstill: BOOL;
    tb_rVelocityTargetMinimum: ARRAY [0..50] OF REAL := 0.0,1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,
        1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,1000.0,
```

Figure 37, *Example of how a test vector is integrated into generated code*

As shown in Figure 37, the test bench is implemented using arrays that are looped through functions and outputs verified.

4.7.1 SoM: Implementation verification with test bench

Verification using test benches is a straightforward process. Because the interface is modelled identically with the reference controller function, a test bench can be used to run test cases with both the reference controller and the generated one.

Running the test bench had some problems in real-time depended operations as the tech bench is generated as discrete values in inputs and expected values in each PLC cycle. When the PLC is performing slower or faster than in the Simulink model (which usually is the case even if only for one cycle), the test bench generates an error. One method of trying to avoid this error is to set PLC task to match the model sample time. But this usually still fails, so the best method is to simply modify the test bench to allow small differences to happen, or manually inspect the test with traces. Test results with running test cases on SoftPLC are shown in Figure 38, TestCase1 SoM “SIL” verification: Implementation compared to model, Figure 39, TestCase2 SoM "SIL" verification: Implementation compared to model, and Figure 40 Testcase3, SoM “SIL” verification: Implementation compared to modeland prefixes “cycle_” and “gen_” signify model and generated implementation respectively.

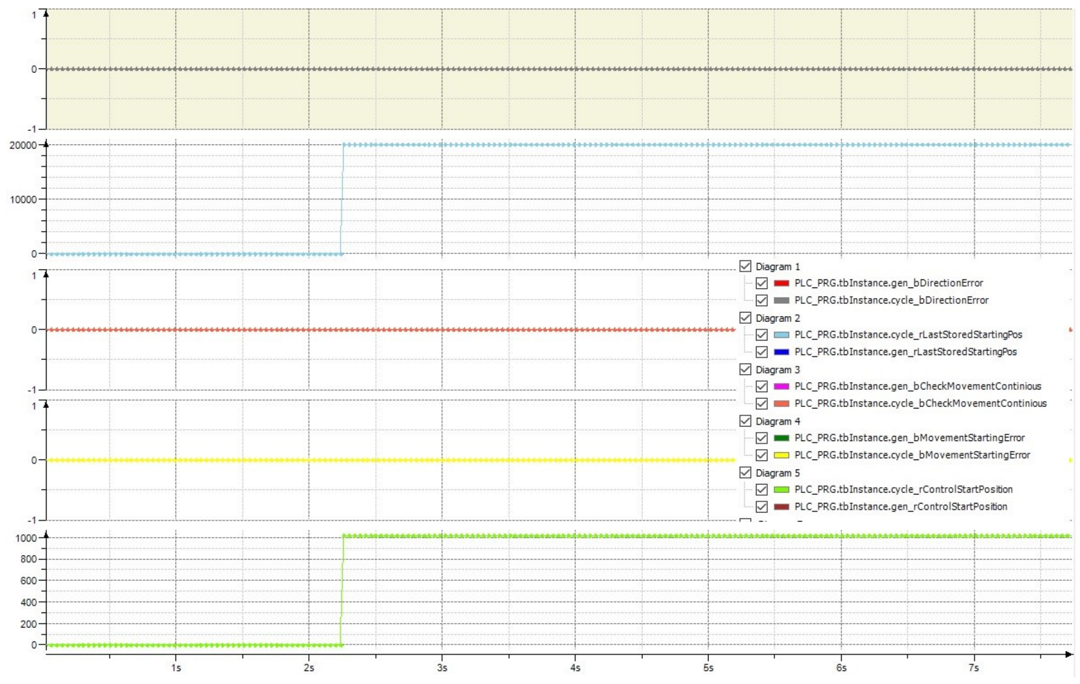


Figure 38, *TestCase1 SoM “SIL” verification: Implementation compared to model*

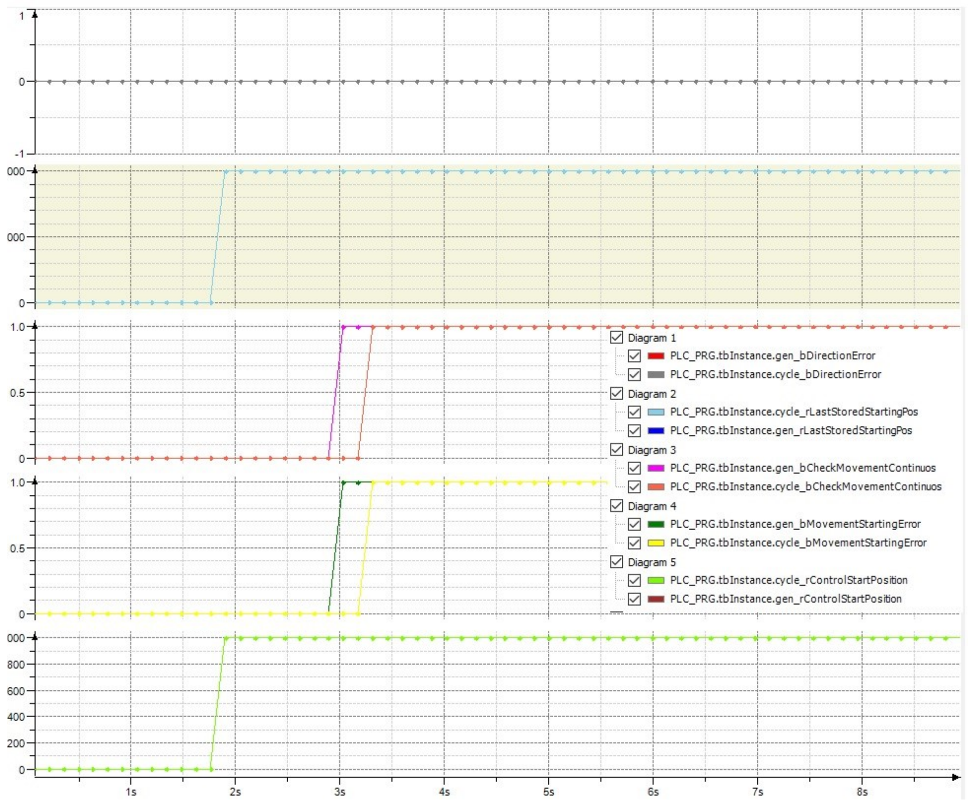


***Figure 39,** TestCase2 SoM "SIL" verification: Implementation compared to model*

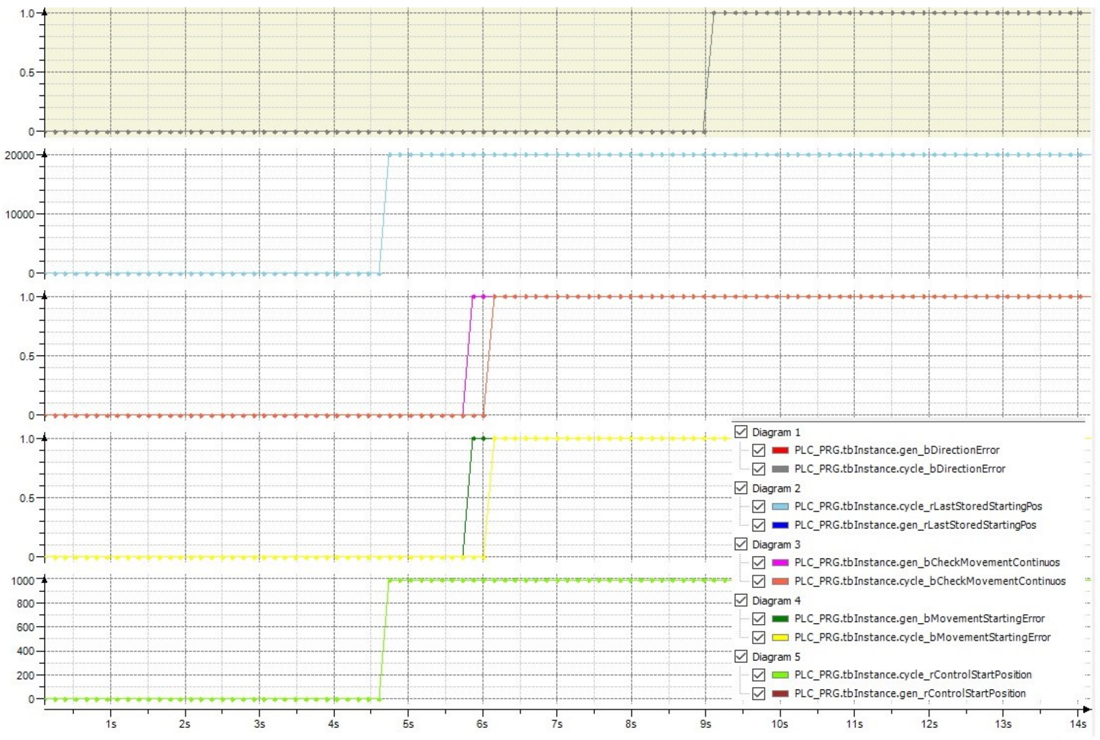
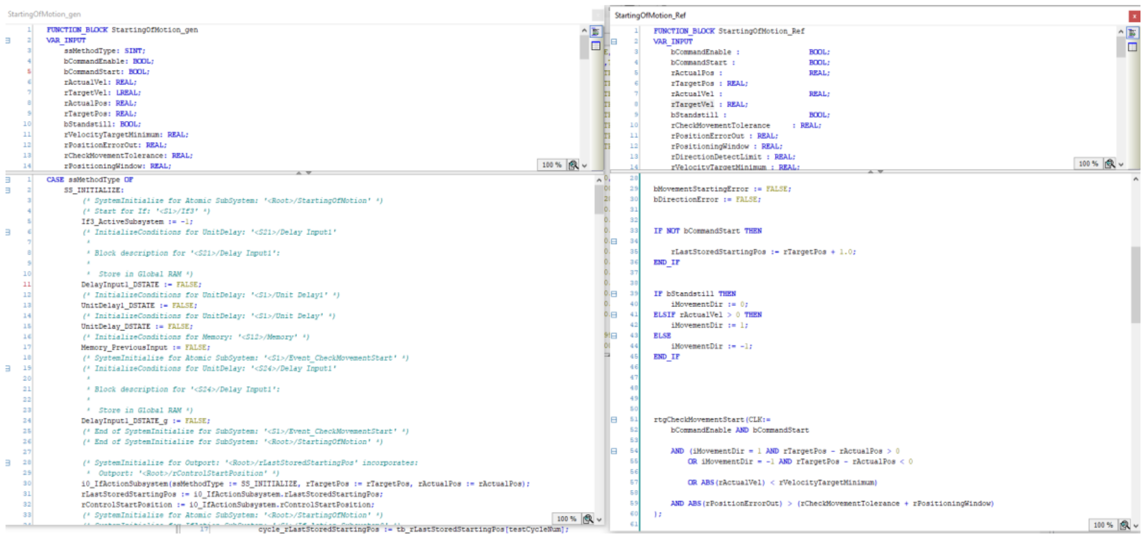


***Figure 40** Testcase3, SoM "SIL" verification: Implementation compared to model*

As can be seen in test results, the automatic tests that are generated by PLC coder will always fail when there are real-time related tests. Usually, this kind of small time variance does not matter in a real-life situation but it’s good to know that the generated testbench is really inflexible in these cases.

4.7.2 SoM: Comparison between reference and new implementation

In the last verification phase, the reference controller and implementation generated from the model are compared. This is also straightforward as the test bench can be run on a reference as well and results compared. A reference controller was imported to generated implementation and tests were run concurrently. As can be seen from Figure 41, Generated implementation on left, Reference (old) on right, generated one is full of comments that refer to the actual model and the interface has changed a bit by implementing a state machine that is not modelled (SSMethodType input). This “extra” input is used in a generated model to initialize variables in beginning but this should not cause errors in comparison.



