MASTER THESIS

Printing for Professionals

EVALUATING THE BEHAVIOR OF EMBEDDED CONTROL SOFTWARE

using a Modular Software-In-The-Loop Simulation Environment and a Domain Specific Modeling Language for Plant Modeling

Christian Terwellen

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE
SOFTWARE ENGINEERING (SE)

EXAMINATION COMMITTEE
Dr. Ir. L.M.J. Bergmans
Dr. L.J.A.M Somers
Dr. Ing. C. Bockisch

UNIVERSITY OF TWENTE.

01-11-2012

Abstract

During the development of high performance mechatronic systems it is necessary to ensure high quality products and short time-to-market. Because the Embedded Software (ESW) is a fundamental part of such systems, it is necessary to provide a well suited testing environment. A challenge in testing ESW is that its hardware environment, called the plant, is needed to properly evaluate its behavior. It is, in many cases, not feasible to test the ESW on the hardware because there are no, or not enough, hardware resources available. To address this problem, the ESW can be tested using a software simulation of the plant behavior. This leads to several advantages. For example, the plant hardware is not needed for testing the ESW and it is possible to test faster, more frequent and more effective.

Since five years, such a simulation environment has been in use at Océ to test the part of the ESW that controls the paper transport inside the printer. This is done by executing the ESW together with a plant model, emulating the hardware included in the paper transport, on a software engineer's workstation. Due to the great success of this simulation environment, the goal of this project is to extend the simulation environment to be able to test the ESW of the whole printer, using simulated plant models.

In order to achieve this goal, the architecture of the simulation environment has been reviewed and newly designed to meet the new requirements. The new version of the simulation environment is well prepared for the future use of multiple different plant models, which is important to reach the goal of testing all printer functions in the simulation environment.

To support the creation of plant models, a plant modeling framework has been developed. This plant modeling framework enables short modeling times and requires only little domain knowledge of the plant concepts and the modeling of physical systems. These are important properties of the framework, since the main users of the framework are software engineers.

The evolution of the testing environment has been an important step towards developing high performance printers without the need of hardware prototypes. This leads to lower cost, since there are no costs for hardware. It also leads to a short time-to-market, since no build and adaption time is needed when the hardware evolves, and the development of ESW can start earlier in the development process.

I dedicate this thesis to Annika and my family,
who supported and encouraged me throughout the years of my study

Acknowledgments

I would like to express my sincere thanks to my supervisors. Dr. Ir. Lodewijk Bergmans, I would like to thank for his outstanding support throughout the project. With his guidance and constructive criticism he contributed tremendously to the success of my project. I also would like to thank Dr. Lou Somers for the technical discussions, which guided me through the different stages of my project. Without his detailed knowledge of the field of high performance printing systems, the project would not have been possible. I would like to express my gratitude towards all my supervisors, Dr. Ir. Lodewijk Bergmans, Dr. Lou Somers and Dr. Ing. Christoph Bockisch for their invaluable assistance, support and guidance during the writing of this thesis.

Besides, I would like to thank Océ for providing me with the opportunity to carry out my final project in such an interesting field of research.

Special thanks also to Henri Hunnekens and Ben van Rens for supporting me during the redesign of the architecture of SIL. Their experience with the simulation environment and their immediate feedback on the different design stages contributed immensely to the results of the redesign of the SIL simulation environment. Besides, I would like to thank Ruud Jackobs and Harald Schwindt for their outstanding support during the design of EZ-VirtualPrinter. I also would like to express my gratitude to Chidi Okwudire, who redesigned MoBasE during the time I carried out my final project. In many meetings we discussed and found the right way to add the information needed in this project to MoBase. The support received from all the colleagues at Océ, who contributed to this project and are not named explicitly, was vital for the success of the project.

Finally, I would like to express my gratitude and love to Annika and my family for their support and encouragement throughout the years of my study.

Contents

List of Figures

List of Tables

Listings

Glossary

RoseRT	IBM Rational Rose RealTime, A development and code-generation environment for the development of real-time software.
ESW	Embedded Software, The software that is developed and deployed on embedded controllers to control the behavior of Mechatronic Systems.
Mechatronic System	System consisting of subsystems from different domains, like Mechanics, Electronics, hydraulics, pneumatics and software engineering.
SIL	Software-In-The-Loop, When using a SIL simulation, the ESW is executed with a model of the plant. SIL is also the internal name of the SIL simulation environment used within Océ.
Named Pipe	Named Pipe or also called FIFO for first in first out is a kind of inter process communication.
CAN bus	The controller area network (CAN) bus, designed by Bosch, is a standard designed for the communication between micro controllers in vehicles. The standard is also used in other applications for communication means between micro controllers.
Main node	Supervisory control that controls several sub nodes
Sub node	Controller that controls I/O-devices and is controlled by a main node.
Paper handling	The handling of sheets as they move through the printer. That comprises accelerating and decelerating sheets according to a timing model, correction of the sheet position on the Z axis, as well as the correction of askew papers (rotation on Y axis)
SheetLogic	SheetLogic is the name of the paper path plant model used in the SIL simulation environment.
FPGA	A Field programmable gate array (FPGA) is a programmable integrated circuit that is programmed in a hardware description language (HDL).

DLL	A DLL (Dynamic Link Library) is a library of functions, which can be used by multiple computer programs. A DLL can be loaded and used dynamically in runtime.
Simulation Module	The term simulation module is used to describe plant models or ESW that are compiled to a DLL and can be loaded and executed by SIL. Plant model module and ESW module are also used if a separation is needed between the two types.
Substrate	Material that is used to print on. For example, paper.

Chapter 1

Introduction

Océ is a company that offers office printing and copying systems, high speed digital production printers and wide format printing systems for both, technical documentation and color display graphics as well as related software. It was founded in 1877, with headquarters in Venlo, The Netherlands. Océ is active in over 100 countries and employs some 20,000 people worldwide. Total revenues in financial 2010 amounted to approximately 2.7 billion. The main Research and Development site is also located in Venlo, The Netherlands. This department develops new printers including mechanics, electronics and embedded control software.

To give an impression of a high volume, high performance printer, figure 1.1 shows a VarioPrint 6250. The printer consists of Paper input modules (PIM) on the left, the Print Engine

Figure 1.1: The VarioPrint 6250.

(PE) and finisher (FIN) modules on the right. The printer is suitable for from 600,000 up to 8,000,000 prints per month and has a print speed of 250 A4/letter prints per minute or 132 A3 prints per minute. The printer can be equipped with up to 12 input trays, which can be refilled during printing and can hold different types of paper (size, coating). The output storage can take on 6000 prints but prints can be unloaded while printing. The toner can also be refilled while printing.

The development of such complex mechatronic systems includes multiple disciplines as mechanical engineering, electronic engineering and software engineering. The role of software engineering is to develop embedded software (ESW) that controls the behavior of the parts of the printer. Testing the ESW is a challenge since it needs its hardware environment, also called the plant, to be evaluated properly. The ESW observes phenomena in the plant via sensors, and controls the plant via actuators. That means that the software needs to be tested on the actual hardware, which has several disadvantages. The development progress of the ESW dependents on the progress of the other engineering groups. This leads to the situation that the ESW cannot be tested in early stages of a project when there is not yet a prototype available.

Even if there is hardware available, it is often not feasible to let every software engineer test the written software on a prototype. Testing on the target hardware can also be ineffective due to long preparation and the fact that error injection and reproduction is difficult. In the case of high performance mechatronic systems it is also likely that the needed hardware is expensive.

To provide a more efficient way of testing, Océ has developed a Software-in-the-loop (SIL) simulation environment (internally simply called SIL), which allows to test the software on an ordinary workstation. In this environment, the ESW is executed with a model of the plant that emulates the behavior of the real plant. Originally, SIL was only used to evaluate the ESW that does the paper handling. The paper handling is the control of the flow of paper through the printer (paper transport) along the so called paper path. Van der Hoest [13] established the foundation of the SIL simulation environment in 2006. After that other works in this context have been performed to add new features or improve the simulation environment. Examples are the simulation of the print process in laser printers [7] or an interactive visualization [8]. Not all those improvements are used in the current SIL simulation environment because some are specific to one kind of printer architecture, and can therefore not be used in all development projects.

The goal of this project is to extend the current SIL simulation framework in a generic way to make modeling and simulation of the behavior of the remaining plant elements of a printer possible. The extension of SIL and the modeling framework, presented in this document, offer several advantages in different stages of the development process. It makes it possible to test more frequent, more controlled and on the developer's workstation and therefore provides shorter feedback cycles during software development. It is also possible to test the software before the actual hardware is build. Those advantages should ultimately lead to a shorter time to market and better quality.

Previous to this work a survey on plant modeling approaches [12] has been created to get a broad overview of different techniques that can be used to model physical systems. The paper also provides a taxonomy for plant modeling approaches to make a clear categorization possible. After the survey part, the paper presents and discusses ways to structure and modularize plant models to enhance maintainability and extensibility.

The remainder of this report is organized as follows. Chapter 2 gives an introduction to the current state of SIL and identifies research questions, based on the goals for the evolution of SIL. Chapter 3 presents the design, evaluation and implementation of a new global architecture of SIL. Chapter 4 presents a plant modeling framework that makes it possible to create plant models for emulating the behavior of several parts of a printer. Chapter 5 discusses the results and presents future work. Chapter 6 draws conclusions of this project.

Chapter 2

Project Context

This section presents the context in which this project is carried out. The first section of this chapter presents the goals for the evolution of SIL and the research questions identified for this project. After that motivations and the problem statement that lead to the initial design of a SIL simulation environment are presented. Last, the current situation of SIL is described.

2.1 Problem Statement

SIL is widely use in several projects to test the embedded software for the paper handling. Several projects use SIL for the major part of the testing work, when developing the ESW of a new printer. Since SIL is used for testing the ESW with great success, there are new goals for the evolution of SIL or the use of simulation in the development process in general. The long term vision is that in the future a virtual printer environment is used to develop new printers. This environment consists of models of different parts of the printer from different domains. This environment is used to test the system even before the first mechanical prototype is built. The following advantages have been identified for the use of such a virtual printer system.

- Critical interactions between disciplines can be studied by combining models of the individual aspects. This fits in a multi-disciplinary Model-Driven-Design approach.
- A virtual printer has the advantage of lower price and shorter lead and adaptation time.
- When the first physical prototype is built, many issues are already solved.
- The virtual printer gives more control over the test conditions (such as climate, error conditions, and tolerances)
- During the development, knowledge is gathered in models, which are used for simulation but also as documentation.
- Better quality due to regression testing.
- A virtual printer consumes less energy and paper.

The short term goal for SIL is to create a virtual version of the whole printer, which is only used to evaluate the ESW and not systems from other disciplines (mechanics, electronics). In this environment software engineers can compose a test system out of ESW modules and plant model modules. All parts of the printer can be easily modeled and used as plant model modules for simulation in SIL. SIL provides easy deployment, resulting in high acceptance and it is part of an integrated tool chain, used in product development.

In order to reach the short term goals and to work towards the vision for SIL, a new plant modeling approach has to be chosen or developed. In addition, the architecture of the SIL simulation environment has to be reviewed to determine if it is suited for the intended use.

Two research questions can be identified that summarize the problems to be solved in order to achieve the short term goals.

- *How to design a plant modeling framework for the creation of plant models for simulation in a Software-In-The-Loop simulation environment that is expressive enough to model a multitude of domain concepts of high performance printers, but also uses an adequate level of abstraction to make the approach usable for non-domain experts?*

- *How to develop a modular and extensible Software-In-The-Loop simulation environment that enables the execution of multiple embedded software modules and multiple plant models with the goal of evaluating the behavior of the embedded software?*

These two questions are addressed in the following two chapters. Chapter 3 describes the design and implementation of a modified and extended SIL. Chapter 4 presents the developed of a plant modeling approach.

2.2 Motivation for the Initial SIL Simulation Design

This section illustrates the motivations for the initial development of SIL. For this project, the documentation of earlier SIL related projects has been examined to get a clear view on motivations, requirements and use cases that has been identified in those projects.

2.2.1 Traditional Development Process of Mechatronic Systems

To better understand the motivations for the initial development of SIL, this section describes the common traditional development process for the development of mechatronic systems. Several steps are performed leading from the concept to a finished product. Figure 2.1 shows the development time of the involved domains in the traditional process of a project.

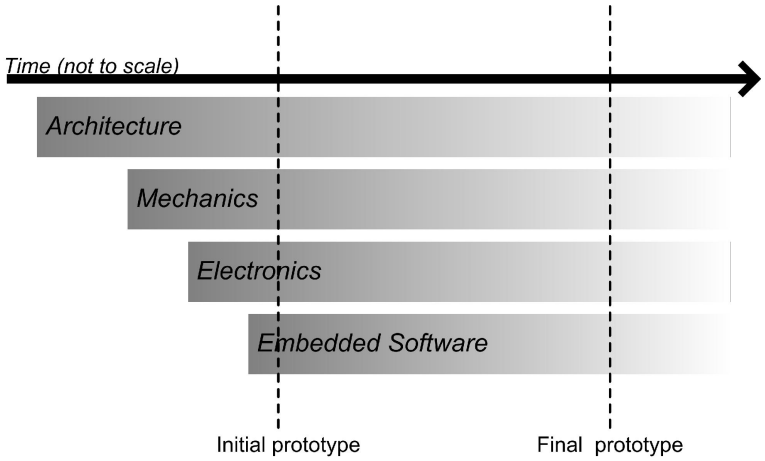

Figure 2.1: The interdisciplinary development process of high performance printers.

The first steps before designing and developing a printer is to analyze the situation on the market and the customers needs. After this analysis, the different functional properties of the printer are identified and the appropriate print technologies are chosen. The whole process of identifying those system requirements is presented by Heemels et al. in [14]. After the concept and goals for the design are clear, a first initial design of the printer is created. This

design is further developed by mechanical engineers. At a certain stage electrical engineering becomes more involved in the process. After some time software engineers begin to develop the corresponding ESW for the printer model. While developing the ESW, it is important to continuously test the ESW with the plant to evaluate its behavior. In early states those tests are executed on hardware test boards that contain the I/O hardware that is going to be used. Those test are done to test the low level control of sensors and actuators rather than the high level logic of ESW. In later stages several prototypes so called lab models are built. These lab models are used to prove that the designed concept works and can be used for testing the high level logic of the ESW. The result of the development is a so called engineering prototype, which is given to the manufacturing department where it is further optimized in terms of ease-of-use, print quality and speed.

Important factors in the development process are:

- Short time-to-market: The time it takes from initial concept to product.
- High printer performance: Such as energy consumption, print speed and print quality.
- High printer reliability: Achieving high quality of hard- and software to increase the mean time between failures.

A problem that can be identified in this development process is that embedded software can be developed from the beginning, but that it can not be tested till the corresponding hardware is created. Thus, the development of the ESW dependents on the progress of the other engineering disciplines. This leads to the situation that the ESW can not be tested in early stages of a project when there is not yet a prototype available.

2.2.2 Drawbacks Regarding Testing Performance

Even if there is hardware available, it is often not feasible to let every software engineer test the written software on a prototype. Testing on the target hardware can also be ineffective due to long preparation and the fact that error injection and reproduction is difficult. In the case of high performance mechatronic systems it is also likely that the needed hardware is expensive.

2.2.3 Advantages of Testing ESW in a Simulation Environment

So, a testing environment, which could be used to test the ESW without the hardware plant would lead to several advantages. A possible approach to solve the above mentioned problems is to model and simulate the behavior of the plant of the target mechatronic system in software. That enables the software engineer to carry out tests in a simulated instead of the real environment. An advantage is that the software can be tested on the software engineer's local system and therefore allow shorter feedback cycles. It is further possible to create and investigate situations that are difficult or not even possible to create and investigate when testing with the real hardware though it is relevant to test them. Another major advantage is that regression testing is possible. So, simulation based testing is likely to improve two of the three important factors identified for the development process of mechatronic systems in section 2.2.1, namely "Short time-to-market" and "High printer reliability". The "time-to-market" gets shorter because the development of the ESW can start earlier. The ESW is likely to have a higher reliability because it is tested more frequently (regression testing).

2.2.4 The Predecessors of SIL

Before SIL was developed, another simulation environment was implemented. This simulation environment served as a reference for the initial SIL implementation. The simulation environment was implemented as a proof of concept to elaborate if a simulation will offer advantages

in the development of ESW for the paper path. The following subsection provides background information about the paper handling of a printer.

Paper Handling

The paper path describes the path of the paper as it moves through the printer. Figure 2.2 shows an example of a printer's internals. The figure shows the way of the paper from the paper input

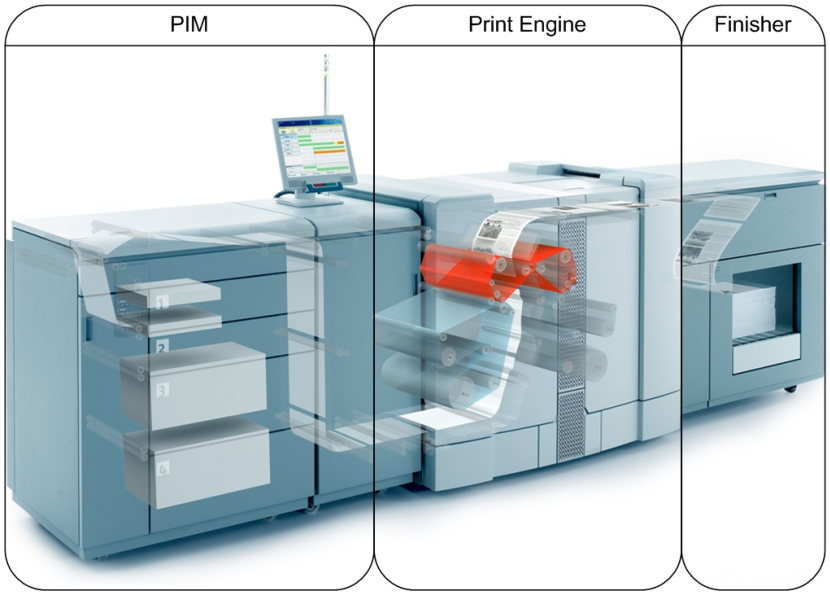

Figure 2.2: An example of the internals of a printing system.

module through the print engine to the finisher, which, in this case, stacks the printed paper. Most printers can have multiple configurations. That means different numbers of PIMs and different numbers and/or types of finishers. In addition, some printers have different versions that differ in speed. Figure 2.3 shows a 2D sketch of an example paper path.

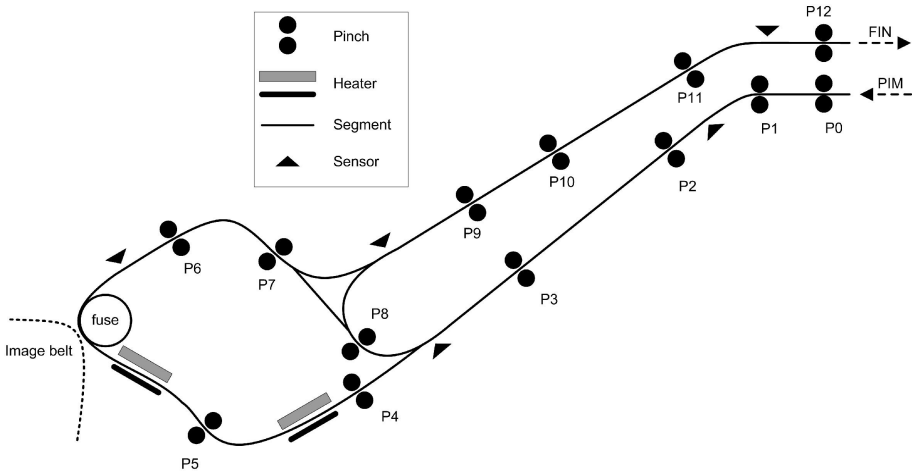

Figure 2.3: A paper path layout.

The paper transport is controlled by ESW with the use of motors, which drive pinches,

which move the paper. Sensors are used to sense the presence or absence of paper. During the design of the physical (mechanical) paper path of a printer, a timing model is created that describes the desired movement of the sheets as a speed profile. So it is defined for the sheet at which point on the paper path it has which speed. In the printer this model is implemented by the use of pinches that are accelerating or decelerating the sheets. All pinches in between such pinches, which accelerate or decelerate a sheet, run at a constant speed. For example, a sheet enters the paper path, described in figure 2.3, with a speed of 1,5 m/s. $P0$ and $P1$ run at a constant speed so that the paper is further transported with 1,5 m/s. After the paper leaves $P1$ and is only in contact with $P3$, $P3$ decelerates the paper to 0,9 m/s. $P4$ again runs at a constant speed so that the paper is transported further with 0,9 m/s. The timing model is used as an input for the ESW, which uses the timing information to control the involved motors in such a way that the accelerations and decelerations are performed. Next to the variation in speed, there are different kinds of checks and corrections of the sheet positions in z-position and to correct occurring rotation of the sheets, as the sheets travel through the printer.

There are several reasons why the development of the paper handling can be difficult.

- Sheets can have different formats.
- Print jobs can contain sheets that are printed simplex (1-sided) and/or duplex (2-sided).
- Some printers support different throughput rates, for example, both 50 and 70 pages per minute.
- During development the paper path typically changes with each new prototype, which leads to repeated redefinition of the timing model.
- When the image is printed on a sheet, it is important that the sheets position is aligned with the image.

The Matlab Simulink Paper Path Simulation

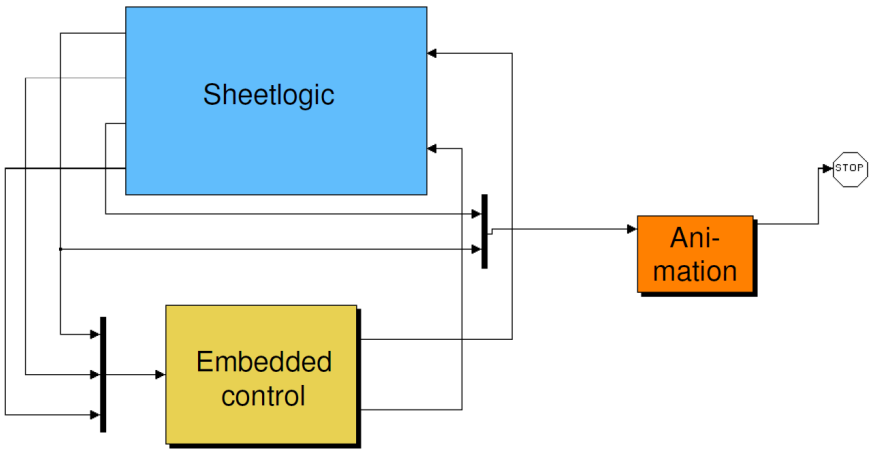

Figure 2.4: A simulation environment for testing the paper path software in Matlab Simulink.

Figure 2.4 shows the high level structure of the simulation environment. The block *SheetLogic* simulates the hardware of a printer, which is in this case sensors, actuators and also the behavior of the sheets based, on actuator signals. The block *Embeddedcontrol* contains the ESW, which is included using the tool TrueTime [1]. The third block, *Animation* is used to visualize the movement of the sheets.

This simulation was evaluated by different software engineers and was considered very helpful in the process of developing ESW. Van der Hoest identified two major drawbacks of this simulation environment, maintainability and license cost [13]. It is hard to maintain the SheetLogic Matlab Simulink block because for doing this one needs to be able to model a physical system in Matlab Simulink. In this case the hardware of the paper path of a printer has to be modeled accurate enough to use the model for evaluating the ESW. This is not feasible because Matlab Simulink is not widely used by the software engineers, which would have to perform this task. Using Matlab Simulink as a simulation environment would also lead to high license cost because a Matlab Simulink license is needed for every software engineer that uses this tooling.

2.2.5 Goals of the Initial SIL Implementation

After identifying flaws in the classical way of testing ESW in the context of a multidisciplinary project and recognizing possibilities and advantages of a simulation based testing approach, the initial SIL was implemented with the following key drivers [13]:

- Ease-of-use. SIL should be easy to use for software engineers and integrators to be accepted by them. This is especially the case for the process of specifying a printer layout and preparing the ESW for simulation.
- Maintainability. Especially for the modeled plant behavior.
- Early feedback. Simulating and testing should provide immediate feedback.
- Reliability. The simulation should be accurate and reproducible. When a certain test case fails, it should be possible to repeat the simulation with exactly the same results, for further analysis.
- Low cost. The use of commercial products that cause high license costs should be avoided.

2.3 Current Situation of SIL

This section presents the initial SIL implementation to give a technical context for this project. The SIL simulation tooling has undergone several extensions and improvements since the first release in 2006. SIL and also the corresponding visualization tool, Argus, where attractive fields of research in several research projects that were carried out by students.

First an overview of the architecture of the software under test is given. After that the structure of SIL and the interfacing with the ESW and the plant models are presented.

2.3.1 The Software under Test (SuT)

The architecture of the ESW and the needed steps to prepare the ESW for execution in the SIL environment, are a major concern because the software should not be changed for the purpose of testing.

Software Architecture

The architecture of the ESW can is divided in so called controllers, main nodes and sub nodes. The controller of a printer is the high level control that communicates with the outside world, provides a local user interface, schedules print jobs and converts the data of the print jobs in a format that is usable by the printer. The main node is the main controlling entity in the embedded software. It receives tasks from the controller and splits these tasks in sub tasks and sends them to the corresponding sub nodes. The sub nodes are low level entities in the printer ESW architecture, and are used to control I/O-devices. This is done with an I/O-layer that

provides functions to communicate with different I/O-devices. Sub nodes have a time sliced structure. That means that a function that initiates all calculations and controls, is called in a defined frequency by the real-time embedded system. This function is called “tick” function. Figure 2.5 shows the high level architecture of the embedded software of most printers. Main

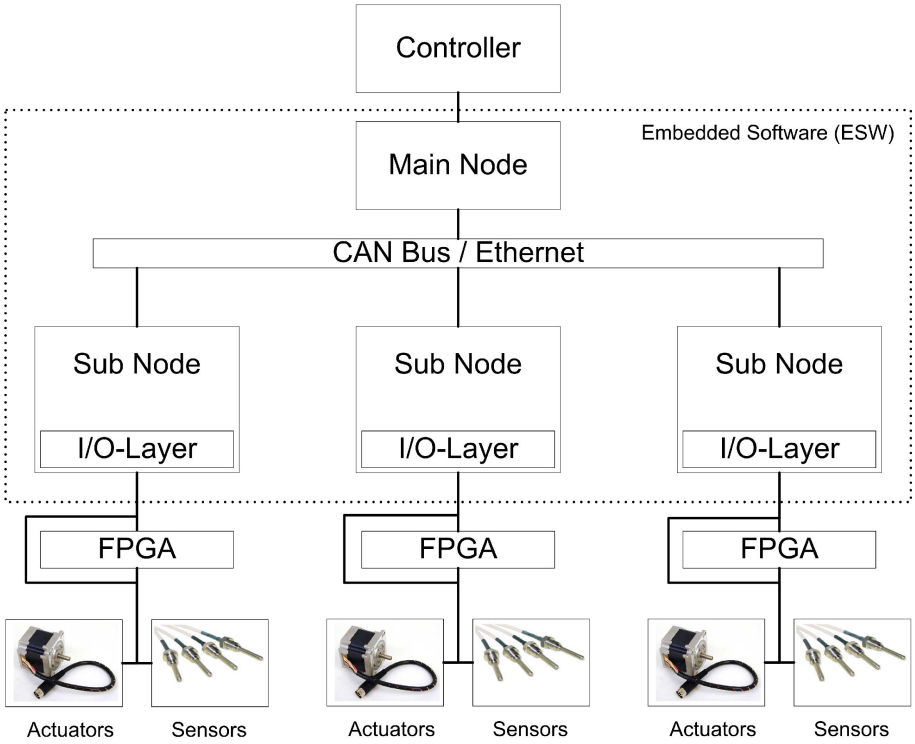

Figure 2.5: Architecture of the embedded control software when in SIL Simulation.

and sub nodes are implemented using C/C++ in RoseRT. The controller software is split up in different parts, which are implemented using different general purpose and scripting languages. The main node communicates with the sub nodes via a CAN bus. The communication between controller and main node is realized using an interface based on high-level commands. For some I/O-devices an additional low level control layer is used, next to the I/O-layer, that implements low level control functionality. This layer is often implemented using FPGAs. This is, for example, used for stepper motor control. If the ESW starts a stepper motor, the I/O-layer calls a function of the low level control that starts the motor at a certain time or position with a certain speed etc. The low level control translates this function call into a low level signal that is send to the stepper motor.

