TAMPERE UNIVERSITY OF TECHNOLOGY

MATIAS SOINI

Modeling and Simulation Practices in Control System Software Development

M.Sc. Thesis

Examiner: Hannu Koivisto
Examiner and topic approved by the Faculty of Engineering Sciences on 09.04.2014

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Automaatiotekniikan koulutusohjelma
MATIAS SOINI: Mallinnus- ja Simulointikäytännöt Ohjausjärjestelmien Ohjelmistokehityksessä
Diplomityö, 73 sivua, 4 liitesivua
Toukokuu 2014
Pääaine: Automaation ohjelmistotekniikka
Tarkastajat: Hannu Koivisto
Avainsanat: mallipohjainen suunnittelu, automaattinen koodingenerointi, mallinnus, simulointi, Simulink, Modelica

Ohjelmistokehityksen rooli nykyaikaisten liikkuvien työkoneiden ohjausjärjestelmäkehityksessä on suuri. Tästä johtuen on ohjelmistokehitysprosessia tavoitteellista pyrkiä tehostamaan. Mallinnus- ja simulointityökalujen avulla voidaan ohjausjärjestelmän osia suunnitella ja testata. Mallipohjainen suunnittelu on kehitysmenetelmä, jossa mallinnustyökalulla kehitettyjä malleja käytetään keskeisenä osana kehitysprosessia. Tämä opinnäytetyö tutkii mahdollisuuksia hyödyntää mallipohjaisen suunnittelun menetelmiä ohjausjärjestelmäkehityksessä. Erityisesti työssä tarkastellaan automaattista koodingenerointia, menetelmää jossa suunnittelumalleista voidaan automaattisesti tuottaa ohjelmakoodia.
Mallipohjainen suunnittelu on hyvin työkaluriippuvaista, joten tässä opinnäytetyössä esitellään markkinoilla olevia työkaluja sekä tutkitaan niiden ominaisuuksia suunnitteluesimerkin kautta. Käsiteltäviä työkaluja ovat Simulink ja OpenModelica. Työssä esitellään mallipohjaisten suunnittelumenetelmien hyötyjä ja haasteita ohjelmistokehitysprosessiin liittyen, sekä tutkitaan työkaluja ja menetelmiä näiden hyötyjen saavuttamiseksi ja haasteiden ratkaisemiseksi.
Työkaluista Simulinkin todettiin olevan soveltuva mallipohjaisuun suunnitteluun ja mahdollistavan automaattisen koodingeneroinnin. OpenModelican mallinnus- ja simulointiympäristö todettiin keskeneräiseksi ja koodingenerointiominaisuus puuttuvaksi. Mallinnustyön tueksi suunniteltiin pohja mallinnuksen tyylisääntökokoelmalle sekä hierarkiselle mallirakenteelle Simulinkiä käytettäessä. Lisäksi esiteltiin mahdollisuuksia jäljitettävyyden toteuttamiseen ja mallien dokumentointiin. Viimeisenä tutkittiin Simulinkin koodigeneraattorin luotettavuutta ja suorituskykyä. Olemassaolevien tutkimustulosten perusteella todettiin koodingeneraattorin olevan luotettava ja toimivan ennustettavasti. Koodigeneraattorin tuottaman ohjelmakoodin suorituskyvyn todettiin olevan verrattavissa ohjelmoijan tuottamaan koodiin.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Automation Technology
MATIAS SOINI: Modeling and Simulation Practices in Control System Software Development
Master of Science Thesis, 73 pages, 4 appendix pages
May 2014
Major: Software in automation
Examiner: Hannu Koivisto
Keywords: Model-based design, automatic code generation, modeling, simulation, Simulink, Modelica

Software development represents a significant portion of the total work effort in control system development, which is why improving the efficiency of the software development process is important. Modeling and simulation tools can be used for design and verification of parts of the control system. Model-based design is a development methodology, that presents models as a central concept in the development process. This thesis explores the opportunities model-based design presents for improving the efficiency of the control system development process. Specifically, the possibility of using automatic production code generation to generate program code representations of design models is of interest.

This thesis presents a selection of the tools available for model-based design and explores their capabilities through a design example. The tools presented are Simulink and OpenModelica. The benefits and challenges of model-based design are discussed with regards to the software development process. Tools and methods for achieving the benefits and addressing the challenges are explored.

Analysis of the tools concluded that Simulink is suitable for model-based design and enables automatic program code generation. OpenModelica was used for basic modeling and simulation work, but the development environment was not mature enough for production use and the tool lacked production code generation capabilities. Methods for supporting the use of modeling practices in control system development were presesented. A draft of a modeling guidelines collection was created and a template for the hierarchical structure of Simulink models was specified. Methods for implementing traceability and documenting models are also presented. Lastly, the reliability and performance of the Simulink code generator was addressed. Based on existing research, it could be deduced that the code generator was reliable and predictable. In terms of performance, the program code generated by the code generator was found to be comparable to code written by a programmer.

ACKNOWLEDGEMENTS

This thesis is done as a part of the Master's degree programme in automation technology at Tampere University of Technology with a Master's thesis grant from the Industrial Research Fund at Tampere University of Technology. It is done in collaboration with Sandvik Mining and Construction Oy in Tampere. The goal of the thesis was to explore the use of model-based design practices in control system development.

Firstly, I would like to thank the examiner of my thesis, professor Hannu Koivisto for his advise and feedback. At Sandvik, I would like to thank Joonas Kaarnametsä who has helped me and provided valuable feedback during the writing my thesis as well as everyone else that I have had the pleasure of working with. Lastly I would like to thank my parents who have supported me during my studies and my friends who have made my study years unforgettable.

CONTENTS

TERMINOLOGY

FSM	Finite-State Machine
HIL	Hardware-in-the-Loop
MBD	Model-Based Design
MBSE	Model-Based Software Engineering
PID	Proportional-Integral-Derivative (control)
PIL	Processor-in-the-Loop
R&D	Research and Development
SICA	Sandvik Intelligent Control system Architecture
SIL	Software-in-the-Loop
TTM	Time-to-Market
V&V	Verification and Validation

1. INTRODUCTION

The trend in technical industries is towards a shorter time-to-market and a more iterative design and development process. This allows companies to react to market realities and to meet the demands set by the growth of feature sets and increased complexity of systems. Reducing time-to-market requires improved efficiency and productivity. In the mechanical industry, the increasing amount of software in control systems enables companies to rapidly develop new machine functions using efficient software development methodologies without the need to undertake traditional, time-consuming mechanical engineering tasks.

This thesis explores the incorporation of modeling and simulation techniques to control system software development practices. In General, modeling and simulation can be understood as developing a model which represents the target system and using simulation practices to solve a specific problem related to that system [4, p. 3]. Model-based design is a software development methodology that emphasizes the use of models as a means of definition and communication. Model-based design enables techniques such as concept validation, early design verification, functional verification through simulation and automatic production code generation, which is of specific interest.

This thesis is done in co-operation with Sandvik Mining and Construction Oy, where there is interest for the model-based design approach and some pilot projects have been carried out. The thesis also continues the research presented in a previous M.Sc. thesis covering model-based design and rapid prototyping in control system design [24]. The goal is to investigate the current technologies in modeling and simulation and explore possibilities for their usage in control system development using existing tools. The tools covered in this thesis are Simulink and OpenModelica. The topics of specific interest are controller design and automatic code generation as well as the challenges they present in terms of modeling and the software development process.

The theoretical information covering software development and modeling and simulation is found in literature, which is plentiful. For the purposes of this thesis, generic software development theory is applicable to control system software development. For model-based design, existing research results along with some industry show cases will be used to establish an understanding of the state of the technology

and the applications of the methodology that exist in the industry and the results that have been achieved.

The second chapter introduces the concepts of control systems and control system software development. It also introduces modeling and simulation and explains the principles behind model-based design. Chapter 3 gives an overview of the state of the art in model-based design. It covers use cases from the industry, the current situation at Sandvik and the tools available on the market. Chapter 4 discusses the actualities of adopting model-based design practices in an existing software development process and outlines a process that incorporates model-based design and the challenges it presents. Chapter 5 presents an analysis of the capabilities of Simulink and OpenModelica through an example design problem with a summary at the end. Chapter 6 addresses the challenges related to using model-based design in control system software development that were identified in Chapter 4. Chapter 7 concludes the thesis by giving and overview of the results and specifies the need for future work.

2. CONTROL SYSTEM DEVELOPMENT AND MODELING AND SIMULATION

This chapter presents the principles of control system design, modeling, simulation and software development. The latter part of this chapter describes a method of integrating modeling and simulation practices into the control system software development process in the form of model-based design.

2.1 Control Systems

A system can be thought of as an entity that exhibits a cause-effect relationship between a provided stimulus and the observable response. The stimuli provided to a system are called inputs, and the observable responses are called outputs. A control system is a group of components, interconnected for the purpose of controlling the behavior of a system. A simple closed-loop feedback control system consists of a controller, an actuator, the target system and a sensor. In the context of a control system, the system whose behavior is being controlled is usually referred to as the process or the plant. The purpose of the control system is to regulate the outputs of the target system by controlling the inputs. The desired output response of the target system is given as an input to the control system, this input is called a set point. The controller uses an algorithm suitable for the characteristics of the target system to generate a control signal, which the actuator converts into a system stimulus. The response of the target system is measured by a sensor, which generates a measurement signal that can be compared to the desired output value, allowing the controller to react to the actual behavior of the system through a feedback connection. A block diagram of such a control system is presented in Figure 2.1. [9, pp. 2-3]

2.2 Digital control

In a digital control system, the controller is a digital computer. In its most basic form, a closed-loop digital control system differs from an analog closed-loop control system by having a digital controller and analog-to-digital (A/D) and digital-to-analog (D/A) converters for feedback input and control signal output. The digital

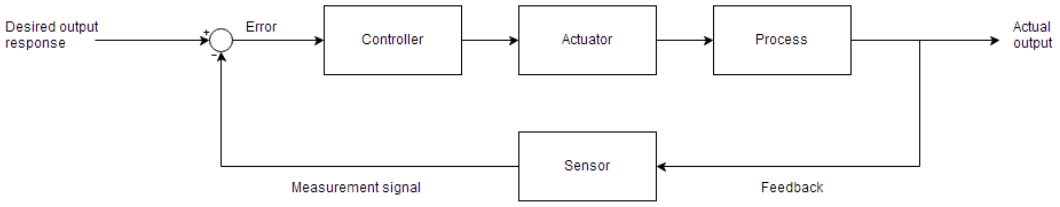


Figure 2.1: The block-diagram presentation of a closed-loop feedback control system, consisting of a controller, an actuator, a process and a sensor. [9, p. 3]

controller is programmed to execute a control algorithm periodically. The block diagram representation of a closed-loop digital control system is shown in Figure 2.2. A basic digital control computer contains a general purpose processor capable of executing program code and some form of memory for storing program code. It usually also contains A/D and D/A converters for connecting input and output signals and may offer connectivity to device communication buses such as CAN, Profibus or Ethernet that can be used for communication in distributed control systems. Depending on the intended application, a digital controller may also contain specialized signal processing units for performing certain types of calculations. Digital control is beneficial because a digital computer can be efficiently programmed to execute a variety of control algorithms and its operation can be adapted to different applications through parameterization, making it possible to conform with tight design-time requirements. The processing capabilities of digital controllers are also good and constantly improving due to advances in computer and digital systems technology. [12, pp. 1-3] A distributed control system is one where the control algorithm is executed on different physical control units.

The digital control computer processes digital signals, which are discrete and quantized. The value of a discrete signal is only updated once in a given sample period and is held constant for the rest of the period. A typical A/D converter samples a continuous signal at the beginning of each sample period and holds the output at that value through a zero-order hold circuit until it is updated again to create a discrete signal. [12, pp. 3-5] The value of a digital signal is also quantized to fit in a finite number of bits. The precision of quantized values is defined by the control computer's maximum operand size. While higher end digital controllers are able to perform calculations for 64-bit values, cheaper devices may be limited to 32, 16 or even 8 bits. Quantization introduces round-off errors which may cause noise in the control output and even instability in some cases, although these effects are mostly negligible for 64 or 32-bit controllers. [12, p. 425]

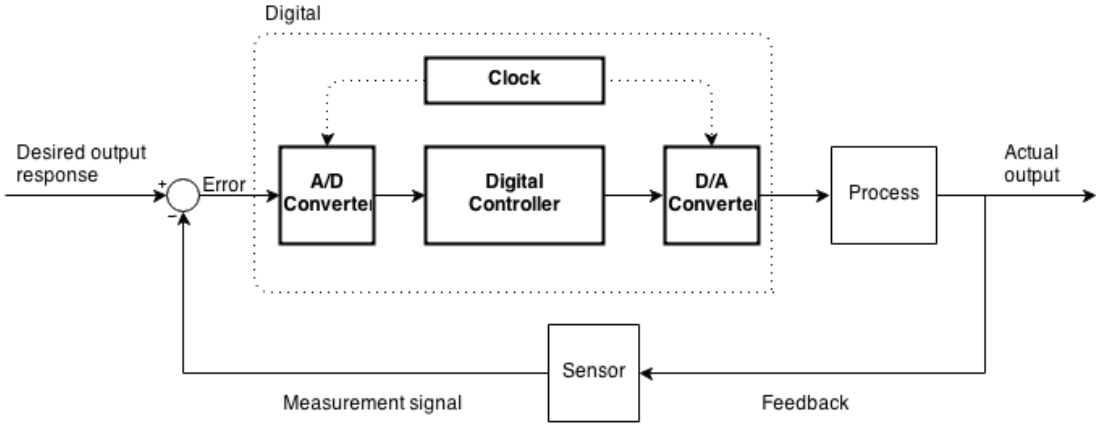


Figure 2.2: The block diagram representation of a closed-loop digital control system. [19, p. 12]

2.3 Modeling and Simulation

The purpose of a model is to act as a surrogate of an actual system in experimental studies. A model describes the behavior of the actual system. This description can be expressed in natural language, mathematical formalisms, rule-based formalisms or symbolically. The main reason for using models in experimental studies instead of actual systems, is the fact that behavioral data is usually much easier to acquire from models. Models may also be used for proof-of-concept studies where the system does not actually exist yet. Using models instead of actual systems can also allow the study to be carried out faster and with less of a financial or safety risk. In some cases, the use of a model instead of the actual system may even be mandatory due to the gravity of the aforementioned risk factors.

The model should accurately describe the behavior of the actual system on a level of detail that is adequate for the problem at hand. Developing the model with a sufficient level of detail ensures that it can provide the data necessary for analysis and problem solving. On the other hand the model should not be more detailed than is necessary to manage model complexity. This puts an emphasis on forming a clear understanding of the purpose of the model before undertaking any modeling work.

Models can be either dynamic or static, the difference being that the behavior of a dynamic model changes with respect to time where the behavior of a static model does not. Most models based on physical systems are dynamic. Models of physical processes are often based on mathematical equations, which in turn are based on the physical properties of the process. The behavior of a linear dynamic system can be described by a group of differential equations [12, p. 12].

In the context of an experimental study concerned with the behavior of a certain

system, the term simulation means exercising the model by driving its inputs and observing its behavior by monitoring its outputs. A dynamic system represented as a group of equations can be simulated using numerical methods [12, p. 11]. The basis for simulation activities is the specification for the model. The knowledge of how the model should behave when exercised in certain ways is extracted from the specification for the model. [4, p. 47] A simulation model is a model that has been implemented as a computer program or as a description that can be compiled into a computer program and executed in a simulation environment. [4, p. 7]

2.4 Software development

In digital control system development, the development of the control logic and control algorithms can be done using software development methologies. The development concepts and methods presented in this chapter are mostly based on [15, pp. 35-58, 91-98].

2.4.1 Software life cycle

The meaning of the term software life cycle is the time between when the software development project is started and when the software is no longer used. The life cycle can be divided into phases. Each phase has explicitly stated goals which need to be fulfilled and deliverables which need to be produced. Quality assurance procedures such as deliverable reviews and testing are also included in each of the phases.

2.4.2 Preliminary analysis

Preliminary analysis is a phase that precedes actual software development practices, but is essential to establishing the goals of the software project. Its goal is to collect system level requirements that describe the core purpose of the software system. Requirements collected in the preliminary analysis phase are often called customer requirements or business requirements, because their purpose is to capture the clients' needs. The goal of preliminary analysis is to present an assessment of the viability of the software project and to successfully capture the customer requirements of the software system.

2.4.3 Requirements analysis

In the requirements analysis phase the goal is to refine the customer requirements of the software system into functional requirements, which define the functionality of the system. Based on the functional requirements of the system, one or more

function is designed such that together they satisfy the functional requirements. System functions are individual points of functionality in the system. The granularity and level of detail of the function descriptions should be such that each function can ideally be individually implemented. With each function point, it should be explicitly stated which functional requirements it satisfies. The function description also usually includes user interface drafts and descriptions of connections to other software systems. Along with the actual functionality, the functional specification also defines any non-functional requirements, which the system should comply with. Such non-functional requirements can be restricting factors such as timing constraints or usability requirements. Functional requirements and system function points are collected into a functional specification document, which describes the functionality of the software system.

2.4.4 Design

The design phase includes architectural design and detailed module design. The goal of the design phase is to describe the implementation of the software system by defining the technical requirements of the software system. Architectural design is concerned with dividing the functionality of the software into individual software modules. A software module is an independent, logically separable part of the software system. Key points of concern for architectural design are module granularity, module interface simplicity and correspondence to individual system functions. The results of architectural design are captured in a document containing a description of each module and their interfaces and which function points they implement. The architectural design document also describes the main operative sequences of the software system. Detailed module design describes the internal implementation details of each module. This includes descriptions of interface function behavior, algorithm descriptions and definitions of implementation techniques such as languages or databases. A module should be specified to such a level of detail that it can be implemented by a programmer. Detailed module specifications are collected into a module design document.

2.4.5 Implementation

The implementation phase consists of the actual programming work of implementing the modules. Each module is implemented based on the module specification and the software system is constructed by integrating the modules through a framework, platform or a control module. The deliverable of the implementation phase is a working implementation of the module specifications compiled successfully, integrated according to the architecture specification and committed to version control.

As a method of quality control, the source code is subject to static and dynamic checks and source code reviews to ensure the quality of the source code.

2.4.6 Testing, verification and validation

Testing activities in a software project can be divided into phases, each corresponding to a level of specification. This is called the testing V-model and it consists of module testing, integration testing, system testing and acceptance testing. These testing phases correspond to the module specification, architecture specification, functional requirement specification and customer requirement specification respectively. The flow of testing according to the V-model is shown in Figure 2.3. The V-model approach is advantageous, because it facilitates finding errors on the finer granularity levels of the system, making it faster to correct them. Module, integration and system testing can be considered verification activities, where the goal is to ensure that the software has been implemented according to its specification. Acceptance testing can be considered a validation activity, where the goal is to make sure that the software requirements actually fulfill the needs of the customer. Testing in general can be thought of as the activity of carrying out the verification or validation work.

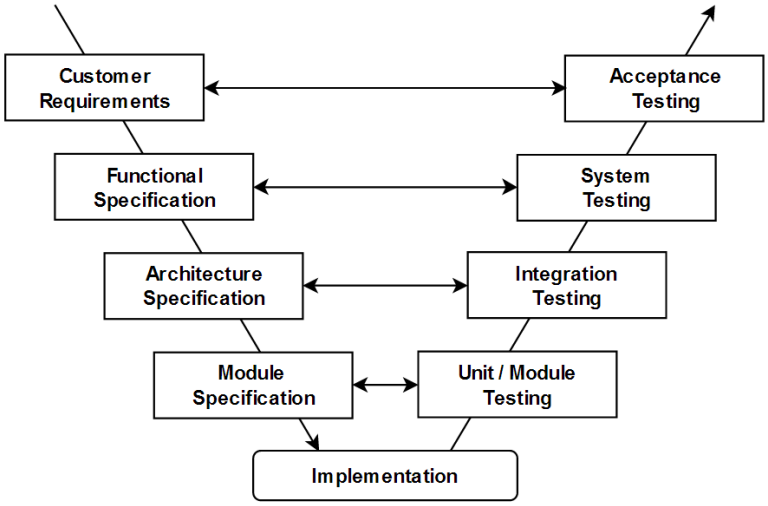


Figure 2.3: The testing V-model commonly used in software development. [15, p. 289]

Cyclomatic Complexity

The cyclomatic complexity measure called McCabe's complexity number was developed to aid in the management of the testability and maintainability of software modules. It denotes the control structure complexity of a program and describes the

amount of different paths the execution of the program can take. The cyclomatic complexity number $v(G)$ is defined using graph theory notation as follows:

$$v(G) = e - n + 2p, \tag{2.1}$$

where the directed graph G consists of n vertices, e edges and p connected components. The control structures in a program can be expressed using the graph notation. McCabe's cyclomatic complexity number provides a way of analysing managing the complexity of software modules. It has been proposed that to ensure the testability and maintainability of a software module, it should have a cyclomatic complexity number of 10 at the maximum. [27]

2.4.7 Maintenance

The maintenance phase of the software life cycle covers the life of the software after it has been designed, implemented, verified and delivered to the customer. Maintenance activities can be divided into three general categories: corrective, adaptive and perfective maintenance. Corrective maintenance includes fixing issues found in the software. Adaptive maintenance means changing the software to adapt to changes in its environment, such as operating system version updates. Perfective maintenance covers adding or changing features of the software system by request or through feedback.

2.4.8 Iterative and Incremental software development

In incremental software development, rather than developing the entire system in one long iteration (waterfall model), the development is divided into multiple smaller iterations. Functionality is added to the software and a working system is the result in each iteration. A key concept of incremental development is that of a core system, which is a simple, working implementation of the software system with minimal functionality. The core system makes it possible to incrementally add functionality to the software system in following iterations, each iteration yielding a tested, working software product. Although there is some overhead from the added requirement for expansibility in the software architecture, those architectural decisions can mostly be validated through the implementation of the core system.[15, pp. 45-47] The ability to present the customer with a working prototype at the end of each iteration greatly improves possibilities for customer interaction and thus, the amount and quality of customer feedback during development [1, 9-13].

In incremental development, the feedback loop is relatively short. This makes it much faster and cheaper to fix specification and design errors and allows for faster verification of critical design decisions. Shorter iterations also make it possible to

react to customer feedback and market developments by steering and prioritizing development effort. It is possible to give more accurate predictions of the cost and delivery date of a project when there is less possibility of a dramatic under or over allocation of resources.[15, pp. 45-47] Schedule planning is based around the idea of delivery milestones, each of which has a goal for what functionality should be included in the milestone release. The development team initially commits to delivering this content by the milestone date, but understands that the goals for the milestone may change on the way. This is called adaptive planning and it is a key concept in so-called agile software development methodologies, which are the focused around the idea of having a software development process capable of reacting to changing customer requirements, environments and market realities.[22, pp. 253-254]

2.4.9 Product management

A software product can be understood as a group of individual, interconnected software modules and the documentation detailing their development, verification and usage [15, p. 52]. A system configuration is the set of versioned modules that make up a specific version of the software system and their documentation. When a module is included in a system delivery, that version of the module should always be available. The version of a software module is typically identified using a unique version number for each consecutive version of the same module. Product management is a software development support process concerned with storing and maintaining different versions of software modules so that different product configurations can be developed and maintained.