```
FUNCTION_BLOCK StartingOfMotion_gen
VAR_INPUT
    ssMethodType: SINT;
    bCommandEnable: BOOL;
    bCommandStart: BOOL;
    rActualVel: REAL;
    rTargetVel: LREAL;
    rActualPos: REAL;
    rTargetPos: REAL;
    bStandstill: BOOL;
    rVelocityTargetMinimum: REAL;
    rPositionErrorOut: REAL;
    rCheckMovementTolerance: REAL;
    rPositioningWindow: REAL;
```

```
CASE ssMethodType OF
    SS_INITIALIZE:
        (* SystemInitialize for Atomic SubSystem: '<Root>/StartingOfMotion' *)
        (* Start for If: '<S1>/If3' *)
        If3_ActiveSubsystem := -1;
        (* InitializeConditions for UnitDelay: '<S21>/Delay Input1'
         *
         * Block description for '<S21>/Delay Input1':
         *
         *  Store in Global RAM *)
        DelayInput1_DSTATE := FALSE;
        (* InitializeConditions for UnitDelay: '<S1>/Unit Delay1' *)
        UnitDelay1_DSTATE := FALSE;
        (* InitializeConditions for UnitDelay: '<S1>/Unit Delay' *)
        UnitDelay_DSTATE := FALSE;
        (* InitializeConditions for Memory: '<S12>/Memory' *)
        Memory_PreviousInput := FALSE;
        (* SystemInitialize for Atomic SubSystem: '<S1>/Event_CheckMovementStart' *)
        (* InitializeConditions for UnitDelay: '<S24>/Delay Input1'
         *
         * Block description for '<S24>/Delay Input1':
         *
         *  Store in Global RAM *)
        DelayInput1_DSTATE_g := FALSE;
        (* End of SystemInitialize for SubSystem: '<S1>/Event_CheckMovementStart' *)
        (* End of SystemInitialize for SubSystem: '<Root>/StartingOfMotion' *)

        (* SystemInitialize for Outport: '<Root>/rLastStoredStartingPos' incorporates:
         *  Outport: '<Root>/rControlStartPosition' *)
        i0_IfActionSubsystem(ssMethodType := SS_INITIALIZE, rTargetPos := rTargetPos, rActualPos := rActualPos);
        rLastStoredStartingPos := i0_IfActionSubsystem.rLastStoredStartingPos;
        rControlStartPosition := i0_IfActionSubsystem.rControlStartPosition;
        (* SystemInitialize for Atomic SubSystem: '<Root>/StartingOfMotion' *)
```

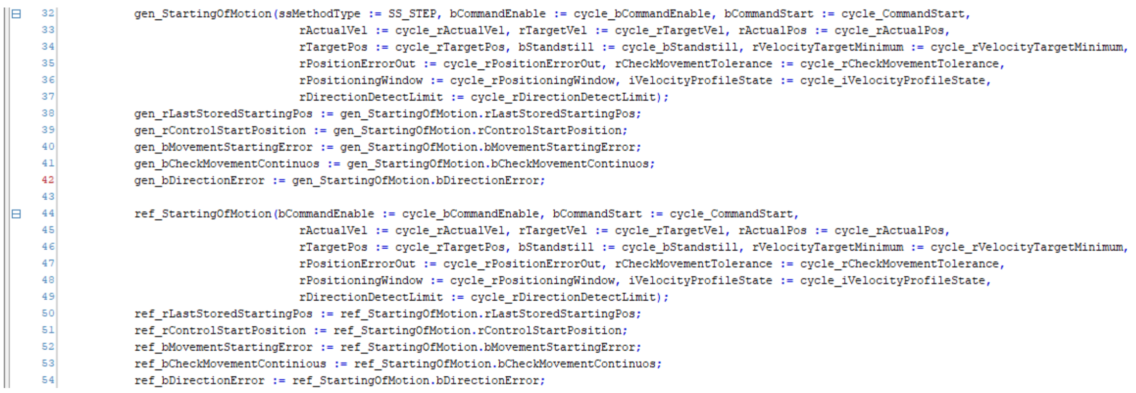
```
cycle_rLastStoredStartingPos := tb_rLastStoredStartingPos[testCycleNum];
```

```
FUNCTION_BLOCK StartingOfMotion_Ref
VAR_INPUT
    bCommandEnable :                BOOL;
    bCommandStart :                 BOOL;
    rActualPos :                    REAL;
    rTargetPos : REAL;
    rActualVel :                    REAL;
    rTargetVel : REAL;
    bStandstill :                   BOOL;
    rCheckMovementTolerance     : REAL;
    rPositionErrorOut : REAL;
    rPositioningWindow : REAL;
    rDirectionDetectLimit : REAL;
    rVelocityTargetMinimum : REAL;
```

```

bMovementStartingError := FALSE;
bDirectionError := FALSE;


IF NOT bCommandStart THEN

    rLastStoredStartingPos := rTargetPos + 1.0;
END_IF


IF bStandstill THEN
    iMovementDir := 0;
ELSIF rActualVel > 0 THEN
    iMovementDir := 1;
ELSE
    iMovementDir := -1;
END_IF





rtgCheckMovementStart(CLK:=
    bCommandEnable AND bCommandStart

    AND (iMovementDir = 1 AND rTargetPos - rActualPos > 0
        OR iMovementDir = -1 AND rTargetPos - rActualPos < 0

        OR ABS(rActualVel) < rVelocityTargetMinimum)

    AND ABS(rPositionErrorOut) > (rCheckMovementTolerance + rPositioningWindow)
);

```

Figure 41, *Generated implementation on left, Reference (old) on right*

Code screenshot In Figure 42, Implementation in test bench for reference and generated SoM shows how the reference was integrated to the test bench.



```
gen_StartingOfMotion(ssMethodType := SS_STEP, bCommandEnable := cycle_bCommandEnable, bCommandStart := cycle_CommandStart,
                        rActualVel := cycle_rActualVel, rTargetVel := cycle_rTargetVel, rActualPos := cycle_rActualPos,
                        rTargetPos := cycle_rTargetPos, bStandstill := cycle_bStandstill, rVelocityTargetMinimum := cycle_rVelocityTargetMinimum,
                        rPositionErrorOut := cycle_rPositionErrorOut, rCheckMovementTolerance := cycle_rCheckMovementTolerance,
                        rPositioningWindow := cycle_rPositioningWindow, iVelocityProfileState := cycle_iVelocityProfileState,
                        rDirectionDetectLimit := cycle_rDirectionDetectLimit);
gen_rLastStoredStartingPos := gen_StartingOfMotion.rLastStoredStartingPos;
gen_rControlStartPosition := gen_StartingOfMotion.rControlStartPosition;
gen_bMovementStartingError := gen_StartingOfMotion.bMovementStartingError;
gen_bCheckMovementContinuos := gen_StartingOfMotion.bCheckMovementContinuos;
gen_bDirectionError := gen_StartingOfMotion.bDirectionError;