Preparing the ESW for SIL Simulation

The part of the software that is tested in SIL consists of the main node and the sub nodes of a printer. In the test setup, the functions of the controller are performed by a java application that communicates with the main node. To be able to use the ESW in the simulation, it has to be compiled for a Microsoft Windows environment that is used on the development workstations. The ESW is compiled as a dynamic link library (DLL). That makes it possible for SIL to load the ESW modules dynamically. For the communication between the main node and the sub nodes, a simulated CAN bus is used that uses a named pipes approach, which is illustrated in figure 2.6. There are more changes necessary to let the ESW communicate with the simulated plant.

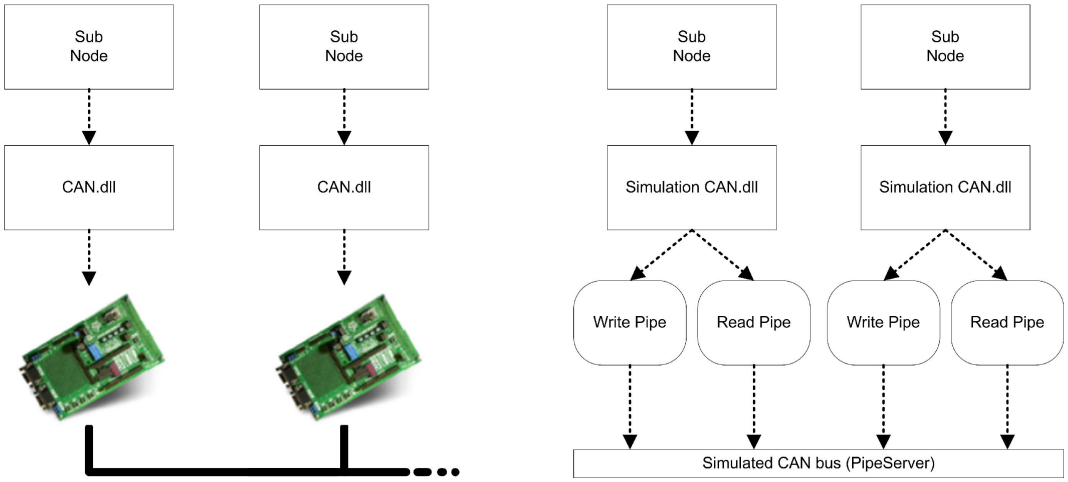

Figure 2.6: The simulated CAN bus.

In the normal case, when the ESW is compiled for an embedded system, an I/O-layer is used to implement the low level communication with I/O-devices or the low level control. For different target embedded systems, different I/O-layers are available. When the ESW is build for SIL, a SIL simulation I/O-layer is used. The interface between the ESW-layer and the I/O-layer stays the same. An overview of the architecture of a sub-node for a target embedded system and for simulation is shown in figure 2.7. The communication between the SIL simulation I/O-layer is based on high-level sensor and actuator signals. Examples for those signals are the logical value of a digital actuator or sensor (ON or OFF) or the "start at time" or "start at position" command of a stepper motor. When the ESW is run and tested in the SIL simulation, it is

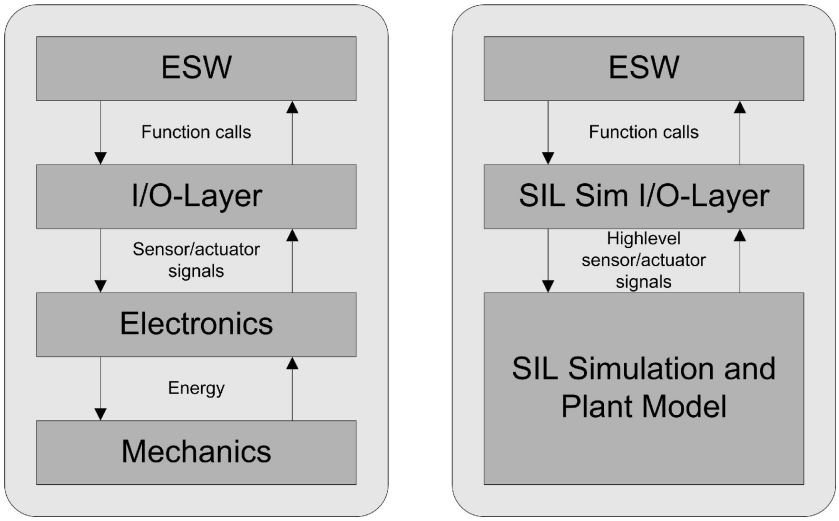

Figure 2.7: Architecture of the embedded control software.

compiled with a simulation specific I/O-layer. The I/O-layer provides an interface between SIL and the ESW. Through this layered structure it is possible to just exchange the I/O-layer of the ESW and leave the ESW itself untouched, as shown in figure 2.7. A drawback is that the other, embedded system specific versions of the I/O-layer, are not tested when the ESW is executed

in SIL. Figure 2.8 shows the concept of the I/O-layer. The different I/O-elements are specified in an I/O-specification header file (left). This file is a list of I/O-devices, that uses macros to specify each I/O-device. These macros (right) are used to create functions, which can be used by the ESW to communicate with I/O-devices. So, every definition in the ESW header creates functions specific for this device function names based on the name of the device. Which functions are generated depends on the type of I/O-device and therefore which macro is used.

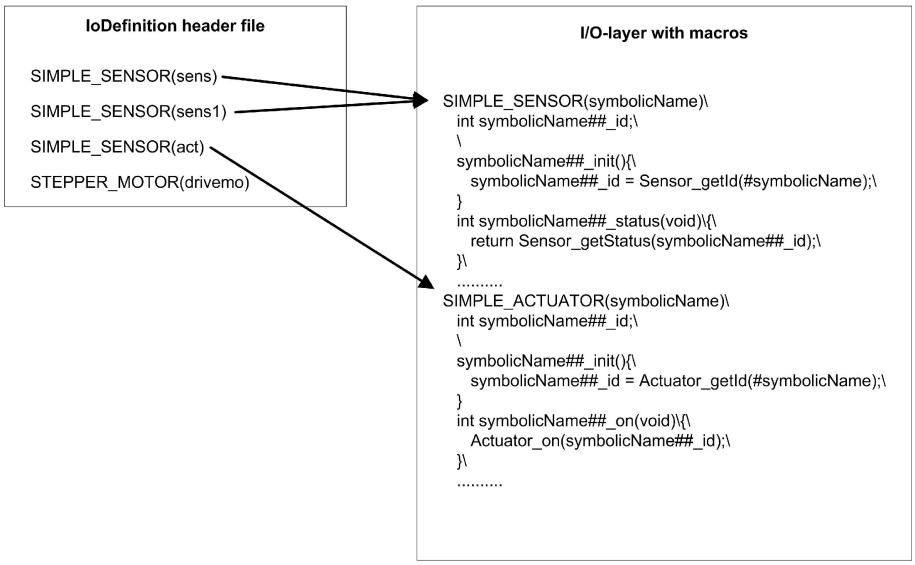

Figure 2.8: I/O-layer macros and the use of them.

2.3.2 Global Structure of SIL

The global structure of SIL did not change significant over the time SIL was used. The structure has been examined in the beginning of this project to be able to evaluate its suitability for the intended extensions. Figure 2.9 shows the different entities of SIL. Subsequently, all entities are described in short.

SIL core and SheetLogic

The SIL simulation environment is a custom made tool implemented in RoseRT using C/C++. In the current SIL simulation environment, the most important plant model that is used, is a model of the paper path of the printer, called SheetLogic. SheetLogic consists of different I/O-devices and mechanical parts that are simulated to emulate the paper path behavior. Examples of those elements are motors, stepper motors, pinches, pinch lift actuators, or paper path segments. Important to mention is that the low level control layer is also included in SheetLogic. SheetLogic uses these elements to create a model of the printer's paper path by reading the specification and properties of different parts from MoBasE (described in the next subsections). It is embedded in the SIL core because SIL was initially designed to only test the ESW for the paper handling. SIL and SheetLogic are coupled because SIL needs information about the I/O-devices used in SheetLogic, to set up the connection between ESW and SheeLogic (section 2.3.3).

The SIL core consists, besides SheetLogic, of functionality to load, execute and schedule ESW modules. The scheduling is done by the clock component, which holds an internal schedule

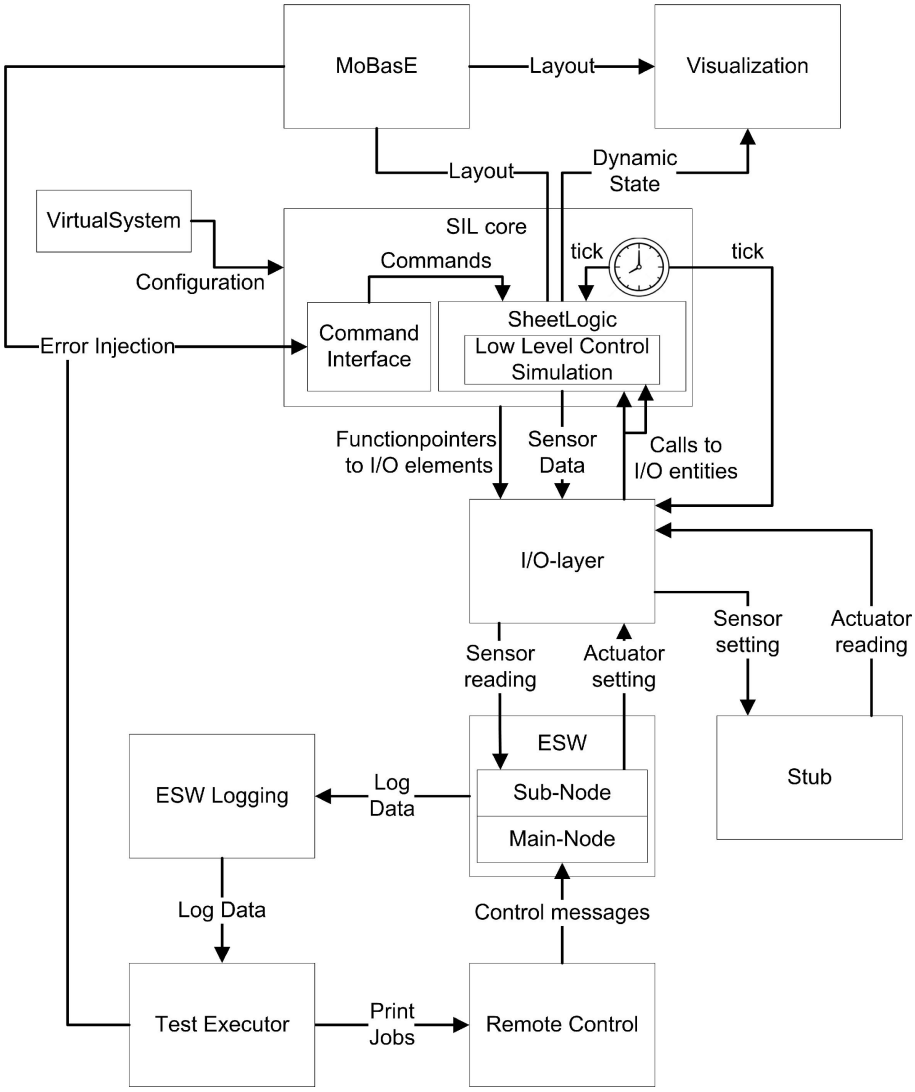

Figure 2.9: The current structure of the simulation environment.

of all loaded ESW modules. Since the ESW of the nodes is time sliced as discussed earlier, the clock can call the "tick" function of the ESW. SheetLogic has also a "tick" function that initiates the calculation of a new state. For SheetLogic and all ESW modules, the sample frequency can be adjusted. While running, the clock calls the "tick" function of all modules and SheetLogic according to the defined schedule.

MoBasE

MoBasE is short for Model Based Engineering. The MoBasE is a data model framework that enables engineers to define a minimalistic data model that spans over disciplines and helps keeping important design information in one place. In the context of SIL, MoBasE is used to read the layout of the paper path of a certain printer for visualization and for generation of the SheetLogic plant model.

Visualization

The visualization front-end of SIL is called Argus. Argus provides different functionality to visualize the state of SheetLogic. The properties of the different parts (pinches, sensors, etc) can be visualized as a list or in a 3D environment. The 3D animation shows the paper path of the tested printer, consisting of segments, pinches, sensors and the moving sheets of paper (figure 2.10). Argus also provides functionality to manipulate the simulation by, for example, changing sensor actuator values or stopping sheets. This is done via the so called command interface.

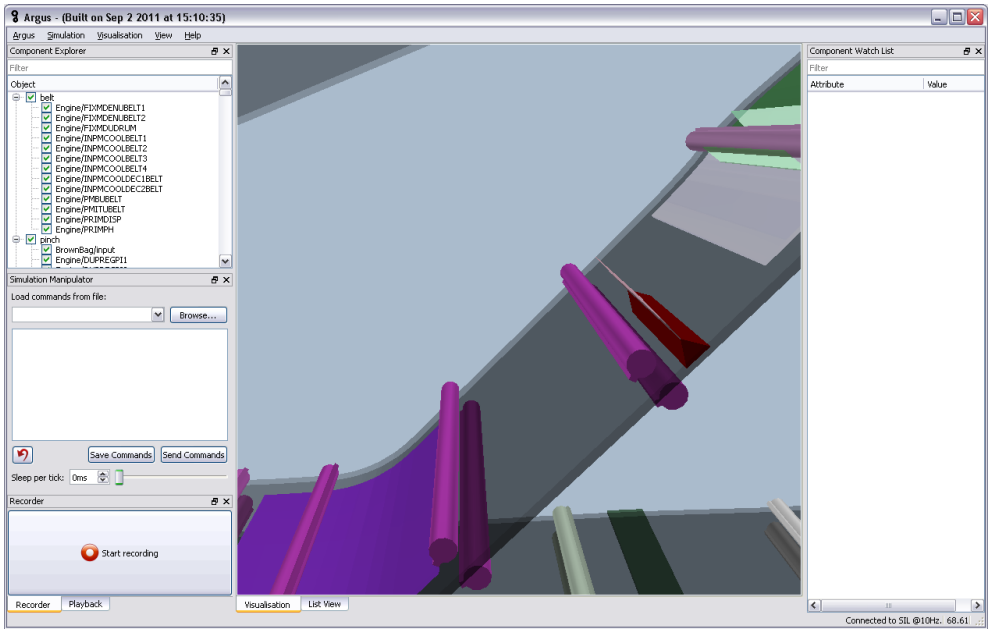

Figure 2.10: The Argus visualization tool.

I/O-layer

The I/O-layer, as mentioned in section 2.3.1, is used to let the ESW communicate with the hardware on the target hardware environment. When the ESW is executed in SIL, the target hardware I/O-layer is replaced with a SIL simulation specific I/O-layer that translates function calls of the ESW to the I/O-layer into function calls to SheetLogic.

ESW

The block ESW contains the software that has to be tested. As described in section 2.3.1, the SuT comprises the main node as well as the sub node(s). But only the sub nodes communicate with SheetLogic via the I/O-layer. The main node in contrast, communicates with the remote control and the sub nodes.

Stub

It is possible to create stubs, which are basically hand coded plant models that use the same I/O-layer as the ESW but with inverted behavior. That means that a stub can set sensor values and read actuator values in SheetLogic instead of reading sensors and writing actuators in the

case of the ESW. This mechanism was added to SIL as a first possibility to emulate the plant behavior of printer parts other than the paper path.

Embedded Software Logging

The ESW has extensive logging capabilities, which are used to debug the embedded software and to evaluate test cases.

Remote Control

The so called remote control is a java application that emulates the printer controller. It can, for example, be used to start print jobs and to read the machine state. The user interface of the remote control is shown in figure 2.11

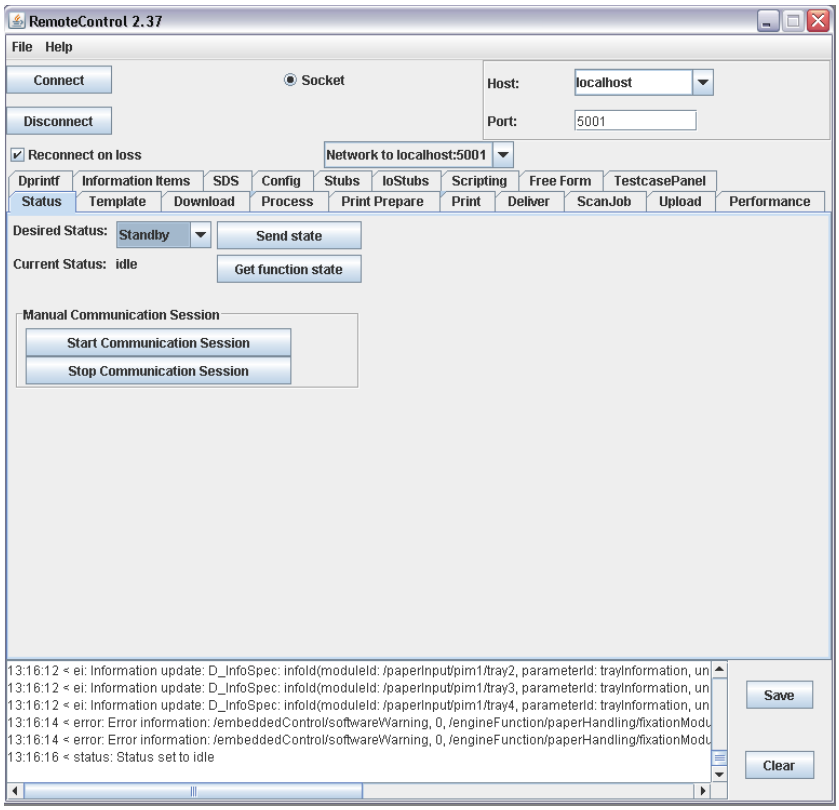

Figure 2.11: The RemoteControl.

Test Executor

The test executor can be used on top of the remote control to automatically run test cases.

VirtualSystem

The VirtualSystem is a XML configuration file specifying, for example, which ESW modules have to be loaded, from where and with which sample frequency.

CommandInterface

The command interface is used to inject error into SheetLogic. It can also be used to request the state and value of sensors and actuators. This functionality can be used by the TestExecutor or the visualization.

2.3.3 Interface between SIL, SheetLogic and ESW

Since the goal of this project is to add new plant models to SIL, the interface of SIL and the ESW is reviewed. The interfacing of the ESW, respectively the I/O-layer, and SheetLogic is set up by the SIL core in the beginning of the simulation. During the initialization of the simulation,

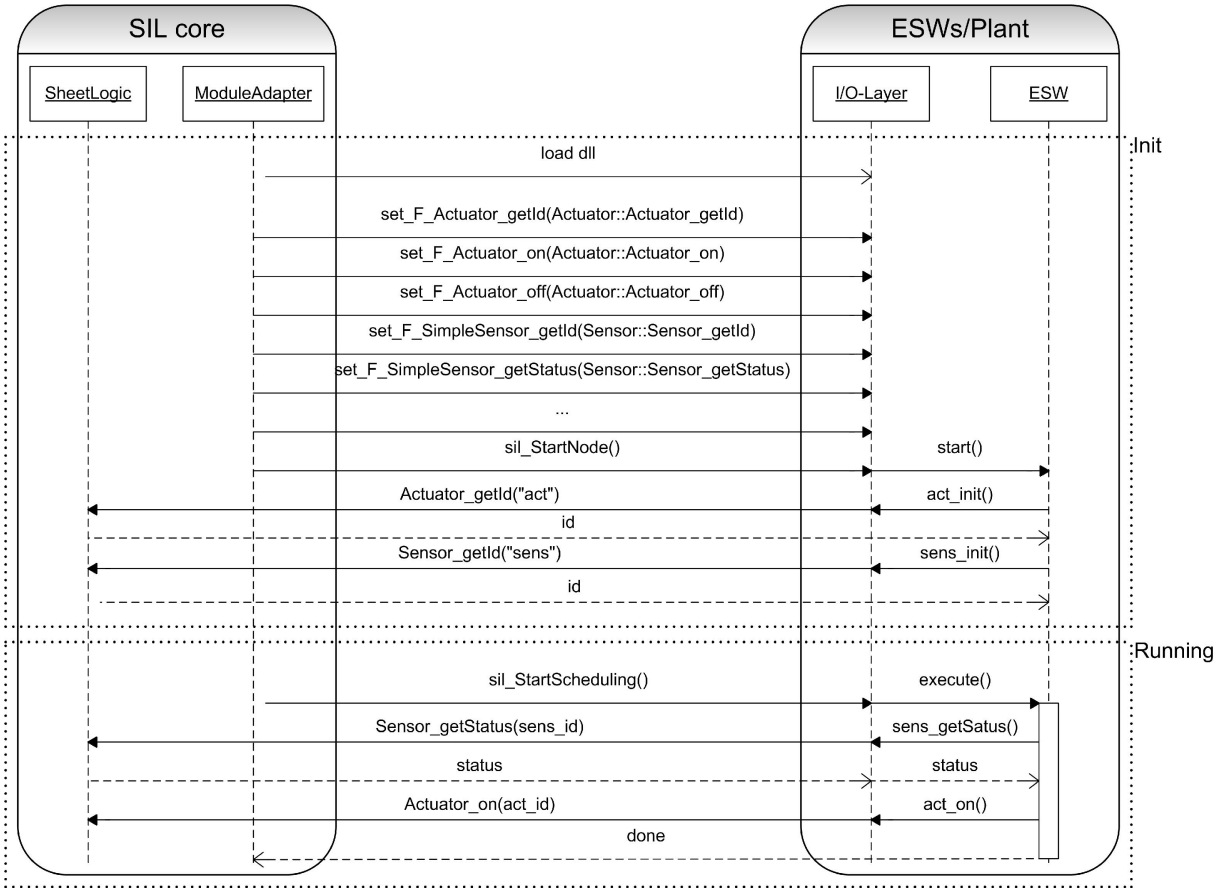

Figure 2.12: The current interface during startup and while running.

the ESW interfacing component of the SIL core, called ModuleAdapter, configures the ESW module. This is done by transmitting several function pointers to the I/O-layer by calling the corresponding "*set_F_*" functions of the I/O-layer. Each type of I/O-device in the SheetLogic has a number of functions to change or read the current state of the device. The function pointers transmitted to the I/O-layer of the ESW are pointing to those functions in SheetLogic. The function pointers are used by the I/O-layer to call the functions in SheetLogic in run-time. Each type of I/O-device has also a "getId" function, which is used during initialization by the I/O-layer to retrieve the id of all I/O-devices, based on the names of the I/O-devices. This id is stored locally and is used as a parameter when calling the other functions of this I/O-device.

2.3.4 Running a Simulation

When running a simulation the simulation can be controlled by the remote control, which emulates the controller of the printer. The RemoteControl application is shown in figure 2.11.

The behavior of the simulated printer and the ESW can be analyzed using Argus, ESW logging and the tooling for plotting actuator and sensor values. Argus provides different tools for visualization like a 3D visualization, list views of all I/O-devices and recording capabilities. In addition, Argus can be used to inject errors into SheetLogic. Argus is shown in figure 2.10.

Figure 2.13 shows the tool dPlot which is used to plot values of I/O-devices from SheetLogic. The SIL core is executed in a console application, which provides a simple menu to plot graphics (lower left corner in figure 2.13).

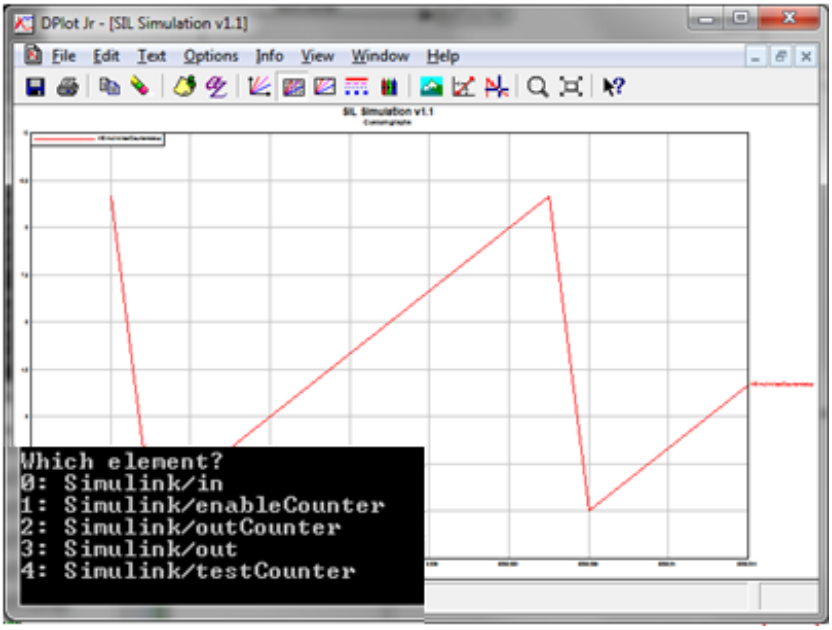

Figure 2.13: Plotting I/O-device values while simulating.

Chapter 3

Refining the Architecture of SIL

This chapter presents the design of a evolved version of SIL, which meets the requirements that are derived from the short term goals from section 2.1. For convenience the new version is referred to as SILv2 and the old as SILv1 throughout the document. The research question, discussed in this chapter is:

- *How to develop a modular and extensible Software-In-The-Loop simulation environment that enables the execution of multiple embedded software modules and multiple plant models with the goal of evaluating the behavior of the embedded software?*

This chapter is organized as follows. First requirements and key drivers for the development of SILv2 are identified. After that, design decisions are discussed and the design for SILv2 is presented and evaluated. Subsequently the implementation and integration into SIL is discussed. Finally some conclusions are drawn.

3.1 Requirements for SILv2

3.1.1 Requirements

From the short term goals for the evolution of SIL (section 2.1), several functional and non-functional requirements for the evolution of SIL have been derived, as shown in table 3.1.

R1	ESW shall be encapsulated within simulation modules.
R2	Plant models shall be encapsulated within simulation modules.
R3	A simulation shall be composed of multiple simulation modules.
R4	Plant models can be implemented using different plant modeling technologies.
R5	Communication between ESW simulation modules and plant model simulation modules shall be based on a generic and well maintainable interface.

Table 3.1: Requirements for SILv2.

3.1.2 Key Drivers

From the short term goals (section 2.1), it can be derived that the new SIL design should be:

- Modular. Logically independent entities of SIL should be separated, and encapsulated in modules.

- Generic. The SIL core and the communication between the SIL core and the simulation modules should be generic.
- Extensible. There are many imaginable extensions and change cases for SIL (new types of ESW or plant models). By Modularizing SIL and using generic ways of communication between the entities, SIL can adapt more easily to future changes and extensions.
- Composable. Simulation modules should be composable to make it possible to run simulations of sub-parts of a printer or of the whole printer.
- Efficient. SIL should have good performance so that it is possible to run the simulation in a decent speed. In the current situation, the main load is caused by the SheetLogic. This can not be changed by SILv2, but a goal is that the performance stays approximately the same compared to SILv1 (+- 5%).

3.2 Detailed Analysis of Problems in SILv1

SIL was originally designed as a dedicated test tool for testing the ESW that controls the paper path of a printer. The short term goals are that SIL evolves to a global test framework for ESW in general. So, SILv1 has been evaluated regarding the requirements and qualities to identify problems that need to be solved in order to meet the requirements. For each problem identified in this section, the related requirement or requirements from table 3.1 are given.

3.2.1 Communication of ESW and SheetLogic

The interface and communication between ESW and SheetLogic, presented in section 2.3.3, is a major concern. The current techniques introduce several problems with respect to the requirements of SILv2.

Coupling of SIL Core and SheetLogic/ other plant models

Related requirements: R1, R2, R3, R4, R5.
The SIL core sets up the connection between the ESW module and SheetLogic by transmitting function pointers, that point to functions of I/O-devices in SheetLogic, in the ESW module. That implies that these functions are known by the SIL core. That hinders the separation of SheetLogic (or plant model simulation modules in general) and the SIL core, which is desirable for the modularization of SIL. Another problem occurs if several plant model simulation modules and several ESW modules are composed in the future. The SIL core needs information about which functions are located in which plant model simulation module and also which function pointers are required by which ESW module. This introduces strong coupling of the SIL core and the different plant model simulation modules.