A version control system is a storage system, where different versions of software modules and configurations can be stored simultaneously. When a module has been completed, or a completed module has been modified, a new version of that module is added to the version control system. Multiple developers may be simultaneously working on different versions of a module without interfering with each other's work.

2.4.10 Requirements management

Correctly understanding the requirements of a software system and then designing it so that it meets its intended goals is essential to a successful software project. Even when requirements are carefully collected in the beginning of the project, it cannot be guaranteed that the intentions of the customer were understood completely. It is also never certain that requirements will stay static during design and implementation. Requirements management is a software development support process whose responsibility it is to collect and maintain the requirements. [15, pp. 91-92]

In the initial requirements collection phase, requirements management is responsible for collecting customer requirements and through an evaluation and analysis process it accepts or denies those requirements with the goal of collecting a set of requirements which together describe a software system that meets the customers' needs and is competitive in the market. The requirements that have been accepted will be further specified and later designed in detail and implemented in a development iteration and included in a software release. Usually, only after the functionality fulfilling a certain requirement has been implemented is it possible to verify that the requirement has been understood as the customer meant it to be. If it is found that a requirement has been misunderstood, a change request will be issued by the customer and processed in requirements management. If the change is approved, the work to carry out that change will be included in one of the future iterations. Customers or other stakeholders may also introduce request for new requirements during the project due to changes in the market or operating environment of the software system. Change requests may also be issued by designers or developers if it is found that a requirement as it has been described is not viable to implement. Figure 2.4 shows an iterative development process where the requirements management process is responsible for the requirements of the software system. [15, pp. 92-94]

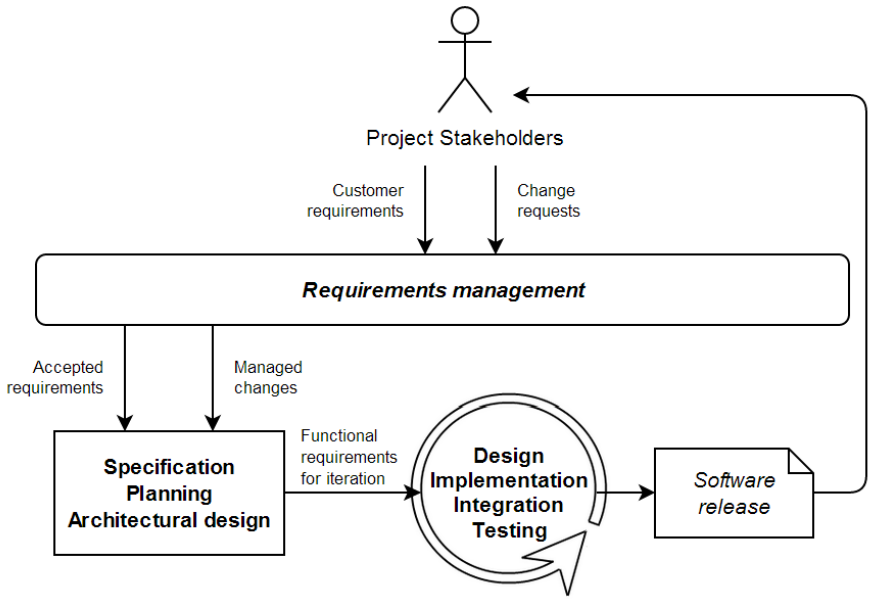


Figure 2.4: An iterative and incremental software development process and a continuous requirements management support process.

It is very likely that changes to the software requirements are needed after they are initially collected, when some of the functionality that implements those requirements may already have been implemented. This puts an emphasis on being able to

effectively analyze the effect that changing a specific requirement has on the overall system and which individual system functions are affected. Requirements traceability means the ability to identify which system functions implement a specific requirement and further, which software modules or functions implement a specific piece of functionality. When doing specification and design work, traceability can be implemented by always explicitly identifying which higher level specification or design artifact or requirement it is related to. Implementing requirement traceability enables impact analysis when analyzing the effect of adding or changing a requirement. [15, p. 97]

2.5 Control System Design Process

Control system design begins by establishing the goals of the system by defining the intended purpose of the system. When written out, these goals should be able to express its intended behavior. The second phase of the process is to identify the system's output variables that need to be controlled to achieve the defined goals. For most systems, identifying the system variables which should be treated as system outputs is somewhat trivial. In some cases, however, if the characteristics of the target system are not well known or if the system is known to be very complex, identifying the outputs that should be controlled to achieve the desired control goals may present a challenge. The quantity and nature of the output variables of the control system affect the complexity and thus the amount of effort required to design and implement the control system, which should be taken into account if the target system is complex and output variable identification is not a trivial task. Based on the purpose and operating environment of the control system, the required accuracy and performance for controlling the outputs is specified. [9, pp. 17-18]

When the purpose of the control system, its outputs and their control requirements have been established, the configuration of the control system can be established. Out of the components seen in Figure 2.1, the target system is the only fixed component. As such, we must choose an actuator, a sensor and a controller to complete the control system configuration. An actuator capable of controlling the behavior of each desired output must be chosen such that it fulfills the performance requirements of the control system. A sensor must be chosen to generate the feedback signal of the closed-loop control system. When choosing the sensor, the control accuracy requirements must be taken into account. The choice of the controller is entirely dependent on the nature and characteristics of the target system. The controller must be capable of executing a control algorithm such that the feedback signal can be read from the sensors and the control signal for the actuator can be generated such that the accuracy and performance requirements are met. The processing capabilities of the digital controller should be considered when choosing

the controller hardware. The complexity of the control algorithm and control accuracy requirements determine what type of controller is best suited for the control system.

Control system design is an iterative process, where the design is refined through several iterations. Initially, the goal is to design a system configuration that meets the primary functional goals of the control system. The design can then be further refined to meet its accuracy and performance requirements by using a different sensor or a different actuator or by changing the control algorithm or its parameters. Ultimately the goal is to have a control system that is feasible to implement and where each component suits their purpose.

Modeling and simulation methods can be used for analysis and verification of the control system. Modeling the process allows the designer to analyze its behavior in different situations and provides information which can be used when selecting the sensors and actuators to be used in the control system. It also allows for the verification of the design when the initial configuration has been chosen.

2.5.1 Distributed embedded control system development

An embedded control system project differs from a pure software project because it has both software and hardware components. The hardware in an embedded control application may include general processing units, specialized calculation units, digital-analog converters and sensors. The group of hardware components in the embedded system form the hardware configuration. In such a project, the division of responsibility between software and hardware and the mapping of software functionality onto hardware processors is described in a system architecture document. System architecture is also concerned with defining the means for hardware abstraction in software. The system architecture design process may by facilitated by the use of hardware and software platforms. A hardware platform in this context means a device family, where a selection of supported configurations is available. This simplifies the system architecture design process by limiting the number of variables. A software platform can be thought of as a an environment that applications can be built on. Software platforms may include hardware abstraction, resource management and inter-process communication services, for example.

After the system architecture has been specified, the software and hardware parts of the system are developed, sometimes in parallel. Co-operation and communication between the software and hardware development teams is required to make sure system as a whole is functional and can be integrated. Integration of software and hardware is done in parts. When the final hardware is not available during software development, partial or temporary hardware (such as FPGA prototypes or hardware simulators) can be used to verify that certain parts of the software can be executed

on the actual hardware. It is also possible to use models as representations of the hardware in software development.

2.6 Modeling and Simulation in control system development

In digital control system design, the use of modeling and simulation practices has extensive benefits. Process modeling allows the control system to be developed independently, disconnected from the actual physical system, whether it exists or not. Modeling the process in such a way that it can be executed and its behavior can be observed in a simulation environment provides a basis for control system development. An executable process model is an essential part of developing the control system as a part of an iterative software development process.

In terms of control design, the process model can be used to enable control design methods such as frequency response and root locus analysis. Transfer function and state-space representations are used to represent the controller in the control design phase. Open and closed-loop simulations with the process model can be used to determine desirable values of control parameters. These control parameter values can be used as requirements for detailed controller design.

In the scope of digital control system development, models are usually understood as descriptions of control system elements written using a modeling language. A modeling language provides the constructs for creating system models and describing concepts such as time and data flow [21, p. 1]. Specialized, domain-specific libraries further facilitate the modeling of systems in a given application domain by providing off-the-shelf components for describing system structure and behavior. The modeling languages used in this thesis are Mathworks' Simulink, a commercial, graphical modeling environment and Modelica, an acausal modeling language with an open specification. Most modeling languages have a specific tool chain for model definition and simulation.

Simulation is the process of executing a model written in an executable modeling language and generating the results of the execution as its output. Simulation requires a simulation environment, which understands the modeling language description of the model and has the means of performing the necessary calculations and displaying the results of the simulation. Dynamic systems are simulated step-by-step with respect to time, so that the simulation results show the evolution of the systems outputs as time progresses. Simulation is a scenario-based activity in the sense that system behavior is triggered by controlling the inputs of the model, then performing the calculations that determine the outputs and finally updating the outputs with the calculated values for each time step. The behavior of a dynamic system model during simulation is determined from the system's characteristics captured in the model definition, the system's inputs and the system's internal state

defined by input values on previous time steps [12, p. 101].

Capturing the behavioral properties of a system in a model is usually based on understanding the physical rules that affect the behavior of the system. The goal is to create a representation of the system using physical equations. One way of achieving this is by analyzing the system specification and deriving a set of equations that represents the physical properties of the system. The parameters for these equations can then be derived from the system specification or acquired through experimental methods. An example of such a modeling process for a mechanical system containing two rigid bodies joined together through a joint would include first formulating an inertial equation to describe the behavior of the system and then calculating the parameters such as length and moment of inertia for each of the components. In the mechanical industry, product development is an iterative process where most new designs are based on old, existing products. This often makes it possible to utilize data from the development of previous products as a basis for a new product being developed.

2.6.1 The Model-based design process

Model-based design is a design methodology which is built around the idea of using models for specification, prototyping, development, verification and communication in an embedded product development project [10, p. 1]. By definition model-based design emphasizes the use of models throughout the development cycle. The intention is to treat models as intellectual property, which is developed and maintained [43, p. 4]. The benefits of model-based design come from increased productivity, improved communication between interest groups and the emphasis on early verification and finding and addressing issues as early as possible [43, p. 2].

It should be noted that the concept of model-based design presented in this thesis is based on MathWorks' idea of its meaning. It takes some of its concepts from the broader model-based software engineering (MBSE) methodology, but its notion of models and their use is different. In MBSE, the goal is to describe the structure and behavior of a system using formal modeling constructs. The models in MBSE can be divided into product models and process models. A product model contains descriptions of the aspects, concepts and relations to build a product in a given application area. A process model can be build using the definitions contained in the product model to describe an actual product. [41, p. 3] Modeling languages such as SysML can be used for systems engineering in MBSE. [17, p. 10]

The central concept of model-based design is an executable specification. It contains the design documentation, executable model and a verification environment for that model. The design documentation includes the textual requirements, describing the functionality which should be implemented by the model, while the executable

model describes the implementation of that specified functionality. The verification environment ensures that the model correctly implements the functionality described in the textual requirements and serves as a reference of how the model is supposed to work. Compared to textual requirements, specifications can describe implementation details through mathematical and behavioral representations. During the development of an executable specification, an executable model is developed to describe the internal structure of the component that is being defined. This internal structure can include definitions of operational states, data flow as well as division of functionality and definitions of atomic subsystems that are to be implemented. Executable specifications are a natural way of representing core functional design details such as algorithms, this way the executable model serves as an extension of the textual requirement. [40, pp. 4-5] Compared to the phase division in the iterative software development approach, the functional specification and design phases overlap in model-based design. Figure 2.5 illustrates the scope of the executable specification as it is presented in this thesis.

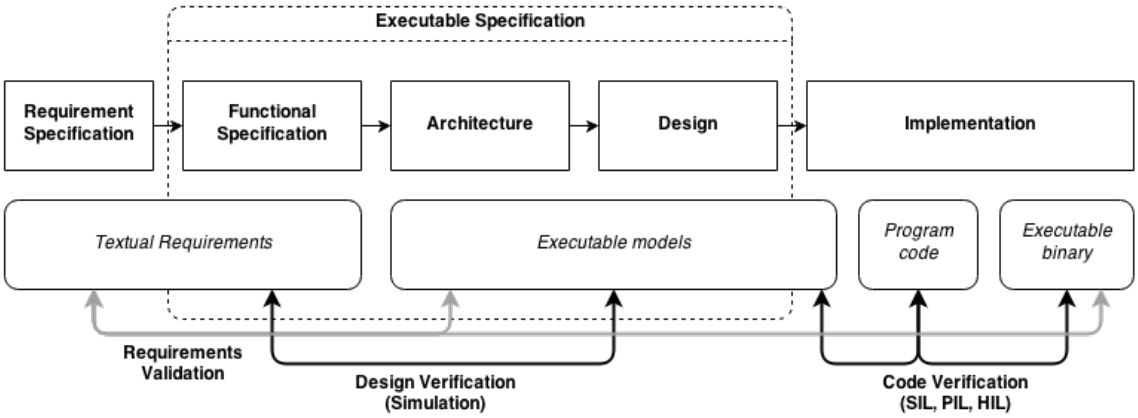


Figure 2.5: The work flow for a generic model-based design process. [3, p. 17]

An executable system model enables continuous verification and validation activities during the development cycle. Initially, the system model can be a high-level description of the features of the target system or process, such as a finite-state machine model. In the requirement specification phase requirements can be validated by creating executable, high-level specifications based on these requirements and executing them with the system model to see if the target system behaves correctly. For a control system, these high-level executable specifications can be rudimentary implementations of the features of the controller, exercising the high-level system model by toggling certain control signals and observing the behavior. This kind of early validation of requirements gives valuable feedback in an early design phase. These high-level models also serve as a base for elaboration in later phases, when more detailed simulation is required for design verification.

In model-based design, the executable model created for the purpose of describing functionality often also contains initial implementations of some system components [40, p. 5]. This blurs the division between the design and implementation phases. In the implementation phase, implementations of atomic subsystems are created and refined. Any required non-functional aspects such as interfaces compliant with actual hardware and timing details are also implemented. The goal of the implementation phase is to automatically generate program code from the executable model. The executable system model is also elaborated and expanded to a level of detail that is sufficient for detailed behavioral analysis. The design of control system functions is greatly facilitated by simulation against a model of the target system. Design evaluation through simulation can give critical feedback on the viability of design decisions on a short feedback loop. It should be noted that the level of detail of the models used for each phase should be carefully evaluated and it should be elaborated if needed. The usefulness of simulation results relies on a sufficiently detailed implementation of the models being executed.

Another part of the product development process is the development of new technologies and algorithms. Especially in the machine industry, where product lines are often iterative, this is also often a separate process. The purpose of algorithm development is to develop prototypes of new functionality based on concepts extracted from business requirements. The model-based design approach enables algorithm development to be more tightly integrated with development process and makes the results more easily applicable to future product development projects. Algorithm development utilizes executable process models and real-time prototyping platforms to analyze and verify control algorithm behavior. The iterative nature of the products in the machine industry has the added benefit of often already having a process model or an existing system available for prototyping. Rapid prototyping of control algorithms has been found to speed up the product development process [24]. The models developed in algorithm development can be used when creating executable specifications for product development projects.

The development and maintenance of models throughout the development process and treating models as actual design entities facilitates communication within the project and to stakeholders outside of the project. A uniform way of storing designs and documenting them reduces the effort required to transfer information between engineers of different backgrounds. Hierarchically developed models where a single model entity contains the specification, design and implementation of a requirement automatically maintains traceability from requirements to implementation. A hierarchical structure also allows for information to be communicated on an appropriate level of abstraction, depending on the situation. It is also possible to automatically generate documentation from models, which alleviates the risk of documentation

getting outdated.

2.6.2 Verification and Testing

Software module tests can be implemented and executed in the modeling and simulation environment. Verification activities in model-based design can be roughly divided into design verification and code verification. Design verification focuses on verifying that the design captured in executable models correctly implements the specification. This supports the principle of early verification, since testing is done on the model level, instead of the source code level. A test suite developed along with the design and expanded as the design evolves can be repeatedly executed in regression testing to make sure the integrity of the design is not affected when it is changed. Such a test suite for the design model can be used to test the operation of the design in varying conditions, much more easily and faster than in traditional module testing. [31, pp. 2-3] In module testing, utilization of the system model allows for the development of descriptive and natural test cases for control applications.

Code verification through various levels of integration with the real target system provide the means to verify the viability and performance of the controller design. Common levels of integration used in code verification are Software-in-the-Loop (SIL), Processor-in-the-Loop (PIL) and Hardware-in-the-loop (HIL) simulation and testing. In SIL testing, program code is generated from the controller model and that program code is compiled and executed in parallel with the model, in the modeling environment. This is the first step to ensuring that program code generated from the model is functionally equal to the original model. When the functional equality of the generated code has been verified on the host PC, the same can be done on the target embedded processor, which will often have more limits on its processing capabilities and use a different instruction set from the host PC running the modeling and simulation tools. In PIL testing, the program code is compiled for the target processor architecture and downloaded. The controller algorithm is then executed in parallel with the original model so that the same test vectors are passed to both, the model and the controller software running on the embedded processor. The outputs of the controller from the embedded processor are then communicated back to the modeling and simulation environment, where the functional equality of the model and the compiled controller software can be verified. The final step in verifying that the controller works in its intended hardware and software environment is running it in real-time, connected to a hardware simulator. The level of detail on the hardware simulator should be sufficient to allow for verification of the functional and non-functional aspects, such as hardware interfaces and timing, of the controller. It is possible to obtain the software for the hardware simulator by generating a program code representation of the system model and executing it on a real-time prototyping

platform. HIL testing done on a hardware simulator makes it possible to verify important safety features before connecting the controller to an actual hardware prototype. The test suite initially created for verifying the controller design model can be used to execute the same test scenarios in SIL, PIL and HIL testing, making the results of each of the testing phases comparable and making it possible to identify errors in translating the model to program code. [31, pp. 2-3] [10, pp. 3-4]

2.6.3 Production Code Generation

One of the biggest individual, measurable benefits of the model-based design approach is automatic production code generation from the designs produced in the detailed design phase. Through the use of the automatic code generation functionality provided by the modeling and simulation tool, a design artifact can be converted from a modeling language description into computer program code, such as C code, that can be compiled and executed on the actual controller hardware. Eliminating the separate implementation phase normally involving manual programming, the implementation and design of a software module are always synchronized, meaning that any changes made to the design are always reflected on the implementation. This enables the approach where models are the focal point of development, which is integral to the idea of considering models as the intellectual property of the organization, not the program code [43, p. 4]. Figure 2.6 presents the general work flow of implementing a software module through modeling and automatic code generation.

This functionality relies on the fact that the code generator understands the modeling language and knows how to translate certain modeling constructs and components into program code, which makes it tool specific. When the program code is automatically generated, its characteristics and quality are defined by the code generator. A versatile code generator will allow qualitative aspects of the program code, such as performance, readability and maintainability, to be configured. As such, it is the configuration of the code generator that is responsible for the quality of the program code, as opposed to the programmer in the conventional approach.

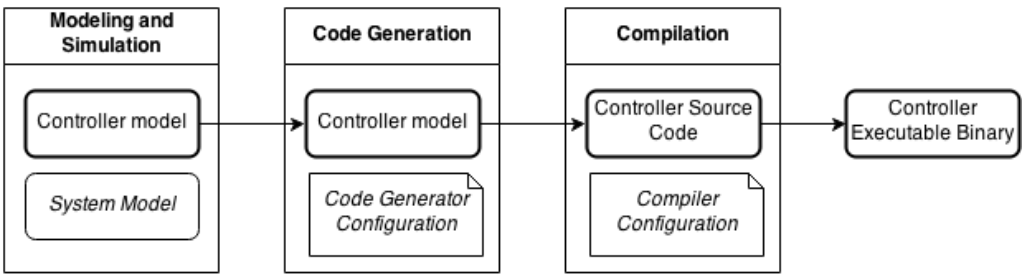


Figure 2.6: The general work flow for implementing a controller through modeling and code generation.

This makes the reliability and quality of the code generator a major concern. The granularity of design artifacts for which code generation can be used varies from individual algorithms to entire controller designs. In an ideal situation, the entire controller design produced in the detailed design phase can be automatically converted to program code, compiled into an executable program and integrated as a part of the control system implementation.

The ability to mostly skip the manual programming phase improves quality and productivity. Productivity is improved when design work is done in the modeling environment and the actual program code representation is automatically generated by the code generator. This is in contrast with a process where the design is translated into program code in a separate implementation phase. Reported productivity gains in both man hours and calendar time vary between 100 and 300 percent [63][47]. Although in model-based design, the most significant gains in software quality are achieved through good modeling practices and verification and validation in the modeling environment, automatic production code generation does provide certain immediate benefits. The use of automatic code generation also, ideally, eliminates the risks of misinterpretation of the design and technical programming errors in the implementation phase, inherently improving software quality [35].

The general conception seems to be, that the motivations for using model-based design methologies are improvement in product quality, advantages in the development of high-complexity functions, shorter development times and cost savings [5, p. 6].

3. STATE OF THE ART

This chapter presents the state of the art for model-based design in the form is use cases and tools. This is done to give an impression of what the starting point is and what is achievable.

3.1 User stories from industry

In the automotive industry, model-based design practices have been adopted, to a varying degree, by numerous companies [5, pp. 5-7]. Several articles describing the use cases, adoption processes and achieved results for model-based design in various industries have been published. Although many of these articles have been published in collaboration with tool providers, they do document actual use cases and trends. The scope of model-based design applications varies from the design and implementation of individual embedded controller functions to entire controllers [47, p. 4]. Ideal application areas for model-based design are ones where the nature of developed control applications is inherently complex or where rigorous verification is required by regulations, such as the aerospace and defense industries [46][11][23, p. 1]. There is also great interest for model-based design in fields such as industrial automation and automotive control, where the goal of developing increasingly intelligent control systems is causing a proportional increase in software complexity [63][61].