ref_StartingOfMotion(bCommandEnable := cycle_bCommandEnable, bCommandStart := cycle_CommandStart,
                        rActualVel := cycle_rActualVel, rTargetVel := cycle_rTargetVel, rActualPos := cycle_rActualPos,
                        rTargetPos := cycle_rTargetPos, bStandstill := cycle_bStandstill, rVelocityTargetMinimum := cycle_rVelocityTargetMinimum,
                        rPositionErrorOut := cycle_rPositionErrorOut, rCheckMovementTolerance := cycle_rCheckMovementTolerance,
                        rPositioningWindow := cycle_rPositioningWindow, iVelocityProfileState := cycle_iVelocityProfileState,
                        rDirectionDetectLimit := cycle_rDirectionDetectLimit);
ref_rLastStoredStartingPos := ref_StartingOfMotion.rLastStoredStartingPos;
ref_rControlStartPosition := ref_StartingOfMotion.rControlStartPosition;
ref_bMovementStartingError := ref_StartingOfMotion.bMovementStartingError;
ref_bCheckMovementContinious := ref_StartingOfMotion.bCheckMovementContinuos;
ref_bDirectionError := ref_StartingOfMotion.bDirectionError;
```

Figure 42, *Implementation in test bench for reference and generated SoM*

The test bench was executed in softPLC and results are shown in Figure 38, Figure 39, and Figure 40.

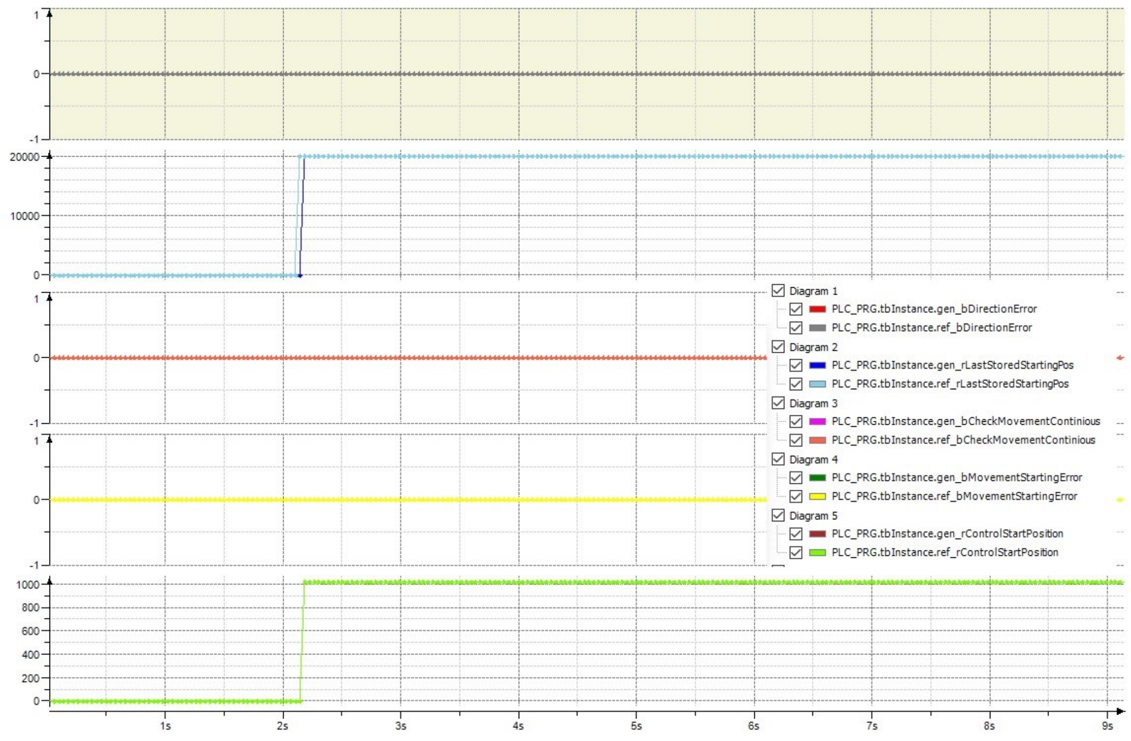


Figure 43, *SoM: TestCase1 verification - Comparison between generated and reference*

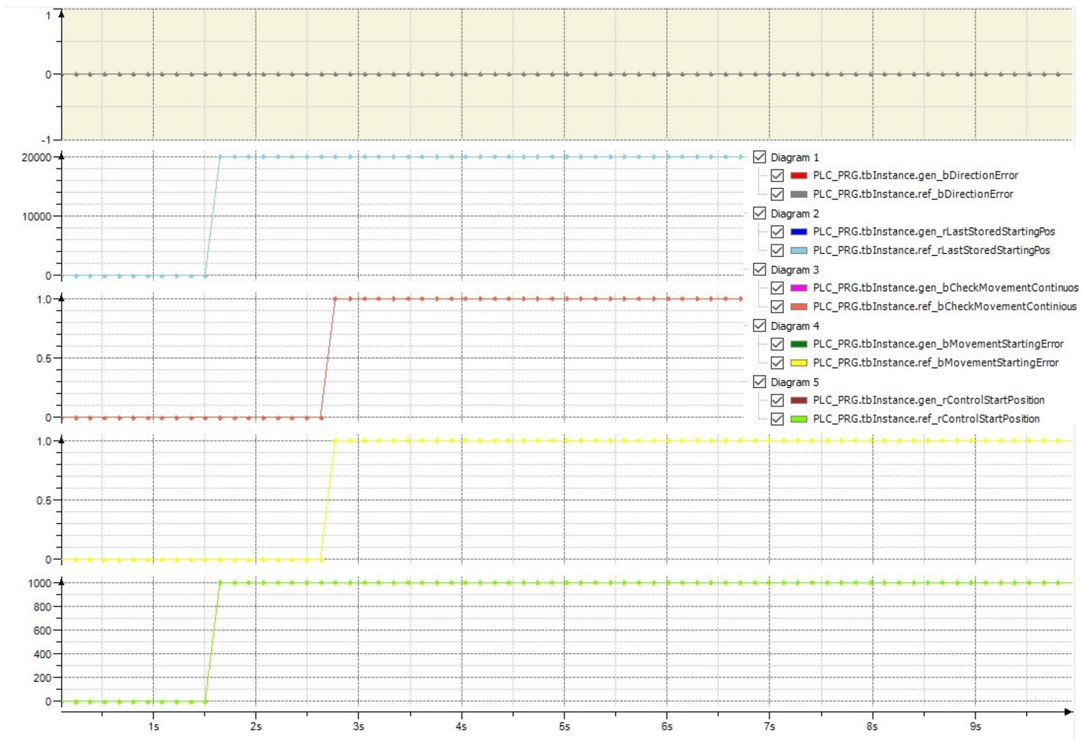


Figure 44, *SoM: TestCase2 Comparison - Comparison between generated and reference*

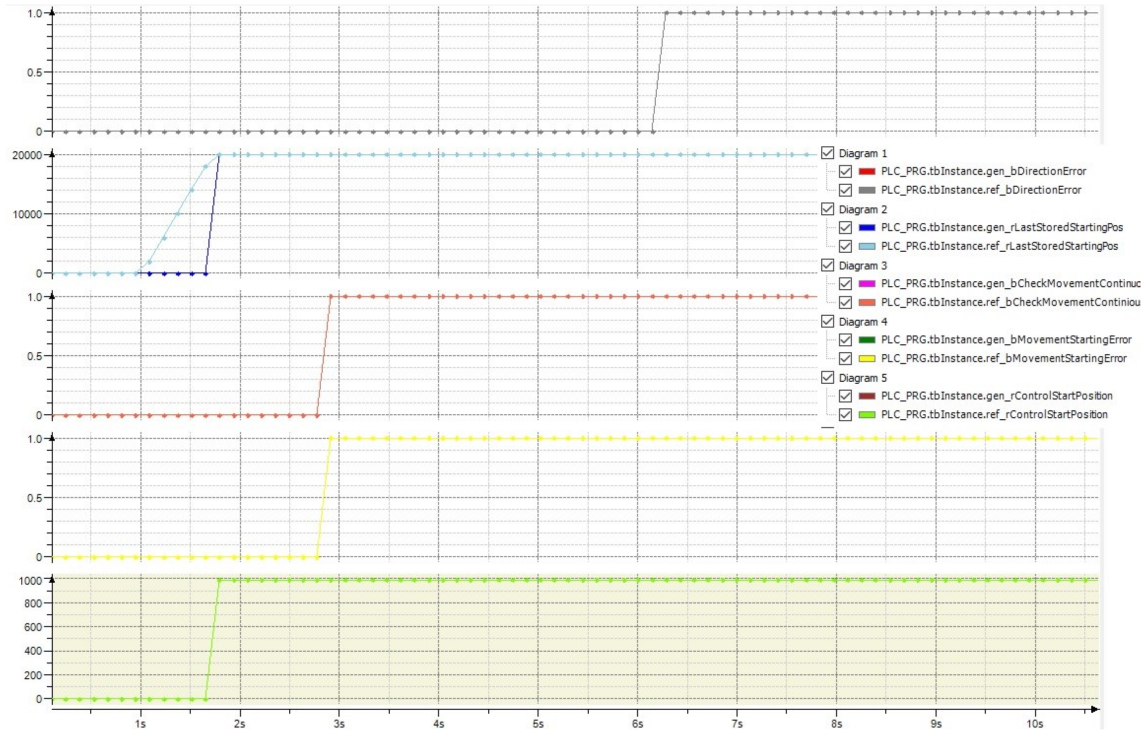


Figure 45, *SoM: TestCase3 Comparison - Comparison between generated and reference*

In Figure 43 and Figure 45, Testcase 1 and Testcase 3 there is an error between generated implementation and referenced. The comparison shows that generated controller has added delay for variable rLastStoredStartingPos, the reference implementation seems to store this variable differently. In the generated implementation, this variable storing happens as a one-shot operation, but the reference model stores this value in multiple cycles. There is also added delay in generated implementation, and this is most likely caused by the difference between model and reference implementation, and there is no error in code generation. The model uses state machines which usually add at least one cycle delay to operation. Diagnosing these differences has increased complexity because the generated code is hard to read by a human and ignored at this point. This could be diagnosed further, what is the exact reason but, in this case, we can determine that even with these errors, requirements are fulfilled with both implementations.

4.7.3 PI-controller: SIL verification with Co-simulation

In SIL verification using co-simulation, Simulink uses Open Platform Communication Unified Architecture (OPC UA) to communicate with soft PLC. In co-simulation, model and implementations (generated and reference) run test vector simultaneously with synchronized signals (Figure 46) and in-loop with the plant models. Synchronization is achieved by changing the timestep input from Simulink internal clock to be received from SoftPLC. This way, the Simulink model is updated with the same values as used in SoftPLC. If timestep is not updated from the PLC, the PI-controller model is not updated although executed. The whole architecture used in co-simulation can be seen in Figure 47

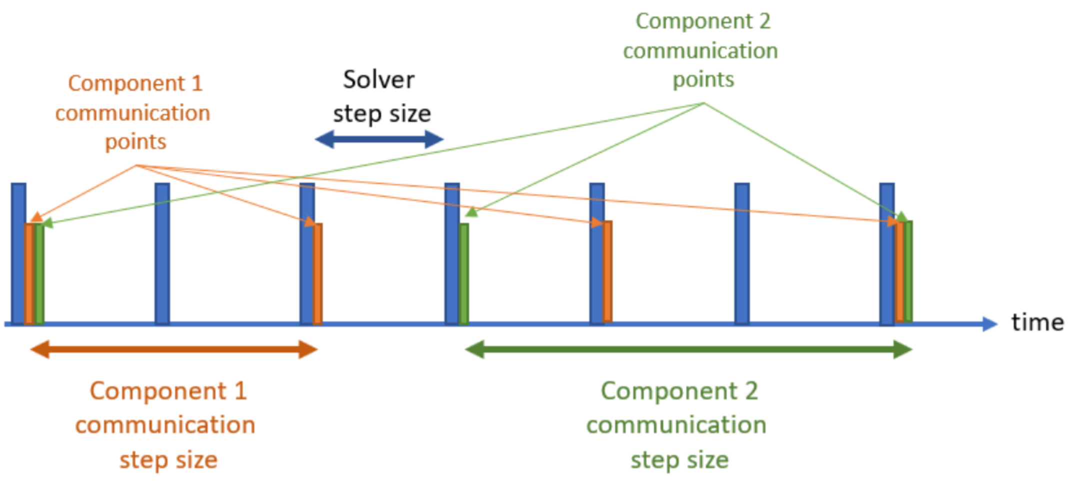


Figure 46, *Co-simulation synchronization [25]*

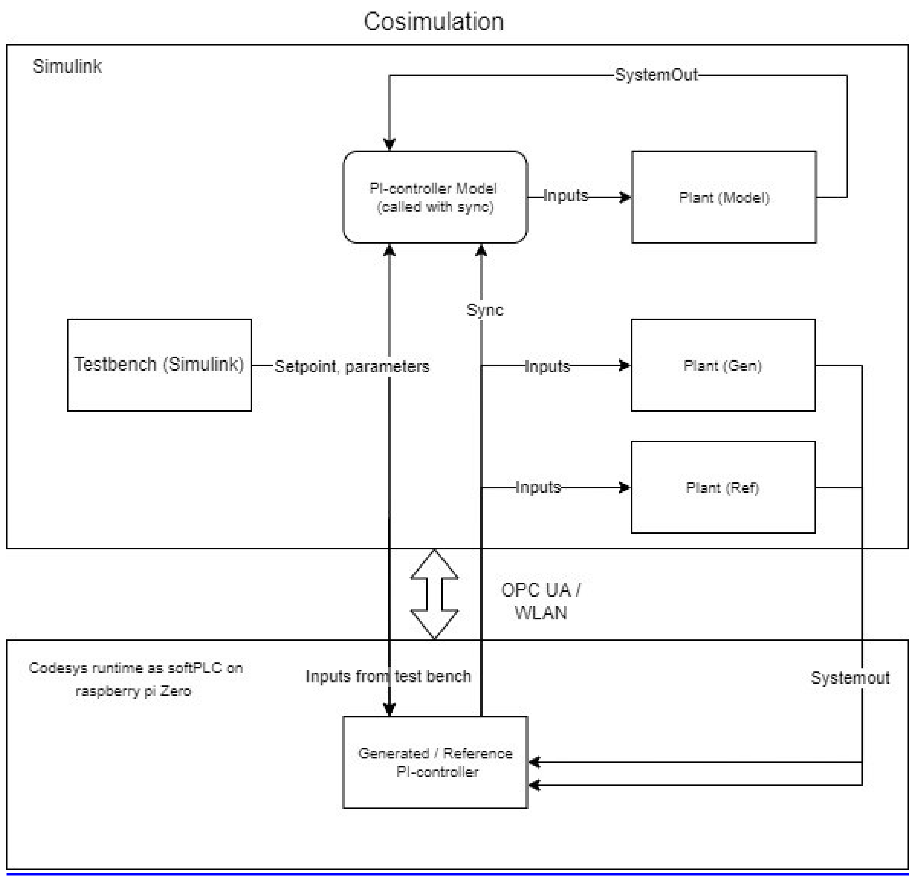


Figure 47, *Co-simulation system used in this thesis*

In this co-simulation, as said before, model, generated, and reference (SuTs) are executed simultaneously. Also, each testable has its dedicated plant model and feedback loop so the differences can be easily observed.

PIController_OPCUA which can be seen in Figure 48, is Matlab System -block that is executing Matlab-script. This script opens the OPC UA communication with the SoftPLC through the wireless network. The script finds predefined variables from the PLC and assigns values from the model. These values are then used in PI-controller in the plc and control -signals are brought back to the model. This OPC-UA script can be found in appendix 2. The generated PI-controller is found in appendix 3.

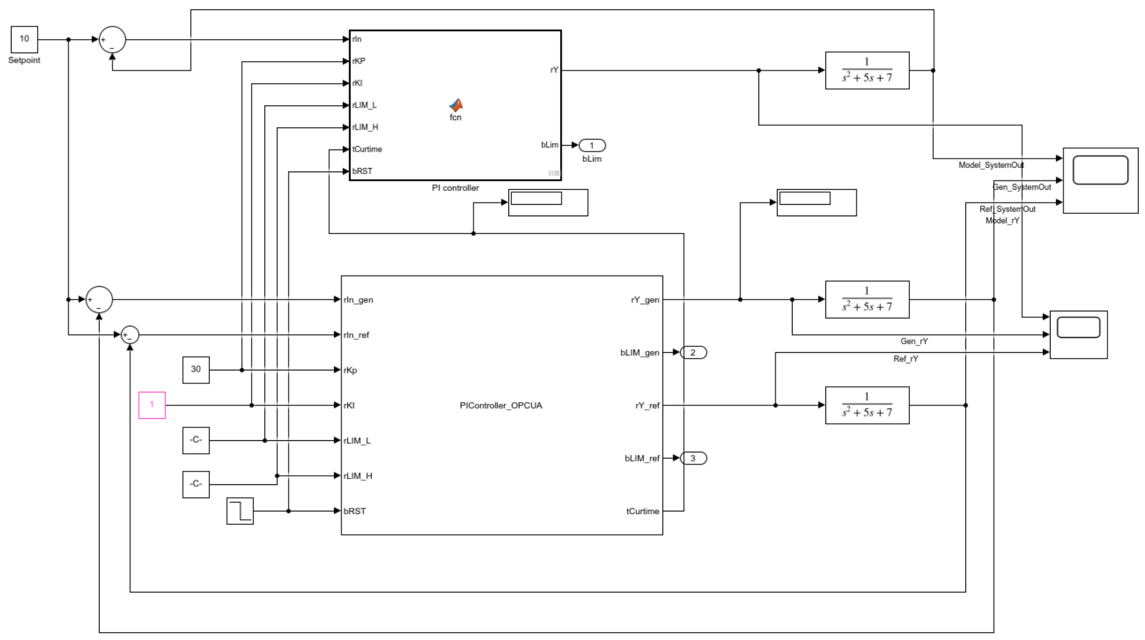


Figure 48, *Co-simulation model in Simulink, full image as appendix 6*

It was beneficial to add both generated, reference, and model to this co-simulation. In this way, separate 2nd verification which was done with SoM-model can be omitted as implementation verification and comparison with the reference controller can be done simultaneously. Results from the co-simulation can be seen in Figure 49 and Figure 50.

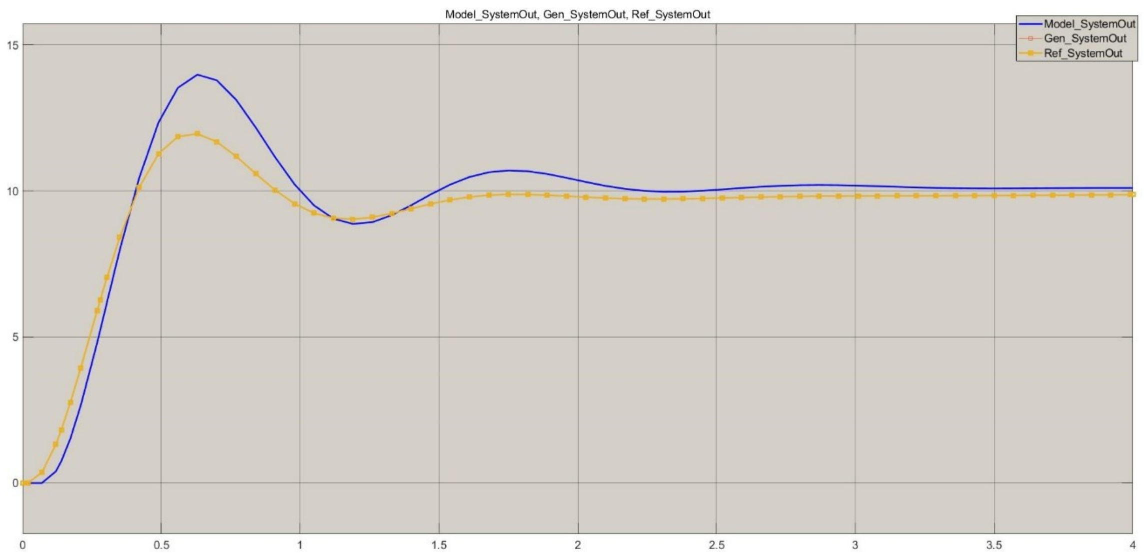


Figure 49, *System response for each controller. Generated and reference overlap completely*

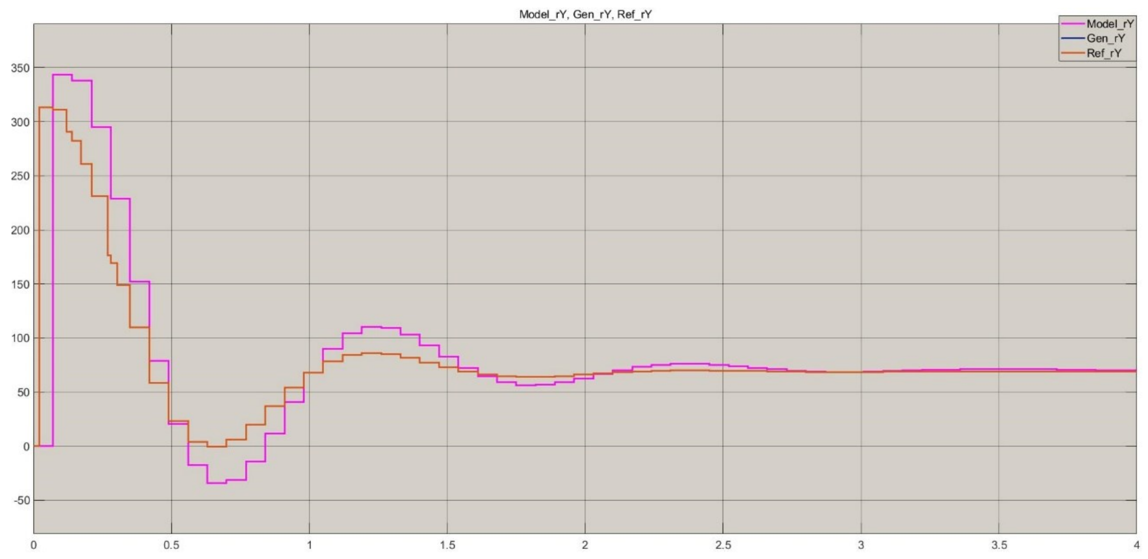


Figure 50, *Controller output rY. Generated and reference overlap*

Results reveal that generated controller and reference are performing identically. Differences can be seen between the model and SoftPLC. The delayed response at the beginning and skipped sync signals reveal that there is a synchronization issue with the test setup. The most likely reason for these differences is that the model block-parameters sample-time was 70ms which was also set in PLC cycle time. But in the co-simulation, model sample time should have been faster or “inherited” as the “sync”-signal from the plc was already controlling the execution. Because of two execution controls, the Simulink model skipped execution sometimes and synchronization did not completely function as expected. Figure 50 confirms this. Regardless of this difference, it can be concluded that each system has a similar response, and generated controller performs as the reference controller which is the goal. Re-testing to fix the synchronization issue seems needless and can be overlooked.

5. DISCUSSION

As the thesis topic looked like a clear subject to study and finding information on this specific problem from different sources should be relatively easy, this was not the case. PLC development seems to fall in between traditional high-level and embedded software development, which made finding PLC-specific studies complicated. This problem was overcome by finding related subjects and handpicking applicable parts from embedded software development and "normal" software development. The re-engineering part is quite a large topic and is quite neglected in the literature. The reason for this could be that re-engineering is still at most an afterthought in software development.

In the case study, Re-engineering two parts of the original controller with methods and the proposed process seemed to function well overall. In this case, requirements were purely copied from reference code comments at the beginning but looking through the code line-by-line revealed that all the functionality was not documented/commented on. This seems to indicate that the original code has seen some changes in its lifetime, as they usually do, and some things have been implemented after-hours. In this case, the "undocumented" part was small enough that it would have not made a difference in the end but following this proposed process, these kinds of requirements or specifications can be easily identified and updated to a new controller.

Modeling the controller to Simulink was done based on new requirements. Normally the old design can be used as a guide for modeling but in this case, following the original design was a mistake. This made the StartingOfMotion-model complex, and it should have been redesigned altogether and changed completely to use Stateflow functions. Stateflow integration works well with the PLC coder -addon and it would have made event / time-based controls easier. For algorithmic controls, the traditional Simulink model with user-defined systems is simple to use and worked well with PI-controller and PLC coder-addon.

The verification between each phase confirmed that the artifact produced from each step is working as expected before proceeding with the process. Re-engineering added one verification step to the traditional V-model. In this case, both controller types were verified to be compatible with reference and could be used in upcoming re-engineering activities. This verification process was most important in re-engineering. Especially in SoM-function, the model seemed to need refactoring right away but was still functioning very similarly and within acceptable limits to the reference controller. This grew confidence in the process itself, as the model worked although the expected result was the opposite.

The simplicity of the case study did cause some drawbacks, mostly from a practical point of view. Although this workflow worked fine for this simple case, it is expected that most real-life situations would not be straightforward. How much more complicated is the process when reference software is high in complexity and done originally with some other platform entirely? This would make the verification process more complicated. Does the more complicated workflow bring the wanted results if the incremental re-engineering method is used? This could mean that the controller needs to be split into two parts, one using MBD and one still developed traditionally. This would make further development unnecessarily complicated when two different workflows are used.

Answering these kinds of questions is hard generally. It could be assumed that transition to MBD has many software qualities improving properties when further developing the software and the proposed re-engineering model, it should increase confidence that all necessary properties are implemented when transitioning to MBD. Another clear benefit is that MBD separates the model from a platform, changing to any other code-generation supported platform is much simpler. But nothing comes without a drawback. Transitioning to MBD is an extensive process, at least with the proposed model. Furthermore, even if this re-engineering process would be only used in larger parts of the software and individual functions would be just tested within the Simulink model, still the process looks time-consuming.

In my personal opinion, the visual method of solving complex control problems is not always the best as there is much more freedom than in text-based code and even some simple problems could take much more visual space than in text format which may affect readability. Another issue that I had with using Simulink, was complicated compile errors. Compile errors were usually of such nature that required Simulink-specific understanding. But this is probably not a large issue and the benefits most likely outweigh this drawback. Overall, using Simulink is straightforward and I believe any software engineer should not have a hard time using this tool, for added benefit, engineers without a software development background should be comfortable as well.

Co-operative development was not tested, which is crucial in any software development with multiple users and should be investigated before diving into MBD. Another possible issue resides in Matlab and Simulink itself; PLC coder is part of the closed source software and adding this as an essential part of the toolchain is problematic. Alternatives to Simulink, at least when writing this, are not as developed and lack features. If MBD does not work as wanted in the long run, changing back to traditional development could cause problems as all the development work is done with Simulink so far and the generated code is not the best starting point for traditional programming.

6. CONCLUSION

This thesis's main objective was to find out how to re-engineer a controller with a model-based design. To answer these questions, a re-engineering process model for MBD was introduced through literature research and a case study was performed with actual controller as reference. The chosen path for answering these questions was complex but should have given the reader understanding of how to transform controller software development to benefit from model-based design through re-engineering.

As a result of this thesis, re-engineering was combined with the V-model software development process and the case-study part proved that the introduced re-engineering process is applicable. Although the re-engineering process was applicable in the simple case study, some costly drawbacks were found. In contrast to benefits, there were some drawbacks. The main issue with this process is the complexity. This would make the process resource-intensive which reduces willingness to use the re-engineering model in full. The case study focused on verification which could be thought of as the most valuable part of re-engineering and different approaches were introduced to this phase. In the end, verifications were completed successfully against the reference controller although a simple case study did not address all the practical questions which could arise. The case study can act as an example of how one could approach this kind of problem, although a more detailed study would be necessary to cover the topic more broadly.

Although model-based design and re-engineering are separate topics and each could be studied separately with more detail, the aim was to combine these topics into one thesis. This combination made the subject a little shallow but should work as a starting point for a more detailed study. Another possibility to further continue this study, implementing safety functions with model-based design and other IEC 61508 Functional Safety-related topics should be interesting as PLCs are often used in safety systems. Verification methods introduced in this thesis could function as a starting point as preliminary lookup proposed that these kinds of verification methods could function with functional safety as well.

7. REFERENCES

[1] W. Bolton, Control systems, 1st ed. Newnes, Oxford, 2002, p.6-7

[2] W. Bolton, Programmable Logic Controllers, Sixth ed. Newnes, 2015, p.4.

[3] A. Crespo, P. Albertos and J. Simo', Embedded Control Systems: From Design To Implementation, Symp. Cost Oriented Automation, 2007, p.2

[4] Naijun Zhan, Shuling wang, Hengjun Zhao, Formal Verification of Simulink/Stateflow Diagrams, Springer, 2017, p.3-4, 8

[5] Clarus Concept of Operations, The National Technical Information Service, Springfield 2005. p.20

[6] V. Socci, Implementing a model-based design and test workflow, IEEE International Symposium on Systems Engineering (ISSE), 2015 p.2.

[7] : Tatiana Kelemenová, Michal Kelemen, Ľubica Miková, Erik Prada, Tomáš Lipták, František Menda and Vladislav Maxim, Model Based Design and HIL Simulations, Iss. Mechanical Engineering 1, 2013, http://pubs.sciepub.com/ajme/1/7/25/.

[8] Global Harmonization Task Force - Quality Management Systems - Process Validation Guidance (GHTF/SG3/N99-10:2004 (Edition 2) p.3.

[9] IEEE Standard Glossary of Software Engineering Terminology, in: IEEE Std 610.12-1990, 1990, p.1-84.

[10] Mathworks, Verification and Validation, (Accessed: 24 March 2022) https://www.mathworks.com/help/slcheck/verification-and-validation.html.

[11] Lettnin, D. & Winterholer, M., Embedded Software Verification and Debugging. 1st ed. New York, NY: Springer New York., 2017, p.10-11

[12] A. Abran, J.W. Moore, SWEBOK 2004: Guide to the Software Engineering Body of Knowledge, IEEE Computer Society Press, 2004, p.4-1.

[13] Y. Singh, Software Testing, Cambridge: Cambridge University, 2011, p.27, 231.

[14] P. Tonella, A. Potrich, Reverse Engineering of Object Oriented Code, 1st ed. Springer New York, New York, NY, 2005. p.1

[15] Dr. Linda H. Rosenberg, Software Re-engineering, Software Assurance Technology Center,p 3 - 10.

[16] M. B. Younis, G. Frey, UML-based Approach for the Re-Engineering of PLC Programs, IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics, p. 1-6.

[17] S. Hassan, U. Qamar, T. Hassan, M. Waqas, Software Reverse Engineering to Requirement Engineering for Evolution of Legacy System, 2015 5th International Conference on IT Convergence and Security (ICITCS), p. 1-2.

[18] Syed Ahsan Fahmi, Ho-Jin Choi, Software Reverse Engineering to Requirements, Information and Communications University, 2007, p.1-6

[19] RBCS 2014 The Agile V Model. (Accessed: 24 March 2022) https://rbcs-us.com/site/assets/files/1203/agile-v-model.pdf

[20] Mathworks, Interface Specification Using Bus Objects, (Accessed: 24 March 2022) https://www.mathworks.com/help/simulink/slref/interface-specification-using-bus-objects.html.

[21] J. Machado, B.Denis, J. Lesage, J. Faure, J.C.L Ferreida Da Silva, Increasing the efficiency of PLC Program Verification using a plant model, Mechanical Engineering Department, University of Minho, 2003, p.6.

[22] Matworks, Test harness, (Accessed: 24 March 2022) https://ww2.mathworks.cn/help/sltest/ug/test-harness-construction-for-specific-model-elements.html.

[23] Mathworks, PLC coder limitations, (Accessed: 24 March 2022) https://www.mathworks.com/help/plccoder/ug/structured-text-code-generation-limitations.html.

[24] M. Conrad, G. Sandmann, A Verification and Validation Workflow for IEC 61508 Applications, Mathworks, 2009, p.2

[25] Mathworks, Co-Simulation Execution, (Accessed: 24 March 2022) https://www.mathworks.com/help/simulink/ug/co-simulation-execution.html

APPENDICES

Appendix 1: Structured text - Testbench code used in verification