Growing Interface between ESW and SheetLogic

Related requirements: R5.
There are several functions for each I/O-device that are used by the ESW to communicate with the device (function pointers). This interface also implies that the ESW module has to provide functions, which can be used by the SIL core to set these function pointers. The results in a very crowded interface, which consists in the current situation of 67 functions in the I/O-layer just for setting function pointers to I/O-devices in SheetLogic. Because of the fact that over time more and more new I/O-devices are added and old I/O-devices still have to be supported, it is likely that the interface keeps growing over time and is hard to maintain.

3.2.2 SheetLogic as the Central Concept of Simulating Plant Behavior

Related requirements: R1, R2, R3.
In SILv1, SheetLogic is the only plant model in use (next to the stub mechanism) and can be seen as the central point of plant behavior simulation. For this reason some features are included in SheetLogic that should be part of the SIL core, when SIL supports multiple plant model simulation modules. Those features are the connection to the visualization and to the so called command interface, which is used for error injection and debugging. Even the plant behavior implemented using the stub mechanism (section 2.3.2) is heavily dependent on the implementation in SheetLogic. This is because of the fact that the I/O-devices, controlled by the stub, are located in the SheetLogic and are controlled by the stub via the same interface that is also used by the ESW (though with inverted behavior, writes sensors and reads actuators).

Another identified problem is that parts of the control software, the low level control, is included in SheetLogic, which makes the interface to some I/O-devices very complex. The interface between I/O-layer and low level control, so between the ESW simulation module and the SheetLogic, is based on function calls (figure 3.1). For a stepper motor the interface consists of 22 functions. Because of the fact that the low level control is totally integrated in SheetLogic, the interface between SIL core and simulation module has to implement those functions. It would also be a better logical separation to make the low level control part of the ESW simulation modules because it is logically part of the control.

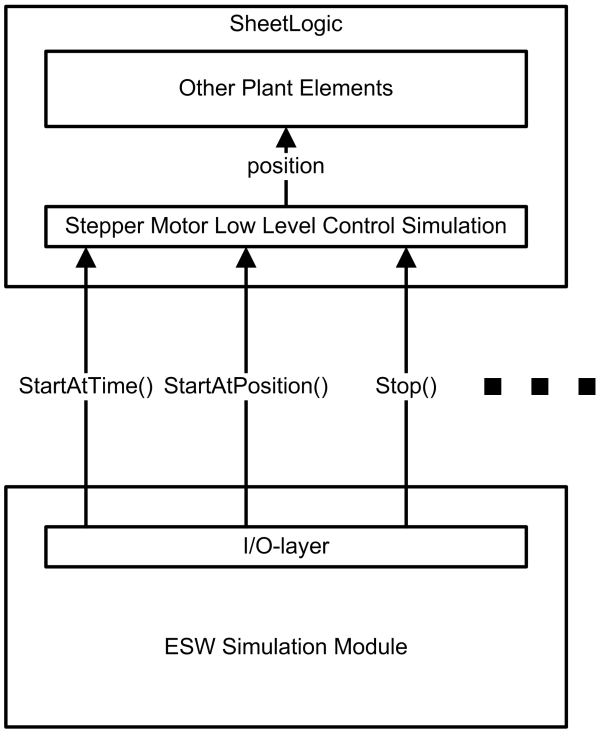

Figure 3.1: Low level control in SILv1.

3.3 Design Decisions

For the identified problems in the previous section, there are several possible solutions. These solutions are discussed in this section and motivations for the final decisions are given.

3.3.1 Generic or Specific Interface

SILv1 uses an I/O-device specific interface for communication between SIL core and simulation modules. That means the communication is based on using function calls to specific simulated I/O-devices. An example is the simple sensor I/O-device, which is a digital sensor that can be in high or low state. Even for this simple I/O-device, four functions in SheetLogic and the four function pointers to these functions in the I/O-layer are needed.

In this communication approach, every kind of information that is communicated needs a dedicated function. So, a function in the plant model, a function pointer to the function in the plant model, and a function in the simulation module to set the corresponding function pointer, is needed. This leads to a growing interface and the coupling of plant model simulation modules and the SIL core, as described in section 3.2. So the way of the communication is regarded as a possible point of improvement.

Options

1. Communicate via function calls to I/O-devices, as in SILv1.

   Advantages: Old communication approach, using function calls to I/O-devices can be kept. The communication approach uses dedicated functions to write and read to and from I/O-devices, which makes usage of the interface in the simulation modules and SIL core simple and consistent.

   Disadvantages: The communication approach has specific functions for all kinds of I/O-devices, which leads to an interface that growth when new kinds of I/O-devices are added. This leads to poor maintainability. The SIL core has to have knowledge about the different I/O-devices in the different plant models to set up the connection, which leads to coupling between plant models and SIL core. The communication approach is little extensible and decreases modularity by introducing coupling.

2. Communicate via generic variables. This means that a simulation module has input and output variables, representing sensors and actuators, that can be identified via the name of the variable. For each I/O-device, a variable is created, which is used to communicate the status of the I/O-device between the simulation modules. In terms of the example of the simple sensor that would mean that a variable with the name of the sensor is created. This variable holds the current state of the sensor. The communication between plant model simulation module and ESW simulation module is realized by sending this variable back and forth between them. The SIL core does not have knowledge what kind of information is communicated.

   Advantages: The generic concept makes it possible to communicate a multitude of information via a simple interface. The SIL core does not need to have knowledge of the used I/O-devices. The communication approach is more extensible and better to maintain due to the fact that it does not depend on used I/O-devices.

   Disadvantages: The usage of this communication approach implies that the communicated variables have to be interpreted properly on the other end of communication.

Decision

Use a more generic interface and communicate via variables representing I/O-devices.

Rationale

The communication based on variables makes SIL more generic because the communication is not limited to I/O-devices. It becomes also more extensible because, the SIL core and the interface do not have to be changed, when new I/O-devices are used. This approach is chosen because these characteristics are key drivers for the development of SILv2.

3.3.2 Storage and Communication of Communication Variables

Since the communication between simulation modules and SIL core is changed to a more generic interface, which uses variables, the options on how to store and communicate these variables are reviewed. The global principle, that simulation modules are loaded by the SIL core as a DLL, should be kept.

Options

1. Communication via pointers to variables, storing them in the SIL core.

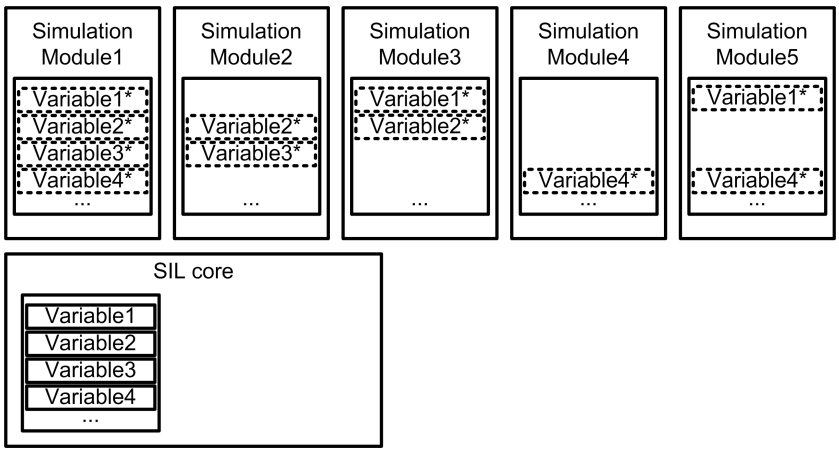

Figure 3.2: Storing variables only in the SIL core.

Figure 3.2 shows the structure when variables are stored in the SIL core and simulation modules access them using references.

Advantages: Simple usage in simulation modules. Minimal storage is needed.

Disadvantages: Pointers may be accessed by multiple entities leading to errors (has to be made thread safe).

2. Communication via pointers to variables, storing them in simulation modules. Figure 3.3 shows the structure when variables are only stored in one simulation module and are communicated using references to these variables. In figure 3.3, simulation module one holds all variables. All other simulation modules have a reference to the variables they need (indicated by the dashed rectangle). The SIL core is just used for setting up the

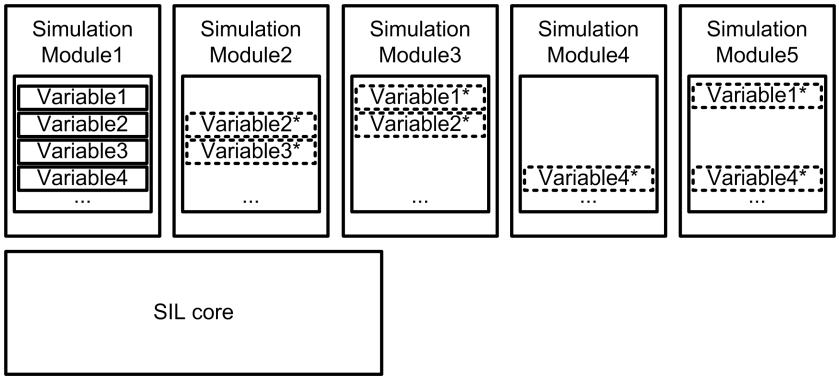

Figure 3.3: Storing variables only in one simulation module.

connections during initialization and is not involved in communication during simulation.

Advantages: Simple usage in simulation modules. Minimal storage is needed.

Disadvantages: Pointers may be accessed by multiple entities leading to errors (has to be made thread safe). Complex configuration of simulation modules during initialization because the SIL core needs to know which simulation module holds which value to set up the connection between simulation modules properly. No central storage.

3. Communication via pointers to variables, storing them in the SIL core and a copy in simulation modules. Figure 3.4 shows the structure of SIL when variables are stored in the

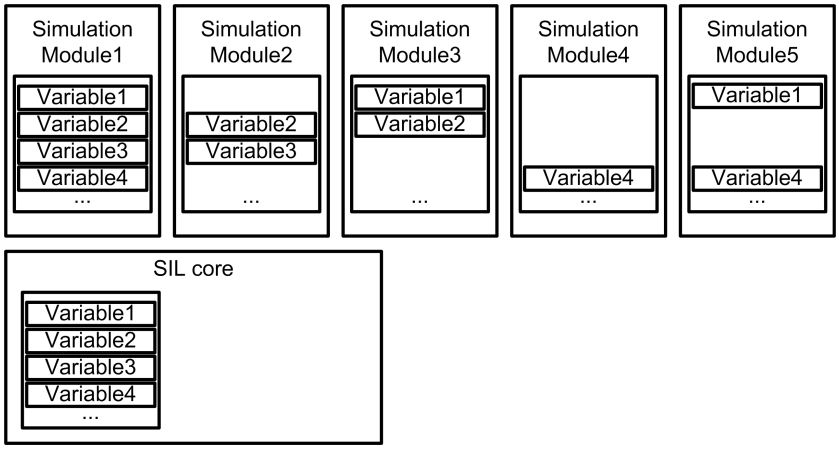

Figure 3.4: Storing variables in the SIL core and a copy in the simulation modules.

SIL core and a copy in the simulation modules. The SIL core takes the active part of the communication while running. This is done by copying values to and from the simulation modules by using a reference to the variables in the simulation module.

Advantages: Simple usage in simulation modules. Parallel execution is possible without much effort. Central variable storage can also be used for other testing purposes, like error injection, debugging and visualization etc. Setup of simulation modules during ini-

tialization is less complex because it has just to be known which variables are written to and read from the SIL core by a simulation module. Communication happens through the SIL core so it has the full control over what is communicated between simulation modules.

Disadvantages: Variable values need to be synchronized properly. Needs more storage because variables are stored multiple times. A realistic estimate is that three times more memory is needed because the variable is stored in the simulation module that writes the value, in the SIL core and in the simulation module that reads the value. In a simulation containing 200 different variables this would be 4.8 KB instead of 1,6 KB for the approaches where each variable is stored only once (assuming a variable size of 8 byte (size of double)).

4. Communication via function pointers to access variables of other simulation modules with storage of variables only in simulation modules. The structure is the same as in figure 3.3. The difference is that the variables are not communicated using a reference to the variable, but a function to get or set a variable.

   Advantages: Simple usage in simulation modules. Minimal storage is needed.

   Disadvantages: Complex configuration of simulation modules during initialization because the SIL core needs knowledge about which simulation module holds which value to set the right function pointers. Pointers may be accessed by multiple entities leading to errors (has to be made thread safe). There is no central storage.

5. Communication via function pointers to access variables stored in the SIL core. The structure is the same as in figure 3.2, with the difference that function pointers are used in stead of references to the variables themselves.

   Advantages: Simple usage in simulation modules. Minimal storage is needed. The setup of simulation modules during initialization is easy. Central variable storage can be used for error injection and visualization etc. Communication happens through the SIL core so it has the full control over what is communicated between simulation modules.

   Disadvantages: Pointers may be accessed by multiple entities leading to errors (has to be made thread safe).

Decision

Use the communication via pointers to variables using multiple copies of the data (option 3).

Rationale

The two options (2. and 4.) are not used because in these cases the SIL core has no control over the communication, which prevents a central point for error injection etc. Additionally, in all options except 3, parallel execution of simulation modules leads to more problems because pointers could be used by multiple simulation modules at the same time. Finally the approach using multiple copies of the variables and communication via references to variables is chosen. The reason is the better decoupling that is provided by this approach in comparison to option 5. The simulation modules do not call functions implemented in the SIL core but the SIL core initiates and performs the communication. The drawback of the additional memory needed

(a few KB) is negligible since the simulation is executed on a workstation PC where enough memory is available.

3.3.3 SheetLogic and the Low Level Control

There are also several options when it comes to the modularization of SheetLogic and the low level control in SILv2.

Options

1. Keep SheetLogic as part of the SIL core and the low level control simulation part of SheetLogic as shown in figures 2.9 and 3.1.

   Advantages: The low level control simulation stays in a central place in the SIL core and can therefore be used by other plant model simulation modules.

   Disadvantages: SheetLogic is part of every simulation, even if it is not needed. Simulation modules can not communicate with low level control simulation via the new communication approach because it requires synchronous communication. Coupling of SheetLogic and low level control simulation and SIL core, which has negative impact on modularity.

2. Extract SheetLogic from the SIL core and keep the low level control simulation as part of SheetLogic (communication still as in figure 3.1).

   Advantages: SheetLogic can be added to a simulation if needed.

   Disadvantages: The low level control simulation is needed by all other plant model simulation modules that use I/O-devices that have low level control. So, SheetLogic has always to be included. Simulation modules can not communicate with low level control simulation via the new communication approach because it requires synchronous communication.

3. Extract SheetLogic from the SIL core and extract the low level control simulation from SheetLogic. The use of the low level control in ESW simulation modules is shown in figure 3.5.

   Advantages: SheetLogic can be added to a simulation, if needed. The low level control simulation can be used independent of SheetLogic. The low level control simulation can be used as a layer beneath the I/O-layer and the output of this low level control layer is communicated to the SIL core, enabling communication only via the new communication approach.

   Disadvantages: The low level control simulation has to be used multiple times, as a part of each ESW simulation module that uses I/O-devices that have low level control, resulting in code duplication.

Decision

Extract SheetLogic from the SIL core and extract the low level control simulation from SheetLogic.

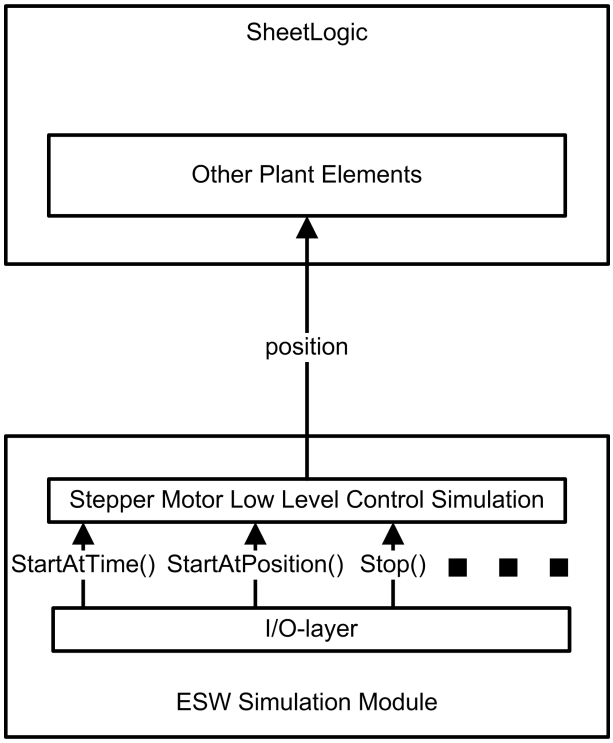

Figure 3.5: Low level control in SILv2.

Rationale

To provide the possibility to freely compose simulation modules with each other, the SheetLogic also has to become a simulation module. Much functionality can be taken from SheetLogic because it is also used by other plant models. SheetLogic becomes also decoupled from the SIL core because the new communication approach does not require the SIL core to know the functions of SheetLogic anymore. So the step to a total decoupling gets smaller. Examples for the extracted functionality are the command interface, visualization and error injection. The low level control simulation can be added as a layer to the ESW simulation modules (as shown in figure 3.5) so that the communication with the SIL core can be realized via the new communication approach.

3.4 New Communication Strategy

With the design decisions from the previous section, a new communication strategy is designed that meets the new requirements and supports the desired modular structure of SIL.

3.4.1 Global Principle

A solution for the problems, identified in section 3.2 is to modify the communication approach within SIL. The suggested solution is to use a more generic interface between simulation modules and the SIL core. The goal is a Plug-and-Play like approach for including simulation modules. The communication of the ESW and the plant models is still established by the SIL core. However, it is not necessary that the SIL core has knowledge of the used types of I/O as in SILv1. The communication in SILv2 is based on the exchange of variables between the

simulation modules. simulation modules have a certain number of I/Os that are represented by double variables. Those variables are created for sensors and actuators, used by the simulation modules. When the simulation is initialized, the simulation modules are loaded and started by the SIL core. During startup each module creates a data structure, called the SIL interface structure, containing the I/O information of the module comprising input and output variables and also variables that are defined as input and output. An input variable is only read by the simulation module, an output variable is only written and a variable that is defined as input and output is read and written by the simulation module. The structure of the SIL interface structure is shown in figure 3.6. The name and value of each I/O variable is stored. A reference to this data structure is requested by the SIL core, and is further used to communicate with the simulation modules. The variables of all simulation modules are stored in a central data storage in the SIL core, the variable database. To set up a connection for communication between simulation modules, the I/O variables from the modules are matched by name. If one module needs to communicate with another, each of them needs to register a variable with the same name. In the case of a digital sensor, for example, the plant model simulation module registers an output variable with the name "sens" and the ESW simulation module registers an input variable with the same name. The communication in runtime is done by the SIL core. The SIL core is responsible for updating the input variables of a simulation module with data from the variable database before a simulation module is executed and to write output variables back to the variable database after a module has been executed. This is done using the reference to the SIL interface structure of the corresponding simulation module. The communication between

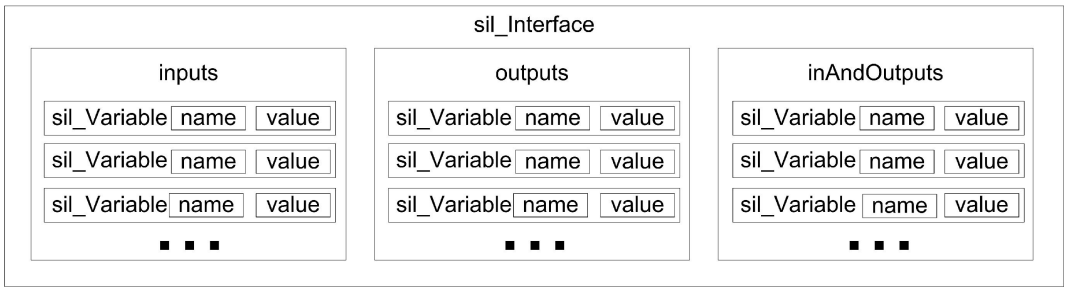

Figure 3.6: The structure of the SIL interface structure.

SIL core and modules is presented in figures 3.7 and 3.8. Figure 3.7 shows the initialization and figure 3.8 shows the communication in running state. During startup of the simulation, the following steps are taken when a module is loaded (numbers refer to figure 3.7):

1. The simulation module is loaded and started by the SIL core by calling "$sil\_StartNode()$".
2. During startup, the SIL interface structure, consisting of lists of input and output variables and variables that are defined as in- and output, is generated by the module. This is done by the ESW by calling the "init" function of all I/O-devices. In the "init" function a variable is created and added to the SIL interface structure.
3. When the simulation module is started successfully, the SIL core retrieves a reference to the SIL interface structure of the module by calling "$sil\_GetInterface()$".
4. The variables from the SIL interface structure are added to or found in the variable database. A list of handles ("HandleList") for the variable values is stored for the module. These handles are used in runtime to access the values of the variables without the need of searching them in the variable database.

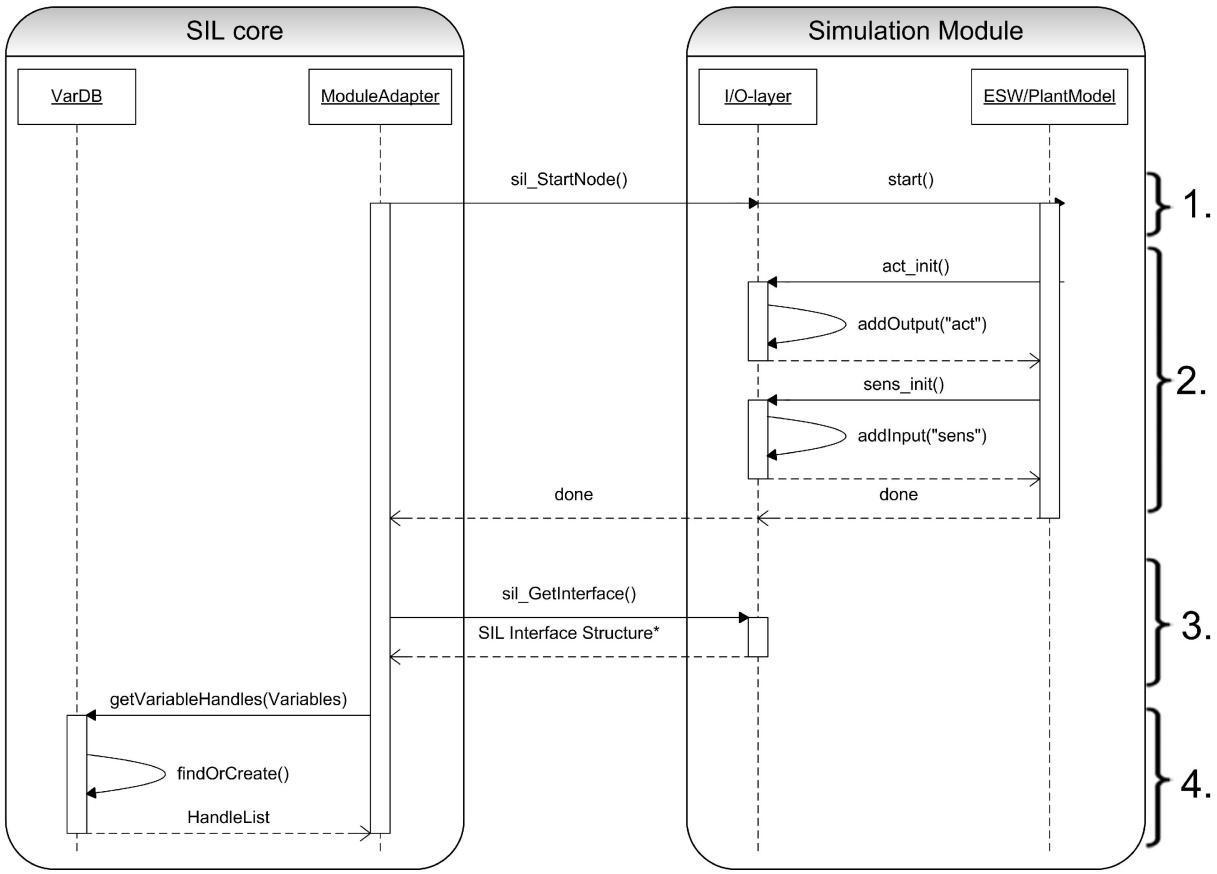

Figure 3.7: The new interface between simulation modules and the SIL core during initialization.

When the simulation is running, the following steps are taken with each computational step (tick function "$sil\_StartScheduling()$") of the module (numbers refer to figure 3.8):

5. In the simulation module, the local values of the input variables and variables that are defined as in- and output, are updated by the simulation core. The new input values are requested from the Variable database by calling "$getValues(HandleList)$".
6. A computational step of the simulation module is executed using "$sil\_StartScheduling()$". The ESW reads from and writes to I/O-devices. In the SIL I/O-layer this means the ESW reads from and writes to the variable values in the SIL interface structure.
7. The new output values and the values of variables that are defined as in- and output, are read by the simulation core and written to the variable database by calling "$setValues(HandleList, ValueList)$".

The structure of SILv2 is shown in figure 3.9. The SIL core and the simulation modules are strictly separated and only communicate via the presented interface (figure 3.7). The interface of simulation modules consists of the following functions:

- "$sil\_GetInterface()$"
- "$sil\_GetVersionInfo()$"
- "$sil\_StartNode()$"
- "$sil\_StartScheduling()$"

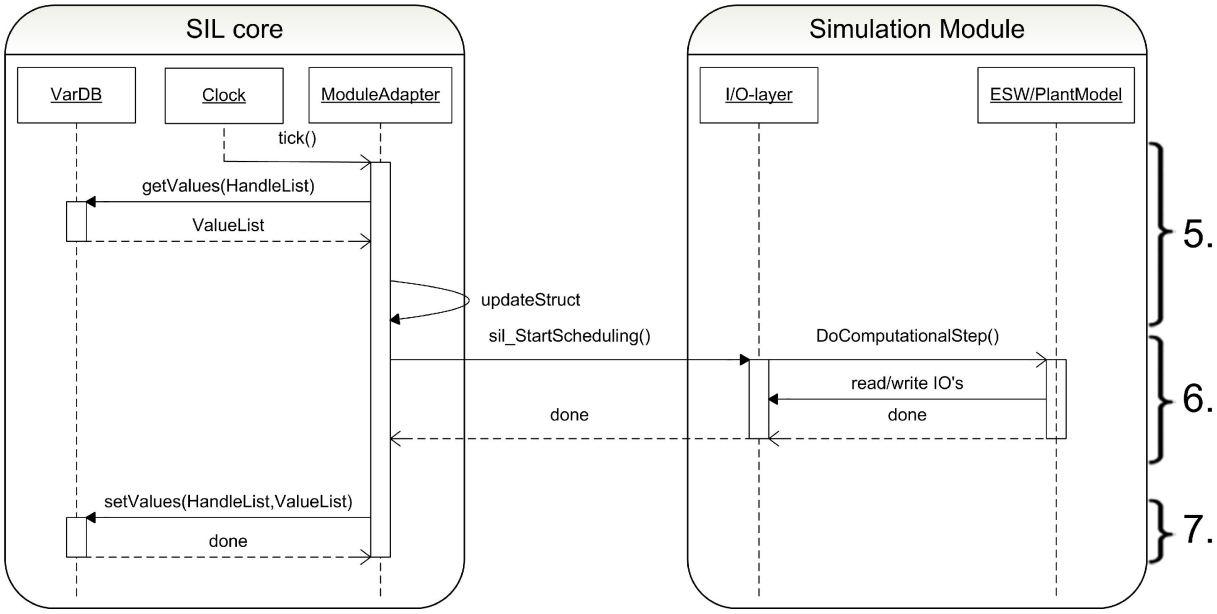

Figure 3.8: The new interface between simulation modules and the SIL core when running.

- "$sil\_StopNode()$"

"$sil\_GetVersionInfo()$" and "$sil\_StopNode()$" are not shown in the sequence diagrams in figures 3.7 and 3.8. "$sil\_GetVersionInfo()$" is optional used to get version information of an ESW simulation module. "$sil\_StopNode()$" is used to stop the simulation module when the simulation is ended.