The reasons for using model-based design also vary, but the common denominator seems to be the use of automatic code generation [5, p. 11]. In general, model-based design is being widely used in the software design and implementation and to a lesser extent, in requirements engineering and architecture design[5, pp. 6, 17]. Software design and implementation include the development of executable specifications, implementation models, automatic code generation, verification and validation. Requirements engineering in model-based design includes describing requirements in the modeling environment as well as techniques such as rapid control prototyping. Architecture design through modeling enables the analysis of architectural design options and supports verification and reusability.

The tool support for model-based design mostly covers software design and implementation. The functionality for requirements engineering and architecture modeling is also there, but these phases of the software development process are not as easi-

ly isolated and migrated to a new environment as the use of external requirements management systems and diagramming tools is prominent in the field of software engineering.

3.2 Pilot projects

Sandvik Intelligent Control system Architecture (SICA) is a control system platform. It facilitates and enables control system application development by providing a software stack and a hardware library along with design guidelines. The goal is to increase the efficiency of R&D and to provide a unified end-user experience for different products and product lines. A machine project using SICA will be able to take advantage of the supported R&D tools.[38] One such tool is Simulink. SICA offers support for model-based design and automatic code generation by providing a Simulink code generator configuration and custom Simulink blocks for interfacing automatically generated code with the control system platform. This way functionality implemented through modeling and code generation can be integrated into the control system. Modeling and automatic code generation have been successfully used in pilot projects.

3.3 Simulink

Simulink is a commercial modeling and simulation tool by The MathWorks and it is widely used in the industry. It is based on the MathWorks' MATLAB computation tool that has its own language. Simulink provides a graphical modeling environment where models can be constructed from a library of modeling elements called blocks, each performing a specific function. Graphical representations of mathematical, causal models describing the relationships between the inputs, outputs and internal states of a dynamic system can be created and simulated. These dynamic models can be descriptions of real world systems such as electronic, mechanical or thermodynamic systems [48, p. 28].

The graphical nature of Simulink models allows for models to naturally contain notions of internal hierarchy and relationships through the use of subsystems, signals and buses. Subsystems and buses can be defined as either virtual or atomic, defining whether they only exist for the purpose of graphical representation. Subsystems are generally used to isolate a system function into a separate diagram, which enables hierarchical model design. A subsystem can either be defined within the model where it is used, or it can be a reference to a subsystem defined in another model or library. A bus is a collection of signals which can be used for the abstraction of connections of multiple signals between subsystems. [48, p. 29]

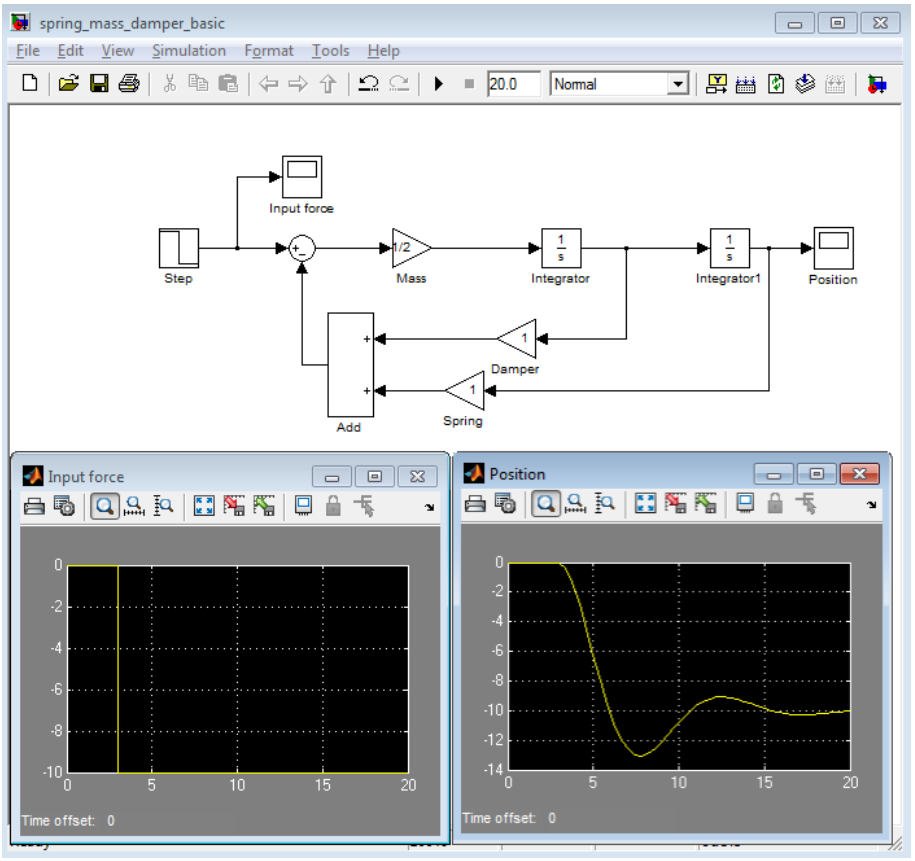


Figure 3.1: A spring-mass-damper system modeled and simulated in Simulink.

In simulation, the states and outputs of a system are being calculated over time. When a Simulink model is simulated, the model is compiled and linked into an executable program which can be executed on the simulation PC. The calculations defined in the compiled model are then successively executed at a specified interval. The difference between two successive computations in the simulation loop is called a time step. The size of the time steps is defined by the equation solver which has been chosen for the simulation. Simulink offers a selection of both variable-step and fixed-step solvers, which are suitable for a variety of different model types. The length of the simulation is determined by the user at the start of the simulation. Simulink provides a graphical simulation environment where simulation options such as simulation duration and solver type can be configured. The results of the simulation can be viewed in signal viewers called scopes or exported to MATLAB for further processing. Figure 3.1 shows the model of a simple spring-mass-damper system being simulated in Simulink. [48, p. 196]

Real-time prototyping platforms can used to test control algorithms and controller designs. Notable manufacturers of such hardware are dSpace and National Instruments. Simulink models can be compiled and executed on real-time proto-

typing hardware that is connected to the actual machine, process or a simulator thereof. This allows for the behavior of the controller to be verified in a realistic environment. It is especially useful in algorithm development. The use of real-time prototyping platforms in control system development is covered in [24] and as such, is not covered further in this thesis.

While the basic functionality of Simulink enables modeling and simulation activities, it is extensible through a number of extensions, called toolboxes. These toolboxes add functionality and analysis capabilities to Simulink. For the needs of model-based design, toolboxes are available for production code generation, design verification, requirements validation and state chart definition. The extensibility of Simulink through purpose-built extensions is one of its selling points. The downside of this is that it is heavily productized to the extent that each toolbox requires a separate license.

3.4 Modelica

Modelica is an open modeling language specification developed by the Modelica Association. It advertises the ability to model complex, multi-domain physical systems through the use of acausal equations describing the systems' properties. It is an object-based language with a syntax similar to that of Java and the MATLAB language, allowing the development and usage of domain-specific component libraries. Modelica Standard Library is a free library developed by Modelica Association, providing components for the modeling of mechanical, electrical, thermal, fluid and control systems as well as hierarchical state machines. [13]

Modelica is only the specification of a modeling language and, as such, relies on actual implementations to allow modeling and simulation of systems. Several implementations of Modelica exist, implementing the language specification to a varying degree. Both commercial and open source implementations are available. The selling point of commercial Modelica tools are a graphical modeling and simulation environment and a plethora of component libraries as well as various other features including model visualization and connectivity to and compatibility with other tools. Commercial Modelica tools include Dymola by Dassault Systèmes [6], MapleSim by Maplesoft [26] and Wolfram SystemModeler by Wolfram [65], among others.

Several projects aiming to offer an open source implementation of the Modelica specification are available. One of these is OpenModelica, a modeling and simulation environment with a graphical user interface. The graphical modeling environment allows for models to be built from components without necessarily having to have knowledge of their implementations. The simulation environment allows for models to be compiled using the provided compiler and executed for the purpose of observing their behavior. In addition to a conventional graphical plotting tool, OpenModelica

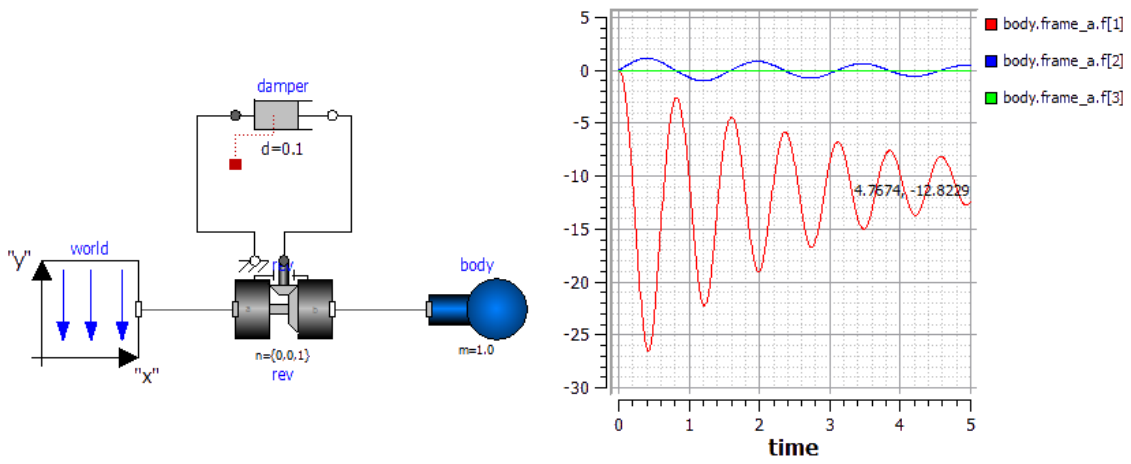


Figure 3.2: The model of a pendulum in the OpenModelica graphical modeling environment and a visualization of its simulation results.

provides the ability to create 3-dimensional visualizations of simulation results for a subset of the available components. Figure 3.2 shows the graphical representation of the model of a pendulum in OpenModelica's graphical modeling environment and a visualization of its simulation results in the plotting tool. By default, OpenModelica provides the Modelica Standard Library. It has raised interest in the industrial and academic communities due to its open source nature, in contrast to most tools which are often commercialized. [33] Another free Modelica implementation is JModelica.org [30] that provides a compilation and simulation environment, but at the time of writing does not offer a graphical modeling tool. Out of this offering of tools OpenModelica is the one that is presented in this thesis as the alternative to Simulink for modeling and simulation. It was chosen because it is open source and it provides a graphical modeling environment.

3.5 Other tools

3.5.1 LabVIEW

LabVIEW by National Instruments is commercial development environment for data acquisition, instrument control and industrial automation. It uses National Instruments' G language, which is a graphical data flow programming language. In the graphical programming language, programs are defined by building block diagrams, in which data is propagated between function nodes through wire connections. [16, p.1] Its strength is in its ability to interface with instrumentation such as sensors and processing units over a variety of buses, allowing for monitoring and data processing. Another central feature is the ability to construct graphical control panels for data visualization and parameter control. Connectivity support also extends to models

developed with other manufacturers' tools. It is possible to visualize data acquired from a Simulink model, for example. For control design, LabVIEW offers a library of control and a system definition components. With these, controller and system models can be defined and then simulated. National Instruments has an extensive hardware offering which is compatible with LabVIEW, including modular embedded hardware prototyping platforms. [32]

4. CHALLENGES OF ADOPTING THE MODEL-BASED DESIGN APPROACH

Trends in the industry indicate that model-based design is being adopted by an increasing number of companies and to an increasingly integral degree in their development processes [37]. At Sandvik the goal is to integrate modeling into the control system development process and to move towards model-based design by supporting the use of Simulink in production projects. Functionality could be designed and implemented by modeling in Simulink and the automatic code generator could be used to generate production code capable of being executed as a part of the control system. The motivations for this are efficiency gains and improved quality in software development as well as a shorter time-to-market for new machine functionality [43, p. 2]. As modeling and simulation practices are already being used for proof-of-concept studies and algorithm development and the results of these are being used as a basis for product development, the use of automatic code generation is a natural progression. The ability to use modeling for detailed design in product development projects makes it possible to make use of the actual model assets developed in research projects. It is also hoped that through the wider adoption of modeling practices in product development, early functional verification made possible by simulation will cause design errors to be caught earlier in the development cycle.

4.1 The work flow for modeling and production code generation

The goal is to allow control systems to be developed in such a way that they consist of software modules implemented through both code generation and manual programming. Figure 4.1 shows a development process where some functionality is implemented through manual programming and some through modeling and automatic code generation. This process focuses specifically on the alternative ways to implement individual software modules. Architectural design is considered a separate design phase where functionality is divided between software modules and it is not necessarily captured in models, although models may have an internal hierarchical structure. Executable specifications in this process include the textual requirements

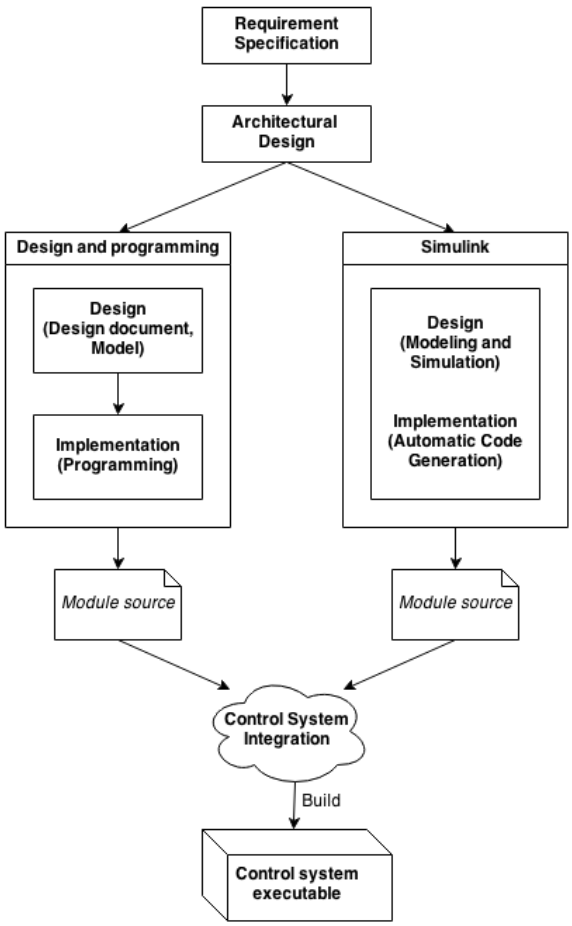


Figure 4.1: A control system software development process where some modules are implemented by manual programming and some through automatic code generation.

and models containing the design and implementation of a software module which has been chosen for design and implementation through modeling. This process includes a manual system integration phase, where the control system is constructed from the individual software modules by connecting them together. System integration can include logically or physically connecting control system components together.

The decision on which design and implementation method to use for each individual module is made in the architectural design phase. Although it is hard to specify definitive rules how to decide whether a specific module should be implemented using modeling and code generation, certain guidelines can be laid out. Primarily, modules that implement a mathematical algorithm or a complex state logic are candidates for implementation through modeling and code generation. In the case of a mathematical algorithm, the ability to iteratively develop and test the algorithm in the modeling environment is highly beneficial because of the short feedback loop. For the development of complex state logic, the ability to visualize the operation of the

finite-state machine and analyze it can help to find design errors such as incorrectly defined transitions or unreachable states. [8, p. 5]

In the work flow where a software module is implemented through manual programming, the design and implementation phases are often separate. Module design in this case is described in a module design document consisting of descriptions of individual functions and sequence diagrams. Sometimes modeling tools such as Simulink can be used in the design phase to verify and refine designs. These module design assets are then used as a basis for the actual implementation as program code. In this work flow, there is a discontinuity between the design and implementation phases which results in the possibility of errors being made in translating the design into program code. It is sometimes also the case that the design and implementation phases of a software module are carried out by different individuals, leaving room for misinterpretation of the design. The design and implementation work flow using Simulink alleviates these issues by containing the design and implementation phases within the modeling and simulation environment. The models representing module design are considered a part of the executable specification for that module.

As such, the implementation phase is reduced to refining and optimizing the design for execution on the target processor and configuring the code generator. Arithmetic errors such as quantization errors, tool errors such as an incorrect or missing code generator configuration and interface errors such as interface mismatches between a control algorithm and its software environment have been identified as common error sources that should be considered in the implementation phase [45, p. 2]. Automatic generation of program code shifts the emphasis of development towards design, which allows the designer to operate on a higher level of abstraction, be less concerned with the details of the implementation and to focus on the creation of intellectual property [8, p. 12]. The lack of a separate implementation phase allows the design and implementation of a module to be done by a single individual, eliminating the risk of miscommunication between design and implementation. The details of the implementation are defined partly in the design model and partly in the code generation configuration.

4.2 Models as a part of the development process

A key factor in providing an alternative method of implementation is integrating the method and its characteristic internal work flow into the existing control system software development process. Seamless integration requires the inputs and outputs of the design and implementation phase of a software module to be similar to what they would be if the module was implemented through manual programming. The input for the design and implementation phase of a module consists of the technical requirement specifying the module's intended functionality and architectural

module description specifying its interface and operation sequences. The output of the design and implementation phase is a source code module implementing the specified functionality. Ideally, there is functionally no difference between a module implemented by manual coding and a one implemented through modeling and code generation if their specifications are the same.

When considering a control system implementation that is a hybrid of manually coded and automatically generated modules, the consistency of module interfaces becomes an important factor to consider. Automatically generated code is for the most part not meant to be readable, meaning that the operating logic of a module is not necessarily very easy to comprehend by studying the source code. This puts an emphasis on the module having well defined and functional interfaces which conform to their specifications to allow integration with the rest of the control system.

The support for design and implementation of individual software modules through modeling and code generation has to be established on the architectural level. Such a partial adoption of the model-based design methodology relies on a clear partitioning of the software modules. The control system software architecture describes the division of functionality between and the granularity of individual software modules. Ideally, this sets clear boundaries for the software modules and makes it possible to assess potential candidates for design and implementation through modeling and code generation. The decisions made in the architectural design phase affect the size and complexity of models and thus also affect qualitative aspects such as maintainability and re-usability of models. The goal should be to find a balance between well defined, manageably sized modules that represent clear functional entities and modules that are fine grained and over-defined, limiting the options in detailed module design.

In a development process where models are used to capture detailed design of software modules, the logical way of developing and maintaining software modules is treating the models as design and implementation artifacts. The reliability of the code generator and the dependability of the generated code have been identified as the key points of concern in automatic production code generation [45, p. 1]. A code generator which has been developed within an established quality management system and certified by an independent organization is more likely to be reliable than one what hasn't [45, p. 2]. The objective reliability of a code generator also improves when its output provenly conforms to a standard such as MISRA C [45, p. 2]. Confidence in the quality of automatically generated code can be improved through the use of modeling guidelines, code reviews, static analysis methods and code verification techniques. Modeling guidelines affect code quality indirectly, by improving qualitative aspects of the models. Code reviews and static analysis methods for the automatically generated code can, on the other hand, be used to initially establish

a certain level of confidence in the quality of the generated code. Code verification techniques such as SIL, PIL and HIL testing can be used to dynamically verify that the generated program code corresponds to the model. [45, p. 4] When such trust can be put into the code generation process that the program code it generates can be assumed to be functionally equal to the model it is based on, models can truly be thought of as the design and development assets that should be maintained.

For models to be treated as design assets, especially in a process where the modeling and code generation work flow exists along with the more traditional design and programming work flow, it should be possible to implement traceability from models to the requirements that specify their purpose, the same way traceability is implemented from requirements to module design and program code. The concept of an executable specification consisting of both textual requirements and executable models implicitly requires traceability between requirements and models. This is especially important since models are developed in a proprietary modeling environment and a link needs to be established between the modeling environment and the requirement specification, which exists as a separate document or in a requirements management system. Since the models are also often of a proprietary file format specific to the modeling environment, it is important that design documentation can be exported so that it can be displayed and stored outside of the modeling environment.

While there are clear benefits to focusing development efforts to the modeling environment, treating models as development assets also imposes certain requirements. As for source code, it should be possible to store models in version control systems where they are stored and available during development and maintained after their development. Storage and maintenance of different versions of the same model is also necessary to support different control system versions and configurations. Along with the actual models, each version of a model should be stored in version control along with appropriate documentation describing the details of its implementation and the program code representation generated by the code generator. With development focused around models, the decisions affecting the qualitative aspects of the development assets also have to made on the model level. Qualitative aspects that the software development process is concerned with are maintainability, readability and re-usability. Modeling guidelines can be used to enforce these positive qualitative aspects which are affected by the internal structure of the model [45, p. 4].

4.3 Organizational considerations

In addition to the technical considerations concerned with developing models and using them as a part of the control system software development process, a number

of organizational considerations related to the adoption of model-based design practices can be identified. The need for organizational change is created by the creation of new engineering tasks, the emphasizing of design and changes in the development process[8, p. 2]. The introduction of the modeling work flow with its specific tools and principles creates a requirement for certain competence in the organization, especially from the point-of-view of a software project. Competence in the areas of modeling, simulation and plant model development are required to take advantage of model-based design. While these skills may be more widely available in the mechanical and hardware engineering disciplines, in control system software development acquiring this competence through recruitment or training may be necessary. [8, pp. 2-3]

In model-based design, the emphasis of effort in the project shifts towards requirements and design [8, p. 3]. From the point-of-view of a software project, especially with the introduction of automatic code generation, this means that development effort needs to be refocused. The most obvious change is the fact that the effort needed for implementation through manual programming is expected to be reduced gradually, as modeling and code generation are adopted. The verification of software modules is also done on model level, eliminating the need to develop module tests on program code level. The software engineers previously fully occupied with software development tasks need to start orientating more towards system architecture design, system integration and model development [43, p. 5].

The introduction of the modeling work flow introduces modeling tools and new ways of work to the software development process on the organizational level. The process changes need to be defined and carried out. The need for new tools in the development tool chain needs to be mapped out and the tool candidates need to be evaluated to reach an informed decision on which tools are the most suited for the purpose. To ensure that the development tools can be used for modeling in production projects, a development environment configuration needs to be established. The organization needs to support the modeling tools by creating and maintaining the development environment and making sure it can be obtained by ensuring that installation files and license keys are available. Especially with the long life cycle expectancy of products in the machine industry, tool configuration management is critically important to ensure that the software can be changed and re-used in the future. Another maintainable aspect of the modeling work flow is the supporting documentation such as standards, guidelines and manuals. While tool manuals are most often provided by the tool supplier, any guidelines and standards have to be defined and maintained. [43, p. 4]

4.4 Summary of goals

The goal of this thesis is to find ways to support the control system software development process described in this chapter and the modeling practices that enable it. This thesis presents an example use case for modeling in the form of a design problem from the machine industry, through which the capabilities of the tools Simulink and OpenModelica are demonstrated. By analyzing the process of solving the design problem with each tool individually, an assessment of their suitability for the modeling and code generation work flow is presented. With the goal of supporting this work flow, the thesis addresses more specific concerns of integrating modeling into the software development process at Sandvik. Traceability, maintainability and quality of models are covered from the point-of-view of the development process. An assessment of the reliability and performance of automatically generated code is presented to support the idea of focusing development efforts in modeling and raising the level of abstraction from program code to models. Verification and validation as a part of the model-based design process are not covered further in this thesis. Organizational and software architectural concerns are also not addressed further than they are in this chapter.