```
IF testVerify THEN
       IF testCycleNum < 201 THEN
           (* TEST CYCLE SETUP  *)
           cycle_bCommandEnable := tb_bCommandEnable[testCycleNum];
           cycle_CommandStart := tb_CommandStart[testCycleNum];
           cycle_rActualVel := tb_rActualVel[testCycleNum];
           cycle_rTargetVel := tb_rTargetVel[testCycleNum];
           cycle_rActualPos := tb_rActualPos[testCycleNum];
           cycle_rTargetPos := tb_rTargetPos[testCycleNum];
           cycle_bStandstill := tb_bStandstill[testCycleNum];
           cycle_rVelocityTargetMinimum := tb_rVelocityTargetMinimum[testCycleNum];
           cycle_rPositionErrorOut := tb_rPositionErrorOut[testCycleNum];
           cycle_rCheckMovementTolerance := tb_rCheckMovementTolerance[testCycleNum];
           cycle_rPositioningWindow := tb_rPositioningWindow[testCycleNum];
           cycle_iVelocityProfileState := tb_iVelocityProfileState[testCycleNum];
           cycle_rDirectionDetectLimit := tb_rDirectionDetectLimit[testCycleNum];
           cycle_rLastStoredStartingPos := tb_rLastStoredStartingPos[testCycleNum];
           cycle_rControlStartPosition := tb_rControlStartPosition[testCycleNum];
           cycle_bMovementStartingError := tb_bMovementStartingError[testCycleNum];
           cycle_bCheckMovementContinuos := tb_bCheckMovementContinuos[testCycleNum];
           cycle_bDirectionError := tb_bDirectionError[testCycleNum];
           IF testCycleNum = 0 THEN
               (* INIT  *)
               gen_StartingOfMotion0(ssMethodType  :=  SS_INITIALIZE,  bCommandEnable  :=  cycle_bCommandEnable,
CommandStart := cycle_CommandStart, rActualVel := cycle_rActualVel, rTargetVel := cycle_rTargetVel, rActualPos
:= cycle_rActualPos, rTargetPos := cycle_rTargetPos, bStandstill := cycle_bStandstill, rVelocityTargetMinimum
:= cycle_rVelocityTargetMinimum, rPositionErrorOut := cycle_rPositionErrorOut, rCheckMovementTolerance := cy-
cle_rCheckMovementTolerance,  rPositioningWindow  :=  cycle_rPositioningWindow,  iVelocityProfileState  :=  cy-
cle_iVelocityProfileState, rDirectionDetectLimit := cycle_rDirectionDetectLimit);
               gen_rLastStoredStartingPos := gen_StartingOfMotion0.rLastStoredStartingPos;
               gen_rControlStartPosition := gen_StartingOfMotion0.rControlStartPosition;
               gen_bMovementStartingError := gen_StartingOfMotion0.bMovementStartingError;
               gen_bCheckMovementContinuos := gen_StartingOfMotion0.bCheckMovementContinuos;
               gen_bDirectionError := gen_StartingOfMotion0.bDirectionError;
           END_IF;
           (* STEP  *)
           gen_StartingOfMotion0(ssMethodType  :=  SS_STEP,  bCommandEnable  :=  cycle_bCommandEnable,  Command-
Start := cycle_CommandStart, rActualVel := cycle_rActualVel, rTargetVel := cycle_rTargetVel, rActualPos := cy-
cle_rActualPos,  rTargetPos  :=  cycle_rTargetPos,  bStandstill  :=  cycle_bStandstill,  rVelocityTargetMinimum  :=
cycle_rVelocityTargetMinimum,  rPositionErrorOut  :=  cycle_rPositionErrorOut,  rCheckMovementTolerance  :=  cy-
cle_rCheckMovementTolerance,  rPositioningWindow  :=  cycle_rPositioningWindow,  iVelocityProfileState  :=  cy-
cle_iVelocityProfileState, rDirectionDetectLimit := cycle_rDirectionDetectLimit);
           gen_rLastStoredStartingPos := gen_StartingOfMotion0.rLastStoredStartingPos;
           gen_rControlStartPosition := gen_StartingOfMotion0.rControlStartPosition;
           gen_bMovementStartingError := gen_StartingOfMotion0.bMovementStartingError;
           gen_bCheckMovementContinuos := gen_StartingOfMotion0.bCheckMovementContinuos;
           gen_bDirectionError := gen_StartingOfMotion0.bDirectionError;