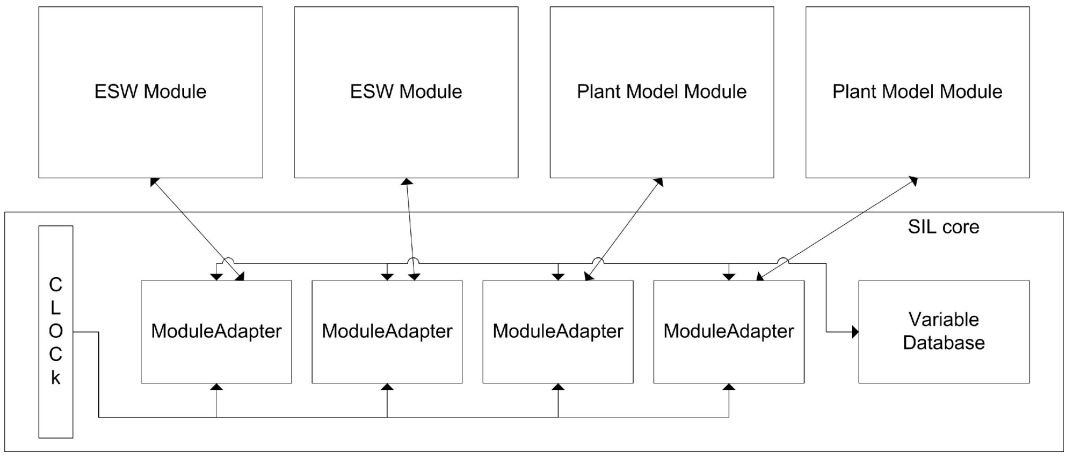

Figure 3.9: The global structure of SILv2.

The communication approach of SILv2 is inspired by the Blackboard architecture and the Publish/Subscribe messaging pattern. The Blackboard pattern is an architectural pattern that is used, for example, for artificial intelligence systems, which uses a central storage, which is used by several agents to communicate with each other. The central storage is called blackboard and the agents are called knowledge sources [11]. The idea of a central storage, similar to a blackboard, and multiple modules, similar to knowledge sources, is taken from this architectural

pattern. The way the simulation module communicates its inputs and outputs during initialization is inspired by the Publish/Subscribe messaging pattern [5]. Theoretically, the simulation module subscribes to some input variables and publishes several output variables.

3.4.2 Changed and added Entities of SIL

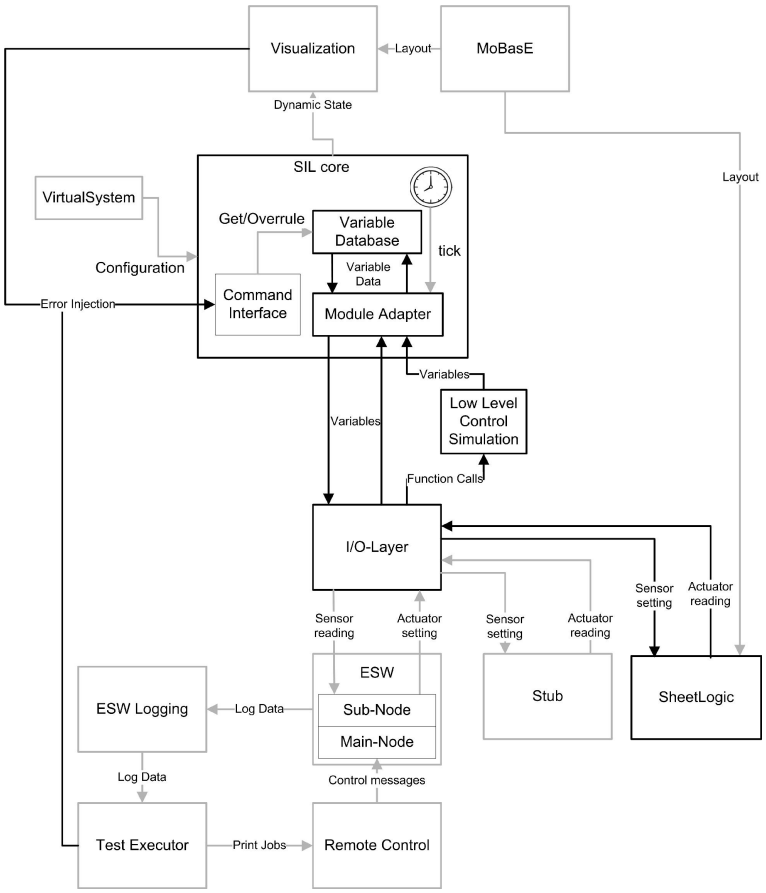

Figure 3.10: The entities changed in this project (everything that is solid black).

SIL Core

To apply the new communication approach, SheetLogic is logically decoupled and extracted from the SIL core. The SIL core becomes a generic simulation core that contains a scheduler, interfacing components to simulation modules and the variable database, which holds the current value of all variables from all modules. In addition there are additional features like the connection to the visualization and the command interface.

Module Adapter, the Interface to Simulation Modules
The module adapter component of the SIL core (called Node in SILv1) is responsible for loading and communicating with simulation modules. The module adapter also communicates with the variable database. In order to implement the new communication approach, this component has been changed to a large degree.

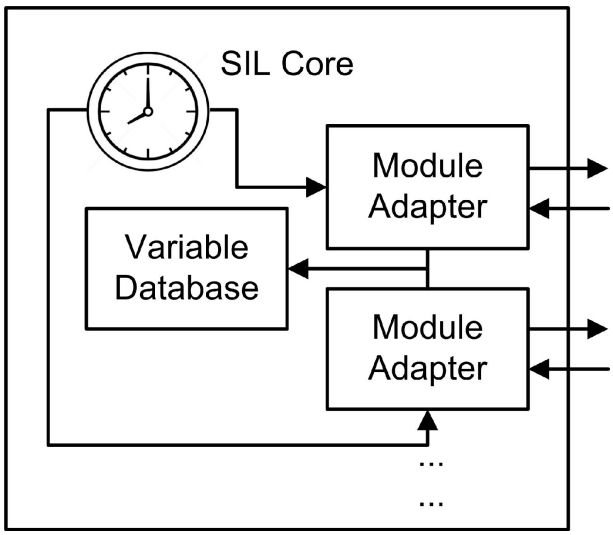

Figure 3.11: The new SIL core.

The most relevant changes are:

- Deletion of function pointer transmission of the old communication approach.
- Adding functionality to request SIL interface structure.
- Adding functionality to communicate with the variable database to write and read variable values.
- Adding functionality to write to and read from the SIL interface structure of simulation modules.

Variable Database
The newly added variable database contains the current value of all variables that are communicated between simulation modules. For each variable, the name and current value is stored. Next to the value and the name, additional information about variables is stored. For example, if a variable is overwritten with a certain value for error injection. The name is used as unique identification. The elementary functionality that has to be integrated is:

- Find or create variables, that are sent to the variable database by the module adapter, based on the variable name. When doing this a list has to be created, containing handles to the variable values, that can be used while simulating to access the value.
- Read variable values from the variable database using a list of handles.
- Write variable values to the variable database using a list of handles.

I/O-layer

The global structure of the I/O-layer not changed because it is not necessary. It is also not desirable because the interface to the ESW has to stay the same, so that the ESW itself has not to be modified. The global concept of the I/O-layer is to provide macros that can be used by the ESW. The macros expand into a number of functions that can be called by the ESW as shown in 2.8. The bodies of those functions have to be changed to implement the new communication approach. The functions do not have to use the function pointers to I/O-devices in SheetLogic anymore. In the functions the values of the variables in the SIL interface structure are read respectively written.

SheetLogic

SheetLogic is extracted and decoupled from the SIL core. SheetLogic is redesigned to use the new communication approach. To use the new communication approach, SheetLogic also uses the I/O-layer. In SILv2 SheetLogic becomes one of many plant model simulation modules. So it is also possible to run simulations without SheetLogic, which has not been possible in SILv1.

Low Level Control Simulation

In SILv1 (figure 3.1), the functionality of low level control is included in SheetLogic. The chosen solution is to extract the low level control from SheetLogic and to add it as a layer to the ESW simulation modules beneath the I/O-layer, like shown in figure 3.5. This solution makes it possible to use the new communication approach, because communication with the SIL core is based on physical variables like position of a motor as shown in figure 3.5. The low level control is included in SILv1 as a simulation. It would be better to include the real code of the low level simulation as a layer to the ESW simulation modules, enabling testing also for the low level control ESW.

3.5 Evaluation

To determine if and how the goal qualities of SILv2 are achieved and if SILv2 meets the identified requirements, the design is evaluated. In addition the design is compared to SILv1.

3.5.1 Change Case Impact Analysis

SILv2 has not only to deal with the day to day use of SIL during testing but also with possible change cases. Because of this, an evaluation and impact analysis of SILv2 and SILv1 is given for some identified change cases. First, for each change case, a high level description is given that describes the goals of the change. After that, an explanation of the necessary changes, is given.

A. Add a new kind of plant model

SheetLogic and also the EZ-VirtualPrinter approach (chapter 4), developed in this project, are implemented using a general purpose language. In is also possible to include Matlab Simulink models in SIL (chapter 4). To extend the simulation environment in the future, other, more specialized system modeling technologies, have to be integrated in SIL. An example is the object oriented physical system modeling language Modelica [2, 12].

SILv1
When a new type of plant model is added to SILv1, all I/O-device functions, which are used by the ESW simulation modules, have to be implemented as in SheetLogic. The low level control simulation has to be integrated in the new plant model simulation module or used from SheetLogic. Finally the new plant model must be integrated into the SIL core so that the SIL core can establish the communication between ESW simulation modules and the new plant model.

SILv2
In SILv2, a new plant model can be added with less effort compared to SILv1. The plant model must implement the SILv2 interface. The SIL interface structure has to be created with the corresponding in- and outputs. There is no need to create functions for different I/O-devices,

but it is necessary to interpret the variables communicated by the ESW simulation modules in the right way. The plant model has to be compiled to a DLL that can be loaded by the SIL core. How new kinds of plant models can be added to SILv2 is also discussed in the following chapter.

B. Add new Kind of I/O-device

The I/O-layer provides functionality to communicate with several I/O-devices. If a new I/O-device is used in a plant model simulation module and the corresponding ESW simulation module, it is necessary to add this I/O-device to the I/O-layer to enable communication. It is also necessary that the plant model knows how to handle the data from the new I/O-device.

SILv1
When a new I/O-device should be used in SILv1, it has to be added in the I/O-layer and the plant model. In addition, all functions to communicate with the I/O-device have to be added to the interface between the SIL core and the ESW simulation module. The transmission of the corresponding function pointers has to be added to the functionality of the SIL core that transmits the function pointers to the I/O-device functions to the simulation modules.

SILv2
In SILv2, the I/O-device has only to be added to the I/O-layer and to the plant model. The SIL core itself is not affected.

C. Add new Visualization

Another imaginable case is that a new visualization tool or an extension to Argus is used to enable visualization of new plant model simulation modules or to simply add visualization functionality.

SILv1
In SILv1, the visualization tool Argus connects directly to SheetLogic. If a new visualization is used it has to meet the requirements of the interface provided by SheetLogic. The other option is to change the interface in SheetLogic or to introduce a central entity in the SIL core that connects to the visualization.

SILv2
In SILv2, a new visualization can use the simulation module interface or a specialized version of it, if additional functionality is needed. The visualization can add all variables of the simulation to its inputs. With this technique the visualization can receive the state of the whole simulation. That means, the state of all variables of all simulation modules can be retrieved by the visualization, without the need of extensions in the SIL core or the interface.

D. Change Settings of Simulation Modules On-The-Fly

Another change case is that simulation modules can change their configurations on-the-fly during simulation (for example, the sample period). This can be useful, if a plant model needs to simulate some process with higher accuracy but other processes can be simulated less accurate. In the current situation the sample time of each module is specified before starting the simulation in the specification in the Virtual System XML file. One time specified, it stays fixed for a simulation run.

A first step would be that the simulation module defines its own sample frequency and communicates it during initialization. The following step would be that a simulation module can change its own configurations on-the-fly every time it is executed.

SILv1
To achieve this change case in SILv1 it is on the one hand necessary to implement the corresponding functionality in the SIL core. On the other hand a function pointer has to be added to the interface and has to be transmitted to the simulation module during initialization. This function pointer can be used to set or change the configurations of the simulation module in the SIL core.

SILv2
In SILv2, it is also necessary to implement the corresponding functionality in the SIL core. In SILv2 it is possible to add a list of configuration parameters to the SIL interface structure to communicate configurations. This can either be done during initialization or every time the simulation module is executed. If changes of the configuration are done on the fly, the SIL core has to check every "tick" if settings are changed in the SIL interface structure.

E. Going Back in Time for Debugging

When simulating, it is possible that the system shows undesired behavior. In this case, it could be desirable to stop the simulation and to rewind the simulation to closer investigate the situation.

In the current situation Argus provides record capabilities that make it possible to record the 3D visualization and to playback it later. However, only the visualization is recorded which can make it difficult to find the reason for the undesired behavior.

SILv1
In SILv1, the used plant model simulation modules could be used to record the state of the plant model entities to be able to replay incidents later. This implies that every plant model has to implement suitable functionality, which can be difficult or needs a lot of effort, if different plant modeling approaches are used.

SILv2
In SILv2, the current state of all I/O-devices is stored in the variable database, which makes it possible to log the current state of all plant models from this central place and to use the logged data to playback the simulation later.

F. Distributed Simulation

When more and more simulation modules and other features are added to SIL, it can be an option to distribute the simulation over two or more computers to enhance performance. A possible case is that the SheetLogic plant model simulation module of a large printing system with several PIMs and FINs is executed on another computer. In this case the communication of a simulation module and the SIL core has to be done via the network or comparable technology.

SILv1
First of all, it is not really possible to distribute the simulation in SILv1 because of the close coupling of plant model simulation modules and the SIL core. So, the SIL core has to be executed

on a computer together with the plant model simulation modules. To implement Distributed Simulation (distribution of ESW simulation modules) in SILv1, it is necessary to implement a proxy for the computer on which the SIL core (and SheetLogic or other plant model simulation modules) is executed and one for the computer on which the ESW simulation modules are executed. The proxy on the simulation module side has to implement all functions for all I/O-devices that are needed by the simulation module and has transmit the corresponding function pointers to the simulation module. The proxy on the SIL core side needs to be configured with the function pointers pointing to the I/O-device functions in SheetLogic by the SIL core. The ESW simulation modules and the SIL core can then communicate with the corresponding proxy. The communication between the proxies can be implement in an intermediate format. Figure 3.12 shows an overview of the proxies in the distributed simulation in SILv1.

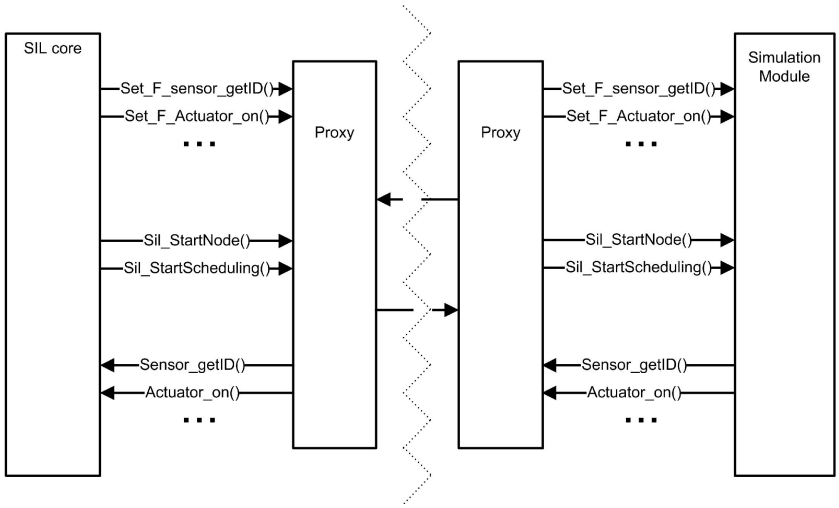

Figure 3.12: Overview of proxies in distributed simulation.

SILv2

In SILv2, the implementation of proxies is easier because less function pointers need to be communicated to the simulation module, which is caused by the higher degree of decoupling compared to SILv1. So less functions need to be implemented in the proxies. The data transfer can be implemented by simply sending the SIL interface structure from proxy to proxy, for example, via Ethernet. The key feature of the proxies has to be the forwarding of the "tick" function and the synchronization of the SIL interface structure. This can again be done in an intermediate format between the proxies. Figure 3.13 shows an overview of the proxies in the distributed simulation in SILv2.

G. Plant Modeling Approach with own Clock

In SILv1, the only plant model that is used is SheetLogic (next to the stub mechanism), which has, like the ESW simulation module, a "tick" function to perform a computational step. Also in SILv2, a requirement for the simulation modules is that the SIL core can call a "tick" function to let the simulation module perform a computational step. If a plant modeling approach is used, which uses an own clock in the created plant models, there must be a possibility to synchronize the SIL clock and the plant model clock.

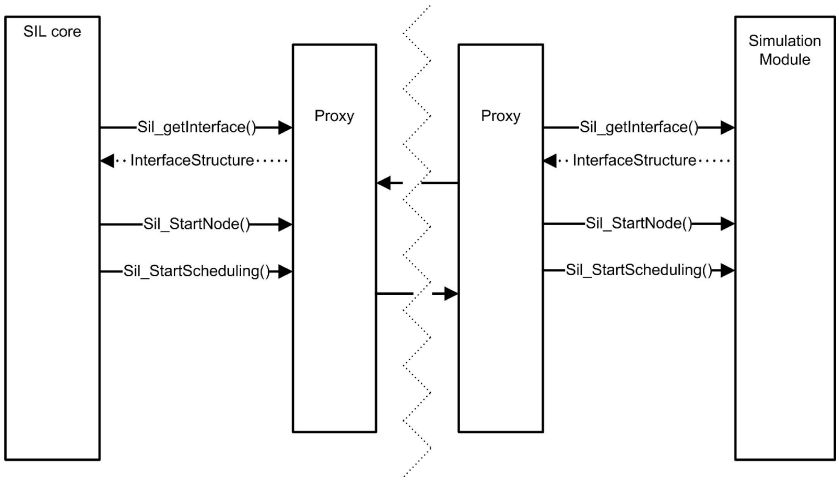

Figure 3.13: Overview of proxies in distributed simulation.

SILv1
This change case requires, similar to change case 4, communication of additional data via the interface of simulation module and SIL core. For SILv1 and SILv2 the corresponding functionality needs to be implemented in the SIL core and the plant model simulation module. The difference in this change case of SILv1 and SILv2 is that the synchronization information is communicated different. For SILv1 the needed function pointers need to be added to the interface and communicated to the simulation module during initialization.

SILv2
In SILv2 it is possible to add a “clock” or “time” variable to the SIL interface structure to synchronize the clocks of SIL and plant model simulation module. This would not lead to changes in the interface.

Impact Analysis

The following table provides an overview that shows the impact of different change scenarios in SILv2 and SILv1. Globally, changes that comprise the communication of additional information have less impact in SILv2. This is because of the new, more generic interface. Again, this ranking is created together with two software engineers, who supervise the evolution of SIL since the first version. The scale that is used consists of high, medium and low impact.

	SILv2	SILv1
A. New plant model type	Medium	High
B. New I/O-device type	Low	Medium
C. New Visualization	Low	High
D. Change Settings of Simulation Modules On-The-Fly	Medium	High
E. Going Back in Time for Debugging	Low	High
F. Distributed Simulation	Low	Medium
G. Plant Modeling Approach with own Clock	Medium	Medium

Table 3.2: Impact analysis of change cases in SILv1 and SILv2.

3.5.2 Key Drivers

This subsections evaluates to what extend the design of SILv2 achieves the identified key drivers. First, for each quality a statement is given how it has been considered in the design and by which decisions it has been affected. Based on these qualities, SILv2 and SILv1 are compared to evaluate if and how the design of SILv2 improves the qualities.

Modularity

Modularity is a key driver throughout the design of SILv2. A modular design is in this case the foundation to meet the other requirements and to achieve the important qualities. Modularity has been achieved by logical separation of logical independent entities of SIL. Examples are the separation of SheetLogic (or plant model simulation modules in general) and the SIL core or the separation of SheetLogic and the low level control. Those modules are more loosely coupled than in SILv1 by using a more generic and minimalistic interface.

Generic

Because the goal is to compose several simulation modules of different types, and to be well prepared for future changes, the SIL core and the communication between SIL core and simulation modules have to be generic. This is achieved by using a new communication approach, which uses a more generic interface and way of communication. This makes it possible to use SIL as a Plug-and-Play environment. Change cases B and D show that the generic interface enables the communication of arbitrary information via the new communication approach as long as the

information can be stored in one or more double variables and no synchronous communication is needed.

Extensibility

To be prepared for future changes, the design has to be extensible. Extensibility is achieved by modularizing the entities of SIL and reduce the coupling between them. So by keeping the design modular and the communication approach generic, the design is also more extensible than SILv1. Several change cases show how low the impact of certain changes in SILv2 is.

Performance

When using SIL to test the behavior of ESW, it is important to have a decent performance to make testing more effective. SILv2 does not enhance or lessen performance drastically. This is because the main load is produced by SheetLogic, other plant models and the ESW simulation modules. Theoretically SILv2 has less performance than SILv1 because all inputs and outputs are copied with each tick of a simulation module, even if they are not changed. In SILv1, if the ESW does not call the I/O-layer function of an I/O-device to change the status of an actuator or to request the status of a sensor, the corresponding function in SheetLogic was also not called by the I/O-layer. In the case when all inputs and outputs are changed by the ESW with each tick, the performance of SILv1 and SILv2 should be the same. In cases in which the ESW changes the state of an actuator or requests the state of a sensor multiple times in a "tick", SILv2 has a slight advantage since the values are, also in this case, only copied one time with each tick. In this case, in SILv1, the callback function to SheetLogic is used every time an I/O-device is accessed by the ESW. In practice, the difference in performance between SILv2 and SILv1 is not noticeable. This has also been confirmed by the users of SIL, working in projects in which SILv2 is already used.

Comparison SILv2 and SILv1

SILv1 and SILv2 have been compared how well the two designs support the key drivers of this project. The ranking has been created together with two software engineers, who supervise the evolution of SIL since the first version. The used scale for evaluation is:

++ very good
\+ good
0 moderate
- bad
– very bad

	SILv2	SILv1
Modularity	++	+
Generic	+	-
Extensibility	++	0
Composability	++	0
Performance	0	0

Table 3.3: Comparison of key qualities in SILv1 and SILv2.

SILv2 has a higher ranking in the most of the key qualities, because it has been designed with those in mind. The requirements for SIL changed drastically over time since SILv1 introduced.

SILv1 has been designed as a dedicated testing tool for testing ESW for the paper handling, so the difference is inevitable.

3.5.3 Meeting Requirements

This subsection evaluates if and how the new design meets the identified requirements.

R1. ESW shall be encapsulated within simulation modules.

The ESW simulation modules were already encapsulated to some degree in SILv1. However the fact that the old interface required the use of function pointers, which point to functions in SheetLogic, within the ESW simulation module, made the functioning of the ESW dependent on the function implementations in SheetLogic.

In SILv1, 67 functions in SheetLogic are called by the ESW simulation modules via function pointers. With the new communication approach, the ESW simulation modules do not need to call any function in SheetLogic or the SIL core. Since the ESW simulation modules do not have function pointers to SheetLogic (or other plant model simulation modules), the interface functions for setting those function pointers are also not needed anymore. So, in SILv2 the ESW simulation modules are even more decoupled by using the communication based on variables.

R2. Plant models shall be encapsulated within simulation modules.

By changing the communication approach, it is possible to encapsulate plant model simulation modules in the same way as ESW simulation modules. This can be done because the coupling of the SIL core, SheetLogic (or some other plant model simulation module) and the low level control, is eliminated by not using function pointers for communication anymore. So, the SIL core does not need to have any knowledge about functions in plant model simulation modules. From the point of view of the SIL core there is no difference between a plant model simulation module or an ESW simulation module because the interface is the same.

R3. A simulation shall be composed of multiple simulation modules.

It has already been possible to compose several ESW modules in SILv1 (R1). But the only usable plant model has been SheetLogic (next to the stub mechanism), which was used by all ESW modules. The communication approach of SILv2 makes it possible to encapsulate plant model simulation modules (R2). These plant model simulation modules use the same interface as the ESW simulation modules, which is based on exchange of variables. In contrast to SILv1, it is possible to use multiple plant model simulation modules in SILv2. All simulation modules used in a simulation setup can communicate with each other by registering communication variables with the same name. A limitation of the composability is that not more than one simulation module can have a certain variable defined as output. So, it is valid if one simulation module has the variable “sens1” defined as output and 3 other simulation modules have the variable “sens1” defined as input. A fifth simulation module that is added to the simulation, which also outputs a value for the variable “sens1”, leads to undesired behavior.

R4. Plant models shall be implemented using different plant modeling technologies.

SILv2 itself does not provide the possibility to use multiple plant modeling approaches for creating plant model simulation modules. However, SILv2 enables the composition of multiple plant model simulation modules (R3), which is a foundation for meeting this requirement. The interface that a plant model must provide (SILv2) is also more compact and generic than the

old interface (SILv1). Integrating it in a plant modeling approach, therefore needs less effort. change case A and also the next chapter provide information about adding a new kind of plant model.

R5. Communication between ESW simulation modules and plant model simulation modules shall be based on a generic and well maintainable interface.

The communication approach of SILv2 introduces a compact and generic interface that does not depend on communicated I/O types. The interface is better maintainable because no functions have to be added if a new I/O-device is added to a ESW or plant model simulation module. Change case B and D show how new information can be communicated via the new communication approach without changing the interface structure.

3.5.4 Additional Advantages of SILv2

SILv2 introduces the variable database as a central storage of the current state of the variables from all simulation modules. In SILv1 the current status was stored in SheetLogic. SheetLogic was the central place of simulating plant behavior in SILv1 and therefore this choice introduced no problems. SILv2 supports multiple plant model modules, which makes it necessary to somehow manage visualization and error injection of plant model simulation modules. Those features have been included in SheetLogic in SILv1. The variable database as central storage entity can be used to implement those feature in SILv2. In SILv1, the visualization tool, Argus, is connected to SheetLogic via a dedicated interface. When using multiple plant model modules it is not feasible to connect Argus, or another visualization, to all used plant model modules. In SILv2 the aim is to connect Argus via the simulation module interface. Argus adds all variables of the variable database to its input variables, so that it receives the current values of all simulation modules. The variable database can also be used to inject errors by overwriting variable values in the variable database.

3.6 Implementation

The implementation of SILv2 is separated in two major parts. The one being the implementation of the new communication approach in the I/O-layer of the simulation modules and the other being the implementation of the new communication approach in the SIL core. The complete implementation of SILv2 is beyond the scope of this project. The complete implementation consists of:

- Extract and modify SheetLogic from the SIL core.
- Extract the low level control simulation from SheetLogic.
- Modify the SIL core (ModuleAdapter) for the new communication approach.
- Add the variable database to the SIL core.
- Modify I/O-layer for the new communication approach.
- Change the I/O-device-macros to use the SIL interface structure.

A major part of these modifications have been implemented in this project:

Extract and modify SheetLogic from the SIL core/ Extract the low level control simulation from SheetLogic: The low level control has not been extracted from SheetLogic because this requires a complete redesign of SheetLogic and of the low level control simulation. SheetLogic has been separated from the SIL core to some extend as a prove of concept. This

has been done by adding a layer that implements the SILv2 interface. Via those layer, the I/O-devices that have no low level control, are communicating with the SIL core via the new communication approach.

Modify the SIL core (ModuleAdapter) for the new communication approach/ Add the variable database to the SIL core: These modifications are implemented in the SIL core and are used in several projects.

Modify I/O-layer for the new communication approach/ Change the I/O-device-macros to use the SIL interface structure: The new communication approach has been integrated in the I/O-layer. The I/O-device-macros are only changed for the I/O-devices that are also supported by the adapted SheetLogic. So, only for those that do not use the low level control simulation.

3.6.1 I/O-layer

The SIL I/O-layer is implemented in C. The implementation is distributed over several separate files of which three are defining the interface and communication approach. These files are:

- SILSimulationItf.h: This file defines all interface functions of a simulation module. It also defines the SIL interface structure and type definitions for the function pointers to SheetLogic, set by the SIL core.
- ln_main.c: This is the main file of the simulation module I/O-layer. It contains the following functions of the interface: "$sil\_GetVersionInfo()''$, "$sil\_StartNode()''$, "$sil\_StartScheduling()''$ and "$sil\_StopNode()''$.
- ln_iolayerTargetMacros.c: This files defines the I/O-device-macros that are used by the ESW to create the functions that are used to communicate with I/O-devices, as described in figure 2.8. In this file the creation and initialization of the SIL interface structure is implemented. It also contains the function "$sil\_GetInterface()''$.

All of those three files are modified to implement the new communication approach.