5. TOOL EVALUATION

The purpose of this chapter is to evaluate the capabilities of the two selected tools, Simulink and OpenModelica, in designing a controller for a physical system with the goal of using the controller model for automatic code generation. In this chapter first presents the main criteria for analyzing the tools' capabilities. Before presenting the actual analysis and summarizing its results, the example design problem is presented. The tool versions used in this thesis are version R2013b of MATLAB and Simulink and version 1.9.0 of OpenModelica.

5.1 Analysis criteria

The modeling capabilities are the primary criterion for analysis of the tools in this Chapter. The developer should be able to describe a system in a language that can be understood by the simulation tool in such a way that its behavior can be observed in simulation. The modeling environment should allow the developer to create both time-continuous and time-discrete models. For example, controller models intended for embedded controller hardware are time-discrete and process models describing the behavior of a dynamic physical system are time-continuous. These can be used together to analyze and verify the behavior of the closed-loop control system, where the outputs of the controller are connected to the process model and the outputs of the process model are fed back to the controller. The modeling language used by the modeling environment needs to support the creation of such control systems. Besides evaluating just the theoretical versatility of the modeling language, for the specific domain of control engineering it is important that the modeling environment provides modeling language constructs which can be easily understood and used by engineers familiar with the domain. As a development tool, the usability of the modeling software is also taken into account.

Besides the modeling environment, the capabilities of the simulation environment determine the quality and quantity of information that can be acquired through simulation. A good simulation environment allows simulation properties such as the solver algorithm, step size and simulation time to be configured so that the simulation results provide adequate information about the system properties under observation. The simulation environment's ability to visualize simulation results for analysis and store them for further processing greatly affects its value as a design

tool.

As tools that are a part of the product development process, the available documentation and product support is a factor that needs to be considered. Support for automatic production code generation is also analyzed, since it is a subject of interest in this thesis.

5.2 Example model

The control design problem used as a means of demonstrating the tools' capabilities is based on the M.Sc. thesis by Arto Sirén covering the design of an automatic leveling controller for a rotary drill rig [42]. It was chosen because it is a design problem representative of Sandvik's engineering domain and because it is a complete, public thesis, giving a detailed explanation of the design problem and the means to solving it. The motivation for the design problem presented in [42] is the need to automate the process of leveling the hull of the drill rig for the purpose of drill hole alignment by controlling its four ground jacks. The hull of a drill rig is also called the carrier. The starting point was, that the leveling was done by manually driving the ground jacks and leveling through measurement and adjustment. The goal of [42] was to design the instrumentation setup and control algorithm for implementing such an automated leveling system. The design approach first introduces the drill rig and presents its key physical properties, after which a physical spring-mass-damper model for the drill rig is defined. This process model is then used in controller design to analyze the effects of different controller designs and control parameter values. [42, pp. 10-11] The control design problem is closely related to the behavior of the physical machine, making it beneficial to use a physical model, which makes it a good candidate for the model-based design approach. It is also beneficial to design and verify the operation of the finite-state machine logic of the controller through modeling and simulation.

This thesis uses the physical model description presented in [42] as a basis for constructing the physical model. Information about machine parameters such as its dimensions, total weight and weight distribution is used in calculating estimates of the physical model parameters. The operation logic of the controller designed in [42] is used as a basis for the controller designed in this thesis with its operation simplified and some of its requirements omitted.

5.2.1 Physical model

The described drill rig consists of two distinct parts: the hull and the mast. When the rig is in its drilling position, the mast is raised and the hull is supported on the ground by four ground jacks, one at each corner. With the mast raised, the highest

point is at the height of 29 meters. The whole rig weighs 150 tonnes with the hull weighing 100 tonnes and the mast weighing 50 tonnes. The ground jacks used for supporting the rig in drilling position are operated by hydraulic cylinders. [42, pp. 10, 12-14]

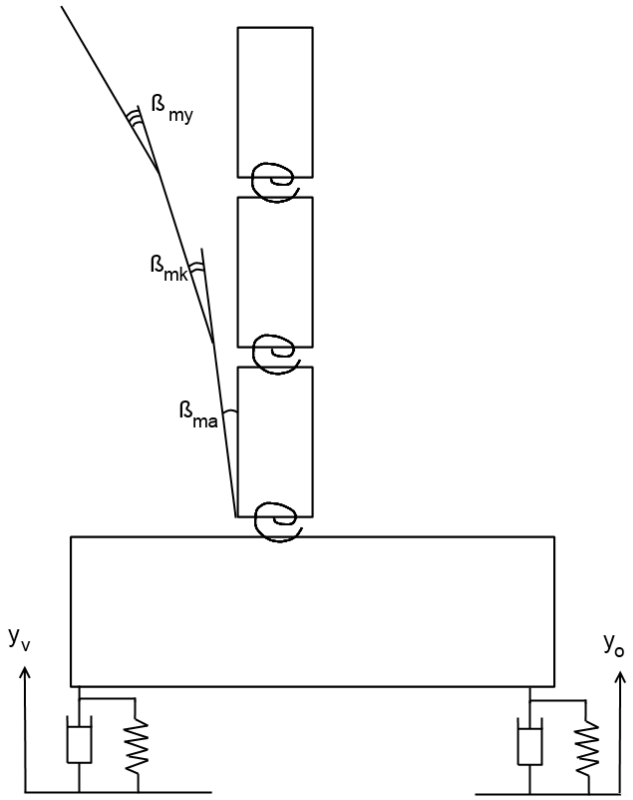


Figure 5.1: A two dimensional projection of the R110 drill rig with its degrees of freedom denoted. [42, p. 20].

Interesting aspects of the mechanical characteristics of the rig are the tall and heavy mast and the heavy and slow hull. When applying forces that cause the heavy hull and the mast to rotate, even a small increase of the angle between the ground and the hull causes the top of the mast to shift. The mast is also not completely rigid and will exhibit a swaying motion when the hull rotates [42, pp. 10-11]. The dynamics of the 3-dimensional rig can be described in terms of two 2-dimensional projections with five degrees of freedom. They are defined as two vertical degrees of freedom, y_v and y_o, representing the ground jacks' displacement from ground level as well as β_{ma}, β_{mk} and β_{my}, representing the bending of the mast. The mast is divided into three sections for the sake of simplicity [42, pp. 19-20]. One projection directly from the side of the machine and another from the front at a slight angle are used. Figure 5.1 shows the angled front projection with the degrees of freedom denoted. The inertial equation for the projection can be formed in terms of these degrees of freedom and their first and second derivatives [42, p. 26]:

$$A\ddot{\mathbf{x}} + B\dot{\mathbf{x}} + C\mathbf{x} = \mathbf{d}, \tag{5.1}$$

where the state variables in vector form $\ddot{\mathbf{x}}$, $\dot{\mathbf{x}}$ and $\mathbf{x}$ are defined as

$$\mathbf{x} = (y_v \; y_o \; \beta_{ma} \; \beta_{mk} \; \beta_{yk})^T \tag{5.2}$$

$$\dot{\mathbf{x}} = (\dot{y_v} \; \dot{y_o} \; \dot{\beta_{ma}} \; \dot{\beta_{mk}} \; \dot{\beta_{yk}})^T \tag{5.3}$$

$$\ddot{\mathbf{x}} = (\ddot{y_v} \; \ddot{y_o} \; \ddot{\beta_{ma}} \; \ddot{\beta_{mk}} \; \ddot{\beta_{yk}})^T. \tag{5.4}$$

The 5×5 co-efficient matrices A, B and C describe the rig's properties, the correlation between the all the freedom degrees in terms of moment of inertia, damping and spring constant. The vector $\mathbf{d}$ contains external forces acting on the system's axes of freedom [42, p. 26].

Simplifications of the physical model

The equations for the physical model are the same for both projections and 3-dimensional control of the physical model can be achieved through the combination of the two. Thus, for demonstration purposes it is considered sufficient to only calculate the parameters of the physical model and design the controller for one of the projections. The purpose of the physical model in this thesis is not to give an accurate representation of the physical properties of the actual mechanical machine, but to serve as a means of demonstrating the process of solving a design problem with modeling tools. As such, a number of simplifications were made to the model. Firstly, the model in this thesis does not take the restrictions of actual actuators and sensors into account. The external forces in vector $\mathbf{d}$ are directly used as control inputs. In order to realistically describe the physical system, a model of the dynamics of the hydraulic actuators would have to be developed. These hydraulic actuators would be controlled by the voltage passed to the hydraulic pump and by the control voltage passed to a proportional directional control valve. For the purposes of this thesis it is not necessary to consider the restrictions set by the physical properties and restrictions of the actuators. We also assume that we are able to measure the position, velocity and acceleration in terms of each of the degrees of freedom, when in reality such sensors might not exist or it might not be feasible to install them on the physical machine.

When considering model simplifications, the nature of the design problem and its requirements should be taken into account. The simplified physical model presented in this chapter is still more than detailed enough for the design problem at hand.

In fact, if the basis for the physical model of the R110 machine was not already available, an even simpler representation of the physical properties of the machine would have been used. The level of detail of a plant model used in controller design should be selected such that it is sufficient for verifying the requirements set for the controller and not more detailed that is necessary.

5.2.2 Controller logic

The system is controlled by exerting external forces on the axes y_v and y_o. Forces acting on these axes together can be used to control the height and angle of the rig in the 2-dimensional projection. Let the components of vector $\mathbf{d}$ representing the forces acting on y_v and y_o be called d_v and d_o respectively. The logic of the controller can be simplified to the finite-state machine (FSM) representation presented in Figure 5.2. In this state logic, the controller starts in the leveled state, regardless of whether or not the rig is actually level. In the leveled state, adjusting the set point for the height will cause the controller to transition to the height adjustment state, where both jacks will be raised until the desired height is reached. When the desired height is reached, the controller transitions to an intermediate height adjusted state, where a decision about which jack should be raised to level the rig is made based on measurement data. The controller proceeds to level the rig by transitioning to one of the two leveling states. Finally, a transition back to the leveled state happens when the left and right sides of the rig are at the same height.

Controlling the height of the left and right sides of the rig can be done by controlling the inputs d_v and d_o and a state logic controller that manages the transition from one control state to another. The actual control of the axes y_v and y_o can be implemented by using a PID controller where the set point and feedback are given in terms of the axes' positional displacement from ground level. Applying a gain to the control signal of the PID controller, calculated based on the error term of the positional displacement value, translates it to a force value proportional to the control value.

Simplifications of the controller logic

Due to simplifications made to the physical model, the controller model also does not take the dynamics of the hydraulic cylinders into account, but rather just controls the force acting on the axes y_v and y_o. The controller design presented in this thesis uses very conservative parameters for PID control since optimization and further analysis of control parameters is not meaningful given the scope of the thesis. Extensive stability analysis of the controller is not performed. Originally, strict requirements regarding maximum safe displacement of the center of mass of the rig due to swaying

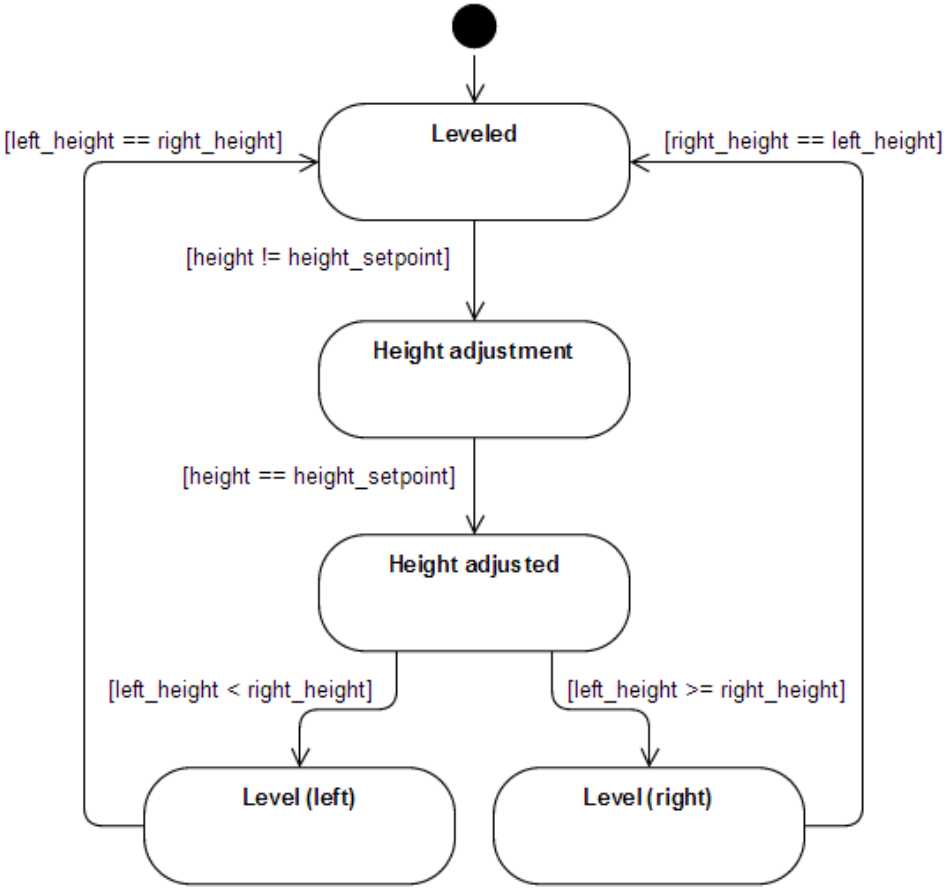


Figure 5.2: A simplified representation of the state logic of the automatic leveling controller [42, p. 57].

of the mast were included to ensure that the machine does not fall over [42, pp. 46-53]. These have been omitted to such an extent that the only verification for this is done by ensuring in basic simulations that the oscillations caused by the swaying of the mast degrade over time.

5.3 Implementation using MATLAB / Simulink

In accordance with the workflow of model-based design, the physical model of the R110 rig was developed first. The starting point is Equation 5.1, which describes the behavior of the physical system as a spring-mass-damper system. The elements of the matrices A, B and C were calculated based on the data that was available of the machine's physical properties, such as weight, dimensions and structural details. The same equations that were originally derived to calculate the element values for the co-efficient matrices [42, Appendix 3] were used in this thesis.

5.3.1 Physical Model in Simulink

The Simulink model to represent the spring-mass-damper equation is formed by using integrators to represent anti derivatives and gain blocks to represent the matrix co-efficients. The input to the system is the vector $\mathbf{d}$, namely its components d_v and d_o. External forces acting on the axes β_{ma}, β_{mk} and β_{my} are assumed to be zero. The main output of the system is the positional information for each of the five axes. This can be though of as measurement data that would be acquired through sensors on the actual machine. Figure 5.3 shows the Simulink block diagram of the physical model of the R110 drill rig. It should be noted that most Simulink blocks can scale to vector and matrix form input signals, either automatically or through a configuration parameter.

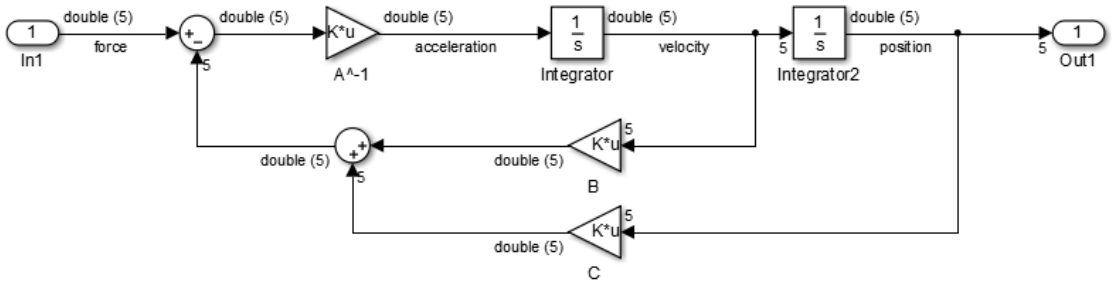


Figure 5.3: Implementation of the 2-dimensional physical model of the R110 drill rig in Simulink. Matrix co-efficients A, B and C are used as gain factors.

A subsystem was created from the contents of the physical model and a subsystem mask was created for it. The mask hides the contents of the subsystem and allows a custom interface to be created for it. The mask of the physical system defines the physical properties of the rig projection as parameters that can be entered in the block parameters dialogue. It also contains the definition for the matrices A, B and C. This way, each instance of the physical subsystem (the side and angled front projections for example) can have its own set of parameters.

5.3.2 Controller Model in Simulink

For discrete controller design, Simulink offers a library of blocks that define discrete states [53]. In addition to this, the sample time of most Simulink blocks can be selected such that the discrete nature of the controller can be modeled. Finite-state machines representing controller state logic can be implemented using Stateflow, a MATLAB and Simulink toolbox for defining state machines and flow charts [59]. Structure and hierarchy can be built into Simulink models by using subsystems [55]

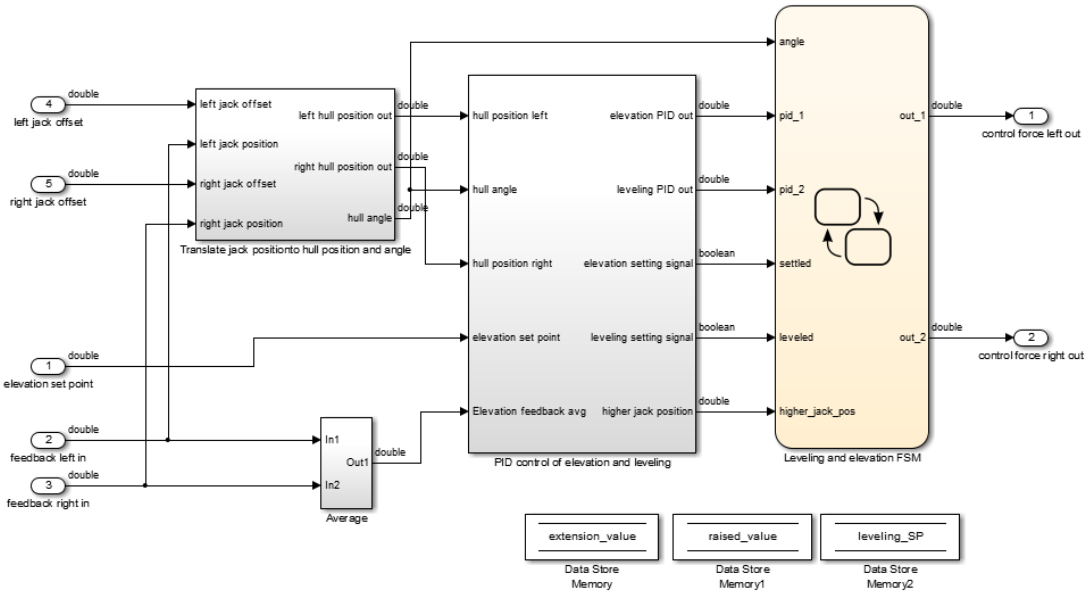


Figure 5.4: Top-level view of the controller for elevation and leveling.

and model references [54].

The controller logic for controlling the elevation and leveling of the R110 drill rig is divided into two main operational subsystems: The PID controller subsystem for calculating the control values and the finite-state machine that determines which controls should be passed to inputs of the physical system. Figure 5.4 shows the top-level view of the R110 elevation and leveling controller. The PID controller subsystem takes five inputs. The inputs required by the elevation phase are the set point for the height of the rig and the feedback signal. In the leveling phase, the PID controller uses the average value of the two positional measurement signals to drive the rig upwards to the desired height. For this purpose, a subsystem that calculates the average value of the two measurement signals (left and right side of the hull) is used. For the leveling phase, the controller uses the same set point value. For determining whether the left or right side ground jack should be used to drive the rig to a position where it is level, the controller uses a measurement of the hull angle. It also uses measurements of the positions of the left and right sides of the hull with respect to height from the ground. For the purpose of representing slopes and bumps in the ground, the values for these measurements are calculated in the controller model based on the measured extension values of the hydraulic cylinders and ground level information originating from the test harness. In reality, this could be implemented by installing dedicated sensors on the drill rig.

The implementation of the PID controller subsystem utilizes two discrete PID controller blocks, one for the elevation phase and another for the leveling phase.

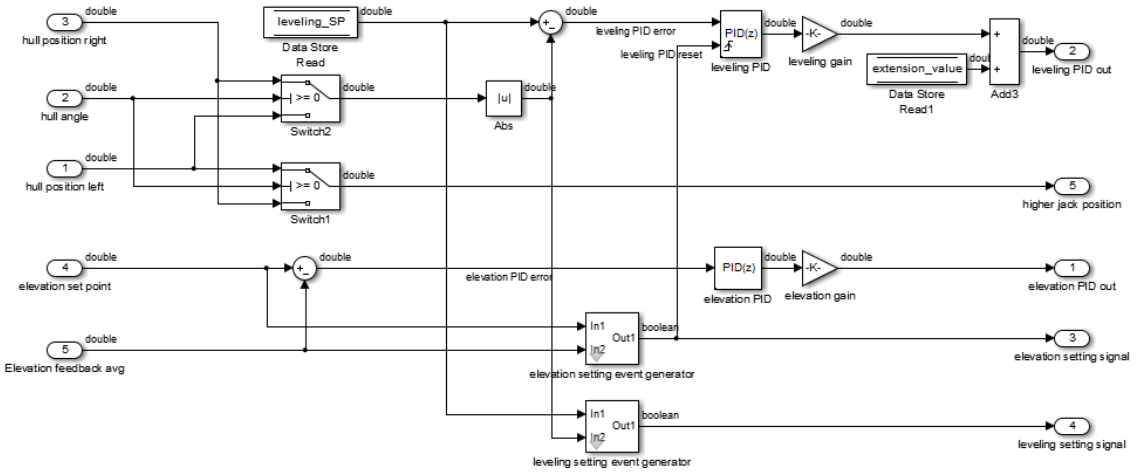


Figure 5.5: Detailed structure of the elevation and leveling PID control subsystem.