                ref_StartingOfMotion0(bCommandEnable     :=     cycle_bCommandEnable,     bCommandStart     :=
cycle_CommandStart,   rActualVel   :=   cycle_rActualVel,   rTargetVel   :=   cycle_rTargetVel,   rActualPos   :=
cycle_rActualPos, rTargetPos := cycle_rTargetPos, bStandstill := cycle_bStandstill, rVelocityTargetMinimum :=
cycle_rVelocityTargetMinimum,   rPositionErrorOut   :=   cycle_rPositionErrorOut,   rCheckMovementTolerance   :=
cycle_rCheckMovementTolerance,   rPositioningWindow   :=   cycle_rPositioningWindow,   iVelocityProfileState   :=
cycle_iVelocityProfileState, rDirectionDetectLimit := cycle_rDirectionDetectLimit);
                ref_rLastStoredStartingPos := ref_StartingOfMotion0.rLastStoredStartingPos;
           ref_rControlStartPosition := ref_StartingOfMotion0.rControlStartPosition;
           ref_bMovementStartingError := ref_StartingOfMotion0.bMovementStartingError;
           ref_bCheckMovementContinious := ref_StartingOfMotion0.bCheckMovementContinuos;
           ref_bDirectionError := ref_StartingOfMotion0.bDirectionError;