SIL Interface Structure

The SIL interface structure has two responsibilities. It is, on the one hand, used to communicate the interface description of the simulation module to the SIL core, and is, on the other hand, used for communication. The SIL interface structure is defined in SILSimulationItf.h using a struct:

```
1  struct sil_Variable
2  {
3      const char* name;
4      double value;
5  };
6
7  struct sil_Interface
8  {
9      struct sil_Variable *inputs;
10     int inpCount;
11     struct sil_Variable *outputs;
12     int outpCount;
13     struct sil_Variable *inAndOutputs;
14     int inAndOutpCount;
15 };
```

Listing 3.1: The SIL interface structure.

I/O-device-target macros

The first step when a simulation module is started is to create the SIL interface structure. After that, the I/O's are initialized by the ESW of the simulation module, which causes that a variable is added to the SIL interface structure. How this is done is demonstrated with the following example. Listing 3.2 shows the initialization part of the macro definition that is used for the simple sensor I/O-device.

```
#define SILSIMULATION_SIMPLE_SENSOR(symbolicName, deviceId, IoId)\
        double* symbolicName##_state = NULL;\
        void symbolicName##_init(void)\
        {\
                if(symbolicName##_state == NULL)\
                {\
                        symbolicName##_state = addInpComponent(#symbolicName);\
                }\
        }\
```

Listing 3.2: Part of the macro for simple sensors.

First the pointer to the state variable of this I/O device is created ("*double∗ symbolicName##_state*"). The function "*void symbolicName##_init(void)*" is called by the ESW of the simulation module after startup. In the this function, the function "*double∗ addInpComponent(const char∗ name)*" is called, which is shown in listing 3.3.

```
static double* addInpComponent(const char* compName)
{
        struct sil_Variable temp={compName, 0};
        outputs[outpCompCount] = temp;
        return &outputs[outpCompCount++].value;
}
```

Listing 3.3: The function addInpComponent.

The parameter of this function is the name of the device, which becomes the name of the variable. The return value is a pointer to the value of the newly created SIL variable in the SIL interface structure. This pointer is further used in the macro, which can be seen in listing 3.4. The initialization step is the same for all I/O-devices.

```
#define SILSIMULATION_SIMPLE_SENSOR(symbolicName, deviceId, IoId)\
        \
        /*create a double pointer that points to the variables value*/\
        double* symbolicName##_state;\
        \
        void symbolicName##_init(void)\
        {\
                if(initialized != 1)\
                {\
                        /*call addInpComponent at init and retrieve*/\
                        /* pointer that points to the variables value.*/\
                        /* Values of inputs are automatically update by SIL before */\
                        /* the software module gets a tick. Outputs are */\
                        /* automatically read by SIL after a ``tick'' */\
                        symbolicName##_state = addInpComponent(#symbolicName);\
                }\
        }\
        \
        E_SimpleStatus symbolicName##_status(void)\
        {\
                /*use the pointer instead of the sheetlogic functions*/\
                if (doubleEqual(*symbolicName##_state, 0.0))\
                {\
                        return SENSOR_INACTIVE;\
                }\
                else\
```

```
27                  {\
28                          return SENSOR_ACTIVE;\
29                  }\
30          }\
31          void symbolicName##_on(void)\
32          {\
33                  *symbolicName##_state = 1.0;\
34          }\
35          void symbolicName##_off(void)\
36          {\
37                  *symbolicName##_state = 0.0;\
38          }\
```

Listing 3.4: The whole simple sensor I/O macro.

The rest of the I/O macro consists of three more functions that are called by the ESW of the simulation modules in runtime. "*E_SimpleStatus symbolicName##_status(void)*" is called by ESW simulation modules to retrieve the current status of the sensor. "*void symbolicName##_on(void)*" and "*void symbolicName ##_off(void)*" are used by stub simulation modules to set a new status of a sensor. It can be seen that in those functions simply the pointer to the value of the SIL variable in the SIL interface structure is used. All other modified I/O-device-target macros can be found in the file ln_iolayerTargetMacros.c in appendix A.

3.6.2 SIL Core

The SIL core is implemented using RoseRT. For the evolution to SILv2 several entities in the SIL core are added or modified.

Overview

The global structure of the SIL core did not significantly change, since SheetLogic has not been fully extracted. Figure 3.14 shows the RoseRT overview of the SIL core. The variable database does not appear in the overview since it is implemented using a passive (normal) class. Those entities are not shown in overviews of active classes (capsules).

Module Adapter

The module adapter component, called A_Node in the implementation and in figure 3.14, is used to load and configure simulation modules during initialization. When the simulation is running, the module adapter communicates with the clock, the simulation module and with the variable database. Figure 3.15 shows the state diagram of the module adapter component. For each simulation module, a module adapter is instantiated. In the initial transition, the module adapter is configured using information from the VirtualSystem XML file. This comprises, for example, the name of the simulation module and the path to the corresponding DLL. If the DLL is not found, the module adapter is stopped and is not further executed ("false" transition of choice point isDllFound). If the DLL is found, the simulation module DLL is loaded. After the DLL is loaded, it is checked if the DLL provides the needed functions of the DLL interface. If not all needed functions are found the, the module adapter is stopped and not further executed ("false" transition of choice point isDllFound). When all functions are found, the simulation module is started by calling "*void sil_StartNode()*". The ModuleAdapter registers itself with the sample rate of the simulation modules with the SIL clock, so that it receives "ticks" according to the sample rate. In the "true" transition from the choice point isDllFound, function pointers are transmitted to the simulation modules for I/O-devices that are not yet changed to the new interface. After that, in the same transition, the SIL interface structure is requested form the

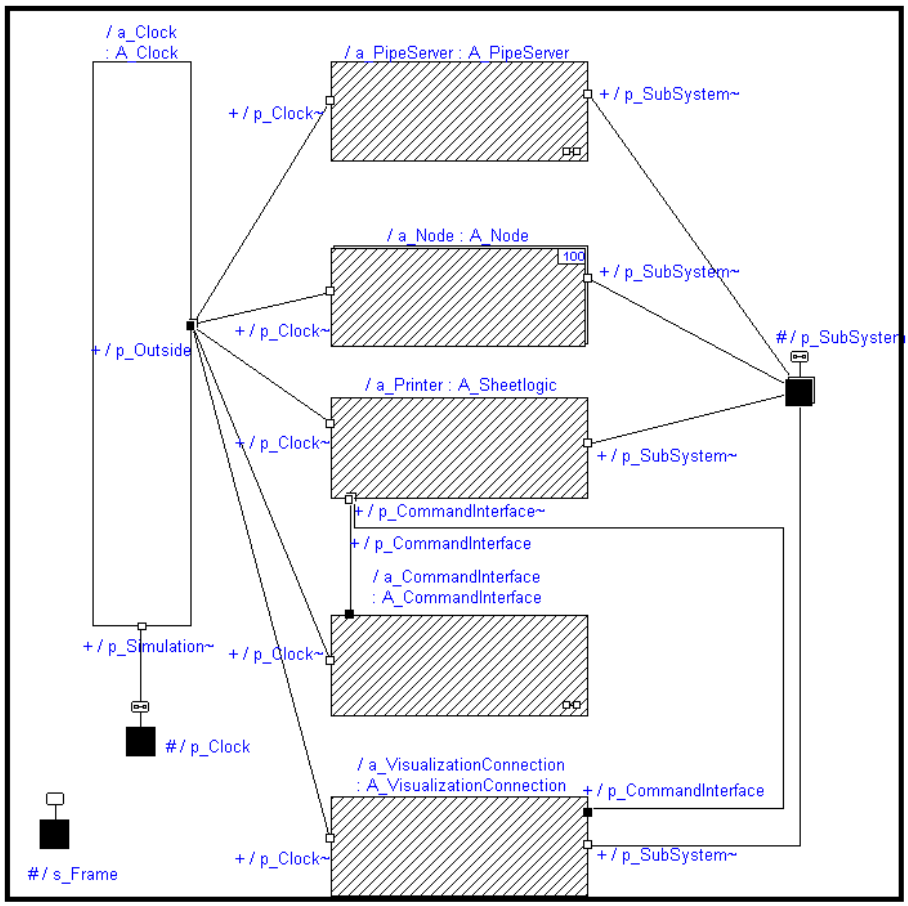

Figure 3.14: RoseRT structure diagram of the SIL core.

simulation module and the contained variables are send to the variable database to receive the handles (listing 3.5). In listing 3.5, three vectors are created, one for each list of variables (inputs, outputs and inAndOutputs). Each variable list is iterated and the name of the variable is send to the variable database. The returned handle is added to the corresponding handle vector.

```
1  vp_VariableStruct->inpCount ; i++){ v_InputHandleVector.push_back(vp_VarDB->
2          getVariableHandle(vp_VariableStruct->inputs[i].name, false));
3  }
4  for( int i = 0 ; i < vp_VariableStruct->outpCount ; i++){
5    v_OutputHandleVector.push_back(vp_VarDB->
6          getVariableHandle(vp_VariableStruct->outputs[i].name, true));
7  }
8  for( int i = 0 ; i < vp_VariableStruct->inAndOutpCount ; i++){
9    v_InAndOutputHandleVector.push_back(vp_VarDB->
10         getVariableHandle(vp_VariableStruct->inAndOutputs[i].name, true));
11 }
```

Listing 3.5: Get variables handles.

After this is done, the module adapter is in running state and waits for the clock to send a "tick" that triggers a "tick" transition. The "tick" transition updates the input variables of the simulation module, calls the "tick" function ("*void sil_StartScheduling*()") of the simulation module and writes the new output values of the simulation module to the variable database (listings 3.6, 3.7, 3.8).

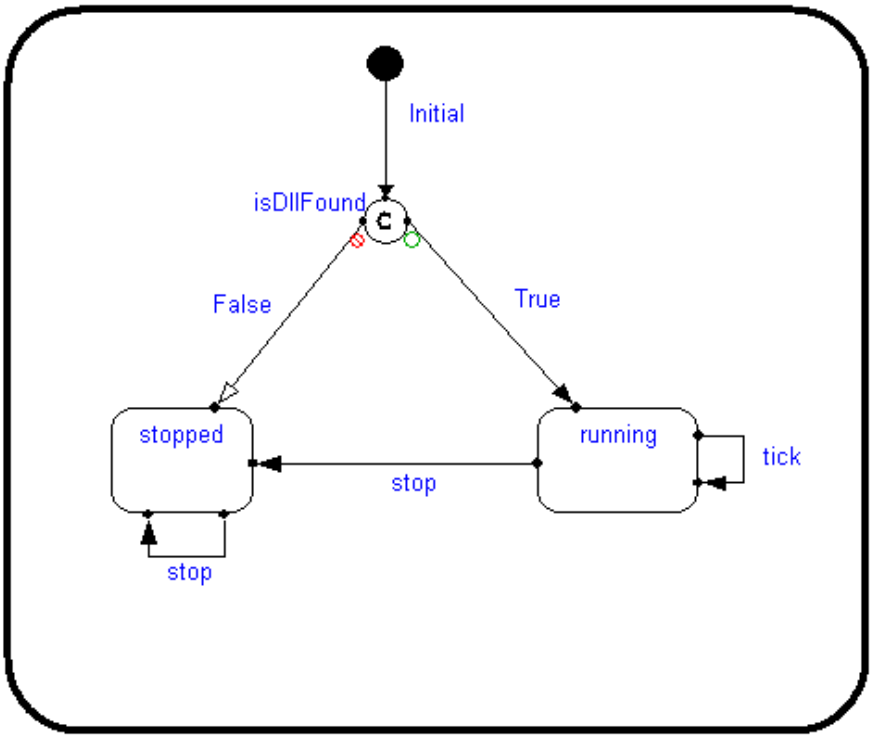

Figure 3.15: RoseRT state diagram of the module adapter component.

```
1  //read values of input variables from variable database and
2  //write them to the SIL interface structure of the simulation module
3  f_UpdateValues();
4  //sil\_StartScheduling()
5  vfp_NodeTrigger();
6  //read values of output variables from SIL interface structure of the
7  //simulation module and send them to variable database
8  f_WriteValues();
9  //Send done to the clock so that the next ModuleAdapter receives a "tick"
10 p_Clock.done().send();
```

Listing 3.6: Tick transition of the ModuleAdapter.

```
1 vp_VarDB->getValues(&v_InputHandleVector,vp_VariableStruct->inputs);
2 vp_VarDB->getValues(&v_InAndOutputHandleVector,vp_VariableStruct->inAndOutputs);
```

Listing 3.7: f_UpdateValues()

```
1 vp_VarDB->setValues(&v_OutputHandleVector,vp_VariableStruct->outputs);
2 vp_VarDB->setValues(&v_InAndOutputHandleVector,vp_VariableStruct->inAndOutputs);
```

Listing 3.8: f_WriteValues()

The communication with the simulation module is simply done by reading and writing to the variable values in the pointer to the SIL interface structure of the corresponding simulation module. The functionality of the functions, provided by the variable database (VarDB), is further explained in the following paragraph.

Variable Database

The Variable database provides an interface for the module adapter for communication during initialization and while running. The variable database is implemented using the singleton pattern to avoid the use of multiple instances. In previous section, describing the module adapter, some functions are mentioned that are used by the module adapter. These functions are:

- "$void\ getValues(sil\_Variable*\ listOfVariables,\ std::vector<int>*\ handleList)$"
- "$void\ setValues(sil\_Variable*\ listOfVariables,\ std::vector<int>*\ handleList)$"
- "$int\ getVariableHandle(char*\ ap\_Name,\ bool\ a\_IsOutputVariable)$"

and are further presented in this section. Two attributes in the context of the communication with the module adapter are "$nameMap$" and "$valueVector$". The "$nameMap$" is implemented using a map data structure with a string (variable name) as key and an integer (index of the variable value in the "$valueVector$") as value. At start it is empty and after initialization it holds the names of all variables from all simulation modules and their corresponding index in the "$valueVector$". The "$valueVector$" is implemented using a vector. It holds objects of the type "$VarInfo$". "$VarInfo$" is a helper class, which stores the variable value and other information about a variable. The function "$int\ getVariableHandle(char*\ ap\_Name,\ bool\ a\_IsOutputVariable)$" is shown in listing 3.9.

```
1  // parameters: char* ap_Name, bool a_IsOutputVariable
2  // check if variable exist in nameMap and returns handle (index)
3  std::map<std::string, int>::const_iterator lc_Iter;
4  lc_Iter = nameMap.find(ap_Name);
5  if( lc_Iter != nameMap.end())
6  {
7    // is in map
8    D_VarInfo l_VarInfo = valueVector[lc_Iter->second];
9    if(a_IsOutputVariable)
10   {
11           // varNames contains variables written by the module
12     if(l_VarInfo.get_isOutputOfModule())
13           {
14             // variables are already written by another value ERROR
15       log->logPrintf(logGroup, LOG_LEVEL_INCARNATION,
16                "WARNING: Variable %s is written by more than one module.\n",ap_Name);
17       return lc_Iter->second;
18     }
19           else
20           {
21           // not written yet. add to handle list and set alreadyWritten to true
22       l_VarInfo.set_isOutputOfModule(true);
23       return lc_Iter->second; // return index of vector
24     }
25   }
26   else
27   {
28  // varNames contains variables that are just input. no check needed
29     return lc_Iter->second; // return index of vector
30   }
31 }
32 else
33 {
34   //not in map
35   D_VarInfo* lp_VarInfo = new D_VarInfo(a_IsOutputVariable, ap_Name, 0);
36   nameMap[ap_Name] = nrOfVariables; // create map entry with name and next available
         index
37   valueVector.push_back(*lp_VarInfo);
38   return nrOfVariables++; // return index in vector, last available index++
39 }
```

Listing 3.9: getVariableHandle()

Basically, the function searches in the "$nameMap''$ for the variable name and determines if the variable is already known. If it is known the index of the variable value, stored in the "$nameMap''$, is returned. Otherwise the variable is added to the "$nameMap''$ and the value to the "$valueVector''$, and the index of the of the value of the new variable in the "$valueVector''$ is returned. The code becomes more complex because it has to be checked if more than one simulation module has defined a certain variable as output, which is an indication for wrong configuration and a warning is written to the log file. After initialization each $''moduleAdapter''$ has called this function and holds lists with handles to the values of its input and output variables.

In runtime the module adapter uses the functions

- "$void\ getValues(sil\_Variable*\ listOfVariables,\ std::vector<int>*\ handleList)''$
- "$void\ setValues(sil\_Variable*\ listOfVariables,\ std::vector<int>*\ handleList)''$

to read fresh values of input variables from the variable database and write new values to output variables in the variable database. The following two listings (3.10, 3.11) show the two functions.

```
1 // parameters: std::vector<int>* ap_Handles, sil_Variable* ap_Vars
2 for ( unsigned i = 0 ; i < ap_Handles->size() ; ++i )
3 {
4   // for each handle: add the value from the list of variables
5   // to the corresponding index in the valueVector
6   valueVector[(*ap_Handles)[i]].setValue(ap_Vars[i].value);
7 }
```

Listing 3.10: setValues()

```
1 // parameters: std::vector<int>* ap_Handles, sil_Variable* ap_Vars
2 for( unsigned i = 0 ; i < ap_Handles->size() ; ++i)
3 {
4   // add the value from the valueVector on the index saved in
5   // the ith  element of the list of indices to the valueList
6   ap_Vars[i].value = valueVector[(*ap_Handles)[i]].getValue();
7 }
```

Listing 3.11: getValues()

While reading or writing the values, there is no need for searching the variables because of the handle list system. This improves performance and makes the implementation of those two function very compact.

Error Injection via the Command Interface

To make it possible to overwrite variables in the variable database during simulation to inject errors, the command interface has been extended. The "$VarInfo''$ class, which is used by the variable database to store information such as the current value of a variable, has also methods and fields that make it possible to overrule variables. The fields involved are:

- "$bool\ overruled''$
- "$double\ overruledValue''$

and the involved methods are:

- "$void\ resetOverruleVariable()''$
- "$void\ overruleVariable(double\ a\_Value)''$

As described in the context of this project (section 2.3.2), the command interface can be used to inject errors from Argus into the SIL core. It can also be used to retrieve sensor or actor values. This command interface is only changed slightly to provide the possibility to overrule a variable in the variable database to enable error injection. Functionality to request variable values from the variable database from Argus has also been implemented. Overruling a variable means to add a overrule value to the variable, which will be returned when the variable value is requested till the overruling is reset. The real value is still set to the variable but the overrule value is returned. To overrule a variable, the variable is searched by name and the function "$void\ overruleVariable(double\ a\_Value)$" is called with the desired value. When a reset overrule command is send, the reset overrule function is called and the real value of the variable is returned again.

Plotting Variable data in Runtime

As described in the context of this project (section 2.3.2), it is possible to plot the values of I/O-devices when the simulation is paused. This functionality has also been added for variables from the variable database. When the simulation is paused, a menu in the SIL console application can be used to navigate through the variables of the system and to plot them. For this purpose the function "$double\ getVariableValue(const\ char*\ name)$", which has been added to the variable database, is used. Figure 3.16 shows the console menu. In the case of figure 3.16, there are three variables available that can be plotted. The program that is used for plotting variables is still the same as in SILv1 as shown in figure 2.13.

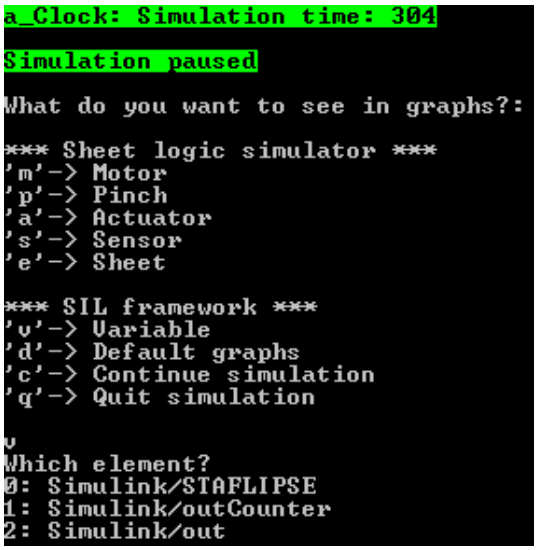

Figure 3.16: The SIL console menu to plot variables.

3.6.3 Adaption of SheetLogic for the new Communication Strategy

To support the new interface, SheetLogic has been modified so that it conforms to the new interface. A total redesign of the SheetLogic plant model has, however not been in the scope of this project. The adaption to the new interface has been done via a "glue layer". This "glue layer" translates communication from the new interface to the old interface (function calls to I/O-devices in SheetLogic) and back again to the new interface. This way SheetLogic

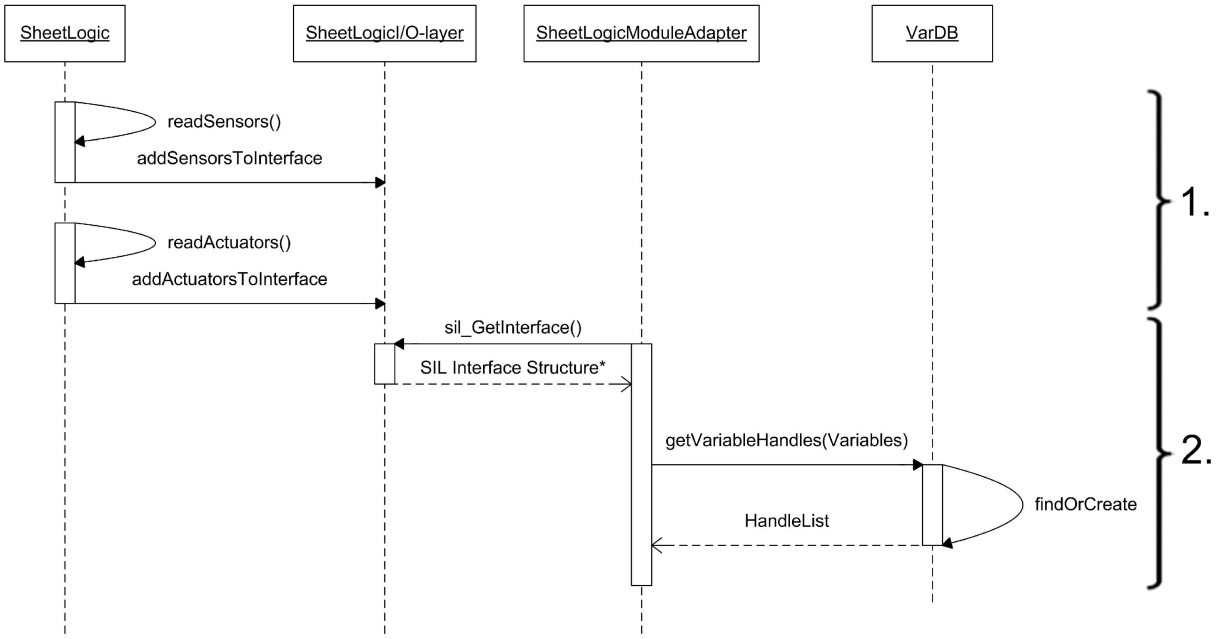

Figure 3.17: The glue layer to adapt the communication of SheetLogic to the new interface.

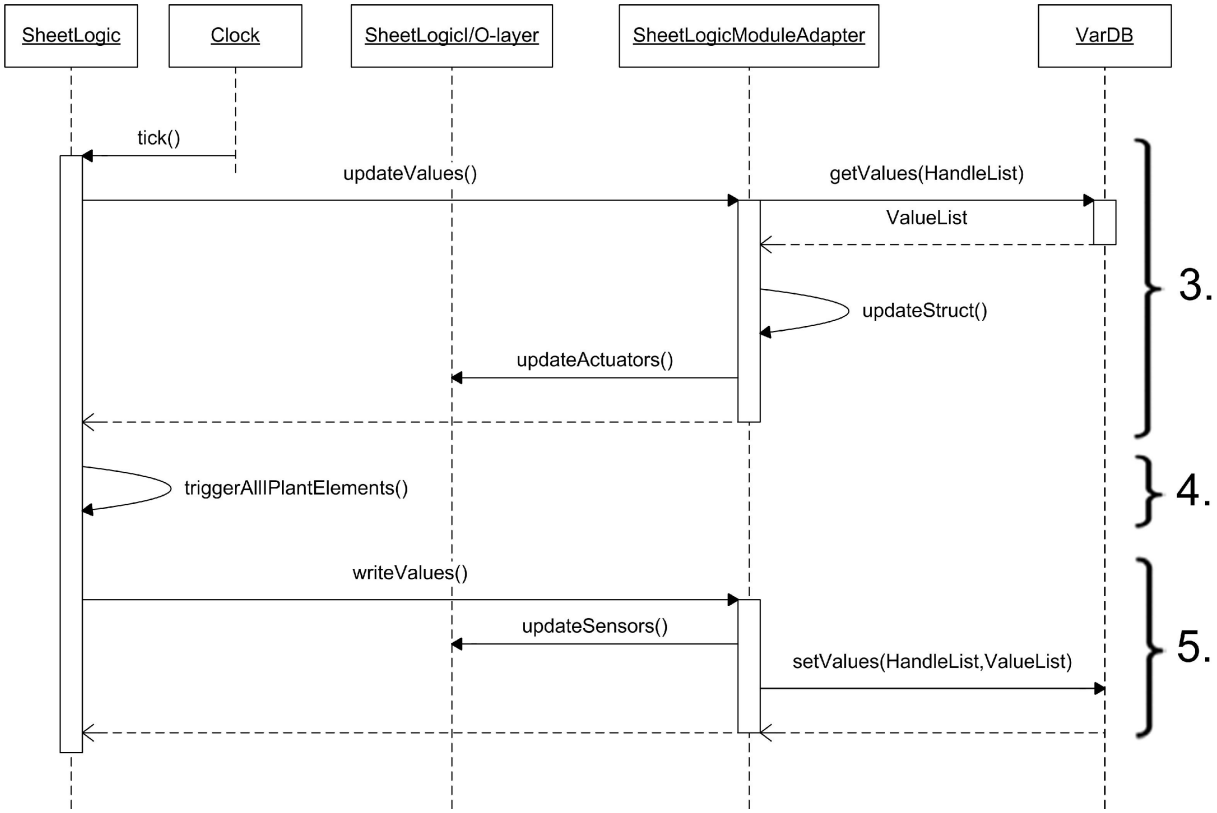

Figure 3.18: The glue layer to adapt the communication of SheetLogic to the new interface.

has been changed only slightly. Figure 3.17 shows the communication during initialization and

figure 3.18 shows the communication in runtime. The glue layer basically simulates the new interface in direction of the SIL core and the old function call based interface in the direction of SheetLogic. The glue layer consists of two classes, the "$SheetLogicModuleAdapter''$ and the "$SheetLogicI/O-layer''$. These to classes are comparable to the I/O-layer and the module adapter for simulation modules.

The initialization corresponds for the most part to the initialization of a simulation module. Differences are that the SheetLogic does not need to be loaded because it is still part of the SIL core and is started together with it. When SheetLogic reads in all I/O-devices from the VirtualSystem XML file and instantiates them during initialization, a variable is created and added to the SIL interface structure for every I/O-device (step 1. in figure 3.17). After that "$sil\_GetInterface()''$, located in the "$SheetLogicI/O-layer''$, is called by the "$SheetLogicModuleAdapter''$. After that the handles for the variables are requested from the variable database (step 2. in figure 3.17). While running, the SheetLogic gets "ticks" from the SIL clock to perform computational steps just as the module adapter. After the "tick" function is called, the input variables of SheetLogic are requested from the variable database and updated in SheetLogic (step 3. in figure 3.18). The input variables of SheetLogic are the variables, created for actuators. In step four, all plant elements in SheetLogic receive a "tick" to calculate the next state. The last step comprises writing the new sensor values (output variables) back to the variable database (step 5. in figure 3.18).