The operation of the controller in the leveling phase is simple. A feedback signal calculated from the difference between the set point value and the measured, averaged hydraulic cylinder extension value is fed to the elevation PID controller block. The control signal calculated by the elevation PID controller is fed to the FSM subsystem. The logic for transitioning from one phase to the other is implemented by using two event generator subsystems that determine when a signal has settled to a certain value. This information is passed to the finite-state machine subsystem to trigger state transitions. The operation of the controller in the leveling phase depends on which ground jack is lower, which can be deduced from the hull angle measurement. The hull angle measurement controls two switch blocks, that pass the correct signals to the finite-state machine subsystem in each case. In the leveling phase, the position of ground jack that is higher is fed directly to the FSM subsystem and the other is fed to the leveling PID controller as the measurement signal. Figure 5.5 shows the block diagram of the PID controller subsystem.

The finite-state machine subsystem is implemented using Stateflow. The default state of the state machine is the elevation state where the both of the hydraulic cylinders are extended identically. In this state, the leveling PID controller input is connected to both, the left and right ground jack outputs. A state transition is triggered by an event signal from the PID controller subsystem telling the FSM that the rig has been raised to the desired height. Upon exit, the value of the elevation PID controller signal and the measurement value for the position of the higher ground jack are stored to global data stores. If the hull is not level (which it will realistically almost never be), the hull angle input is used to decide whether the FSM transitions to the left or right jack leveling state. In the leveling states, the

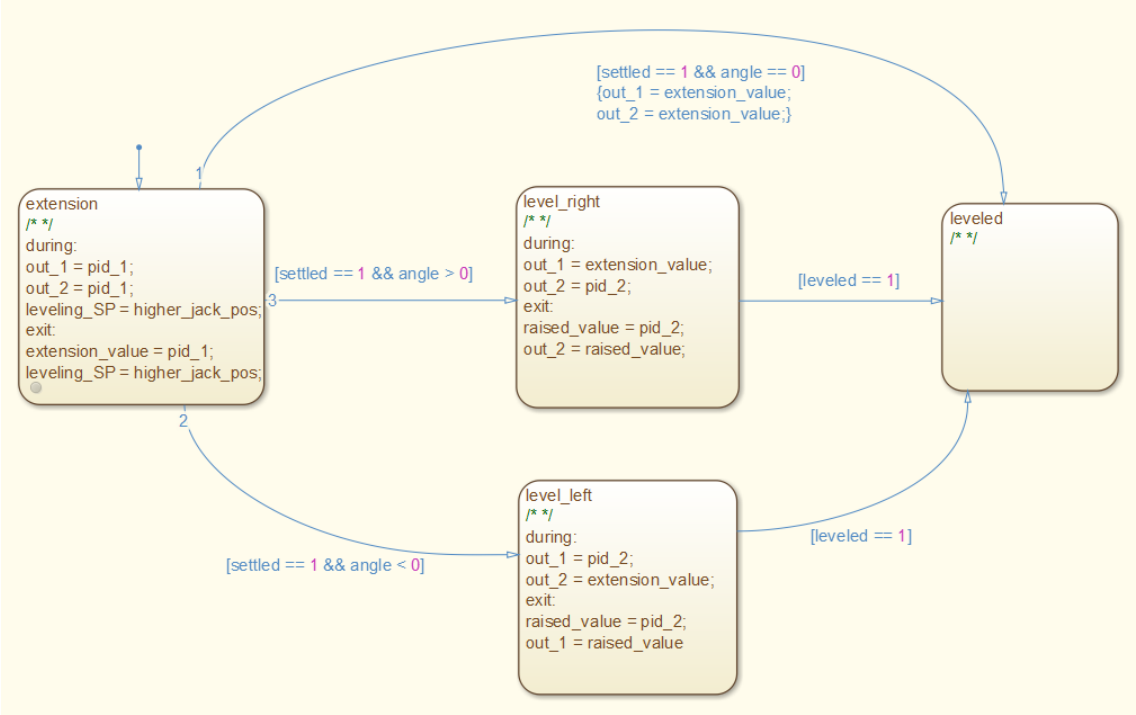


Figure 5.6: The finite-state machine controlling the elevation and leveling states and outputs of the controller.

control signal for the ground jack that was higher is fixed to the control value that was stored upon exiting the elevation state. The control signal from the leveling PID controller is passed to the other ground jack. A signal that monitors the settling of the position of the lower ground jack to its target height triggers the transition to the final state, where the drill rig has been raised and leveled. Figure 5.6 shows the Stateflow chart of the leveling and elevation FSM subsystem.

5.3.3 Simulation in Simulink

Simulink provides a simulation environment that can be used to execute models and observe their outputs. Simulation data can be displayed using scope blocks or exported to the MATLAB workspace. The simulation can be controlled by selecting the simulation time and the solver algorithm to be used for computing the behavior of the model.

A test harness connecting the controller to the physical model was created for simulation. It is constructed such that the outputs of the controller are connected to the inputs of the physical model. As the controller only generates control signals for the two ground jacks, the inputs for the other three degrees of freedom are zeros. The test harness utilizes rate transition blocks between the time-discrete controller and the time-continuous physical model. The feedback connection goes from the

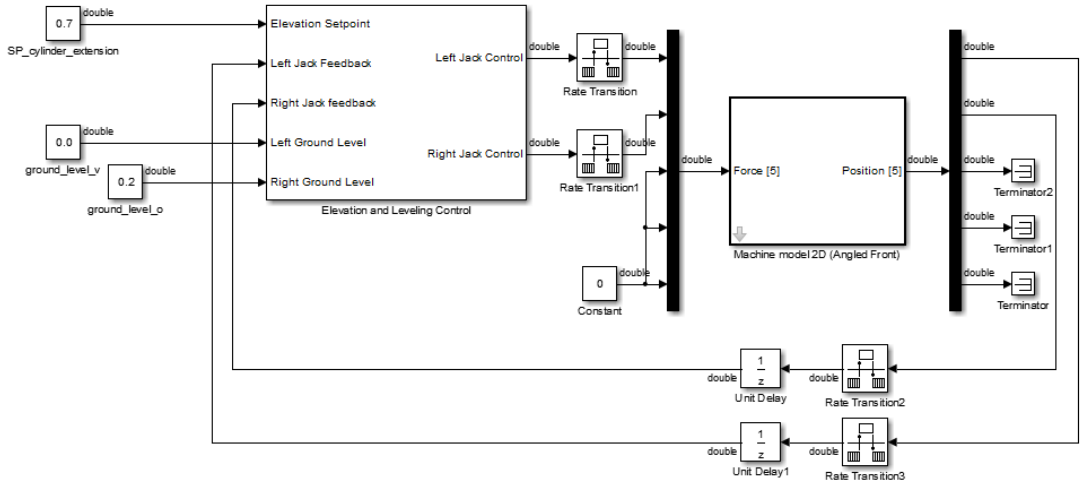


Figure 5.7: The system level view of the test setup for the r110 controller.

physical model's output to the inputs of the controller. Discrete unit delay blocks are used in the feedback path to simulate sensor delay. The simulation scenario is defined by the elevation set point value, which can be used to set the desired height to which the rig should be raised as well as the ground level offset values, which can be used to simulate scenarios where the ground, on which the rig is standing, is not level. Figure 5.7 shows the test harness setup.

The first step in simulating the test harness is selecting a solver that is suitable for the model. The test harness model in this case contains both time-discrete and time-continuous sections, specifically the integrator blocks in the physical model have time-continuous states, so a variable-step continuous solver is required. Initially, simulating the test harness was really slow. The cause for this was found to be the way the variable-step solver operated on the physical model, where each of the elements of the co-efficient matrices affects the output. The co-efficient matrices' elements are not equal in magnitude or their effect on the overall output of the physical model. For example, the values for elements of the matrix C representing spring constants range from the magnitude of 10^5 to 10^9. This causes the step-size of the variable-step solver to get extremely small. The R110 physical model is what can be called a stiff system, so the variable-step, continuous $ode23t$ solver was selected based on MathWorks' documentation for solver selection [51] and experimentation. To further improve simulation performance, the relative tolerance solver option was changed to allow more error in the states of the system. This was found not to affect the results of the simulation in any significant way, but the time to simulate the test harness for 60 seconds was reduced from over 2 hours to less than 2 minutes.

Simulations were run to ensure that the controller worked as intended. Sample

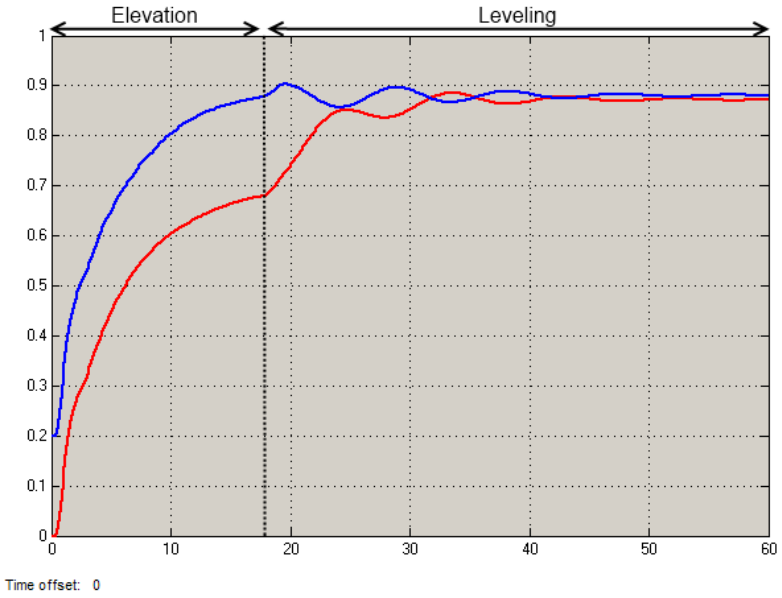


Figure 5.8: Simulation output utilizing the test setup for a scenario where the machine hull is raised 0.7 meters and the ground level is 0.2 meters higher on the right side. The figure also illustrates the state transition from elevation to leveling.

time of the time-discrete blocks was set to 0.1 for the simulations. The parameters of the PID controllers were selected such that the control was sufficiently fast and clearly stable in the simple simulation scenarios that were run. The simulation scenarios were focused on verifying that the PID control for each of the states worked correctly and that the FSM subsystem correctly transitioned from one state to the other. Figure 5.8 shows a simulation scenario where the rig sits on ground where the right side is 0.2 meters higher than the left side and the elevation set point is at 0.7 meters. The plotted data sets represent the height of the right and left sides of the hull. The elevation state of the controller, where both ground jacks are being given the same control signal, is visible in the plot. The transition to the leveling state happens at around 18 seconds simulation time. At that time, the control of the right ground jack is fixed and the left jack is controlled to level the drill rig. While controlling both of the ground jacks with equal input causes very little oscillation in the rig, controlling only one jack does cause the mast to sway and oscillation occurs. The leveling procedure is done when the height of the left side of the rig has settled within a given margin of the target height.

5.3.4 Code Generation in Simulink

Automatic code generation will be used to generated a program code representation of the PID controller subsystem. The intention is to generate a program code representation of the controller and to compile it in an external development environment.

The development environment used in this thesis is Microsoft Visual Studio 2012.

The code generation options in Simulink allow the developer to configure a number of parameters related to the code generation process and the characteristics of the generated code. The Target Selection option can be used to select the system target file, a Target Language Compiler configuration for code generation. The system target file specifies the execution environment of the generated code and its characteristics [49]. Additional parameters for the selected system target can be configured in the sub menus of the code generation configuration. A default selection for each of these parameters is defined in the system target file. For code generation of the controller model, the Embedded Real-time Target configuration was selected. The code generator was configured to use C++ as the target language and to allow code generation for blocks that use continuous time. Other configuration options were left to their default values. As for options that affect the process, code generator can be configured to execute model checks before actually performing code generation. It can also be set to automatically create and display a detailed code generation report.

Simulink Coder only supports fixed step solvers for code generation [51]. This is most often feasible, since controllers targeted for embedded controller hardware should be time-discrete. For code generation, the documentation suggests that the solver should be changed in the configuration options. The coge generator was triggered for the PID controller subsystem with and without explicitly changing the solver type to fixed-step, and in both cases it resulted in the generated code using a fixed-step discrete solver. For production code generation it is recommended to explicitly change the solver type.

As a result of the code generation process, Simulink Coder creates a folder containing the multiple header files and a source file for the controller. Table 5.1 lists source and header files generated in the example case. The structure of the generated program code module is such that it contains all of the functionality of the controller in one module. It should be noted that the output of the code generator can be changed through the configuration options, mainly the Target Language Compiler configuration. The source files have dependencies to Simulink-specific code modules, meaning they either have to set up in the compiler path or copied to the project directory in Visual Studio.

The basic structure of the generated code module consists of three functions, initialization, stepping and termination. Initialization is executed before and termination after executing the model. The step function first reads the inputs of the model, executes the model code for one time step and finally updates the outputs and internal states. The actual operation logic of the controller is contained in the step function. The code in the step function is annotated for traceability to the

File Name	Type	Size (kB)
Elevation0.cpp	source	17
Elevation0.h	header	8
Elevation0_private.h	header	2
Elevation0_types.h	header	1
rtwtypes.h	header	7

Table 5.1: A list of the files for the C++ program code representation of the controller model created by the Simulink Coder code generator.

source model. Each annotation denotes which model element it implements.

To test how easily the code generated with Simulink Coder can be integrated into an external software project, a new C++ software project was created in Visual Studio. A reference to the Simulink library modules located in the MATLAB installation directory was added to the project. To test the operation of the controller, a very simple function representing the process model, consisting of a delay and a gain, was written. A main function to call the initialization, stepping and termination functions of the controller module was written. Before the simulation is executed, the input variables are set to certain constant values to set up the simulation scenario. During the actual simulation, the execution is done in a loop where the inputs of the controller are updated, a simulation step of the controller is executed, the inputs of the process model are updated and finally a step of the process model function is executed. The length of the simulation is determined by the limit of iterations of the simulation loop, determined by a constant value.

To analyze the operation of the controller, logging functionality was added to the simulation program. A matching simulation scenario was set up to be able to compare the operation of the controller program to that of the controller model. In this scenario, the elevation height set point was set to 0.7 meters, the ground levels were set so that the right side was 0.2 meters higher than the left. Simulation time was set to be sufficiently long, since values could be omitted from the end of the log if needed. The simulation was executed and the log values were read into MATLAB and plotted. Figure 5.9 shows this plot. As in the Simulink simulation, the transition from the elevation state to the leveling state is clearly noticeably, meaning that the operational modes of the controller are working as they should be. The difference in the forms of the Simulink simulation plot and the simulation program is caused by the fact that there is no actual physical model in the control loop in the simulation program.

To ensure that the program code implementation is also numerically correct, a test setup was created in Simulink. This setup utilizes the Simulink S-function functionality, that allows program code to be wrapped into a simulink block. An S-

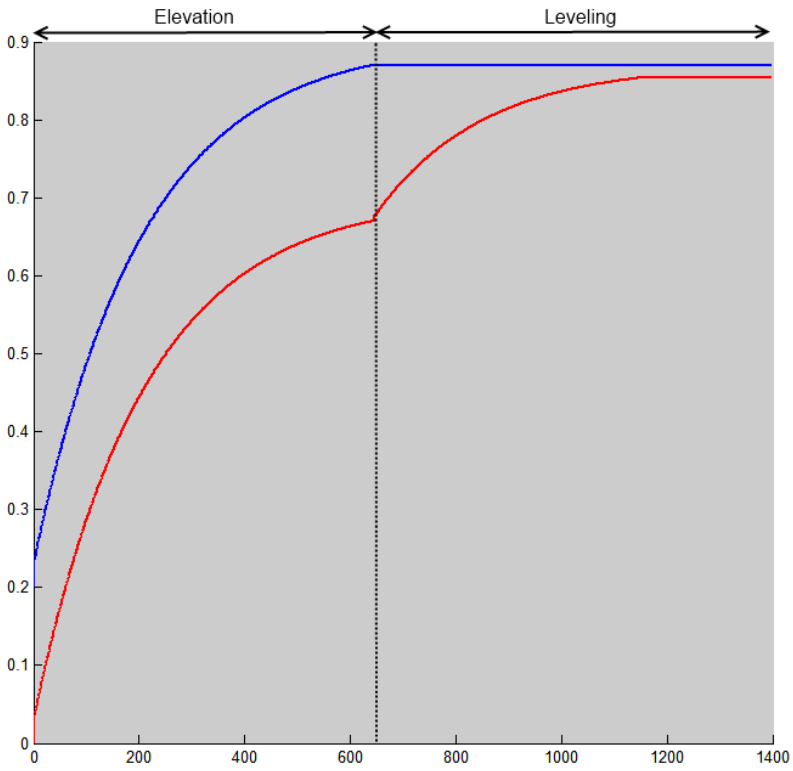


Figure 5.9: Output values from the simple external simulation program utilizing the controller source code generated in Simulink plotted in Matlab. The figure illustrates the correct operation and state transitions of the controller.

function representation of the controller program module was created. Two parallel test harness structures were created, where one was identical to the one presented in Figure 5.7 and the other had the controller model replaced with the controller S-function block. Scope blocks were connected to show the difference between the outputs of the physical model in each case. Figure 5.10 shows the plot of this difference for the extension of the left hydraulic cylinder. The plot shows that the error in the simulation result for the discrete controller model and the S-function is in the magnitude of 10^{-16}, which is negligible.

5.4 Implementation using OpenModelica

To demonstrate its strengths, the approach for modeling the physical system with OpenModelica makes use of the Modelica Standard Library. Specifically, the Multibody mechanics library [28] was used to model the side projection of the R110 drill rig. Model parameters such as spring constants and damping factor values for the side projection were calculated the same way they were for the angled front projection in the Simulink example. The other alternative would have been to implement the mathematical spring-mass-damper model as in Simulink, but this would not ha-

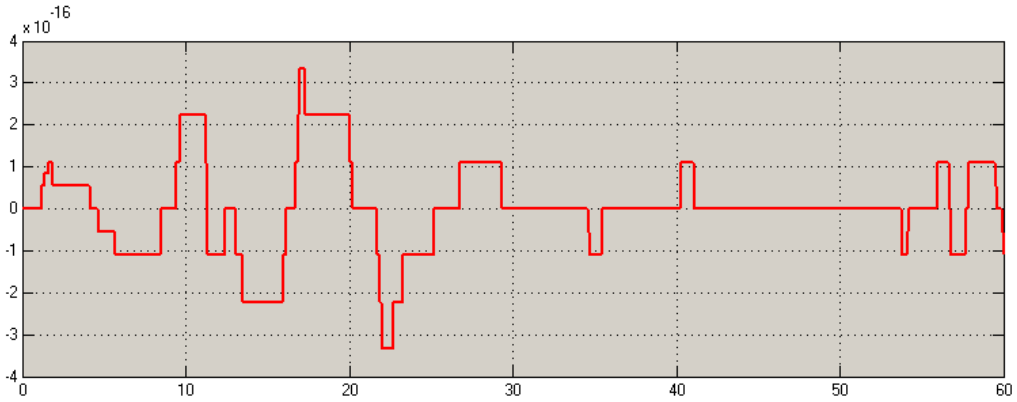


Figure 5.10: Plot of the error between the simulation results for the discrete controller model and the controller S-function block.

ve been the natural way to represent such a system with OpenModelica given the available tools.

The modeling tool in OpenModelica is OpenModelica Connection Editor. It provides a graphical modeling environment for constructing models from library components. A component can be added to be model by dragging it from the library browser onto the model diagram, where its parameters can be edited in the attributes dialogue. Components can be graphically connected together by drawing a connection line from one connection port to the other.

The hull and mast of the rig are modeled as a rigid body, consisting of three elements: two body elements making up the hull and the third body element perpendicular to the hull representing the mast. The dimensions of the entire structure are 9 meters in width and 28 meters in height. The hydraulic actuators were modeled using components that represent a spring and damper connected in parallel. These take into account the spring and damping effects of the hydraulic actuators themselves and those of the ground. Fixed grounding points were added to the model to fix the spring-damper components to given points in space. These fixed coordinates are propagated through connections and positional translation properties of components. As a result, the positions of each of the components in space can be calculated. The spring-damper components were connected to the fixed grounding points and perpendicularly to each end of the hull at their other end. A World component was added to represent a world coordinate system and to define the gravity field affecting the model. To model the force exerted on the hull by the hydraulic actuators, two world force components were connected to the connection points at opposite ends of the hull. Two input ports were added to the model and connected to the world force components. Two output ports were added to the model and connected to the vertical position measurements of the left and right sides of the

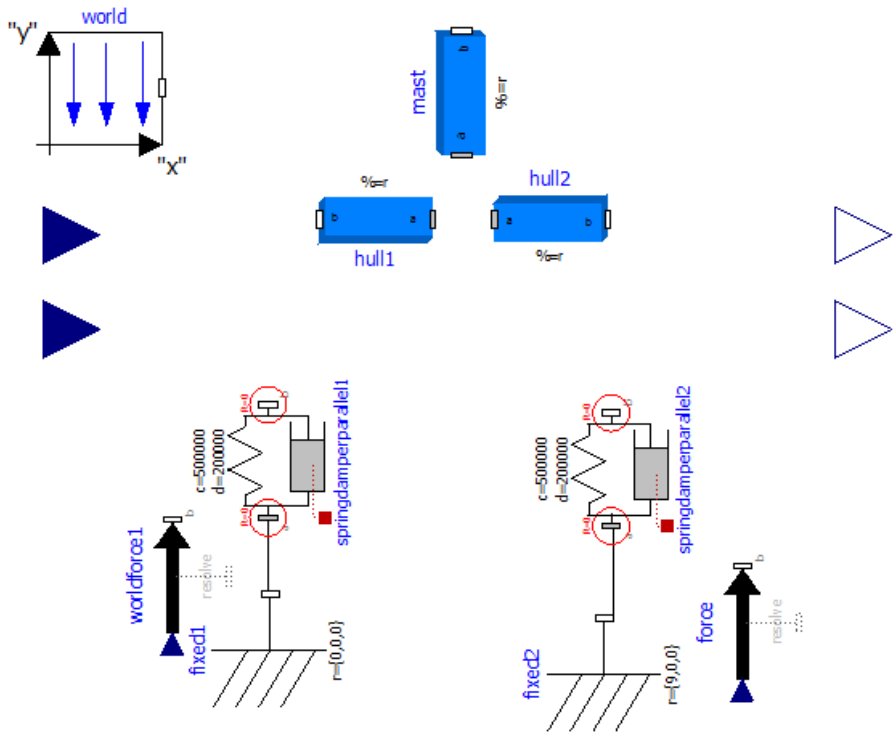


Figure 5.11: Graphical representation of the physical model of the R110 drill rig in OpenModelica Connection Editor.

hull.