           (* VERIFY  *)
           IF testVerify THEN
               IF ABS(cycle_rLastStoredStartingPos) < 1.0E-11 THEN
                   IF ABS(gen_rLastStoredStartingPos) > 0.0001 THEN
                       testVerify := FALSE;
                       testVarName := 'rLastStoredStartingPos';
                   END_IF;
               ELSIF    ABS(gen_rLastStoredStartingPos    -    cycle_rLastStoredStartingPos)    >    (0.0001    *
ABS(cycle_rLastStoredStartingPos)) THEN
                   testVerify := FALSE;
                   testVarName := 'rLastStoredStartingPos';
               END_IF;
           END_IF;
           IF testVerify THEN
               IF ABS(cycle_rControlStartPosition) < 1.0E-11 THEN
                   IF ABS(gen_rControlStartPosition) > 0.0001 THEN
                       testVerify := FALSE;
                       testVarName := 'rControlStartPosition';
                   END_IF;
               ELSIF    ABS(gen_rControlStartPosition    -    cycle_rControlStartPosition)    >    (0.0001    *
ABS(cycle_rControlStartPosition)) THEN
                   testVerify := FALSE;
                   testVarName := 'rControlStartPosition';
               END_IF;
           END_IF;
           IF testVerify AND (gen_bMovementStartingError <> cycle_bMovementStartingError) THEN
               testVerify := FALSE;
               testVarName := 'bMovementStartingError';
           END_IF;
           IF testVerify AND (gen_bCheckMovementContinuos <> cycle_bCheckMovementContinuos) THEN

               testVarName := 'bCheckMovementContinuos';
           END_IF;
           IF testVerify AND (gen_bDirectionError <> cycle_bDirectionError) THEN
               testVerify := FALSE;
               testVarName := 'bDirectionError';
           END_IF;
           testCycleNum := testCycleNum + 1;
       END_IF;
   END_IF;
```

Appendix 2: Matlab script: OPC-UA linking PLC and matlab

```
classdef PLCCOSIM < matlab.System
    % PLCCOSIM Add summary here
    %
    % This template includes the minimum set of functions required
    % to define a System object with discrete state.

    % Public, tunable properties
    properties

    end

    properties(DiscreteState)
        CycleNum;
    end

    % Pre-computed constants
    properties(Access = private)
        UAObj;
        DeviceNode;
        Cycle_U;
        Cycle_Y;
        Cycle_rConstSpeedTime;
        Cycle_rAcc;
        Cycle_rDec;
        Cycle_diDistance;
        Cycle_rInitialVel;
        Cycle_rMinVel;
        Cycle_rMaxVel;
        Cycle_rVelTarget_gen;
        Cycle_bSuccess_gen;
        Cycle_rVelTarget_ref;
        Cycle_bSuccess_ref;
        TestCycleNum;
        PreviousCycleNum;
    end