The functions "$void\ updateActuators()''$ and "$void\ updateSensors()''$ perform the translation from the new interface to function calls and vice versa. Listing 3.12 shows the translation from variables to function calls for SheetLogic. Listing 3.13 shows the translation from function calls to variables for the new interface.

```
1  std::map<int, int>::iterator l_IterAct;
2  for(l_IterAct = simpleActuatorMap.begin(); l_IterAct != simpleActuatorMap.end();
       l_IterAct++)
3  {// iterate all Simple Actuators in SheetLogic
4    if((uint16_t)interfaceStruct.inputs[l_IterAct->first].value != Actuator::
         Actuator_getStatus(l_IterAct->second))
5    {// if the variable value differs from the value of the actuator in SheetLogic
6      if(interfaceStruct.inputs[l_IterAct->first].value == 0.0)
7          {// If value is 0
8        Actuator::Actuator_off(l_IterAct->second);// turn actuator off
9      }
10         else
11         {// If value is 1
12       Actuator::Actuator_on(l_IterAct->second);// turn actuator on
13     }
14   }
15 }
16
17 // update analog actuator in sheetlogic if the variable value is changed
18 std::map<int, int>::iterator l_IterAnaAct;
19 for(l_IterAnaAct = analogActuatorMap.begin(); l_IterAnaAct != analogActuatorMap.end();
       l_IterAnaAct++)
20 {// iterate all analog Actuators in SheetLogic
21   if((int16_t)interfaceStruct.inputs[l_IterAnaAct->first].value != AnalogActuator::
         AnalogActuator_getStatus(l_IterAnaAct->second))
22   {// if the variable value differs from the value of the actuator in SheetLogic
23     if(interfaceStruct.inputs[l_IterAnaAct->first].value == 0.0)
24         {// If value is 0
25           // turn actuator off
26       AnalogActuator::AnalogActuator_off(l_IterAnaAct->second);
27     }
28         else
29         {// If value is something else
30           // turn actuator on with specific value
31       AnalogActuator::AnalogActuator_on(l_IterAnaAct->second, (int16_t)interfaceStruct.
             inputs[l_IterAnaAct->first].value);
32     }
```

```
33   }
34 }
```

Listing 3.12: updateActuators()

```
35 std::map<int, int>::iterator l_IterSens;
36 //for each sensor in SheetLogic
37 for(l_IterSens = simpleSensorsMap.begin(); l_IterSens != simpleSensorsMap.end();
       l_IterSens++) {
38   // write sensor value to interface structure
39   interfaceStruct.outputs[l_IterSens->first].value = (double)Sensor::
         SimpleSensor_getStatus(l_IterSens->second);
40 }
41 //update analogSensors
42 std::map<int, int>::iterator l_IterAnaSens;
43 //for each sensor in SheetLogic
44 for(l_IterAnaSens = analogSensorsMap.begin(); l_IterAnaSens != analogSensorsMap.end();
       l_IterAnaSens++) {
45   // write sensor value to interface structure
46   interfaceStruct.outputs[l_IterAnaSens->first].value = (double)AnalogSensor::
         AnalogSensor_getStatus(l_IterAnaSens->second);
47 }
```

Listing 3.13: updateSensors()

3.7 Conclusion

To get a clear view on the short and long term goals of SIL, several users of SIL have been interviewed (section 2.1). From those goals, requirements for the SILv2 simulation environment have been derived (section 3.1). After identifying those requirements, the old situation of the SIL simulation environment has been reviewed and several problems, regarding the requirements, have been identified in its structure (section 3.2). To meet the requirements, a new communication approach between the SIL core and the simulation modules as well as a corresponding interface has been designed, implemented and evaluated (chapter 3). After the redesign of the overall structure of SIL by developing a new communication approach between simulation modules and SIL core, SIL is well prepared for the future use of different plant model simulation modules. The evaluation (3.5) shows that SILv2 meets the qualities that have been the key drivers in the development (Modular, Generic, Extensible, Composable, Efficient) and the identified requirements for the evolution of SIL.

Chapter 4

The SILv2 Plant Modeling Framework

The previous chapter presented the design of SILv2, which is an evolution of SILv1 that changes the communication between the SIL core and the simulation modules to create a generic simulation environment to test ESW. This section presents the design of a plant modeling framework that can be used to create plant model simulation modules. Previous to this project, a literature study has been done, that analyzes and categorizes several plant modeling approaches in the context of SIL simulation [12]. The research question treated in this chapter is:

- *How to design a plant modeling framework for the creation of plant models for simulation in a Software-In-The-Loop simulation environment that is expressive enough to model a multitude of domain concepts of high performance printers, but also uses an adequate level of abstraction to make the approach usable for non domain experts?*

First an analysis is presented that identifies stakeholders and requirements. After that the design of the plant modeling framework is presented. After presenting the implementation, the framework is evaluated. Last, conclusions are drawn.

4.1 Requirement Analysis

In preparation for the design of the plant modeling framework, an analysis is done to get an overview of who is going to use the approach.

4.1.1 Stakeholders

There are several people that use SIL and the plant modeling framework. The following roles in a project have different usages and requirements for the plant modeling framework.

Software Engineer

The software engineer uses SIL on a regular basis to test the implemented ESW. The ESW can be tested directly by the software engineer after implementing a new part of ESW or after modification, providing immediate feedback.

Architect

Architects use SIL, for example, to analyze the impact of changes in the ESW or in the plant. To do this, an architect needs to create or change plant models, and change the simulation

setup to perform the experiments.

Maintainer

The software engineer that designs and implements a certain part of the ESW, is also responsible to create the corresponding plant model simulation module for this part of the ESW. This has been the case in SILv1. However, it is imaginable that a software engineer takes the role of a maintainer, who is responsible for several plant models in SILv2, because more and more plant model simulation modules are added and have to be maintained. This includes creating new plant model simulation modules as well as updating plant model simulation modules and the VitualSystem setup file, as the plant is changed throughout the development of a printer.

Integrator

The ESW is implemented in different modules. There are different parts of ESW for different functions of the printer. The integrator puts together all different parts of the ESW. The integrator uses SIL to test the interaction between the different parts of the ESW.

Test Engineer

The test engineer defines, performs and evaluates test cases for SIL to evaluate the ESW of a printer.

Domain Expert

In the context of the modeling framework, a domain expert is an engineer from another discipline (mechanics, electronics). The domain expert provides domain knowledge for the plant modeling process, if necessary.

4.1.2 Requirements

The global high level requirement for the new plant modeling framework is that it can be used to model the "remaining plant elements" that are necessary to properly evaluate the ESW's behavior. "Remaining plant elements" means, everything that is not covered by SheetLogic. The following requirements have been identified by interviewing stakeholders that are going to use the plant modeling framework.

R1, The plant modeling framework shall support short modeling times.

Stakeholder(s): Software Engineer, Architect, Maintainer, Integrator
Since the creation of plant model simulation modules is not a main activity of the day-to-day work of a software engineer, there is not much time available for creating plant model simulation modules. The time needed to create a plant model should be less than the time needed to create a plant model, for the same sub system of the plant, using the stub mechanism (section 2.3.2).

R2, It shall be possible for a non-domain expert to create plant model simulation modules.

Stakeholder(s): Software Engineer, Architect, Maintainer, Integrator
The plant model simulation modules are going to be created by software engineers. A software engineer has typically not much detailed domain knowledge of the domains involved in the plant development, like mechanics, electronics and chemistry, or the modeling of physical

systems. So, the modeling process should not require, for example, knowledge of mathematical representations of mechanical processes.

R3, The modeling framework shall provide capabilities to model a plant in enough detail to properly evaluate the ESW.

Stakeholder(s): Software Engineer, Architect, Maintainer, Integrator
The main goal of the plant models, created with the modeling framework, is the evaluation of the ESW. Requirement two states that no detailed domain knowledge of domains like mechanics, electronics and chemistry, or the modeling of physical systems, should be required in the modeling process. However, the level of detail should be high enough to be able to properly test the ESW.

R4, The modeling framework shall be well maintainable.

Stakeholder(s): Maintainer
The maintenance of the plant modeling framework is also done by software engineers next to their day-to-day work. So, not much time is available and therefore the maintenance effort has to be kept as low as possible.

R5, The models created with the modeling framework shall be well maintainable.

Stakeholder(s): Maintainer
Throughout the development process of a printer, the plant is changed multiple times. It is necessary that the models, created with the SIL plant modeling framework are easy to update and to modify.

4.2 Design Decisions

4.2.1 Level of Abstraction

There are many approaches for modeling physical systems as shown in [12]. In the literature study, approaches are categorized using a novel taxonomy that divides the presented approaches in continuous, discrete and hybrid approaches, on a high level. Continuous modeling is used to describe systems in terms of differential equations. These equations are used to examine the evolution of physically significant variables, for example, velocity or torque, over time. This approach has its origins in classical mechanics. Example approaches are: Modelica, Matlab Simulink and Bond graphs. Discrete modeling approaches model systems using discrete events and state transitions. The underlying theory is automata theory. In these approaches, a system is always in a certain state. If a certain input is received, the state of the system can change. If the state is changed, actions can be performed like the generation of outputs. An example approach is the modeling with state charts. Hybrid system modeling is, as the name suggests, a mix of continuous and discrete modeling. The two possible approaches for hybrid modeling are to include continuous models in the states of discrete models or to switch the behavior of a continuous equation system by using a discrete model [12].

When testing ESW it can be enough to have a plant model with a high level of abstraction, because the ESW is most of the time only interested in high level information. An example is the movement of the sheet in the paper path of a printer. The ESW starts a motor and expects a sensor to sense a sheet after a defined time. In this case it is sufficient to move the sheet according to the speed of the motor, without taking the loss of energy through friction in the

motor and in the gearbox, or similar phenomena, in account. So a detailed model is not needed in this case.

However, some processes that are controlled by the ESW are more complex and can require a precise and physically correct simulation. An example is the pre-heating of paper when it is printed in a laser printer. This process depends on many different physical variables that are controlled by the ESW. It is therefore necessary to model this process precisely to properly evaluate the behavior of the ESW.

Generally, continuous, discrete and hybrid modeling can be used to model a physical system on a high or low level of abstraction. However, continuous modeling is a very detailed and physically correct approach to model physical systems, and is therefore very suitable for modeling the detailed behavior of mechatronic systems. Discrete modeling is more suitable to model a system on a high level of abstraction.

Options

1. Use a *low level of abstraction* to model the plant with high detail.

   Advantages: Physically precise models can be created. It is possible to model everything in the plant.

   Disadvantages: Creation and execution of models takes long. Detailed domain knowledge is needed.

2. Use a *High level of abstraction* to model the plant with lower detail.

   Advantages: Creation of models requires little time. Approaches with a high level of abstraction are easier to use, also for non domain experts because no, or less detailed, domain knowledge and knowledge of physical system modeling, is needed.

   Disadvantages: A high level approach is typically not as expressive as a low level approach. The created models are typically less precise. So, it is possible that necessary detail is lost.

Decision

Use both; an approach with a low-level of abstraction and an approach with a high-level of abstraction, according to the needed level of detail in different situations.

Rationale

The use of a high level and a low level approach makes the plant modeling framework more flexible. Since in SILv2 it is possible to compose multiple plant model simulation modules with multiple ESW simulation modules, it is possible to use different plant modeling approaches for different plant model simulation modules, in different situations, to adapt to the needed level of abstraction. The remaining design decisions are made separately for the high-level and the low-level approach.

4.2.2 (C)OTS vs. Custom-Made

Another decision has to be made about if a (commercial) of the shelf ((C)OTS) or a custom-made approach should be used. This decision has to be made for the low-level and the high-level

approach.

Options

1. Use a (C)OTS.

   Advantages: No time for development is needed, and there are no development costs. Support from the vendor is available. Tool is updated by the vendor.

   Disadvantages: The tool possibly has high License cost. A COTS tool is likely to be less flexible.

2. Use a Custom-Made.

   Advantages/Disadvantages are the exact opposite of using a (C)OTS tool.

Decision

Use a COTS for the plant modeling approach with a low level of abstraction and a custom-made approach for the plant modeling approach with a high-level of abstraction.

Rationale

For the low-level approach the decision is made to use a COTS approach because many suitable approaches are available and it would take a long time to develop a custom one. The software that is going to be used is Matlab Simulink. This tool is used in the whole company and so there is no disadvantage of high license cost because floating licenses are available. Several models of printer plant systems are already available that are used to evaluate the plant behavior from a mechanical point of view (from mechanical engineers). These could be reused as plant model simulation modules for SIL simulation.

For the high-level plant modeling approach, the decision has been made to develop a custom-made plant modeling environment. The reason for this is that the high-level approach should be a modeling environment suitable for the printer domain. This approach can be used to create high-level models as a combination of general printer plant concepts, like moving elements and the transport of toner or ink through the printer. The decision for the high-level plant modeling approach is to create a domain specific modeling language (DSML). This is done because a DSML can be created specifically for the domain of printers and provides the desired functionality of combining high level concepts.

4.2.3 Modeling by Hand and Generation of Plant Models

As mentioned in the context of this project (section 2.3.2), information about the paper path structure is taken from MoBasE to automatically generate an instance of SheetLogic of the current printer. So, the SheetLogic can be seen as a meta model of a paper path, holding the basic concepts, which is used together with information from MoBasE to instantiate a paper path. This leads obviously to short or no modeling time, if the information is available in MoBasE. MoBasE is used to store and exchange information about the printer between disciplines. So, information is added and kept up to date. This information can be reused in SIL for plant modeling. A similar approach is imaginable for the DSML. However, in the

current situation, MoBasE is not prepared to hold all the information needed to generate plant models for other printer parts than the paper path. So the structure of MoBasE would have to be modified.

Options

1. Generate high level of abstraction plant models from MoBasE.

   Advantages: Little or no modeling time is needed, if information is available from MoBasE.

   Disadvantages: MoBasE has to be modified to store the new information. There is a possibility that errors are introduced in the process of generating models, so correctness has to be checked. Only the current state of the printer and no alternative versions are stored in MoBasE.

2. Create high level of abstraction plant models by hand.

   Advantages: Creating models by hand is more flexible than the creation from MoBasE because alternative versions of the printer can be modeled.

   Disadvantages: More time is needed to create models. A suitable editor has to be developed to support the modeling process.

Decision

Once more, the decision is to use both options; to generate high level of abstraction plant models from MoBasE and to provide a modeling environment to create plant models by hand.

Rationale

Since in the time this project is carried out, another project has been done to extend MoBasE, it has been possible to add information, needed for generating plant models, to MoBasE. However, there are cases in which the creation of models by hand is desirable. An imaginable case is that an architect wants to analyze the impact of a change in the plant on the ESW. Since only the current state of the printer development is stored in MoBasE, alternative plant model simulation modules have to be created by hand. Figure 4.1 shows a schema, explaining the work-flow of creating plant model simulation modules with the DSML.

4.2.4 Implementation Environment/Tools for Implementing the DSML Tools

There are two choices to make for the implementation of the DSML tools. The DSML tooling consists of two main parts, the front-end and the back-end. The front-end provides functionality to create plant models. The back-end provides functionality to generate plant model simulation modules (DLL that can be loaded by SIL) from the plant models created in the front-end or from MoBasE. So, one decision has to be made about the implementation of the back-end and another about the implementation of the editor. The structure of the tools of EZ-VP can be found in figure 4.21.

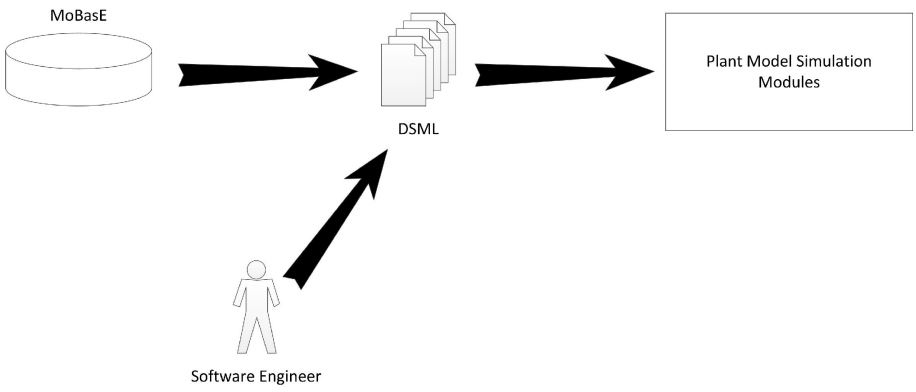

Figure 4.1: Creation of plant model simulation modules with the DSML.

Options, implementation environment/tools for the Back-end

1. $C++$ using RoseRT.

   Advantages: It is widely used within the company. It can easily compile into a DLL. There is the possibility to reuse from SheetLogic (also implemented in RoseRT).

   Disadvantages: There is no dedicated support for DSML development.

2. Eclipse Modeling Framework. The Eclipse modeling framework (EMF) [4] is a tool suite for the Eclipse IDE that supports DS(M)L development.

   Advantages: It supports DSML development. DLLs can be generated but not as easy as with RoseRT. It is free.

   Disadvantages: Eclipse is used very little within the company, so there is no experience.

3. Another general purpose language like *Java* or *C#*.

   Advantages: Free choice.

   Disadvantages: Not all languages prove support to create DLLs. There is no dedicated support for DSML development.

Options, implementation environment/tools for the Editor (Front-end)

1. $C++$ using RoseRT.

   Advantages: It is widely used within the company.

   Disadvantages: There is no support for graphical user interfaces.

2. Eclipse Modeling Framework.

Advantages: Provides dedicated support for the implementation of DSML editors.

Disadvantages: Eclipse is not used within the company, so there is no experience.

3. Another general purpose language like *Java* or *C#*.

   Advantages: Many languages support easy graphical user interface creation.

   Disadvantages: No dedicated support for DSML development available.

Decision

Use RoseRT for the implementation of the back-end and the EMF for the implementation of the editor.

Rationale

Since the back-end implements the plant behavior, which is a big part of the implementation, and it is possible to reuse from SheetLogic, RoseRT has been chosen for the implementation. Another important advantage is that nearly all users of the DSML have experience with RoseRT, so that maintenance is easier.

The EMF offers sophisticated functionality for the implementation of DS(M)L editors. When a meta model is created as a language definition, an editor for the corresponding DSML can be generated. So, it is possible, also for users with little experience in Eclipse, to update the DSML editor during maintenance.

4.2.5 Including the SIL Interface

Plant model simulation modules created with Matlab Simulink and the DSML have to comply to the SILv2 interface. Since the SIL simulation specific I/O-layer of the ESW simulation modules, as described in section 2.3.1, could be reused, this possibility has been contemplated for the DSML and the Matlab Simulink approach.

DSML Approach

Options

1. Use the ESW simulation module I/O-layer.

   Advantages: No development time is needed. Can be included without great effort, because the approach is implemented in RoseRT (as the ESW). Only one I/O-layer has to be maintained.

   Disadvantages: The solution is less flexible, if additional functionality is needed in the plant modeling approach.

2. Use a custom made I/O-layer for the approach.

Advantages: The solution is more flexible, if additional functionality is needed.

Disadvantages: Has to be maintained separately. Takes time to develop.

Matlab Simulink Approach

Options

1. Use the ESW simulation module I/O-layer.

   Advantages: No development time is needed. Only one I/O-layer has to be maintained.

   Disadvantages: It is possibly hard to integrate in the code generation process of Matlab Simulink because it was developed for RoseRT. Less flexible, if additional functionality is needed in the plant modeling approach.

2. Use a custom made I/O-layer for the approach.

   Advantages/Disadvantages: Same as for DSML approach.

Decision

Use the ESW simulation module I/O-layer for the DSML and a custom one for Matlab Simulink.

Rationale

Since the DSML back-end is developed entirely in RoseRT and the ESW simulation module I/O-layer can be included with very low effort, it is advantageous to reuse this I/O-layer. If flexibility is needed, it is possible to add generic functions to the I/O-layer to add and use output and/or input variable next to the I/O-device specific functions.

For the Matlab Simulink approach, the choice is to implement a custom I/O-layer. This is done mostly because the ESW I/O-layer can not be included in the Matlab Simulink approach easily. Since I/O-devices are not modeled explicitly, the target macros are not used and the use of the ESW simulation module I/O-layer would not offer advantages.

4.3 The Custom-Made High-Level Plant DSML Approach EZ-VirtualPrinter(EZ-VP)

This section presents the design of the high-level domain specific modeling language called EZ-VirtualPrinter (EZ-VP spoken as, easy virtual printer). The main purpose of DSMLs in general is to make the creation of models easier for people from the domain by using domain specific names and concepts. When designing a DSML, the specific domain has to be analyzed to identify common domain concepts and constructs and the connection between them. These concepts are used to create a meta model that serves as a language definition. An editor has to be provided that supports the modeling process in the corresponding DSML. The models are parsed by some kind of compiler or interpreter, which transforms the model into a form that can be executed or in some other way used for calculation. In a DS(M)L the model is often transformed to a general purpose language in this step.

EZ-VP is used to create high level plant model simulation modules for the evaluation of ESW in SIL. It provides the modeler with generic high-level concepts from the domain of high-performance printers. Those concepts can be combined, without the need that the modeler known the underlying behavior, which makes modeling easy and fast. EZ-VP has a front-end, which consists of a model editor that outputs a model in an intermediate format that is used by the back-end. The back-end uses the model in the intermediate format to create a simulation module for SIL. Intermediate models can not only be created using the EZ-VP editor but can also be taken from MoBasE. This structure is shown in figure 4.21.

The following sections present the development of EZ-VP, starting with a domain analysis to identify overlapping concepts of the different printer models. Selected concepts are abstracted to be more generic and therefore applicable in more situations. These concepts are subsequently implemented in EZ-VP.

4.3.1 Example Printer Models

To identify generic concepts that are used in many printers, several printer models have been inspected. The chosen models are of different types to have a broad view on the used hardware and concepts used in the plant. These four models have been inspected for the domain analysis:

1. A continuous feed ink-jet printer called Color Stream (left in fig. 4.2).
2. A wide format printer that jets melted toner balls called Color Wave (right in fig. 4.2).
3. A cut-sheet laser printer called VarioPrint 6250 (figure 1.1 in the introduction).
4. A display graphics ink-jet printer called Arizona (beneath in fig. 4.2).

Those printers represent the different kinds of printers very well.

4.3.2 Identification of Overlapping Concepts

Even though the inspected printers partially use completely different technology, some global similarities can be identified. This is done by reviewing the different technologies and by interviewing experts.

Print Process

All printers use some material, like toner or ink, to form a picture on some kind of substrate (material the printer prints on). The Color Stream, Color Wave and the Arizona use print heads to jet a liquid on the substrate, whereas the VarioPrint uses another technique with which the image is applied from a drum to the substrate using heat and pressure.

Figure 4.2: Three of the four printers inspected for the domain analysis.

When using print heads to jet liquids on the substrate, a plate of microscopic nozzles is used to control where ink/toner is applied and where not, thereby forming the image on the substrate. In the VarioPrint a photosensitive drum is used that first has a negative electrostatic charge. Then, light is used to form the image on the drum. This is done by exposing the parts of the drum to light where toner should stick, eliminating the negative charge. Then, toner is applied to the drum that will stick in the places of the drum that are not negatively charged. The created toner image is transferred to the substrate, using pressure and heat.

In those processes, the ESW controls the creation of the image that is transferred and the conditions in the printer, so that the image can be transferred. These conditions comprise, for example, the position of moving parts of the printer, if enough material is available and if the right temperature is reached. The print process for the different printers, using print heads, is similar. This is also the case for the different laser printers.

Temperature/Humidity Conditioning

In different processes in the printers, the control of temperature and humidity is needed. An example is the print process of the VarioPrint in which the substrate and the drums are heated to transfer the image. Another example is the heater used to melt the toner balls in the print head of the Color Wave before jetting the melted toner. Some printers, like the Color Stream, have humidity and temperature control in their cabinets to ensure good external conditions while printing.

Movement

All inspected printers have moving parts for different purposes. Movements can be separated in rotational movements, like the movement of the drums in the VarioPrint, and linear movements, like the movements of the carriage of the Arizona and the Color Wave.

Transport of Ink and Toner through the Printer

Independent of the used technology (toner, toner balls or Ink), it is always necessary to transport the material from some stock to the print process. Dependent on the used technology, this process can be more or less complex. For example in the Color Wave, the transport is limited to letting toner balls drop in the print head, if the temporary reservoir in the print head is almost empty. In ink jet printers the transport tends to be more complex due to the use of multiple tanks and loops in the system for ink conditioning. In addition to a pump, pressure in the tank can be used to initiate flow.

4.3.3 Controlling the Plant

The control of the concepts has the form of a feedback loop from one or more actuators, via several hardware parts of the concept, to one or more sensors, as shown in figure 4.3. To identify important parts of the concept, the causality of phenomena from the actuators to the sensors has been reviewed. As a result, a concept that is used throughout the printer and all

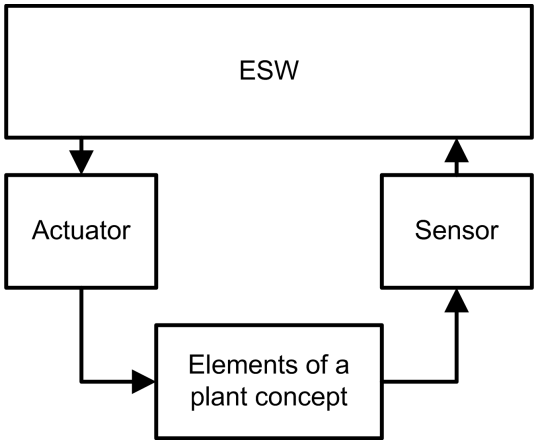

Figure 4.3: General form of a control loop in the plant.

earlier presented concepts, is the concept of I/O-devices. I/O-devices are used by the ESW to control the printer parts in the plant. They can be used to sense (sensor) or to cause (actuator) certain phenomena in the plant. So the I/O-devices provide an interface between ESW and the plant. This section shows how to abstract from multiple I/O-devices to a small number of I/O-devices by identifying elementary similarities and abstracting from hardware specific differences. The identification of different I/O-devices has been done by reviewing the SIL specific I/O-layer, which provides an interface for the ESW to all I/O-devices. The SIL specific I/O-layer abstracts several I/O-devices. An example is the use of analog sensors. An analog sensor senses an analog value in the plant and makes this value accessible to the ESW. Several different kinds of analog sensors are used that need different control from the ESW I/O-layer. This is, for example, caused by different electrical wiring etc. So, for each type of analog sensor, a different I/O-macro is used even if the underlying logic is the same. However, in the SIL

specific I/O-layer this is not necessary because the values of all analog sensors can be retrieved in the same way from the SIL interface structure. So the number of I/O-devices that need to be considered is much smaller in the SIL I/O-layer than in the real ESW hardware I/O-layer.

In addition to the above described differences in I/O-device control, there are several kinds of sensors and actuators that require certain special control from the point of view of the ESW. This is caused by the low level control that is discussed in section 3.4.2. An example for such a sensor is a "time capture sensor". The behavior of this sensor is implemented in the low level control (FPGA), which is still part of SheetLogic, since SILv2 is not yet completely implemented. This sensor can be used to capture the time a sensor is activated or deactivated very fine grained. This sensor is used because the ESW of a sub node is typically executed with 200Hz, which is not exact enough in some cases when measuring the time of certain events (the low level control is triggered more often). When the time capture sensor is used by the ESW, first the sensor is set to capture mode. This means that if a change in state happens, the precise time is stored in the low level control. Every following tick, after enabling the capture sensor, the sub node checks if a capture has been done. If a capture was done, the time value can be retrieved from the time capture sensor. This value can further be used by the ESW. So for the ESW this is a special kind of sensor, but from the point of view of the plant it is just a digital sensor, which is read by the low level control.

From the point of view of the plant model, the types of I/O-devices that have to be supported are the following, and every I/O-device, used by the ESW and the low level control, can be abstracted to one of the following sensors.

- Analog Sensor; measures an analog value in the plant.
- Analog Actuator; causes phenomena in the plant based on an analog value.
- Digital Sensor; measures a digital value (ON, OFF).
- Digital Actuator; causes phenomena in the plant based on a digital value(ON, OFF).

The following two sections present the two plant concepts, chosen to be included in EZ-VP. The identified I/O-devices are used to provide an interface between the ESW and the identified parts of the concepts.

4.3.4 Abstracting the Plant Concept of Moving Elements

For the initial version of EZ-VP the concepts of linear moving parts and the transport of ink/toner are further worked out, abstracted as far as possible and implemented. To do this, for each concept, important properties and characteristics are identified. The different involved I/O-devices and their relation to these two concepts are also investigated. For each concept, a part of the meta model for the language definition of EZ-VP, is created. This meta model is used to instantiate concrete examples of systems.

In printers, it is often the case that parts of the printer, like print head carriages, are moved. Those movement are controlled by the ESW, using some kind of actuator to initiate the movement, for example, a motor. The movement is observed using a sensor, for example, a digital sensor, sensing if the carriage is in home position. Figure 4.4 shows the movement of the carriage of the Arizona. In case of the ColorWave, the carriage only moves in X direction over the breadth of the printer.