During the modeling of the physical system, an issue was discovered with the graphical model editor. The force values given to the model as inputs are formatted vectors that contain the forces acting on the x, y and z axes. Where the component specifications state that they should be able to operate on vector inputs, the graphical editor would give an error when a vector format input was connected to a world force component. Using the text editor, it was possible to explicitly define the dimensions of the components' connection ports. Using the text editor to add connections to the model caused an issue with the graphical editor, where it would not allow any connections to be made until OpenModelica Connection Editor was restarted. This was such an inconvenience that most of the development after this point was done in the text editor. Figure 5.11 shows the incomplete graphical representation of the physical model of the R110 drill rig in OpenModelica Connection Editor. The source code listing of the physical model is presented in Appendix A.1.

The controller implementation in OpenModelica was done entirely in the text editor, due to the problems with the graphical editor. Modelica Standard Library does not provide sufficient tools to implement a discrete control system. An external library that contains discrete components, a discrete PID controller for example,

exists, but was not used in this thesis. The control system was implemented using time-continuous control components, mainly the PID controller. The intention was to translate the controller design from the Simulink example to OpenModelica. Modelica Standard Library provides standard signal processing and control components. It also provides the StateGraph library for implementing state machines [29]. Modelica Standard library does, however, not directly provide a way of observing signal settling to control the state transitions of the controller the way it was done in the Simulink model. It was decided that the leveling controller would not be implemented in OpenModelica.

The elevation controller takes the measured values for the positions of the left and right sides of the machine's hull as inputs. The average value of the inputs is passed as feedback to a PID controller component. The set point of the PID controller connected to the other input of the PID controller. The output of the controller is fed through a gain component. The elevation controller controls both ground jacks equally, so the force vector passed to both outputs is constructed such that the control signal is assigned to the y axis of the force vector while the x and z axes are set to zero. A constant force value is added to the Y axis to represent the counter force the ground exerts on the machine's body due to its mass, so that it does not fall through the ground. The counter force effect was modeled in the controller model because it was more convenient, logically it would be better to include it in the physical model. The set point of the controller is defined in the controller model instead of it being an input to the controller. The source code listing of the controller is presented in Appendix A.2.

A test harness was created to construct the closed-loop control system. The outputs of the physical model representing the measured values of the vertical positions of the left and right sides of the hull were connected to the inputs of the controller. The outputs of the controller representing the control signals for the left and right ground jack actuators were connected to the inputs of the physical model. The source code listing of the test harness is presented in Appendix A.3. Figure 5.12 shows the results of the compiled model executed in the OpenModelica simulation environment and plotted. The plotting tool allows for any property of any of the components within the model to be plotted, which makes it possible to view and analyze the internal states of the model during simulation. The Multibody library in Modelica Standard Library also includes defitions of visualization properties for some of its components. OpenModelica offers a visualization tool that can be used to visualize the simulation of a model throught the Modelica3D library [18]. Figure 5.13 shows the 3-dimensional visualization of the machine during the execution of the test harness model.

The simulation environment in OpenModelica uses OpenModelica Compiler to

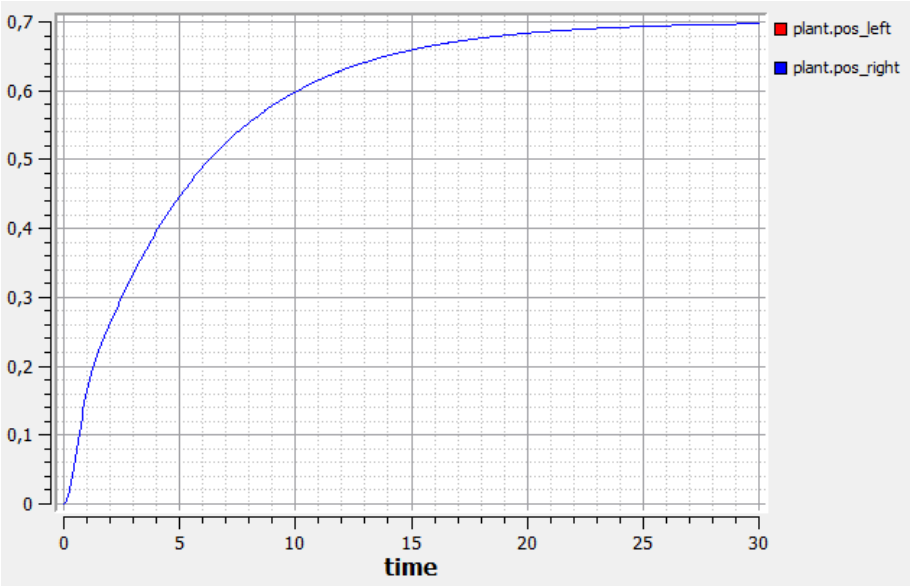


Figure 5.12: The simulation output from the OpenModelica simulation environment for a scenario where the machine hull is raised 0.7 meters. The machine model is symmetrical and the ground levels on both sides are equal, resulting in the left and right sides being raised identically.

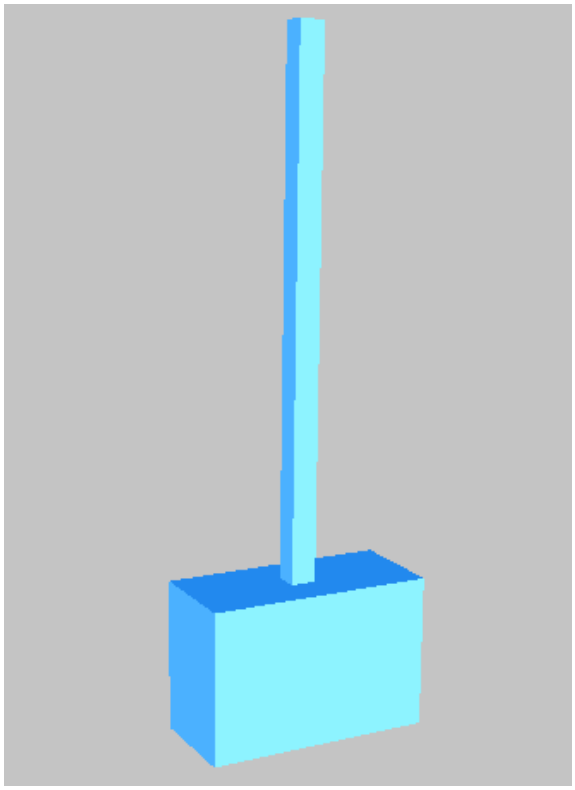


Figure 5.13: 3-dimensional visualization of the simulation of the test harness model created using Modelica3D. The object shown is the machine body consisting of the hull and mast.

compile the Modelica model into executable program code. The possibility of using the compiler to generate code that could be executed outside of the OpenModelica simulation environment was researched. OpenModelica compiler was executed from the command line to generate the program code used for simulation. When inspecting the generated code, the first observation was that the file was extremely large in size and contained a lot of programmatic annotation related to the simulation process. The other observation was that the code was closely dependent on the OpenModelica simulation libraries for execution. As a conclusion of its code generation capabilities, OpenModelica does currently not offer the ability to generate program code that could be integrated in to an external software project.

5.5 Analysis and conclusion

The analysis of the tools' suitability for Sandvik's use case can be divided into two parts. Firstly, the tools capabilities in actual design and implementation work in the modeling environment presented in this chapter are analyzed. Secondly, the tools' suitability for adoption and use as a part of the software development tool chain are assessed. The tools' suitability for control system design is the main requirement. This includes the ability to define system models and to design and implement controllers through modeling and automatic code generation. It must also be possible to test the controller design against the system model. Both Simulink and OpenModelica can be used to develop both system and controller models and to simulate the behavior of the system model when its inputs are being controlled by the controller model. A strong point of OpenModelica is its ability to describe physical systems through the application area specific libraries. The physical system models are descriptive for engineers of a specific domain, with modeling components representing actual physical components and their properties, which makes the design process easier. In Simulink, creating a design often requires the additional step of translating the design idea into Simulink modeling constructs. For example, in Modelica an electrical circuit could be modeled using components from the Electrical component library, inherently describing the electrical properties of the circuit model, where as in Simulink the electrical properties of the circuit would have to be explicitly defined through model structure and parameters.

For controller design, Simulink offers a far superior tool set in terms of its block library and analysis tools. The block library in Simulink offers a large variety of both time-continuous and time-discrete blocks that integrate with Stateflow charts. This allows the development of algorithmically complex, control systems with state machine logic. Simulink also supports the development of structured models through its subsystem and model reference features. OpenModelica offers the basic tools for creating controller models, but lacks the library components for time-discrete cont-

roller design at the time of writing. In addition to simulation result visualization, Simulink provides the ability to perform further data analysis and visualization in MATLAB, making it more versatile in terms of analysis. It also offers a variety of tools for verification, validation and testing of designs and implementation models as separate toolboxes. Simulink models' ability to access the MATLAB work space makes it possible to import and export simulation data, which can be useful for control system testing. OpenModelica's tools for analysis in controller design are effectively limited to its simulation environment. Lastly, the code generation functionality that the Simulink Coder and Embedded Coder toolboxes provide in Simulink seems versatile and effective to the point, where not using the code generation option to some extent seems wasteful. At the time of writing, the option of using the developed controller models for automatic production code generation does not exist in OpenModelica.

The other aspects to consider about the modeling tools are related to their viability as software development tools with regards to usability, support and cost. In usability, Simulink is far ahead of OpenModelica. The graphical editor in OpenModelica seems more like a demo than an actual development tool. During the development, several cases were encountered that required the model to be edited in the text editor. The graphical editor would also not work together with the text editor as making changes to the model in the text editor would often put the graphical editor into a state where it was unusable. Even for the cases where the graphical editor did work, there were issues with objects sticking to the mouse cursor, selecting objects by clicking, moving objects and opening context menus. Simulink's user interface is functional and can be used efficiently once the user gets familiar with it.

The concepts of tool cost and support go hand-in-hand for both Simulink and OpenModelica. Simulink, being a commercial tool offers full product support and extensive documentation complemented by examples for its features. Its status on the market also makes it possible to find reference designs and answers to specific problems. Literature covering the use of Simulink for specific design problems is also plentiful. For OpenModelica, the situation is the opposite. The available documentation is limited to the manual available for the OpenModelica tools, the documentation and examples provided with Modelica Standard Library and a handful of publications. Finding answers to specific design problems when using Modelica is challenging.

On the other hand, Simulink is expensive and highly productized. The functionality that it provides is divided into different toolboxes, each requiring the purchase of a separate license. For an organization that wants to adopt Simulink as a part of the development tool chain the cost of adoption will be significant. The licensing of toolboxes creates additional work, when the decision to purchase Simulink licenses

is not enough, but the need for specific toolboxes needs to be assessed. OpenModelica in this respect is a desirable option, with its open source license it is available for developers to download and use. Lastly, adopting either tool for use as a part of the software development process will require commitment to that specific tool. Both tools use a language and work flow which are proprietary, meaning that models developed with either tool are not portable.

Based on the experiences of the work carried out in this chapter, Simulink seems like it is suitable for its intended use of modeling, simulation and code generation. Its strengths lie in its versatility, support and the competence available on the market, resulting from its wide-spread use in the industry. OpenModelica, at this point, cannot be considered a viable alternative. It is not versatile or mature enough for production use and the missing code generation functionality makes it unsuitable for the use case presented in this thesis.

6. SOLUTIONS TO PRACTICAL CHALLENGES

6.1 Model structure and style

Developed models should be structured, have a consistent style and be properly defined and configured. As for the consistency of modeling style, there is value in purely ensuring that the models developed within a project and across the organization have a uniform structure and style. Stylistic aspects include both structural and implementation details which improve model readability. This facilitates development when multiple developers may work on the same model and improves maintainability and reusability because it is easier for a developer to form an understanding of an unfamiliar model.

The other goal of managing model structure, style and configuration is ensuring compliance with requirements imposed by the intended software and hardware environments of the controller executable compiled from generated code. The requirements imposed by the software environment are satisfied partly by the Simulink Target Language compiler configuration and partly in the model. Maintaining structure in the model allows the parts of the model affecting its software interface to be clearly defined, checked and maintained. Requirements mandated by the hardware environment are concerned with the target hardware's program and variable memory size, processing speed and its supported and optimal data types. The program's size and execution speed are affected by the the configuration of the code generator, mainly the optimization options. While the Simulink code generator is capable of automatically converting the data types used in the model to ones that are supported by the target hardware, it is a good practice to use supported data types in the model. Namely, it is common practice to use floating-point variables in the development phase of a model and then convert them to fixed-point in the implementation phase, before code generation.

The basis for the use of rules and guidelines is a style guide document, which specifies the stylistic and structural rules that models should comply with. The rules presented in the style guide document should be universal and should, as such, not affect functional aspects of the model. The purpose of the style guide is to ensure that functional program code can be generated from models developed according to the style guide and to provide a basis for the uniformity of models across the organization. Examples should be available of correct interpretations and

implementations of the design rules. These examples should demonstrate how each of the rules can be implemented when developing a solution for an actual design problem. A way of doing this is to have a working reference modeling project, where implementations of the design rules can be found. Specification and implementation of the reference modeling project is not covered further in this thesis.

The style guide serves as a basis for communicating design rules. One way of communicating the design rules is organized training, where the rules are presented and rationales behind the rules are explained. Another way of enforcing the developed models' compliance with the rules specified in the style guide is the use of automatic checks within the development environment. Certain aspects of the model's structure can be automatically checked and the developer can be notified of any non-compliant models. This eliminates some of the risk of human error in the development process. The checks that are automatically executed on the model should be carefully chosen and maintained, since any unnecessary or out-of-date checks only cause unnecessary error messages and will get ignored by the developers over time.

6.1.1 Sandvik Modeling Guidelines

Rather than start the development of their style guide from scratch, many organizations have chosen to use the style guide developed by the MathWorks Automotive Advisory Board (MAAB) as a basis [46, p. 1][39]. To determine whether the MAAB modeling guideline collection [60] could be used as a basis for Sandvik's modeling guidelines, an initial analysis of the MAAB guidelines was conducted. The guidelines in the collection are divided into categories, each covering an area of the modeling process. Each guideline is denoted by a title and a unique identifier label. The guidelines have been assigned priorities, enumerated as either mandatory, strongly recommended or recommended. In addition to being under a certain chapter in the guidelines document, each guideline is also categorized by the rationales behind its existence. The rationale categories used in the MAAB guidelines document are readability, workflow, simulation, verification and validation and code generation. Based on this content overview, it was chosen that Sandvik's modeling guidelines collection would be based on the MAAB modeling guidelines.

During initial analysis of the MAAB guidelines, it was determined that each entry in the guideline collection should be analyzed and an assessment should be made on whether or not it is suitable and beneficial for Sandvik's control system development process. As such, the goal was set to selecting a subset of these guidelines as the baseline for Sandvik's modeling guidelines. The analysis done as a part of this thesis, as described in this chapter, serves as a recommendation and a basis for further analysis.

The focus of the analysis was to identify suitable guidelines related to model structure and hierarchy, automatic code generation output and model readability and to eliminate guidelines which didn't seem suitable or were too specific or unclear. Some guidelines were excluded because it was apparent that they were only relevant in an organization employing modeling on a very large scale. Others were excluded because their purpose wasn't completely clear from the description or because it seemed that their provided benefit was significantly less than the effort required to comply with them. During the analysis process, the guidelines in the MAAB collection were categorized as either suitable, partly suitable or non-suitable for Sandvik's development process and needs. A comment specifying the reasoning behind the categorization of each guideline was also written down for future reference. The results were stored into a spreadsheet where each guideline was identified by their title and MAAB identifier and categorized based on which chapter they appear in as well as the keywords that are associated to them. With the guidelines specific to only Simulink Stateflow being omitted from the analysis, Table 6.1 shows the counts for suitable, partially suitable and non-suitable guidelines respectively. Version 3.0 of the MAAB modeling guidelines document dated 31.8.2012 was used.

Suitable	48
Partially Suitable	16
Non-Suitable	12

Table 6.1: The results of the MAAB modeling guidelines analysis in the form of counts of suitable, partially suitable and non-suitable guidelines.

Guidelines covering the software environment for model development and naming rules for modeling entities were accepted with the exception of the naming rules for files and directories, which were deemed too restrictive. The section covering model architecture presents some good general rules regarding the division of functionality between Simulink and Stateflow and the use of subsystems in Simulink, most of which were accepted. The J-MAAB controller model architecture presented in the architecture section was not accepted in the form in which it is presented in the MAAB guidelines. The controller model is, however, used as a basis for the reference model architecture presented in the section covering model architecture. The MAAB guidelines document presents a good number of specific, mostly readability focused, Simulink modeling rules. Only a few of these were rejected due to not being very clear or beneficial while most were accepted. The sections covering the use of enumerated data and MATLAB functions were accepted with minor modifications.

6.1.2 Model checks

Simulink provides the ability to automatically check certain aspects of the model within the modeling environment. Simulink offers a tool called Model Advisor which allows for models and subsystems to be checked for structural properties, conditions and configuration settings that can result in inaccuracy or inefficiency in simulation and execution [52]. Model Advisor checks can be run separately, or be configured so that they are automatically run before simulation. By default, the tool offers checks that help the developer to ensure that their model has been properly configured for simulation. In addition to these default checks, MathWorks offers purpose-built compilations of checks for more specific purposes as parts of different toolboxes. The Simulink Verification and Validation toolbox provides a collection of automated checks, allowing the model to be checked for MAAB guideline compliance. It also allows for the composition of a custom collection of checks from the MAAB guideline checks as well as the authoring of new checks.

A subset of the available MAAB guideline checks was chosen based on the results of the earlier analysis of the MAAB guidelines document. A new Model Advisor configuration was created, containing checks for guidelines which were earlier categorized as suitable. A total of 24 checks were included in the configuration. The newly composed check configuration was executed against a model from one of the pilot projects, not developed in accordance to the MAAB guidelines, to see whether any discrepancies would be found. As a result, 12 checks were passed and 12 returned warnings about violations of MAAB guidelines. None of the checks returned errors. This collection of MAAB guideline checks should be refined through experiences from imposing these checks on development models along with further analysis of the MAAB guidelines.

To demonstrate the ability to author custom checks, a check that verifies the existence of custom control system platform interface blocks connected to model inputs and outputs was written. Model Advisor checks can be written in the MATLAB language and have the ability to access models through the MATLAB API, which provides methods for searching subsystems and blocks and reading their properties. This allows the developer to programmatically navigate the connections between subsystems to verify aspects of model structure, such as the presence of certain connected blocks. It is reasonable to assume that the custom check authoring capabilities offered by Simulink are versatile enough to satisfy most needs for custom checks dealing with structural or configurational aspects of a model.

As a part of the Simulink Coder toolbox, a collection of checks for ensuring that a model or subsystem is technically compatible with the code generator is provided. These are contained in a separate tool called Code Generation Advisor,

which allows the user to emphasize certain qualitative aspects of the generated code, execution efficiency over program size for example. The selected point of emphasis affects which checks are executed and at which priority. It was decided that the collection of available Code Generation Advisor checks should be analyzed and that a predefined collection should be selected or a custom collection of checks should be composed to enforce good practices in modeling with regards to the technical aspects of code generation compatibility. Detailed analysis of the predefined Code Generation Advisor configurations is not included in this thesis.

6.1.3 Hierarchical and structural models

Hierarchical models are partitioned into subsystems in a way that is analogous to the way functionality is divided into functions in a software module. Each of the subsystems implements a logically separable part of the model's functionality. Generally, a hierarchical model has a top-level layer, which connects the subsystems together and contains the interface definitions for the whole module, as well as a variable number of more detailed layers containing the subsystem designs or references.

Hierarchical structuring of models has considerable benefits for organizing development work during project execution. Hierarchical and structured models are also more maintainable and reusable. A model which has been partitioned into subsystems enables multiple project members to work on different subsystems simultaneously. While this is not really a concern when the scope of models is rather small, it becomes important for larger models. An integral part of an incremental development process is the ability to store and maintain different versions of the software modules for the purposes of configuration management. Models should be stored into the same version control system as the rest of the project's assets when possible. With the ability to store different versions of the same model, it becomes possible to create different product configurations utilizing these distinguished versions. The componentization of the entire model as well as the individual subsystems makes it possible to maintain model configurations and to reuse components in future designs.

This section presents an adaptation of the J-MAAB Model Architecture Decomposition presented in chapter 5 of the MAAB modeling guidelines document [60, p. 32]. The model architecture presented in the MAAB guidelines suggests that a model should be composed of four distinct layers: a top layer, a trigger layer, a structure layer and a data flow layer. The top layer serves the purpose of defining the model interface consisting of its input and output variables. The trigger layer defines the timing and priority of periodical and triggered subsystems or function calls and as such, serves as a way of managing timing and execution order in the model. The structure layer presents the logical division of model functionality into subsystems

as well as the interconnectivity of subsystems and local signals within the model. Finally, the data flow layer presents the detailed implementation of each subsystem in the model.

The top layer is suitable for Sandvik's reference hierarchical model architecture as is. The definition of the inputs and outputs of the model on a dedicated top layer is a good practice and makes it simple to manage the software interface of the generated code. The most significant adjustment to the J-MAAB Model Architecture Decomposition is the omission of the trigger layer. Sandvik's control system platform is responsible for the periodical execution and timing of tasks, thus it should not be a concern on the model level. Eliminating the trigger layer simplifies the model architecture. The structure layer is also suitable as it is presented in the MAAB guidelines. A specific thing to note about the structure layer is that all of the subsystems on the structure layer should be defined as atomic subsystems representing logically separable sections of the control system. While the MAAB guidelines document does not discuss the use of Stateflow charts in its layered model architecture, it can be argued that they should be allowed on the structure layer. The nature of the charts on the structure layer should be such that, with regards to size, complexity and reasoning for logical division, they are roughly equal to the subsystems defined on the structure layer. The data flow layer is also suitable for Sandvik's model architecture with the addition of allowing subsystems to be used as parts of data flow layer definitions. The reason for this is that in practical applications, the diagram representation of the data flow layer is likely to be very large in size if it is not allowed to be further split into subsystems and components. When the use of subsystems is allowed in the data flow layer, the readability of diagrams can be maintained. Subsystems on the data flow layer may be defined as atomic. Stateflow charts may also be used on the data flow layer. The reasoning behind the use of atomic subsystems is discussed in section 6.4.3. Figure 6.1 shows Sandvik's adaptation of the J-MAAB hierarchical model architecture, consisting of the three layers: top layer, structure layer and data flow layer.