    methods(Access = protected)
        function setupImpl(obj)
            % Perform one-time calculations, such as computing constants
            % init opc UA server connection
```

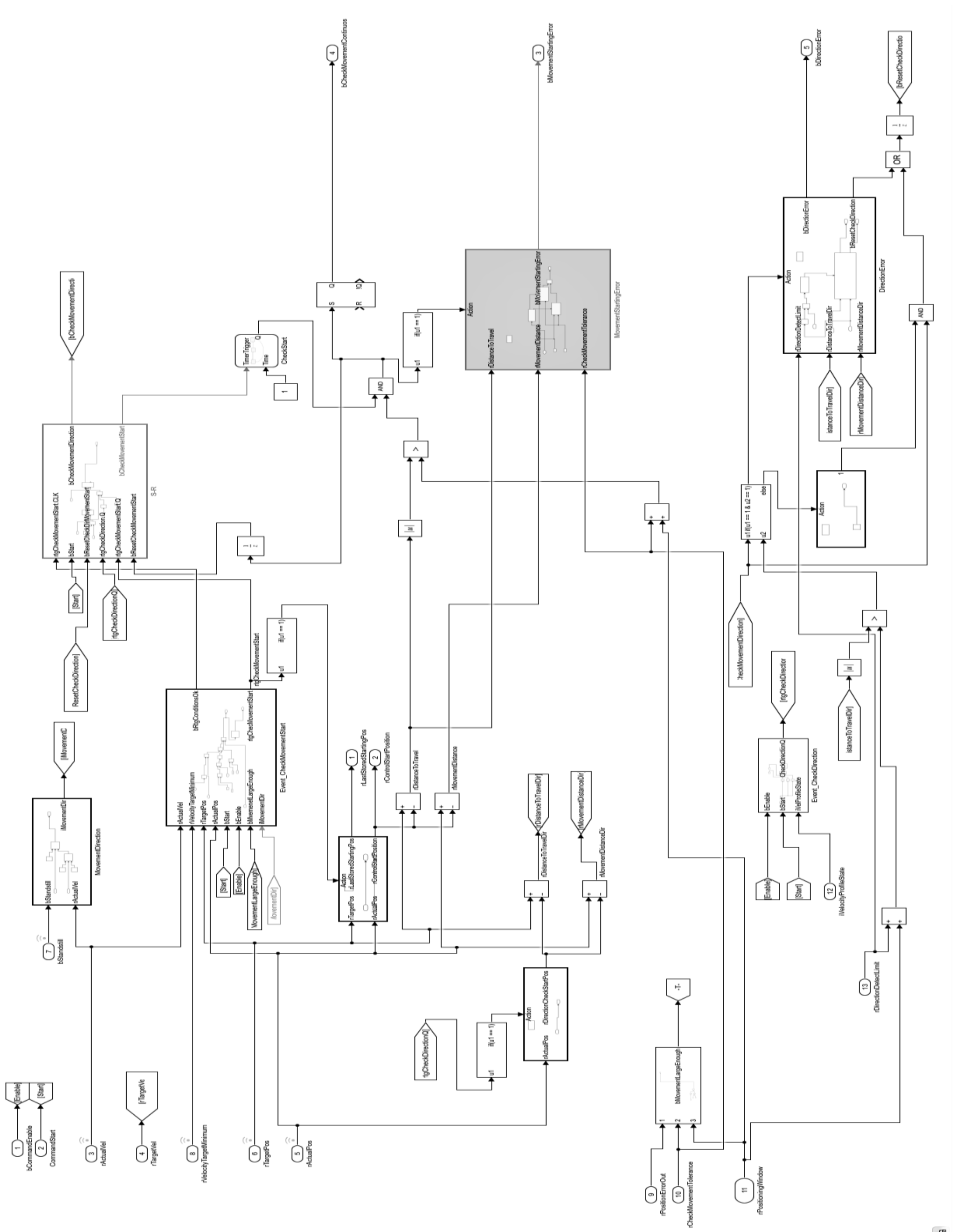
```
                obj.UAObj = opcua('opc.tcp://192.168.254.127:4840');
                obj.UAObj.Endpoints(1) = [];
                connect(obj.UAObj);
                obj.DeviceNode = findNodeByName(obj.UAObj.Namespace,'DeviceSet','-once');
                obj.Cycle_U = findNodeByName(obj.DeviceNode,'cycle_U');
                obj.Cycle_Y = findNodeByName(obj.DeviceNode,'cycle_Y');
                obj.Cycle_rConstSpeedTime                    =                    findNodeBy-
Name(obj.DeviceNode,'cycle_rConstSpeedTime');
                obj.Cycle_rAcc = findNodeByName(obj.DeviceNode,'cycle_rAcc');
                obj.Cycle_rDec = findNodeByName(obj.DeviceNode,'cycle_rDec');
                obj.Cycle_diDistance = findNodeByName(obj.DeviceNode,'cycle_diDistance');
                obj.Cycle_rInitialVel = findNodeByName(obj.DeviceNode,'cycle_rInitialVel');
                obj.Cycle_rMinVel = findNodeByName(obj.DeviceNode,'cycle_rMinVel');
                obj.Cycle_rMaxVel = findNodeByName(obj.DeviceNode,'cycle_rMaxVel');
                obj.Cycle_rVelTarget_gen = findNodeByName(obj.DeviceNode,'cycle_rVelTarget_gen');
                obj.Cycle_bSuccess_gen = findNodeByName(obj.DeviceNode,'cycle_bSuccess_gen');
                obj.Cycle_rVelTarget_ref = findNodeByName(obj.DeviceNode,'cycle_rVelTarget_ref');
                obj.Cycle_bSuccess_ref = findNodeByName(obj.DeviceNode,'cycle_bSuccess_ref');
                obj.TestCycleNum = findNodeByName(obj.DeviceNode,'testCycleNum');
                obj.PreviousCycleNum = findNodeByName(obj.DeviceNode,'previousCycleNum');
            end

            function  [rVeltarget_gen,bSuccess_gen,rVeltarget_ref,bSuccess_ref]  =  stepImpl(obj,
rConstSpeedTime, rAcc, rDec, diDistance,  rInitialVel, rMinVel, rMaxVel)
                % Implement algorithm. Calculate y as a function of input u and
                % discrete states.
                obj.CycleNum = obj.CycleNum+1;
                writeValue(obj.UAObj, obj.Cycle_rConstSpeedTime, rConstSpeedTime);
                writeValue(obj.UAObj, obj.Cycle_rAcc, rAcc);
                writeValue(obj.UAObj, obj.Cycle_rDec, rDec);
                writeValue(obj.UAObj, obj.Cycle_diDistance, diDistance);
                writeValue(obj.UAObj, obj.Cycle_rInitialVel, rInitialVel);
                writeValue(obj.UAObj, obj.Cycle_rMinVel, rMinVel);
                writeValue(obj.UAObj, obj.Cycle_rMaxVel, rMaxVel);
                writeValue(obj.UAObj, obj.TestCycleNum, obj.CycleNum);

                valueUpdated = false;
                for rct = 1:100
                    previousCycleNumValue = readValue(obj.UAObj, obj.PreviousCycleNum);
                    if previousCycleNumValue == obj.CycleNum
                        valueUpdated = true;
                        rVeltarget_gen = readValue(obj.UAObj, obj.Cycle_rVelTarget_gen);
                        bSuccess_gen = readValue(obj.UAObj, obj.Cycle_bSuccess_gen);
                        rVeltarget_ref = readValue(obj.UAObj, obj.Cycle_rVelTarget_ref);
                        bSuccess_ref = readValue(obj.UAObj, obj.Cycle_bSuccess_ref);
                        break
                    end
                    pause(0.001)
                end

                if ~valueUpdated
                    error('not get the value for cycle number %d', obj.CycleNum);
                end
            end

            function resetImpl(obj)
                % Initialize / reset discrete-state properties
                obj.CycleNum = 0;
                writeValue(obj.UAObj, obj.TestCycleNum, obj.CycleNum);
                writeValue(obj.UAObj, obj.PreviousCycleNum, obj.CycleNum);
            end

            function [out1,out2,out3,out4] = getOutputSizeImpl(obj)
                out1 = propagatedInputSize(obj,1);
                out2 = propagatedInputSize(obj,2);
                out3 = propagatedInputSize(obj,1);
                out4 = propagatedInputSize(obj,2);

            end

            function [out1,out2,out3,out4] = getOutputDataTypeImpl(obj)
                out1 = 'single';
                out2 = 'boolean';
                out3 = 'single';
                out4 = 'boolean';
            end

            function [out1,out2,out3,out4]= isOutputComplexImpl(obj)
                out1 = false;
                out2 = false;
                out3 = false;
                out4 = false;
            end

            function [out1,out2,out3,out4] = isOutputFixedSizeImpl(obj)
                out1 = true;
                out2 = true;
                out3 = true;
                out4 = true;

            end

            function s = getDiscreteStateImpl(obj)
                s = obj.CycleNum;
            end
        end
    end
```

Appendix 3. Structured text: PI-controller generated from model

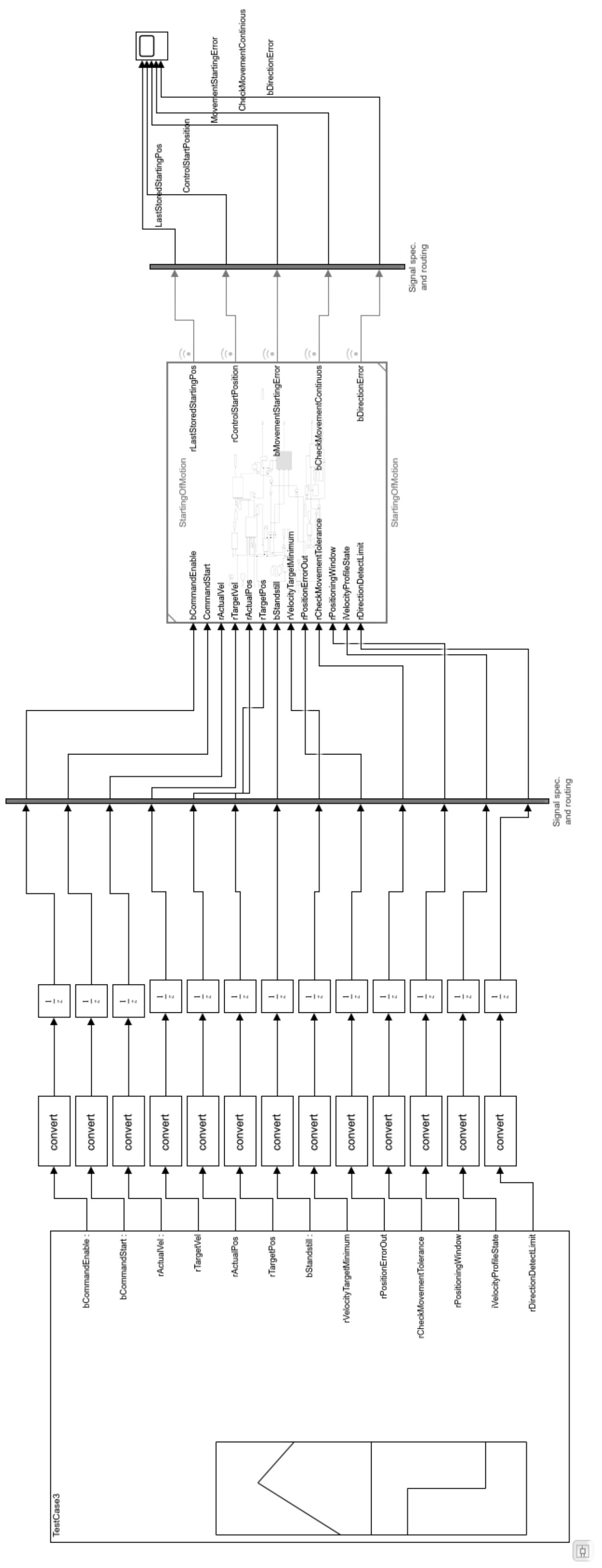
```
IF need_init THEN
    (* SystemInitialize for MATLAB Function: '<Root>/PI controller' *)
    init_not_empty := FALSE;
    need_init := FALSE;
END_IF;