Identifying Parts and Properties

To be able to model this concept, the important parts and their properties are identified. A movement is always made along some kind of guidance. The position of the moving element can always be seen relative to the guidance. It is also possible that multiple movements are

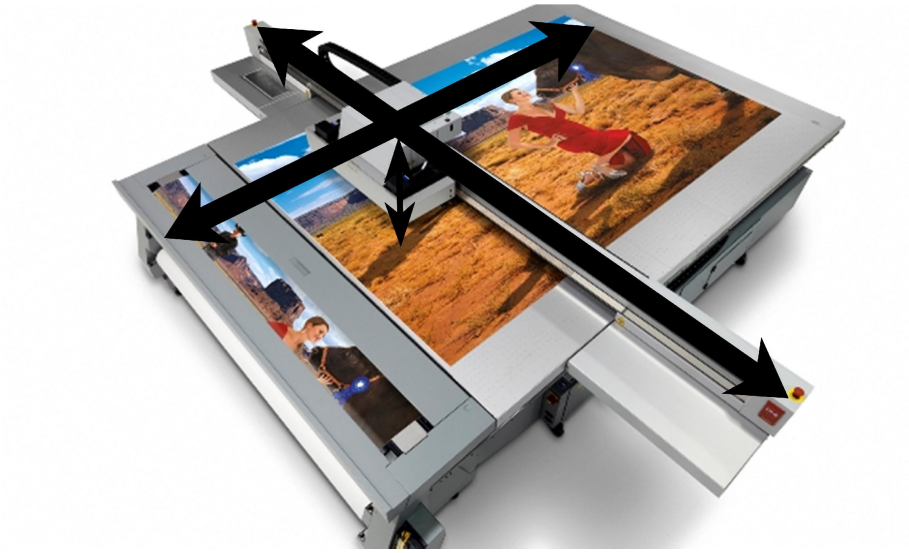

Figure 4.4: The carriage of an Arizona wide format printer.

combined by using multiple guidances, like in the case of the Arizona carriage (figure 4.4). In this case, the carriage can move in three dimensions (when looked at from above), along the X and Y axis to move the carriage over the printed material, and a small amount in Z direction to lift and lower the carriage. The involved I/O-devices are motors, encoders (analog sensors that count the rotations of the motor) and sensors to sense the position of the moving element relative to the guidance.

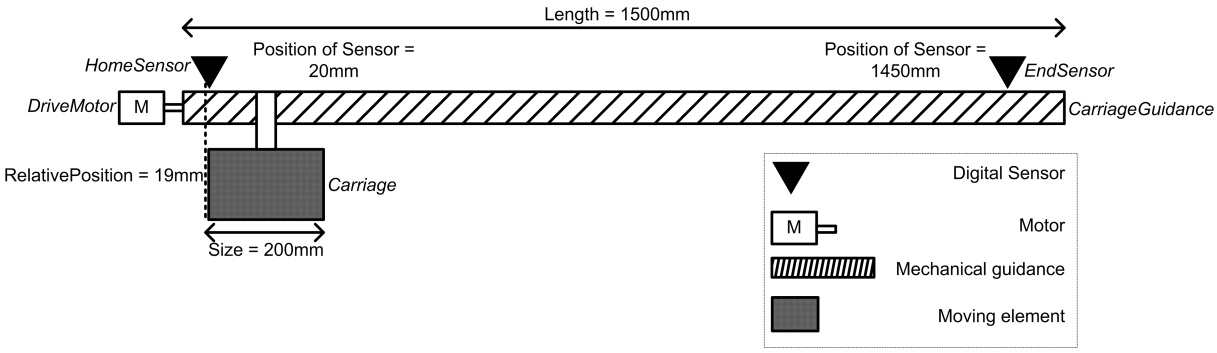

Figure 4.5: Simple example system.

Figure 4.5 shows an abstraction of the print head carriage of the ColorWave that moves along a mechanical guidance (values of the properties are randomly chosen, names of parts are shown in italic letters). The important entities here are:

- A motor, which drives the carriage.
- A mechanical guidance.
- The moving element itself.
- Two sensors on the mechanical guidance to sense the moving element.

For these elements the following properties are identified.

- The sensors have a position on the mechanical guidance.
- The moving element has a position relative to the mechanical guidance.
- The size of the moving element has to be known to determine when it is sensed by the sensors.
- There is a ratio that describes how much the moving element moves when the motor runs.
- The mechanical guidance has a length.

Those parts and properties where found in several of the example printers. However, there are some differences in the number of sensors that are used and the number of motors that drive the moving element.

The Meta Model of Moving Elements

Those parts and properties are captured in the part of the meta model, shown in figure 4.6. The figure shows the meta model implemented in the EMF. For each entity in the meta model

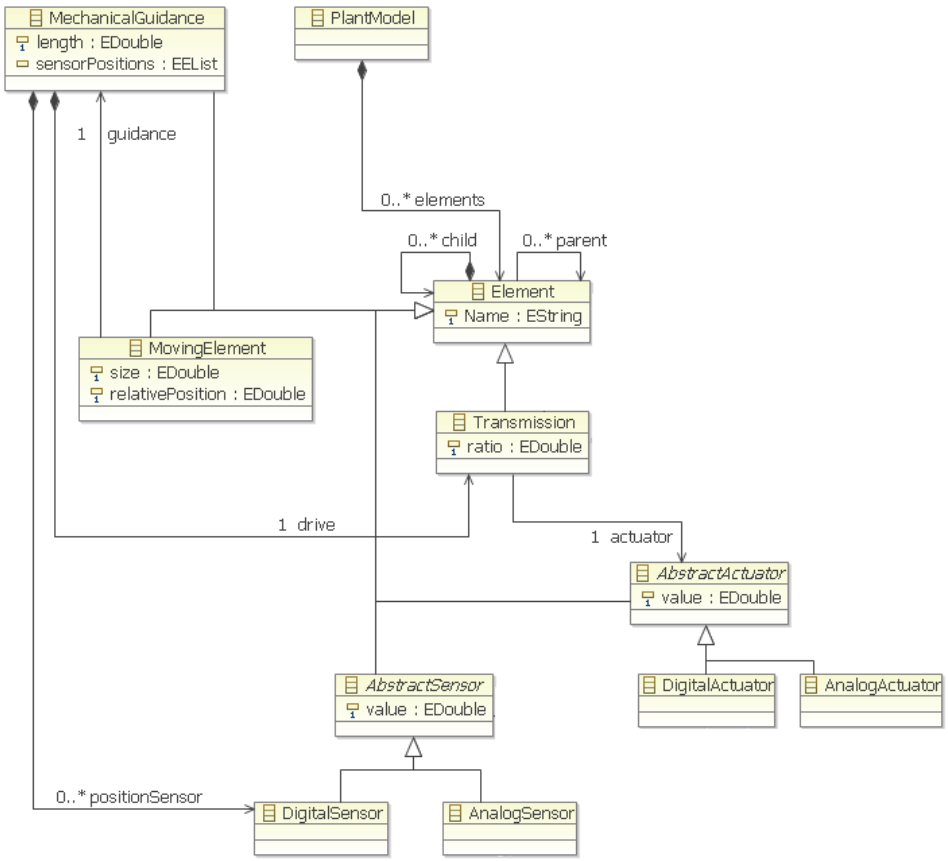

Figure 4.6: Part of the meta model capturing the concept of moving elements.

a detailed description is given. Additionally, examples of specific printer parts that can be abstracted to the corresponding meta model entity, are given.

- *MovingElement* is the central entity in the concept of moving elements. It is a generic concept describing a part or group of parts in the printer that is moving. The *MovingElement*

can be seen as a container for other plant elements that are moving. An example is the frame of a print carriage. To model a print carriage, a *MovingElement* is created and all elements that are part of the carriage in reality, can be added as children to it (child relation of *Element*). The *MovingElement* has two attributes, a *size* and a *relativePosition*. The attribute *size* holds the size of the surface of the *MovingElement* that can be sensed by the *Sensors* on the *MechanicalGuidance* as shown in figure 4.5. The attribute *relativePosition* holds the position of the *MovingElement*, relative to the *MechanicalGuidance* it is connected to.

- *MechanicalGuidance* is a concept that defines the movement of a moving element. Another possible name for this concept could be movement, because it abstracts from the mechanical domain to a generic description of a movement that is initiated by an *Actuator* and sensed by some *Sensors*. However, since all other entities in the meta model are physical parts and to keep the naming consistent, the concept is named *MechanicalGuidance*. The *Mechanical Guidance* is connected to an *Actuator* via a *Transmission*. The *MechanicalGuidance* has zero or more *Sensors* positioned on it, which are used to determine the position of the *MovingElement* as it moves along the guidance. The positions of the *Sensors* are stored in the attribute *sensorPositions*. The *length* of the *MechanicalGuidance* is stored in the attribute *length*. Examples of specific plant elements that are abstracted to a *MechanicalGuidance* are a spindle drive or a toothed belt. A spindle drive is basically a threaded rod driven by a motor on which a screw nut is placed. The part that should be moved is attached to the screw nut and can be moved by turning the threaded rod. Another specific version of a *MechanicalGuidance* is a toothed belt, which is driven on two sides by gear wheels. The *MovingElement* is fastened to the belt and can be moved by turning the gear wheels.
- The entity *Transmission* defines a *ratio* between the *Actuator* and the *MovingElement*. This *ratio* defines the distance the *MovingElement* moves when the motor makes one turn. So the ratio not only defines the ratio of a probably used gearbox, but abstracts from the whole connection from *Motor* to the *MovingElement*. So the *Transmission* comprises properties like a gearbox used with the motor and the thread pitch of a spindle drive and combines them to one ratio.
- *DigitalSensor*, *AnalogActuator* and *Actuator* are the used I/O-devices in the concept. They hold an attribute *value*, which represents an digital or analog *value* based on the type.
- In addition to the concept of moving elements, a superclass for all elements in the meta model is added. The *Element* is the central entity of the model. Every entity in the meta model is an *Element*. An *Element* can be connected to another *Element*, which creates a parent-child relation. In reality this relation means that the parent *Element* is somehow mechanically connected to the child *Element*, for example, bolted or screwed to the child *Element*. The *Element* has an attribute *name* so that every element in the meta model has the attribute *name*.

Example Instantiations

Figure 4.7 shows an instantiation of the system shown in figure 4.5. Figure 4.8 shows another example for a moving element. This system is an abstraction of a print head carriage similar to the one of the Arizona. In this system, the carriage can move in X and Y direction. Figure 4.9 shows the instantiation of the meta model for this system. The guidance for the movement in Y direction is connected to the moving element “attachmentOfYguidance”. This way the guidance for the movement in Y direction becomes the moving element that moves in X Direction. The

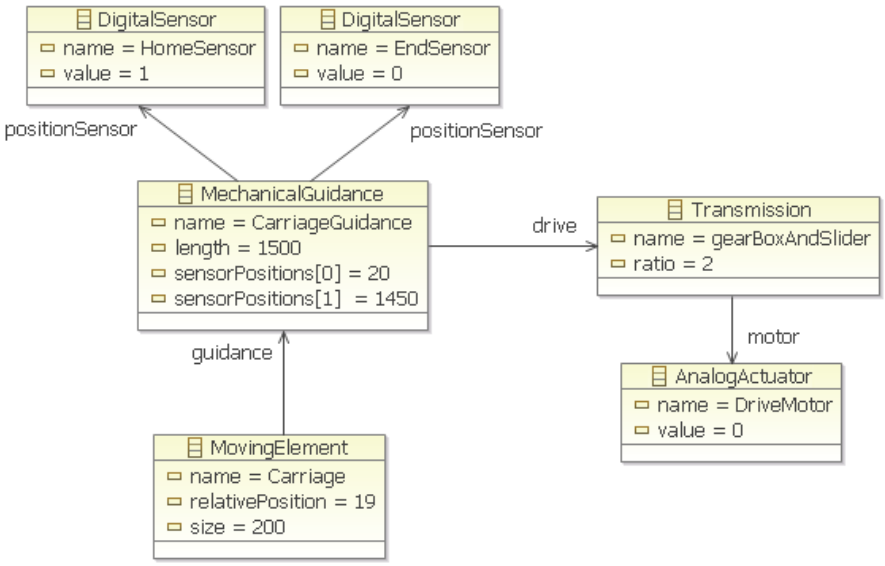

Figure 4.7: Instantiation of the meta model with the system of figure 4.5.

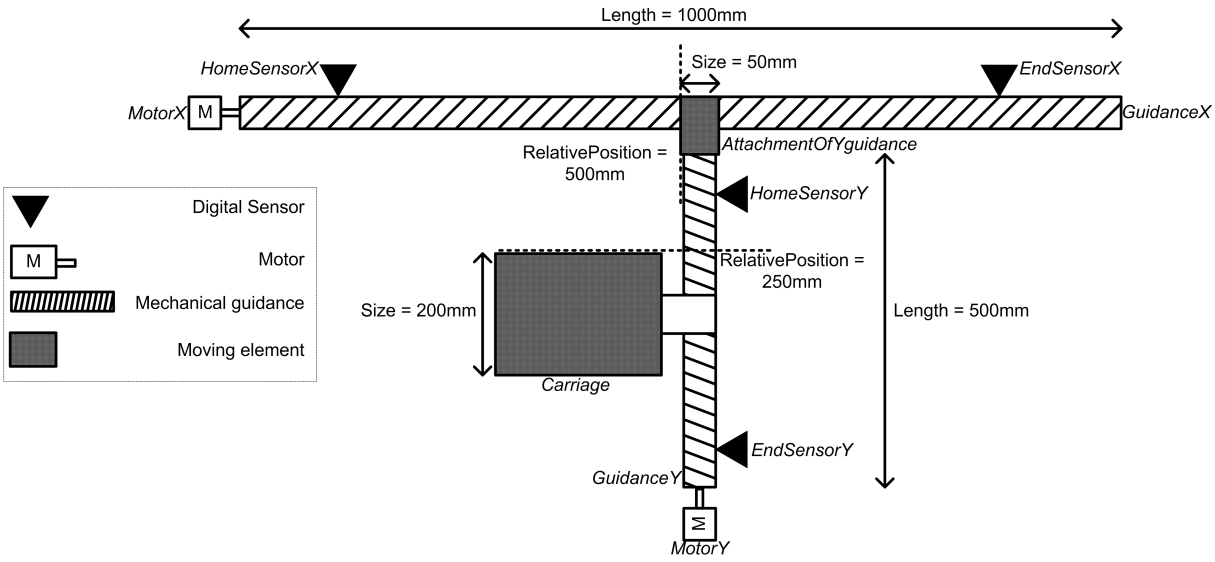

Figure 4.8: Another example system for the concept of moving elements.

carriage is the moving element attached to the guidance in Y direction. This way the carriage can move in X and Y direction. This way it is also possible to add yet another guidance for movement in Z direction. Then the carriage could move in three dimensions. This system is also used in a case study (section 4.6.1). In this case study a plant model simulation module based on this system is controlled by ESW.

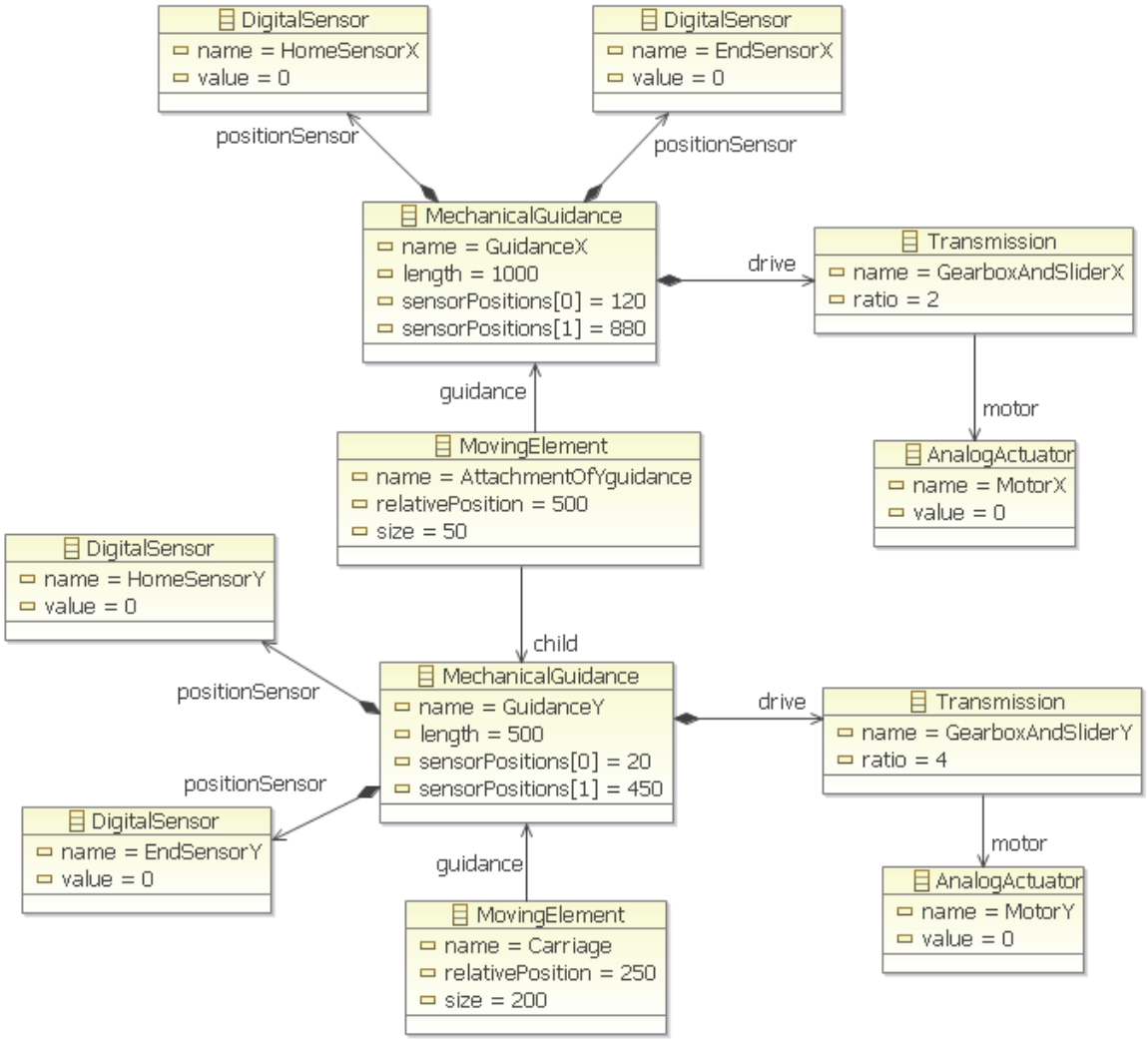

Figure 4.9: Instantiation of the meta model with the system of figure 4.8.

Behavior of the Plant Model

Figure 4.10 presents the behavior of the model when it gets a "tick" from SIL. (Numbers in the enumeration refer to the numbers in figure 4.10).

1. The position of the *MovingElement* is updated.
2. The *Sensors* of the *MechanicalGuidance* are updated based on the position of the *MovingElement*.

In this version, the main element of the concept that gets the first "tick" from the class *PlantModel*, is the *MovingElement*. Alternatives for this are that not the MovingElement but the motor gets a "tick" and the calculation is done in the sequence of the causality of the movement, or the other way around, from the sensor to the actuator. The reason why the version from figure 4.10 is chosen, is that it makes the code more readable and understandable, since the two core entities of the concept (MovingElement and MechanicalGuidance) initiate the calculation.

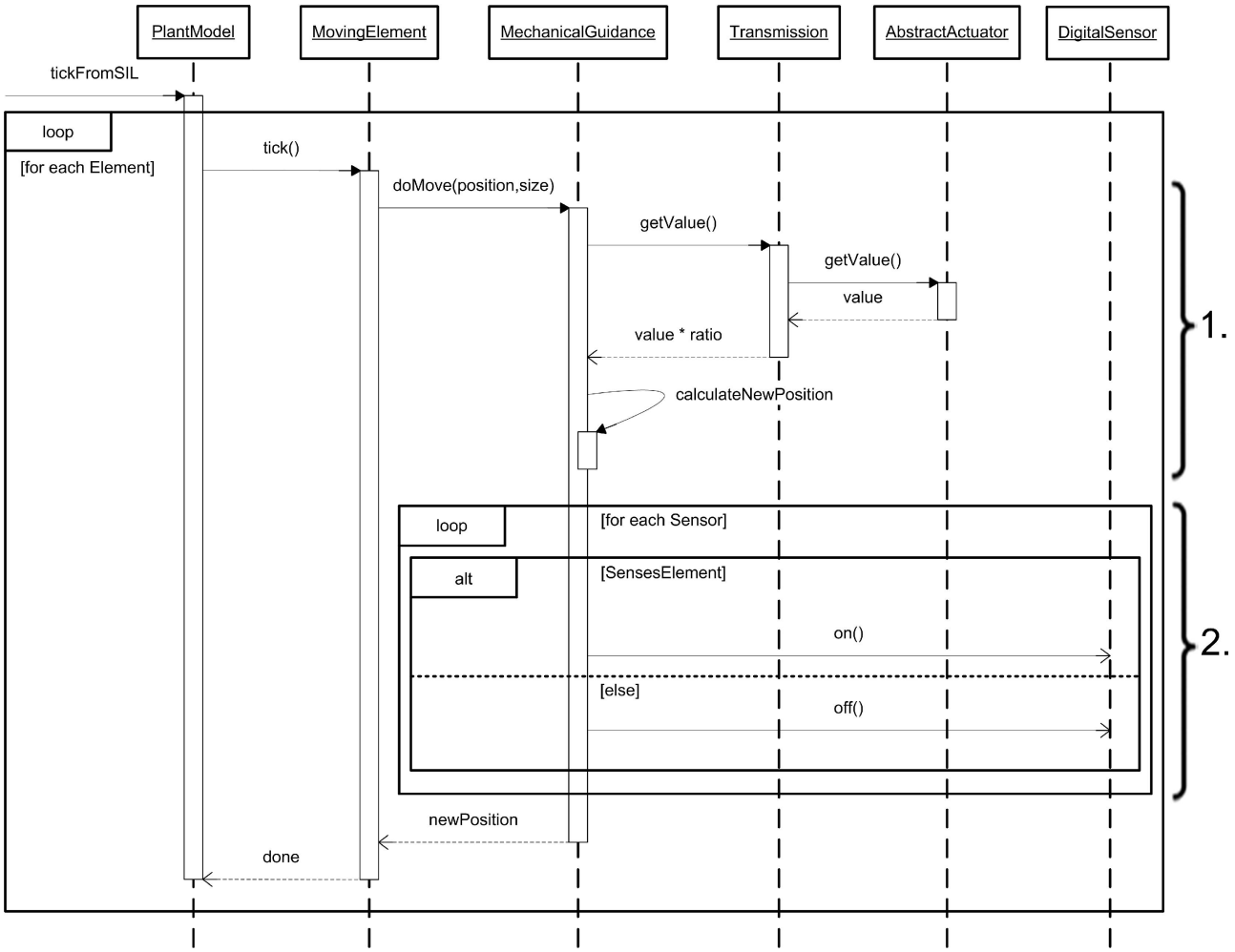

Figure 4.10: Behavior of the *MovingElement* in EZ-VP.

4.3.5 Abstracting the Plant Concept of Transport of Material through the Printer

As identified earlier, in all printers there is the need to transport ink or toner from a stock to the print process. Even though the transport of toner and ink is quite different in reality due to the fact that ink is a liquid and toner is a powder, some analogies can be found. The material is in some kind of container. From this container it is moved to another container based on some actuator controlled by the ESW. The amount in the containers is sensed by some kind of sensor. At a certain point the material leaves the system (print process).

Identifying Parts and Properties

The first system that is reviewed is an example ink handling system of an ink jet printer. This system is shown in figure 4.11. The system consists of two tanks. The left is the stock tank and the right is a buffer tank before the print head. Ink is transported from the first to the second tank with a pump. If there is pressure in the second tank and the valve in the connector to the way out of the system is opened and the system is printing (print head), the ink leaves the system.

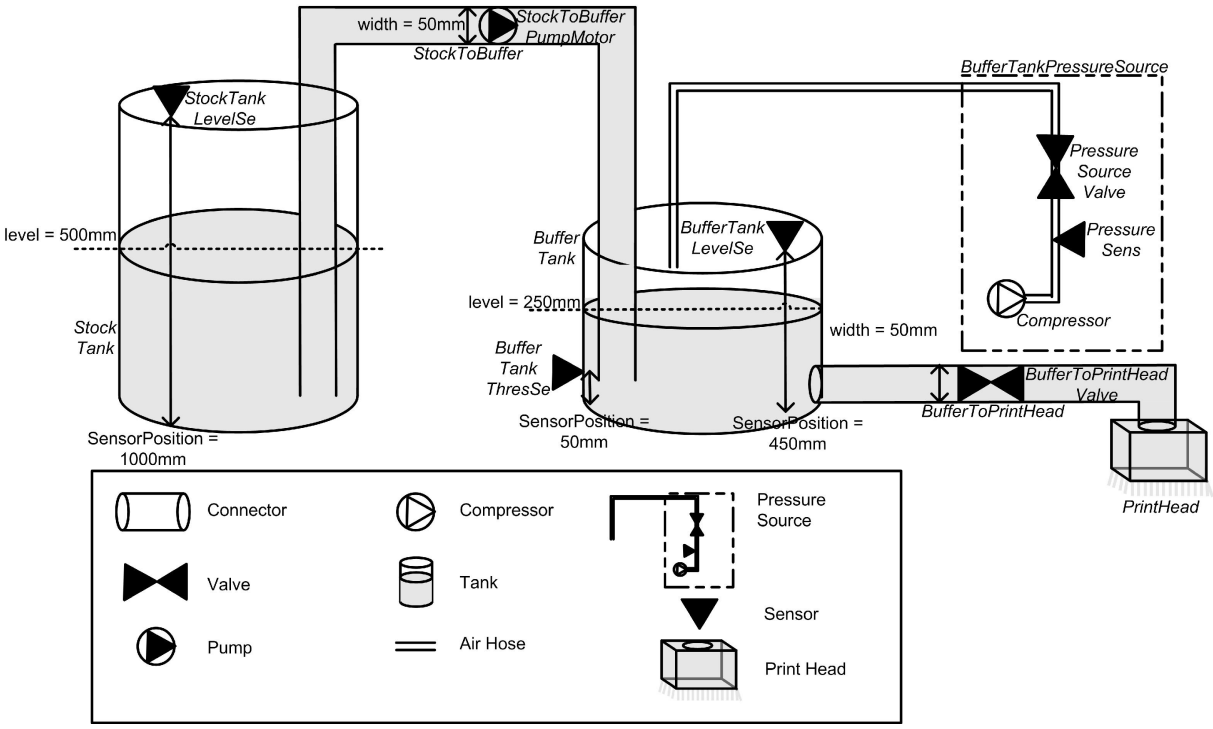

Figure 4.11: An example material transport system in a printer.

Again, the important parts of this concept are identified:

- A tank, which is the container holding the material (ink).
- A connector, which connects two tanks, enabling the transport of material between the tanks, or that connects a tank to the print process.
- In a connector there can be pumps and/or valves.
- A tank can have analog level sensors and/or digital threshold sensors.
- A tank can be connected to a pressure source, which adds pressure to the tank (negative or positive).
- A pressure source contains a compressor, can contain valves and/or pressure sensors.
- Material leaves the system at a certain point based on if the system is printing or not.

These elements have important properties:

- A tank has a volume, a current level of material and a current pressure.
- The level sensor in a tank have a certain height of placement (digital and analog sensors).
- A connector has a certain width, which is needed to determine how much material flows at which pressure.
- Flow between tanks is initiated by pumps or pressure in the tanks. A pump has a certain delivery rate. The delivery rate caused by pressure depends on the pressure in the tanks and the width of the connector.

The Meta Model of Material Transport

Figure 4.12 shows the part of the meta model capturing the concept of material transport. For each entity in the meta model a detailed description is given. Additionally, examples of specific printer parts, that can be abstracted to the corresponding meta model entity, are given.