6.2 Traceability

The use of executable specifications inherently maintains a level of traceability between functional requirement specifications, design and implementation in the modeling environment. However, traceability between high-level requirements and the executable specifications needs to be created and maintained manually. At Sandvik, high-level requirements are stored in a separate web-based application life cycle management system, Polarion. For implementing traceability between requirements that exist in Polarion and Simulink models, a Simulink plugin is available. This plugin, called Polarion Connector for MATLAB Simulink, enables developers to link

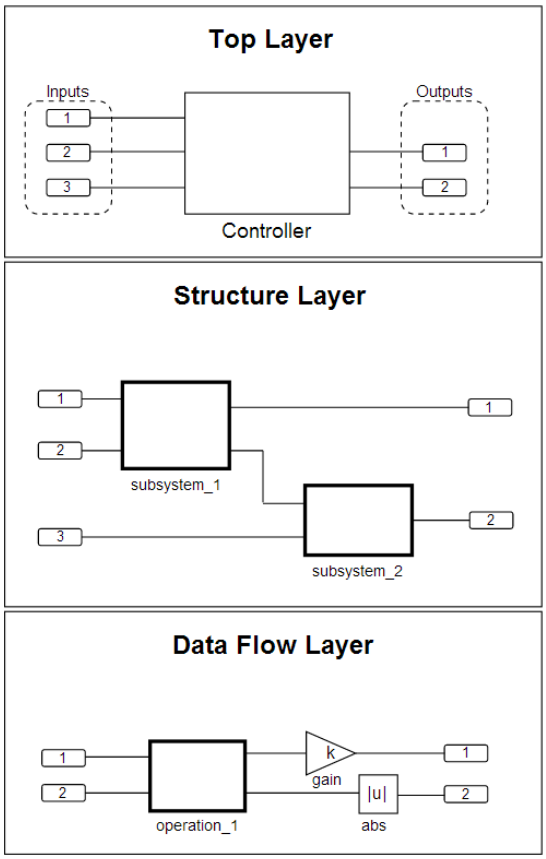


Figure 6.1: Sandvik's hierarchical model architecture, as adapted from the MAAB modeling guidelines [60, p. 32].

Simulink models or subsystems to Polarion work items through the Polarion Web-Service API. When a Simulink model or subsystem is linked to a Polarion requirement, it is possible, within Simulink, to see which requirements a specific Simulink asset satisfies. A direct link to view the requirement is also shown to the developer working within Simulink. From Polarion's point-of-view, the Simulink assets linked to a specific requirement are shown in that requirement's details. If the developer chooses to do so, it is also possible to publish the block diagram representation of the model or subsystem so that it is viewable directly in Polarion. [34]

The code generation process also creates a discontinuity in the traceability of development assets. It is desirable that functionality, which is implemented in automatically generated program code is traceable to its source model or subsystem and, further, to its specification and requirements. Traceability through code generation is implemented by Simulink Coder. The code generator can be configured to automatically include information about the origin of each distinct section of code as comments in the generated source code. Even though readability of automatically generated source code is not a major concern, model-code traceability can be

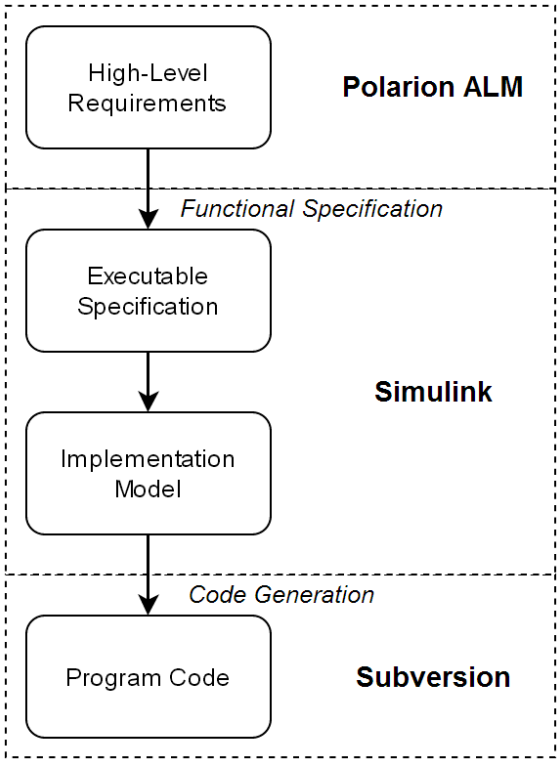


Figure 6.2: Traceability in Sandvik's control system software development process.

useful when issues arise in system-level verification and it is necessary to trace the error to its origin in the model. Tracing the source of an issue found during program execution is also facilitated by consistent, hierarchical model architecture. Figure 6.2 shows the traceability chain, and the points of discontinuity between different environments.

In practice, the Polarion Connector plugin delivers most of what it promises, yet, in the most recent version available at the time of writing, it has issues that stand out. The most obvious problem is with the way that Simulink items are displayed in Polarion, when they are linked to an existing requirement. The only reference that is added to the requirement is a hyperlink, which if opened on a machine that has a Simulink installation, will open the linked Simulink item for viewing. The hyperlinks are, however, not useful or descriptive for users without Simulink. Another problem is that while it is possible to publish a screenshot of a Simulink diagram as a new work item in Polarion, it is not possible to do this so that the screenshot would be added to an existing work item. The diagram publishing feature also suffers from the issue of limited image size, which is described in the following section. The two main problems coupled together cause the traceability from Polarion to Simulink to be only partially implemented by the Polarion Connector plugin.

6.3 Model Documentation

Documenting the design artifacts developed in the modeling and simulation environment is necessary in order to enable reuse and maintenance and to allow communication to peers who may not have access to the modeling tools. An example of this would be when the control logic for a machine has been implemented as a model and engineers of other disciplines, electrical or hydraulic for example, are taking part in reviews of the control logic. Model documentation should cover the architecture, implementation and interface of the model. In cases where there are multiple versions of a model, documentation specific for each version should be available. Simulink offers the ability to automatically generate documentation of the contents of a model as a navigable HTML report. This HTML report covers an entire model or subsystem from the top layer to the data flow layers containing the implementation, which can serve users with different interests. The generated HTML report can be stored in version control along with the models, allowing documentation to be versioned. Automatically generated documentation puts an emphasis on the descriptive naming of subsystems, signals and interface ports.

In practice, the HTML report generated by Simulink is functional for smaller models, where diagrams are not as expansive as in larger models. The problem with using Simulink's HTML report generator for larger models is that the diagrams used in the HTML report are stored as images in the Portable Network Graphics (PNG) format. Although the PNG format is lossless, the HTML report generator appears to limit the pixel size of the images it renders, making it so that details in larger diagrams become unreadable. This is especially harmful for signal names which, by default, use a smaller font. As a result of this, the HTML report is not a sufficient solution for model documentation.

MathWorks offers a separate Simulink toolbox focused on generating model documentation. The toolbox is called Simulink Report Generator and it offers a variety of configuration options for the automatic generation of documentation from Simulink models. According to the documentation, it offers the ability to configure the layout and contents of the generated report in a variety of formats, specifically the Scalable Vector Graphics (SVG) format is available as an output format for renderings of Simulink model diagrams [57]. Its main advantage is direct integration into Simulink and product support from The MathWorks. The downside is, however, the fact that it does require the purchase of a separate license only for the purpose of report generation. Simulink Report Generator was not evaluated as a part of this thesis, but it is an option which is worth considering in the future.

Another way of satisfying the requirements for model documentation is using an external, independent model viewer which does not require MATLAB and Simu-

link licenses so it can be made available throughout the organization. One such tool is DiffPlug by DiffPlug LLC, which offers the ability to view and browse Simulink models and Stateflow charts [7]. The use of an external tool does not have the convenience of a HTML or PDF report, but it does provide a way of viewing proprietary model files, giving its users access to design and implementation information contained within. An external viewer tool also supports the idea of using model assets as executable specifications. It allows model assets to be accessed directly in the version control repository, without the additional step of generating the documentation. Besides its main functionality of viewing Simulink models, DiffPlug can analyze the model's dependencies on external libraries and track signals through different diagram layers. All of the features described here are included in the free version, which makes it a desirable option due to not requiring the purchase of any extra licenses. DiffPlug was tried with the R110 example Simulink model developed in chapter 5 as well as a number of Sandvik's Simulink models. During initial evaluation of the tool, all of the Simulink models and Stateflow charts used in the evaluation process were displayed correctly. Although this evaluation doesn't provide definitive proof of the correctness of the viewer's interpretation of Simulink models and Stateflow charts, it suggests that DiffPlug is suitable for use as a means of providing visibility into proprietary Simulink model files as well as communicating design and implementation information.

6.4 Reliability of the code generator

The central role of the code generator makes its reliability essential to the model-based development process. Raising the level of abstraction in development relies on the trustworthiness of the code generator. This trust can be built on results from trials and research done within the organization as well as research and experiences from the industry.

6.4.1 Existing Research

Simulink Coder has a long presence on the market [25] and is widely used in the industry for prototyping and production code generation [37] [47] [63] [61] [46]. This in itself suggests that it is suitable for larger scale adoption and viable for production code generation of even safety-related software applications. It has been indicated that it can produce program code that is functionally equal to and qualitatively comparable to hand-written code [2, pp. 15-16, 43-47].

Research done on the topic of the quality of automatically generated code shows that compared to manual programming, programming errors are reduced [14] [44, p. 5] [35]. With automatic code generation in general, it has been shown that syn-

tactic and data flow errors are significantly reduced. The nature of the errors found in automatically generated code is systematic. Typical errors found in automatically generated code are related to incorrect configuration of the code generator or incorrect modeling representations of structures and data in the models [44, pp. 3-4].

The MathWorks claims that using Simulink Coder, the cyclomatic complexity of the automatically generated code should be in correspondance to that of the source model, although the measured complexity of the code could be slightly higher than the indicated cyclomatic complexity of the model due to certain error checks [58]. This is backed by research, which shows that for Simulink Coder, the cyclomatic complexity measurements are equal between the generated program code and the source model in the cases that were a part of the study [36, p. 38]. Research conducted using a competing code generator, TargetLink by dSpace GmbH, show that the complexity of the generated code could even be lower than that of the model and that it is highly affected by the optimization options of the code generator [44, pp. 5-7]. While the results of the research conducted using the TargetLink code generator may not be directly applicable, it gives an indication of the general characteristics of code generated from Simulink models. Simulink Coder and TargetLink have been found to be comparable [2, p. 48].

In addition to the the technical support offered by the supplier, Simulink Coder has been certified by TÜV SÜD Automotive GmbH, for use in development of IEC-61508 part 3 compliant safety-related software systems. This implies that the model-based design process showcased by MathWorks, utilizing Simulink Coder for code generation, complies with the traceability and quality requirements imposed by the standard. Simulink also provides support for development processes compliant with other standards such as DO-178B and MISRA C. While these are not definitive indicators of quality, they do give an indication of the level of maturity of the code generator and the level of supplier involvement and support for the development processes of their customers.

Based on the assessment of existing research and material on the topic of code generation from Simulink models, it is feasible to assume that Simulink Coder is reliable enough to warrant a development approach where it is assumed that the code generator correctly translates the model into program code. Any issues or inconsistencies found in the software modules should be fixed in the model, not in the code.

6.4.2 Metrics from projects

To support the assessments made based on existing research, data was collected from the pilot projects carried out at Sandvik. Both static and dynamic metrics

for software modules were collected from a few different projects with the purpose of evaluating the properties of automatically generated software modules to those of hand-written ones. While a direct comparison, where the same module had been implemented using both methods, was not available, they do have similar properties. The modules that were implemented through modeling and code generation are similar in the sense that they implement a mathematically complex algorithm, which performs an intelligent machine function. The hand-coded modules chosen for comparison implement similar pieces of machine functionality. It should be noted that Module D is responsible for implementing a core part of the machine's functionality and is generally larger than the others.

The dynamic metric which was chosen for this analysis is application cycle time. It shows how much time each execution of the cyclical application takes on the controller hardware. The static metrics collected for this analysis are code size, McGabe's cyclomatic complexity number and the complexity of the software module's interface, described by the number of signals going into and coming out of each software module. Together these metrics give an idea of the module's size and complexity. Tables 6.2 and 6.3 present the collected data.

The first observation that can be made is the fact that the automatically generated modules have a significantly higher cyclomatic complexity number than the hand-written modules do. This can be attributed to three factors, the first of which is the fact that the algorithms they represent are mathematically complex by nature, often resulting in implementations involving nested loop structures. The second factor which affects this is the structure of the software modules. While the hand-written software modules are often split into multiple files and functions, the code generator generates one file where most of the functionality is contained in a single function. The third factor which has been found to affect the complexity of the generated program code are the optimizations carried out by the code generator [44, p. 5]. While cyclomatic complexity numbers in excess of 10 are generally considered too high, the model-based design approach justifies sacrificing readability for compact and efficient program code since readability of automatically generated code is not something that needs to be considered.

The sizes of the automatically generated modules are fairly consistent. Compared to the hand-coded modules they are generally slightly more compact, taking into account their relative complexities denoted by the cyclomatic complexity number and the number of input and output signals, which would also indicate that the Simulink Coder code generator creates compact code. The measured execution times are fairly similar for generated and hand-coded applications. Although not directly comparable or definitive, it is reasonable to assume, based on the measured execution times, that the code generator generally produces acceptably efficient code.

Module	Cycle Time (μs)	Lines of Code	Avg. Cyclomatic Complexity No.	Signals In	Signals Out
Module A	200	7600	18,34	21	10
Module B	1500	5495	13,58	17	9
Module C	450	3330	25,03	26	8

Table 6.2: Metrics for program modules generated from Simulink models.

Module	Cycle Time (μs)	Lines of Code	Avg. Cyclomatic Complexity No.	Signals In	Signals Out
Module D	600	22300	6,24	58	82
Module E	140	6500	6,82	37	39

Table 6.3: Metrics for hand-coded program modules.

Together, the research done in the industry and the data gathered from in-house trials indicate that the overall reliability of the Simulink Coder code generator is good. With this information, the transition to the model-based approach is warranted. This means that the automatically generated program code assets should mainly be treated as an intermediate representation of the models and as such, should not be subject to detailed source code reviews or module testing on the program code level. Omitting these quality assurance activities on source code level means that they should be carried out on model level.

6.4.3 Resolving performance issues

The model-based approach emphasizes tackling issues on the model level. In practice, it is likely that certain issues are only discovered when the control algorithm is executed on the target controller, in its proper hardware and software environment during system integration or system testing. Traceability between the model and the automatically generated and then separately compiled source code for software modules, as stated before, is a key factor in tracking down and fixing these issues. This is true for both functional and non-functional issues. Tracking a functional issue from a compiled software module to the specific part of its source model which is causing it is facilitated by consistent, hierarchical system and model architecture. When the source for the issue has been identified, the part of the model implementing the misbehaving system functionality can be fixed, the code generator invoked and the control system executable recompiled.

Locating and fixing issues caused by non-functional aspects of the control system presents a challenge. One specific issue that arises from the discontinuity caused by the automatic code generation process between the model and the program compi-

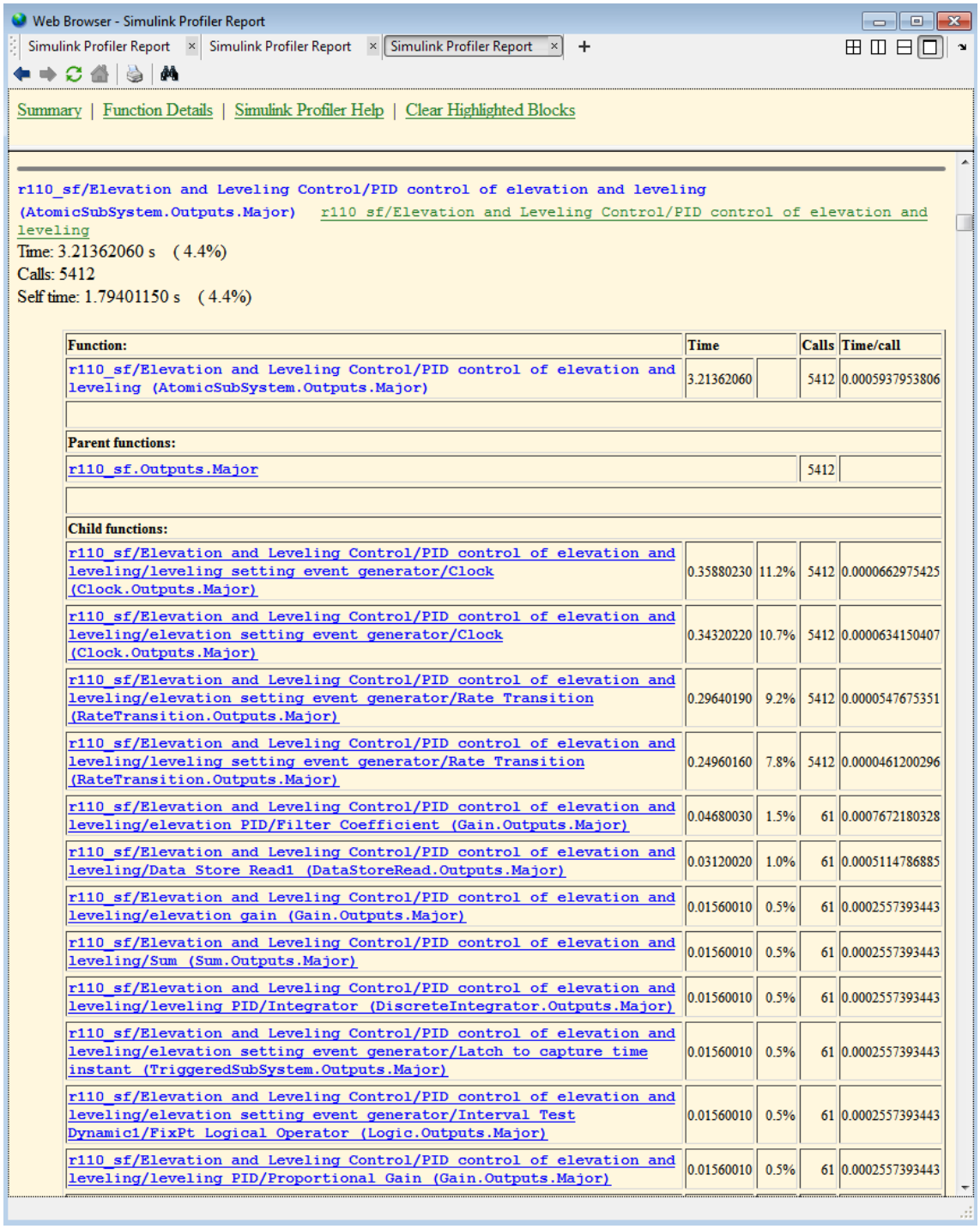
led for the controller hardware is that of performance. While the model may behave correctly in simulation, it may not be possible to execute it in real-time on the controller hardware due to a lack of or contention for resources such as memory or execution time on the processor. The actual execution environment of a software module may have other applications running in parallel with it, restricting the amount of resources a single application can take up for the entire system to be able to run without real-time deadline violations. While the system architecture should take into account how software is allocated to computation units, cases where a software module takes up more execution time than was estimated will arise. These may by caused by unoptimized algorithms where real-time execution is possible on the simulation PC, but is too computationally intensive for the embedded controller or a faulty code generator configuration, where the characteristics, such as floating-point calculation capabilities, of the embedded controller have not been properly captured. It is therefor important to have the ability to resolve such performance issues.

The first step in identifying the source of a performance issue within a model is performance profiling in the simulation environment. Simulink offers a tool called Simulink Profiler for profiling model execution during simulation. During the execution of a simulation scenario, the profiler collects data on how much time was spent executing each atomic block and subsystem in the model. The notion of the atomic subsystem is important here, because it allows for hierarchical structures to be defined for the simulation. Without defining subsystems as atomic, it can be very difficult to see how execution time of the model is divided between the logically separated sections of the model. The contents of an atomic subsystem are evaluated together, therefor generating a separate entry in the profiler report [55]. Each subsystem whose performance is to be individually analyzed needs to be defined as atomic.

The Simulink Performance Profiler report shows the division of accumulated execution time between the atomic blocks and subsystems in the model. The pseudo code scheme for how the model is executed during simulation is presented in Listing 6.1. For a time-discrete controller, the main concern should be the *Outputs.Major* time, denoting the time it takes to calculate the outputs of a specific atomic subsystem on a major time step [56]. The goal when studying the performance profiler report should be to identify any subsystems which are taking exceedingly long to execute, possibly indicating a problem with their internal implementation. In situations where the software module is exhibiting performance issues on the embedded controller hardware and the performance profile doesn't reveal any significant inconsistencies in subsystem execution times, it may be the case that the code generator has not been configured properly or the proper configuration has not been selected.

```
Sim()
    ModelInitialize().
    ModelExecute()
          for t = tStart to tEnd
          Output()
         Update()
         Integrate()
             Compute states from derivs by repeatedly calling:
                 MinorOutput()
                 MinorDeriv()
             Locate any zero crossings by repeatedly calling:
                 MinorOutput()
                 MinorZeroCrossings()
         EndIntegrate
         Set time t = tNew.
    EndModelExecute
    ModelTerminate
EndSim
```

Listing 6.1: Pseudocode representation of model execution during simulation in Simulink [50].



Function:	Time		Calls	Time/call
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling (AtomicSubSystem.Outputs.Major)	3.21362060		5412	0.0005937953806
Parent functions:				
r110_sf.Outputs.Major			5412	
Child functions:				
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling/leveling setting event generator/Clock (Clock.Outputs.Major)	0.35880230	11.2%	5412	0.0000662975425
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling/elevation setting event generator/Clock (Clock.Outputs.Major)	0.34320220	10.7%	5412	0.0000634150407
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling/elevation setting event generator/Rate Transition (RateTransition.Outputs.Major)	0.29640190	9.2%	5412	0.0000547675351
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling/leveling setting event generator/Rate Transition (RateTransition.Outputs.Major)	0.24960160	7.8%	5412	0.0000461200296
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling/elevation PID/Filter Coefficient (Gain.Outputs.Major)	0.04680030	1.5%	61	0.0007672180328
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling/Data Store Read1 (DataStoreRead.Outputs.Major)	0.03120020	1.0%	61	0.0005114786885
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling/elevation gain (Gain.Outputs.Major)	0.01560010	0.5%	61	0.0002557393443
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling/Sum (Sum.Outputs.Major)	0.01560010	0.5%	61	0.0002557393443
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling/leveling PID/Integrator (DiscreteIntegrator.Outputs.Major)	0.01560010	0.5%	61	0.0002557393443
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling/elevation setting event generator/Latch to capture time instant (TriggeredSubSystem.Outputs.Major)	0.01560010	0.5%	61	0.0002557393443
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling/elevation setting event generator/Interval Test Dynamic1/FixPt Logical Operator (Logic.Outputs.Major)	0.01560010	0.5%	61	0.0002557393443
r110_sf/Elevation and Leveling Control/PID control of elevation and leveling/leveling PID/Proportional Gain (Gain.Outputs.Major)	0.01560010	0.5%	61	0.0002557393443

Figure 6.3: Simulink Profiler report for the Tamrock R110 ground jack controller.