(* MATLAB Function: '<Root>/PI controller' *)

(* rIn : REAL;                                   // Controller input (setpoint - actual meas-
ured value) *)
(* rKP : REAL := 1.0;                    // Gain *)
(* rKI : REAL := 1.0;                    // Integral Ki (Integral time) *)
(* MATLAB Function 'PI controller': '<S1>:1' *)
(* rLIM_L : REAL := -1.0E38;      // Lower limit for output *)
(* rLIM_H : REAL := 1.0E38;              // Higher limit for output *)
(* bRST : BOOL;                                  // Resets internal integrator *)
(* rY : REAL;                                    // Controller output *)
(* bLIM : BOOL;                                  // indicates that the Output Y runs to one
of the limits rLIM_L or rLIM_H *)
(* init: BOOL; *)
(* tx: DWORD; *)
(* tc : REAL; *)
(* t_last: DWORD; *)
(* in_last : REAL; *)
(* i: REAL; *)
(* p: REAL; *)
(* (* initialize at power_up *) *)
(* '<S1>:1:31' if isempty(init) || bRST == 1 *)

IF ( NOT init_not_empty) OR (bRST = 1.0) THEN
    (* '<S1>:1:33' init = 1; *)
    init_not_empty := TRUE;
    (* '<S1>:1:34' in_last = rIn; *)
    in_last := rIn;
    (* '<S1>:1:35' t_last = tCurtime; *)
    t_last := tCurtime;
    (* '<S1>:1:36' i = 0.0; *)
    i := 0.0;
    (* '<S1>:1:37' tc = 0.0; *)
    (* '<S1>:1:38' rY = 0; *)
    rY := 0.0;
    (* Outport: '<Root>/bLim' *)
    (* '<S1>:1:39' bLim = 0; *)
    bLim := 0.0;
```

```
ELSE
    (* '<S1>:1:40' else *)
    (* ( read last cycle time in Microseconds *)
    (* '<S1>:1:42' tx = tCurtime; *)
    (* '<S1>:1:43' tc = tx - t_last; *)
    tc := tCurtime - t_last;
    (* '<S1>:1:44' t_last = tx; *)
    t_last := tCurtime;
    (* (* calculate proportional part *) *)
    (* '<S1>:1:47' p = rKP * rIn; *)
    b_p := rKP * rIn;
    (* (* run integrator *) *)
    (* '<S1>:1:50' i = (rIn + in_last) * 5.0E-7 * rKI * tc + i; *)
    i := ((((rIn + in_last) * 5.0E-7) * rKI) * tc) + i;
    (* '<S1>:1:51' in_last = rIn; *)
    in_last := rIn;
    (* (* calculate output Y *) *)
    (* '<S1>:1:55' rY = p + i; *)
    rY := b_p + i;
    (* (* check output for limits *) *)
    (* '<S1>:1:58' if rY >= rLIM_H *)

    IF rY >= rLIM_H THEN
        (* '<S1>:1:59' rY = rLIM_H; *)
        rY := rLIM_H;
        (* '<S1>:1:60' if rKI ~= 0.0 *)

        IF rKI <> 0.0 THEN
            (* '<S1>:1:61' i = rLIM_H - p; *)
            i := rLIM_H - b_p;
        ELSE
            (* '<S1>:1:62' else *)
            (* '<S1>:1:63' i = 0.0; *)
            i := 0.0;
        END_IF;

        (* Outport: '<Root>/bLim' *)
        (* '<S1>:1:65' bLim = 1; *)
        bLim := 1.0;
    ELSIF rY <= rLIM_L THEN
        (* '<S1>:1:66' elseif rY <= rLIM_L *)
        (* '<S1>:1:67' rY = rLIM_L; *)
        rY := rLIM_L;
        (* '<S1>:1:68' if rKI ~= 0.0 *)

        IF rKI <> 0.0 THEN
            (* '<S1>:1:69' i = rLIM_L - p; *)
            i := rLIM_L - b_p;
        ELSE
            (* '<S1>:1:70' else *)
            (* '<S1>:1:71' i = 0.0; *)
            i := 0.0;
        END_IF;

        (* Outport: '<Root>/bLim' *)
        (* '<S1>:1:73' bLim = 1; *)
        bLim := 1.0;
    ELSE
        (* Outport: '<Root>/bLim' *)
        (* '<S1>:1:74' else *)
        (* '<S1>:1:75' bLim = 0; *)
        bLim := 0.0;
    END_IF;

END_IF;

(* End of MATLAB Function: '<Root>/PI controller' *)
```

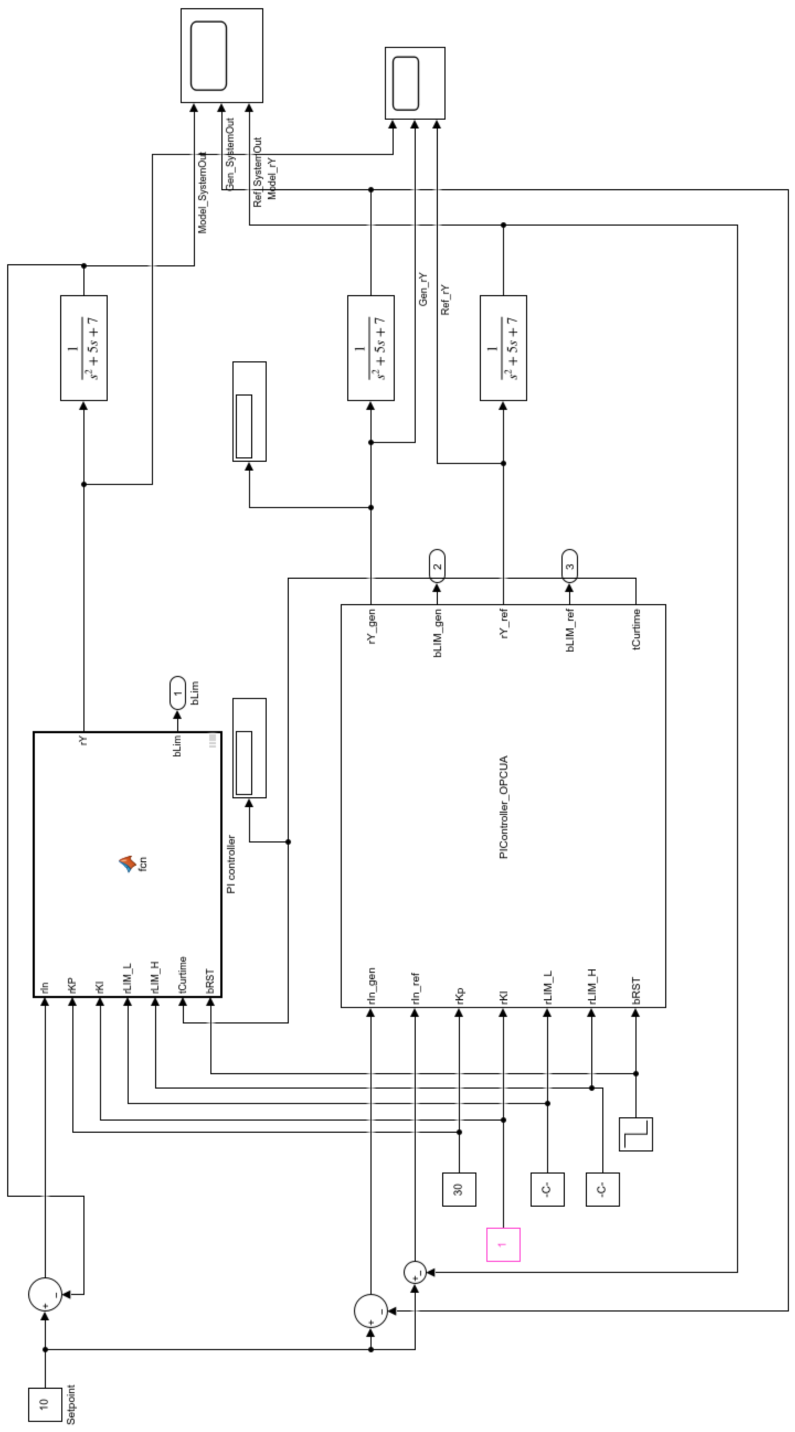
Appendix 4. StartingOfMotion-model



Appendix 5. StartingOfMotion-model test harness



Appendix 6. Cosimulation-model for PI-controller


Model_SystemOut
Gen_SystemOut
Ref_SystemOut
Model_rY
Gen_rY
Ref_rY
$\frac{1}{s^2+5s+7}$
$\frac{1}{s^2+5s+7}$
$\frac{1}{s^2+5s+7}$
rY_gen
bLIM_gen
rY_ref
bLIM_ref
tCurtime
2
3
1
bLim
rY
fcn
PI controller
rIn
rKP
rKI
rLIM_L
rLIM_H
tCurtime
bRST
PIController_OPCUA
rIn_gen
rIn_ref
rKp
rKI
rLIM_L
rLIM_H
bRST
30
-C-
-C-
1
10
Setpoint