7. CONCLUSIONS

The goal of the thesis was to explore ways to support the use of model-based design, especially automatic production code generation, in control system software development. The work flow of a development process where functionality could be designed and implemented by modeling and code generation is outlined and considerations regarding the use of models as design and development assets are presented.

As modeling and simulation tools, Simulink and OpenModelica were analyzed through an example design problem. OpenModelica shows promise in the area of physical systems modeling, but lacks in the areas of controller design, analysis tools, product support and code generation capability. The design software is also not mature enough for production use. Simulink supports model-based design through its core functionality and a selection of toolboxes. A controller was designed in Simulink and a program code implementation was created using the code generator. The program code representation of the controller model was then used as a part of an external software project to ensure that it is portable and integrateable. While both tools use proprietary modeling languages and file formats and as such require commitment, the product support for Simulink as a commercial tool helps to mitigate the risk.

To support model-based design on the organizational level, establising general modeling standards is recommended. It was found that the MAAB modeling guidelines are a good basis for establishing modeling standards that ensure maintainability, readability and reusability of models. Simulink also offers automated checks to enforce the MAAB modeling rules. A reference for building hierarchical and structured models is also presented in the thesis as an adaptation of the J-MAAB model architecture decomposition. To ensure that models developed for code generation comply with requirements set by software integration and their intended hardware and software environments, custom model checks for essential properties of the models are recommended in addition to documentation. Enforcing modeling standards and rules becomes more important when scope and degree of modeling expand in the organization.

To bridge the traceability gap between the proprietary modeling environment and requirement specifications, Simulink offers the ability to link models to external requirements. For implementing traceability between Simulink models and the

generated program code, the code generator can be configured to insert annotations into the source code that denote the model components that sections of the code originate from. Basic model documentation can be created using the report generator function included in Simulink. In cases where a simple HTML report is not sufficient, DiffPlug, a third party viewer program can be used to view Simulink model files.

Lastly, the thesis addresses the concern for the reliability and performance of the code generator. As a part of the thesis, existing research documents and other materials were analyzed and it was concluded that it can be feasibly assumed that the the code generator is reliable and consistent. To support the research, metrics from projects carried out at Sandvik are presented that show no significant differences between hand coded and automatically generated program modules. This warrants a development approach where it can be assumed that performance issues in generated program code can be fixed in the models. To support this, the thesis covers performance analysis of Simulink models to identify model sections that are taking longer than expected to execute.

7.1 Future Development

To make use of the results of this thesis and to enable further development, responsibilities supporting model-based design efforts needs to be appointed within the organization. These individuals need to be responsible for developing and maintaining modeling practices, tools and documentation. The organization also needs to be concerned with developing or acquiring the required competence in the area of modeling.

Based on the results, the requirements for the modeling standards should be further specified and used as a basis for writing the Sandvik modeling guidelines document and creating a modeling project template. The need for custom model checks that enforce model properties related to proprietary hardware or software environment requirements should be mapped out and the check collection should be created. To support the use of Simulink's code generation functionality, the Code Generation Advisor tool should be analyzed and taken into use.

While automatic production code generation is considered an immediate benefit, it does not realize the full potential of model-based design. After incorporating automatic code generation, other methods, especially early design verification and functional verification in simulation should be explored to achieve further gains from the already model-focused development process.

BIBLIOGRAPHY

[1] Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J. Agile software development methods: Review and analysis. VTT Publications, 2002. pp. 9-15.

[2] Ajwad, N. Evaluation of Automatic Code Generation Tools. M.Sc. Dissertation. Lund University, Department of Automatic Control, 2007. Available: http://www.control.lth.se/documents/2007/5793.pdf. Visited: 24.2.2014. 74 p.

[3] Ananthan, A., Tate, S. Model based systems engineering - Verification and Validation of Medical Device Systems. The Mathworks, Inc., 2013. Available: http://www.incose.org/chicagoland/docs/2013_08%20MBD.pdf. Visited: 17.2.2014. 37 p.

[4] Birta, L.G., Arbez, G. Modelling and Simulation: Exploring Dynamic System Behaviour. Springer, 2007. pp. 3-47.

[5] Broy, M., Kirstan, S., Krcmar, H., Schätz, B., Zimmermann, J. What is the benefit of a model-based design of embedded software systems in the car industry? In: Rech, J., Bunse, C. Emerging Technologies for the Evolution and Maintenance of Software Models. Igi Global, 2011. pp. 343-369.

[6] Dassault Systèmes: Dymola website. Available: http://www.3ds.com/products-services/catia/capabilities/systems-engineering/modelica-systems-simulation/dymola. Visited: 8.4.2014.

[7] DiffPlug LLC. DiffPlug Website. Available: http://www.diffplug.com/. Visited: 19.3.2014.

[8] Dillaber, E., Kendrick, L., Jin, W., Reddy, V. Pragmatic Strategies for Adopting Model-Based Design for Embedded Applications. SAE International, 2010. Available: http://www.mathworks.com/tagteam/63207_Strategies%20for%20Adopting%20MBD%20for%20Embedded%20Apps.pdf. Visited: 3.2.2014. 16 p.

[9] Dorf, R. C., Bishop, R. H. Modern Control Systems. Pearson, 2008. pp. 2-3.

[10] Erkkinen, T., Conrad, M. Verification, Validation, and Test with Model-Based Design. The Mathworks, 2008. Available: http://www.mathworks.com/tagteam/53246_COMVEC%2008%20-%20VVnT%20with%20MBD.pdf. Visited: 5.5.2014. 6 p.

[11] Fiehler, D., Collins, B., Carlaftes, J. Using Model-driven Engineering Techniques for Integrated Flight Simulation Development. Raytheon Missile Systems, 2009. Available: http://www.dtic.mil/ndia/2009systemengr/8980ThursdayTrack4Fiehler.pdf. Visited: 20.3.2014. 15 p.

[12] Franklin, G. F., Powell, J. D., Workman, M. L. Digital Control of Dynamic Systems 3rd Edition. Addison-Wesley, 1998. 737 p.

[13] Fritzon, P., Rogovchenko, O. Introduction to Object-Oriented Modeling, Simulation and Control with Modelica. Workshop at MODPROD 2012, Linköping University. Available: http://www.modprod.liu.se/modprod2012/1.322778/modprod2012-tutorial1-Peter-Fritzson-ModelicaTutorial.pdf. 65 p.

[14] German, A. Software Static Code Analysis Lessons Learned. In: CrossTalk, Volume 16, Issue 11, November 2003. pp. 13-17.

[15] Haikala, I. Märijärvi, J. Ohjelmistotuotanto, 11th edition. Talentum, 2006. pp. 35-98.

[16] Halvorsen, H-P. Control and Simulation in LabVIEW. Telemark University College, 2011. Available: http://home.hit.no/~hansha/documents/labview/training/Control%20and%20Simulation%20in%20LabVIEW.pdf. Visited: 9.4.2014. 50 p.

[17] Hästbacka, D. Developing Modern Industrial Control Applications: On Information Models, Methods and Processes for Distributed Engineering. Ph.D. Dissertation. Tampere, 2013. Tampere University of Technology. Publication 1143. 99 p.

[18] Höger, C., Mehlhase, A., Nytsch-Geusen, C., Isakovic, K., Kubiak, R. Modelica3D - Platform Independent Simulation Visualization. In: Proceedings of the 9th International Modelica Conference, Munich, 2012. pp. 485-494.

[19] Koivisto, H. ASE-2110 Systeemit ja Säätö: Lecture 1. Tampere University of Technology, 2014. Available: https://moodle2.tut.fi/pluginfile.php/174596/mod_resource/content/1/Luento_1_johdanto_2p.pdf. Visited 15.1.2014. 14 p.

[20] Kolu, A. Development of automatic testing concept and administration in mobile machine control system testing. M.Sc. Dissertation. Tampere University of Technology, 2010. 58 p.

[21] Kuhn, T., Kemmann, S., Trapp, M., Schäfer, C. Multi-Language Development of Embedded Systems. OOPSLA Workshop on Domain-Specific Modeling, 2009. 7 p.

[22] Larman, C. Agile and Iterative Development: A Manager's Guide. Addison-Wesley Professional, 2004. pp. 253-254.

[23] Lind, I., Andersson, H. Model Based Systems Engineering for Aircraft Systems - How does Modelica Based Tools Fit?. SAAB Aeronautics, International Modelica Conference, Dresden, 2011. Available: http://www.modelon.com/fileadmin/user_upload/Products/DS/Dymola/references/42_poster_ID_152_a_fv.pdf. Visited: 20.3.2014. 8 p.

[24] Loimusalo, M. Use of real-time simulation in algorithm development of mobile work machine. M.Sc. Dissertation. Tampere University of Technology, 2011. 64 p.

[25] Maclay, D. Click and Autocode. In: IEE Review. Volume 46, Issue 3, May 2000. pp. 25-28.

[26] Maplesoft: MapleSim website. Available: http://www.maplesoft.com/products/maplesim/index.aspx. Visited: 8.4.2014.

[27] McCabe, T. A Complexity Measure. In: IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976. pp. 308-320.

[28] Modelica Association: Modelica.Mechanics.Multibody library documentation. Available: https://build.openmodelica.org/Documentation/Modelica.Mechanics.MultiBody.html. Visited: 16.4.2014.

[29] Modelica Association: Modelica.StateGraph library documentation. Available: https://build.openmodelica.org/Documentation/Modelica.StateGraph.html. Visited: 16.4.2014.

[30] Modelon AB: JModelica.org website. Available: http://www.jmodelica.org/. Visited: 8.4.2014.

[31] Murphy, B., Wakefield, A., Friedman, J. Best Practices for Verification, Validation and Test in Model-Based Design. The Mathworks Inc., 2008. Available: http://www.mathworks.com/tagteam/49392_2008-01-1469_VVTinMBD_SAE08_FINAL_1%2010%2008.pdf. Visited: 12.2.2014. 6 p.

[32] National Instruments Corporation: LabVIEW website. Available: http://www.ni.com/labview/. Visited: 9.4.2014.

[33] Open Source Modelica Consortium: OpenModelica website. Available: https://openmodelica.org/. Visited: 8.4.2014.

[34] Polarion Software: Polarion Connector for Simulink Website. Available: http://extensions.polarion.com/extensions/173-polarion-connector-for-simulink. Visited: 1.4.2014.

[35] Potter, B. Achieving Six Sigma Software Quality Through the Use of Automatic Code Generation. The Mathworks International Aerospace and Defence Conference, 2005. Available: http://www.mathworks.com/aerospace-defense/miadc05/presentations/potter.pdf. Visited: 3.2.2014. 11 p.

[36] Prabhu, J. Complexity Analysis of Simulink Models to improve Quality of Outsourcing in an Automotive Company. Manipal University, 2010. 47 p.

[37] Prabhu, S. M. Deploying Model-Based Design: 20+ Years of Partnership. MathWorks Automotive Conference 2012, The Mathworks Inc., 2012. Available: https://www.mathworks.se/company/events/conferences/automotive-conference-michigan/2012/proceedings/deploying-model-based-design-20-years-of-partnership.pdf. Visited: 4.2.2014. 29 p.

[38] Rantanen, A. Experiences in Platform Approach in Machine Control Systems. Seminar Presentation at Viksut Vekottimet 2013, Tampere University of Technology, 2013. Available: http://www.tut.fi/fi/tietoa-yliopistosta/laitokset/tietotekniikka/laitoksen-esittely/tapahtumia/viksut-vekottimet/index.htm. Visited: 1.4.2014. 10 p.

[39] Saikkonen, A. Embedded Controls In Simulink. Wärtsilä, 2013. Presentation at MathWorks Model-Based Design Conference, Tampere, May 2013. 31 p.

[40] Schubert, P. J., Vitkin, L., Winters, F. Executable Specs: What Makes One, and How are They Used?. SAE International, 2006. Available: https://delphiauto.com/pdf/techpapers/2006-01-1357.pdf. Visited: 12.2.2014. 10 p.

[41] Schätz, B., Spies, K. Model Based Software Engineering - Linking mode and less formal product models to a structured process . In: Proceedings of ICSSEA 2002 15th International Conference on Software Systems Engineering and their Applications. 8p.

[42] Sirén, A. Porauslaitteen automaattinen vaakatasoonajojärjestelmä. M.Sc. Dissertation. Tampereen Teknillinen Korkeakoulu, 1989. 88 p.

[43] Smith, P. F., Prabhu, S. M., Friedman, J. H. Best Practices for Establising a Model-Based Design Culture. The Mathworks, Inc., 2007. Available: http://www.mathworks.com/tagteam/40538_SAE-2007-01-0777-Best-Practices-for-MBD-Culture.pdf. Visited: 4.2.2014. 7 p.

[44] Stürmer, I. Conrad, M., Fey, I., Dörr, H. Experiences with Model and Autocode Reviews in Model-based Software Development. ACM Digital Library, 2006. 7 p.

[45] Stürmer, I., Weinberg, D., Conrad, M. Overview of Existing Safeguarding Techniques for Automatically Generated Code. 2nd International Workshop on Software Engineering for Automotive Systems (SEAS'05). ACM Digital Library, 2005. 6 p.

[46] Tamblyn, S. Henry, J., Rapp, J. Accelerating NASA GN&C Flight Software development. The MathWorks, Inc., 2011. Available: http://www.mathworks.com/tagteam/69872_accelerating-nasa-gn&c-flight-software-development-91876v01.pdf. Visited: 24.2.2014. 3 p.

[47] Thate, J. M., Kendrick, L. E. Caterpillar Automatic Code Generation. SAE World Congress, 2004. Available: http://www.mathworks.com/tagteam/20303_91198_Caterpillar_2004-01-0894.pdf. Visited: 13.1.2014. 8 p.

[48] The MathWorks, Inc. Learning Simulink 6, 5th edition. 2005. Available: http://www.mathworks.com/academia/student_version/learnsimulink_sp3.pdf. 367 p.

[49] The MathWorks, Inc. Simulink Coder: Selecting and Working with Targets. Available: http://www.mathworks.se/products/simulink-coder/description2.html. Visited: 15.4.2014.

[50] The MathWorks, Inc. Simulink Documentation: Capture Performance Data. Available: http://www.mathworks.se/help/simulink/ug/capturing-performance-data.html. Visited: 3.3.2014.

[51] The MathWorks, Inc. Simulink Documentation: Choose a Solver. Available: http://www.mathworks.se/help/simulink/ug/choosing-a-solver.html. Visited: 14.4.2014.

[52] The MathWorks, Inc. Simulink Documentation: Consult the Model Advisor. Available: http://www.mathworks.se/help/simulink/ug/consulting-the-model-advisor.html. Visited: 12.3.2014.

[53] The MathWorks, Inc. Simulink Documentation: Discrete Blocks. Available: http://www.mathworks.se/help/simulink/discrete.html. Visited: 28.4.2014.

[54] The MathWorks, Inc. Simulink Documentation: Overview of Model Referencing. Available: http://www.mathworks.se/help/simulink/ug/overview-of-model-referencing.html. Visited: 28.4.2014.

[55] The MathWorks, Inc. Simulink Documentation: Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CoreReuse Subsystem. Available: http://www.mathworks.se/help/simulink/slref/subsystem.html. Visited: 3.3.2014.

[56] The MathWorks, Inc. Simulink Documentation: Types of Sample Time. Available: http://www.mathworks.se/help/simulink/ug/types-of-sample-time.html. Visited: 27.2.2014.

[57] The MathWorks, Inc. Simulink Report Generator Documentation: Report Generation Options. Available: http://www.mathworks.se/help/rptgenext/ug/set-report-output-options.html. Visited: 19.3.2014.

[58] The MathWorks, Inc. Simulink Verification and Validation Documentation: Types of Model Coverage. Available: http://www.mathworks.se/help/slvnv/ug/types-of-model-coverage.html. Visited: 24.2.2014.

[59] The MathWorks, Inc. Stateflow product website. Available: http://www.mathworks.se/products/stateflow/. Visited: 14.4.2014.

[60] The MathWorks, Inc. The MathWorks Automotive Advisory Board modeling guidelines, version 3.0, 2012. Available: http://www.mathworks.se/automotive/standards/maab.html. Visited: 5.5.2014. 128 p.

[61] The MathWorks, Inc. User Story: Metso Develops Controller for Energy-Saving Digital Hydraulic System for Papermaking Equipment Using Model-Based Design, 2013. Available: http://www.mathworks.com/tagteam/74783_92071v00_Metso_UserStory_final.pdf. Visited: 20.3.2014. 2 p.

[62] The MathWorks, Inc. User Story: Scania Develops Fuel-Saving Driver Support System for Award-Winning Long-Haulage Trucks, 2011. Available: http://www.mathworks.com/tagteam/67381_91835v01_Scania_UserStory_final.pdf. Visited: 20.3.2014. 2 p.

[63] The MathWorks, Inc. User Story: Wärtsilä Accelerates Engine Control Development Using Production Code Generation, 2005. Available: http://www.mathworks.se/company/user_stories/Wrtsil-Accelerates-Engine-Control-Development-Using-Production-Code-Generation.html. Visited: 13.1.2014. 2 p.

[64] Vasaiely, P. Interactive Simulation of SysML Models using Modelica. B.Sc. dissertation. Hamburg University of Applied Sciences, 2009. 62 p.

[65] Wolfram: Wolfram SystemModeler website. Available: http://www.wolfram.com/system-modeler/. Visited: 8.4.2014.

A. APPENDICES

A.1 Modelica Physical Model

```
model r110_physical
  annotation(Diagram(), Icon());
  inner Modelica.Mechanics.MultiBody.World world;
  // Body
  Modelica.Mechanics.MultiBody.Parts.BodyBox hull1(
                        r = {4.5,0,0}, r_shape = {0,0,0},
                        density = 500, length = 4.5,
                        width = 6, height = 4,
                        lengthDirection = {1,0,0});
  Modelica.Mechanics.MultiBody.Parts.BodyBox hull2(
                        r = {4.5,0,0}, r_shape = {0,0,0},
                        density = 500, length = 4.5,
                        width = 6, height = 4,
                        lengthDirection = {1,0,0});
  Modelica.Mechanics.MultiBody.Parts.BodyBox mast(
                        r = {0,24,0}, r_shape = {0,0,0},
                        density = 2000, length = 24,
                        width = 1, height = 1,
                        lengthDirection = {0,1,0},
                        widthDirection = {1,0,0});
  // Inputs
  Modelica.Blocks.Interfaces.RealInput input_left[3];
  Modelica.Blocks.Interfaces.RealInput input_right[3];
  // Outputs
  Modelica.Blocks.Interfaces.RealOutput pos_left;
  Modelica.Blocks.Interfaces.RealOutput pos_right;
  // Spring Damper Parallels
  Modelica.Mechanics.MultiBody.Forces.SpringDamperParallel SD1(
                        c = 500000, s_unstretched = 0.0,
                        d = 200000, numberOfWindings = 50,
```

```
                              animation = false);
  Modelica.Mechanics.MultiBody.Forces.SpringDamperParallel SD2(
                              c = 500000, s_unstretched = 0.0,
                              d = 200000, numberOfWindings = 50,
                              animation = false);
  // Actuation forces
  Modelica.Mechanics.MultiBody.Forces.WorldForce worldforce1;
  Modelica.Mechanics.MultiBody.Forces.WorldForce force;
  // Fixed Anchor points
  // Default: r = {0,0,0}
  Modelica.Mechanics.MultiBody.Parts.Fixed fixed1;
  Modelica.Mechanics.MultiBody.Parts.Fixed fixed2(r = {9,0,0});
equation
  // Connect inputs
  connect(input_left,worldforce1.force);
  connect(input_right,force.force);
  // other
  connect(fixed2.frame_b,SD2.frame_a);
  connect(fixed1.frame_b,SD1.frame_a);
  connect(hull1.frame_b,hull2.frame_a);
  connect(hull1.frame_b,mast.frame_a);
  connect(force.frame_b,hull1.frame_a);
  connect(worldforce1.frame_b,hull2.frame_b);
  connect(SD1.frame_b,hull1.frame_a);
  connect(SD2.frame_b,hull2.frame_b);
  // Connect outputs
  connect(hull1.frame_a.r_0[2],pos_left);
  connect(hull2.frame_b.r_0[2],pos_right);
end r110_physical;
```

A.2 Modelica Controller Model

```
model r110_controller
  annotation(Icon(), Diagram());
  // Constants
  Modelica.Blocks.Sources.Constant const[3](k = {0,765100,0});
  Modelica.Blocks.Sources.Constant constant_two(k = 2.0);
  Modelica.Blocks.Sources.Constant constant_zero(k = 0);
  Modelica.Blocks.Sources.Constant setpoint1(k = 0.7);
  // Inputs
```

```
  Modelica.Blocks.Interfaces.RealInput measurement_left;
  Modelica.Blocks.Interfaces.RealInput measurement_right;
  // Outputs
  Modelica.Blocks.Interfaces.RealOutput output_right[3];
  Modelica.Blocks.Interfaces.RealOutput output_left[3];
  // Other
  Modelica.Blocks.Math.Add add2[3];
  Modelica.Blocks.Math.Add add1[3];
  Modelica.Blocks.Math.Gain gain1(k = 100000);
  Modelica.Blocks.Routing.Multiplex3 pid1_mux;
  Modelica.Blocks.Continuous.LimPID PID1(
                        k = 1.0, Ti = 1.0, Td = 1.0,
                        yMax = 10.0, yMin = 0.0);
  Modelica.Blocks.Math.Add add3;
  Modelica.Blocks.Math.Division division1;
equation
  // Connect inputs and calculate average
  connect(measurement_left,add3.u1);
  connect(measurement_right,add3.u2);
  connect(constant_two.y,division1.u2);
  connect(add3.y,division1.u1);
  // Controller input
  connect(division1.y,PID1.u_m);
  connect(setpoint1.y,PID1.u_s);
  // Connect constant force to adders
  connect(const.y,add1.u1);
  connect(const.y,add2.u1);
  // Control routing
  connect(PID1.y,gain1.u);
  connect(constant_zero.y,pid1_mux.u1[1]);
  connect(constant_zero.y,pid1_mux.u3[1]);
  connect(gain1.y,pid1_mux.u2[1]);
  // Same control passed to both outputs
  connect(pid1_mux.y,add1.u2);
  connect(pid1_mux.y,add2.u2);
  // Connect adders to outputs
  connect(add1.y,output_left);
  connect(add2.y,output_right);
end r110_controller;
```

A.3 Modelica Test Harness

```
model r110_top
  r110_1.r110_controller controller;
  r110_1.r110_physical plant;
equation
  connect(controller.output_left,plant.input_left);
  connect(controller.output_right,plant.input_right);
  connect(plant.pos_left,controller.measurement_left);
  connect(plant.pos_right,controller.measurement_right);
end r110_top;
```