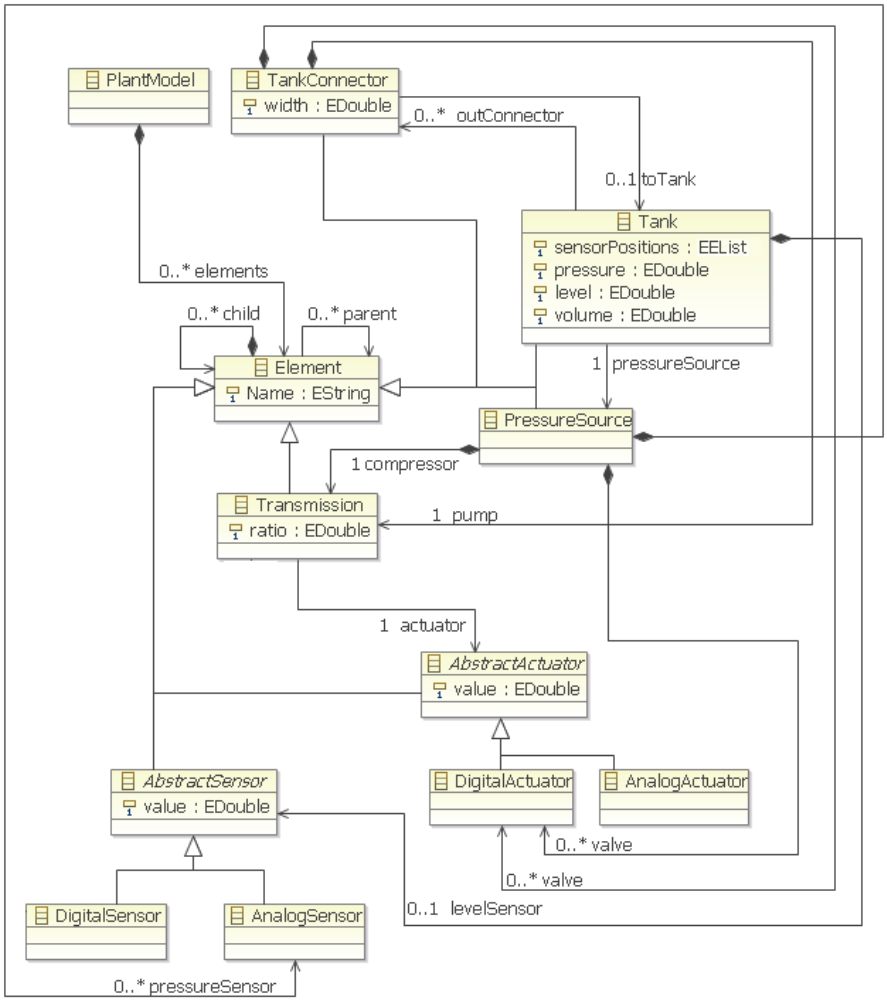

Figure 4.12: Meta model of the material transport concept.

- The core of the concept is represented by the entity *Tank*. The *Tank* is the container, which holds the material. A *Tank* has a *volume*, a current *level* of material and a current *pressure*. The *pressure* in the *Tank* is generated by a *PressureSource*. In a *Tank* there are *Sensors* that sense the current *level* of material in it. For these *Sensors*, the placement height is saved with the attribute *SensorPositions*. The used *Sensors* can be digital, which sense if the *level* of material exceeds a certain threshold, or *AnalogSensors*, which sense the current *level* of material. In reality, *Tanks* have different forms and different materials. It is assumed that the relation between *volume* and *level* of the material and *Tank* is linear. That means if the *Tank* holds 25% material of its *volume* the *level* is also at 25%. When 50% of the *volumen* is in the *Tank* the *level* is 50% etc. *Tanks* can have multiple in- and outgoing *TankConnectors*.
- The entity *TankConnector* represents a connection between *Tanks*, enabling transport of material. A *TankConnector* has a certain *width*, which is used to determine the flow of material, based on *pressure* in the *Tank*. A *TankConnector* can have valves, which are represented by *Actuator* and can close the *TankConnector* so that no material can flow. *TankConnectors* can also have pumps to initiate flow. *TankConnectors* connect

two *Tanks*, or a *Tank* and the print process. In reality *TankConnectors* can be elastic tubes or pipes.

- *PressureSources* are used to create a positive or negative *pressure* in the *Tanks*. A *PressureSource* has a *Compressor*, which is an *Actuator* that is connected to the *PressureSource* via a *Transmission*. A *PressureSource* can also have *Valves* that are represented by *Actuators*. *Sensors* can be added to the *PressureSource* to measure the generated *pressure*. In reality, a *PressureSource* is a *Compressor*, which is connected to a *Tank* with a hose.
- As in the concept of moving elements, in this concept the entity *Transmission* is used. The *Transmission* is used for the *Compressor* in the *PressureSource* and for the *pump* in the *Connector*. For the *pump* the *ratio* in the *Transmission* defines how much material is transported when the *Pump* is turned on (mm^3/ms) and for the *Compressor* how much *pressure* ($mBar/ms$) is added when the *Compressor* is turned on. So again the *Transmission* is an abstract concept to describe a *ratio* between an *Actuator* value and an effect in the plant.

The print head, respectively the print process is not included, since this can also be modeled as a valve in the connector to the exit of the system. The entities *Element* and *PlantModel* are the same as in the part of the meta model describing the moving elements concept.

Example Instantiations

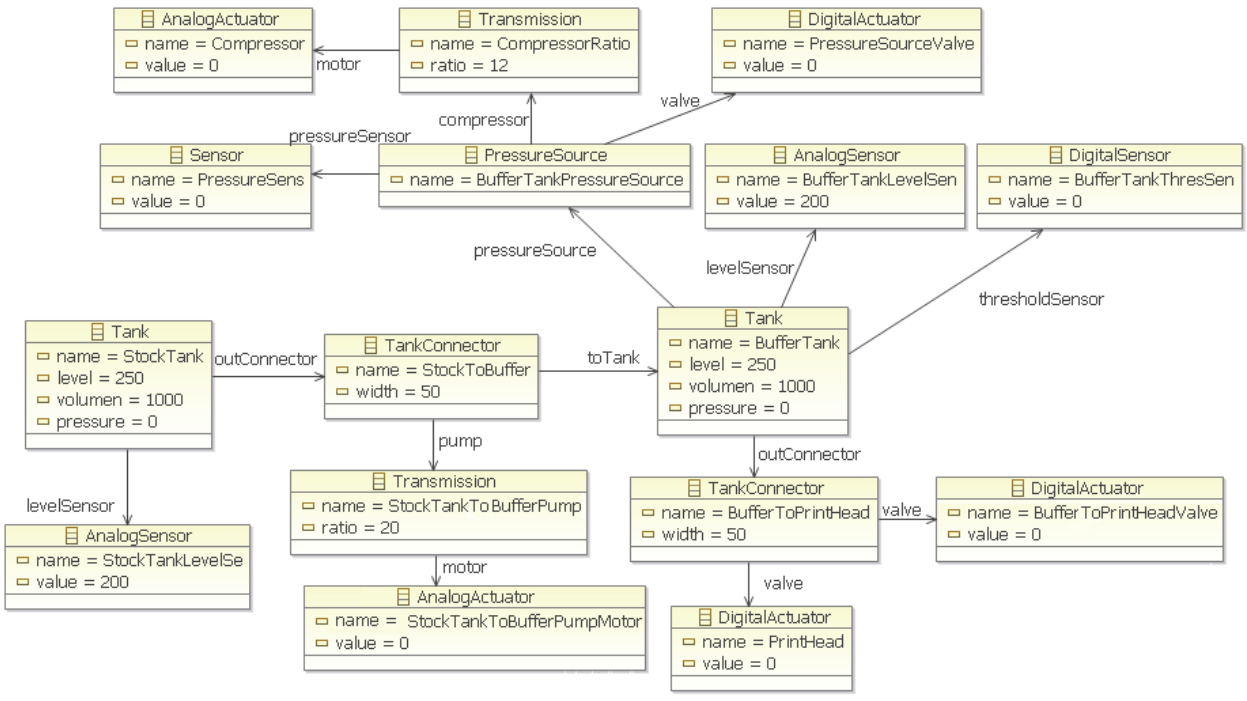

Figure 4.13: Instantiation of the example material transport system from figure 4.11.

Figure 4.13 shows the instantiation of the meta model for the example system from figure 4.11. Figure 4.14 shows another example system. This system transports, in contrast to the

first system, not ink but toner. It consists of two tanks, a storage and a buffer tank. Both tanks have level sensors. When the actuator between the two tanks is activated, toner falls from the storage to the buffer tank. From the buffer tank the toner leaves the system, if the actuator, representing the print process, is activated. Figure 4.15 shows the instantiation of the second example material transport system.

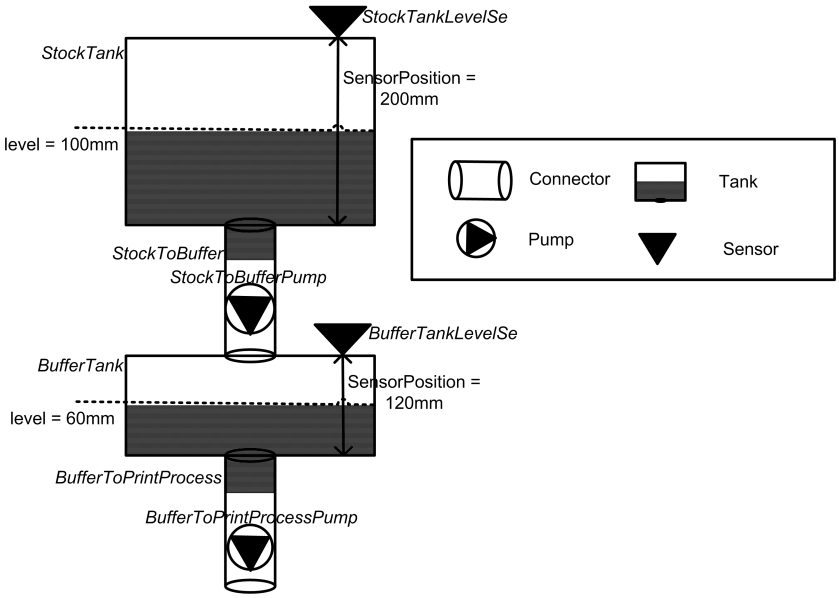

Figure 4.14: Example material transport system.

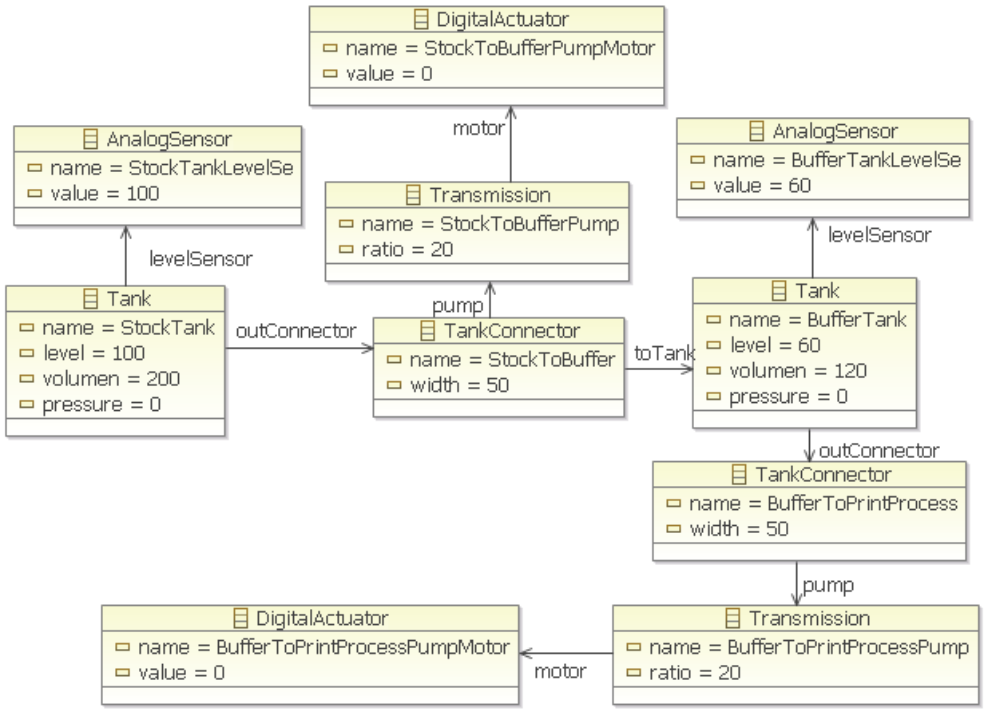

Figure 4.15: Instantiation of the example material transport system from figure 4.14.

Behavior of the Plant Model

The behavior of a plant model of the material transport for a computational step is as follows (Figure 4.16, 4.17, 4.18 and 4.19). The numbers on the right in the figures corresponds to the numbers of the items in the following enumerations.

1. The plant model module receives a “tick” from the SIL core.
2. The plant model forwards the “tick” to every tank in the plant model successively.
3. The tank requests the generated pressure from the pressure source, if a pressure source is installed. The value of the analog actuator (compressor) is requested by the pressure source.
4. The value of the pressure sensors of the pressure source are updated.
5. The state of the valves of the pressure source is checked. If one or more are closed, a pressure value of zero is returned. Otherwise the calculated pressure value is returned.

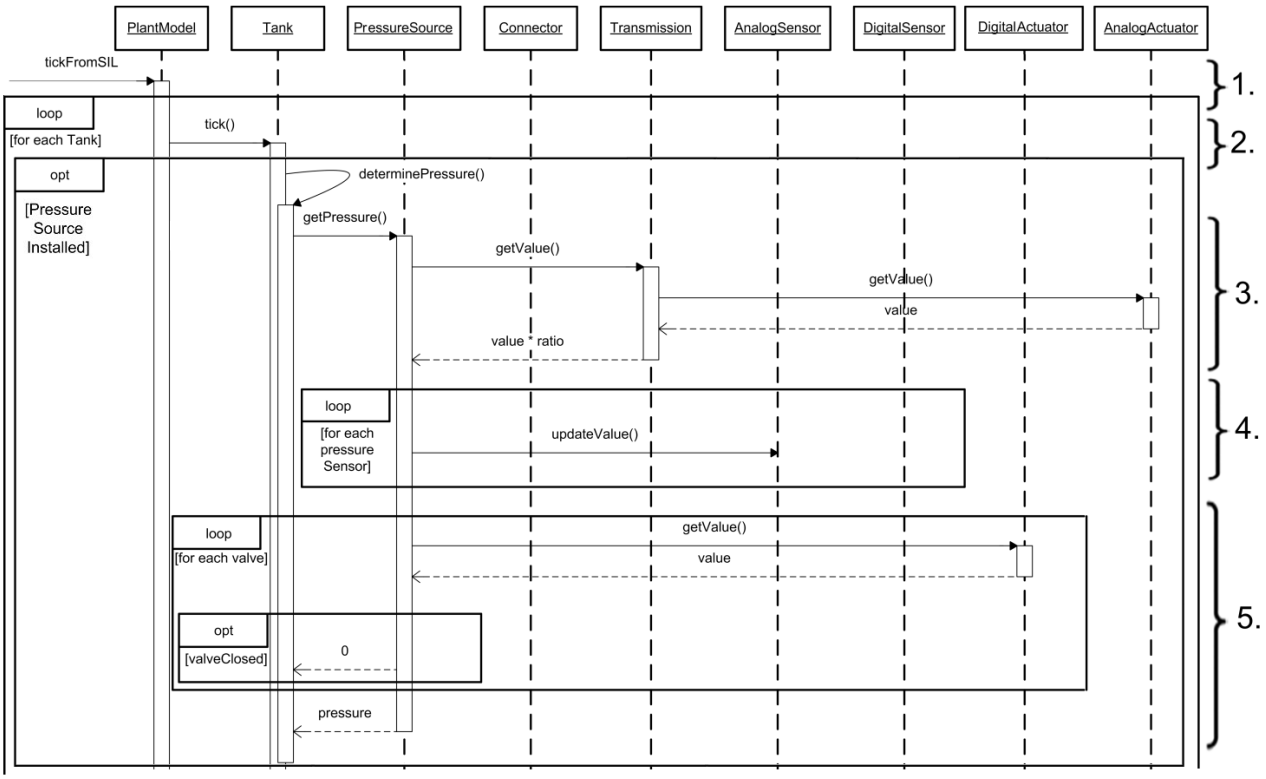

Figure 4.16: Material transport plant model behavior.

6. The transport of material is initiated for each outgoing connector. Material is only transported, if all valves of the connector are opened.
7. The amount of material that is transported is determined. If a pump is installed in the connector, it is determined by the delivery rate of the pump, otherwise by the pressure in the tanks it connects.

8. Material is transported based on the previously calculated amount.

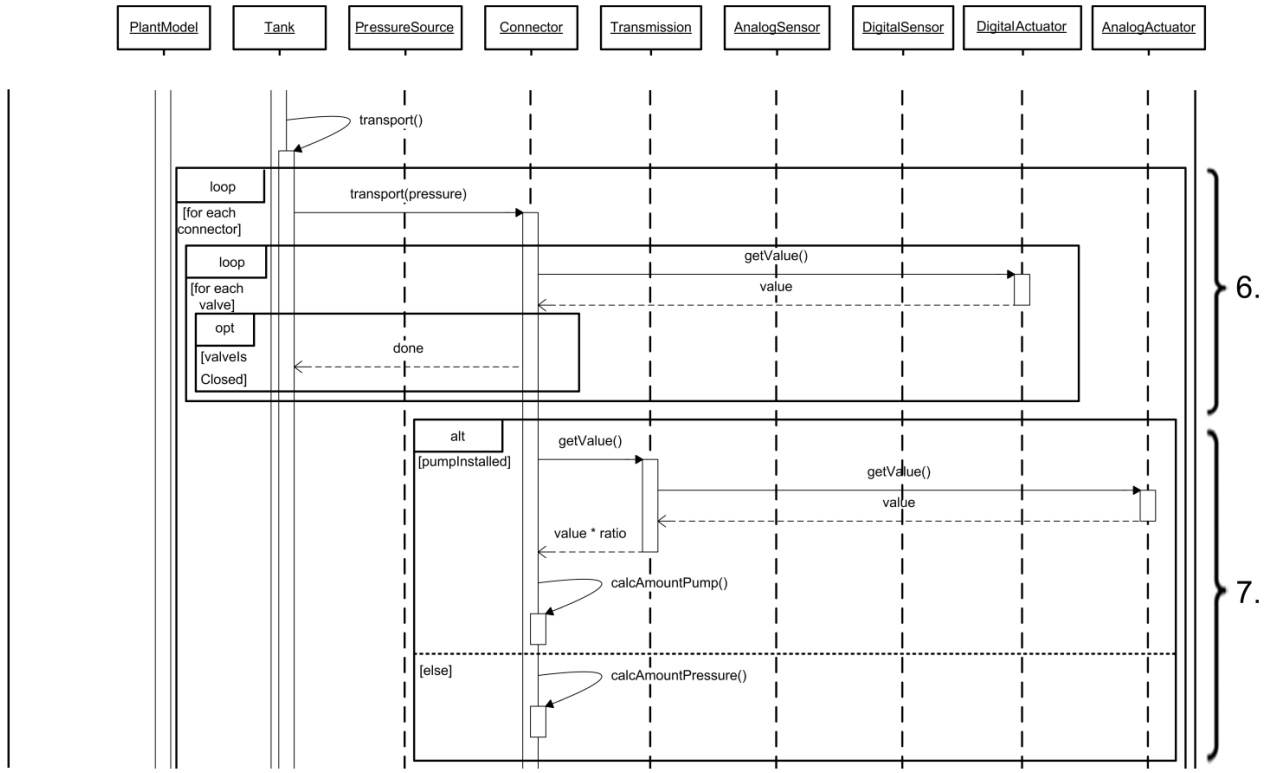

Figure 4.17: Material transport plant model behavior.

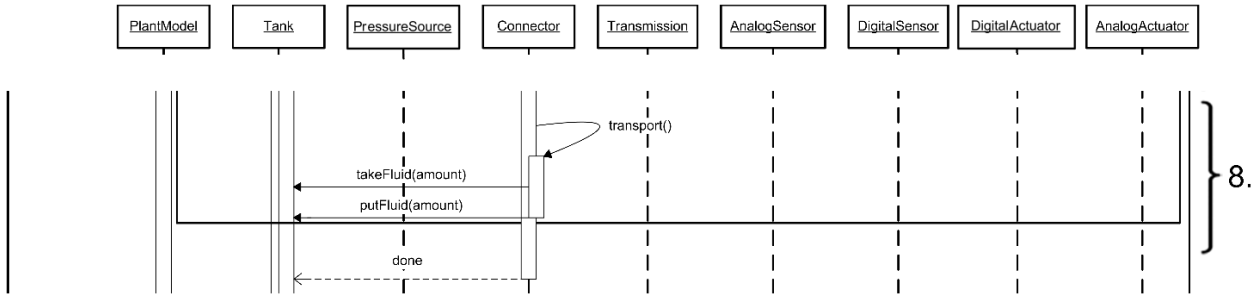

Figure 4.18: Material transport plant model behavior.

9. The digital level sensors of the tank are updated based on if the material level is higher than the position of the sensor.
10. The analog level sensors are updated based on the position of the sensor and the level of the material in the tank.

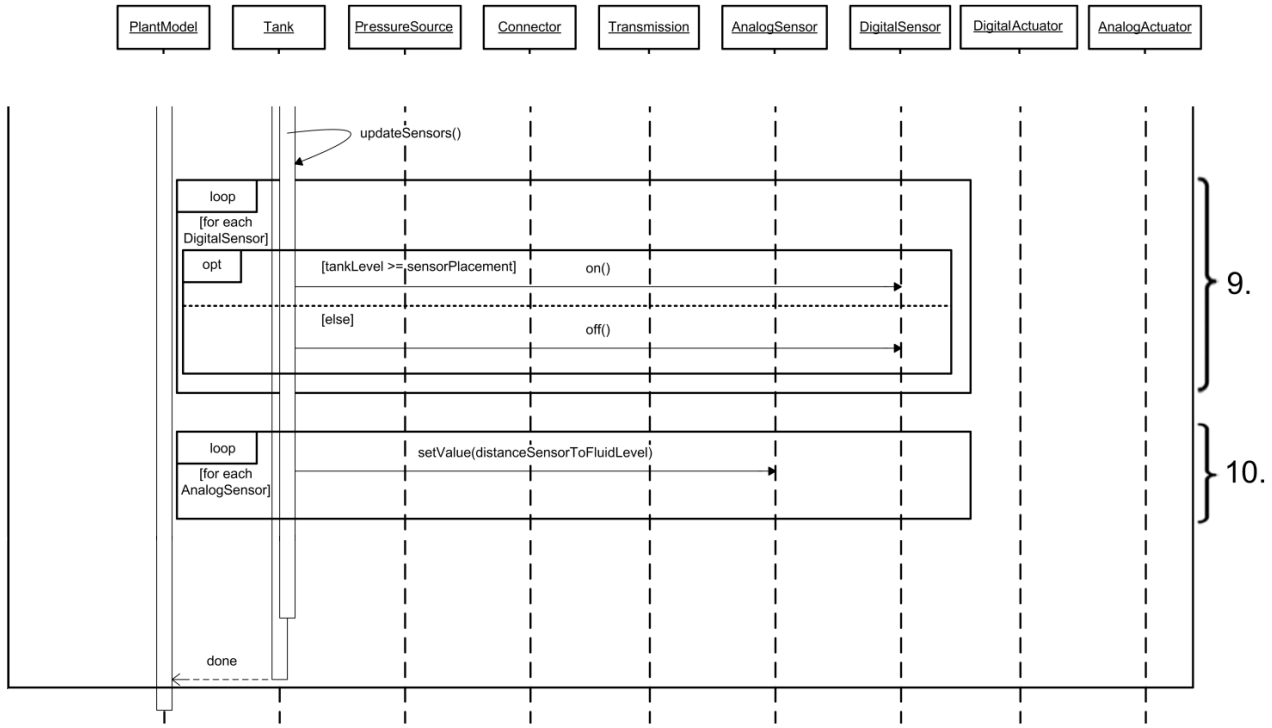

Figure 4.19: Material transport plant model behavior.

4.3.6 The Whole Meta Model

Figure 4.20 shows the whole meta model, comprising the two presented concepts. It can be seen that the I/O-devices, *Element*, *PlantModel* and the *Transmission* are used by both concepts. This meta model is further used as the language definition for EZ-VP.

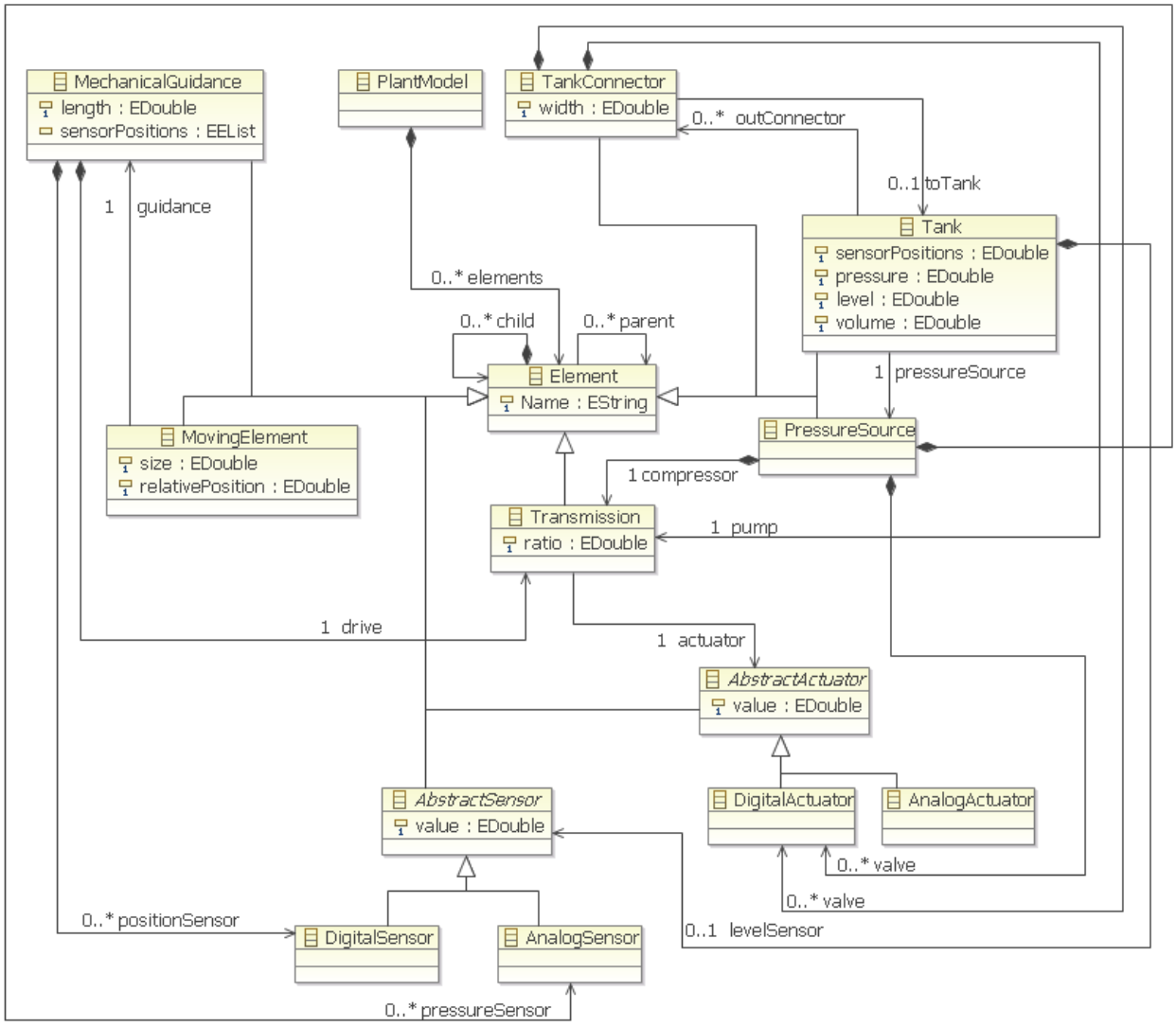

Figure 4.20: The whole meta model, comprising the concept of moving elements and the transport of material through the printer.

4.3.7 Tools of EZ-VP

EZ-VP is separated in different tools, which together are used to create plant model simulation modules. Figure 4.21 shows an overview of the EZ-VP framework and the plant model simulation module creation process. The front-end end of EZ-VP consists of two parts. A DSML editor, created with the EMF, and MoBasE. Both parts produce output that is used by the back-end to create SIL plant model simulation modules.

EZ-VirtualPrinter Edit

EZ-VirtualPrinter Edit is created as an Eclipse plug-in using the EMF [4]. The EMF offers functionality to create a meta model, which is used to automatically generate an Eclipse plug-in. This plug-in comprises an editor that can be used to combine and parameterize the elements specified in the meta model and therefore to create instantiations of it. Figure 4.22 shows

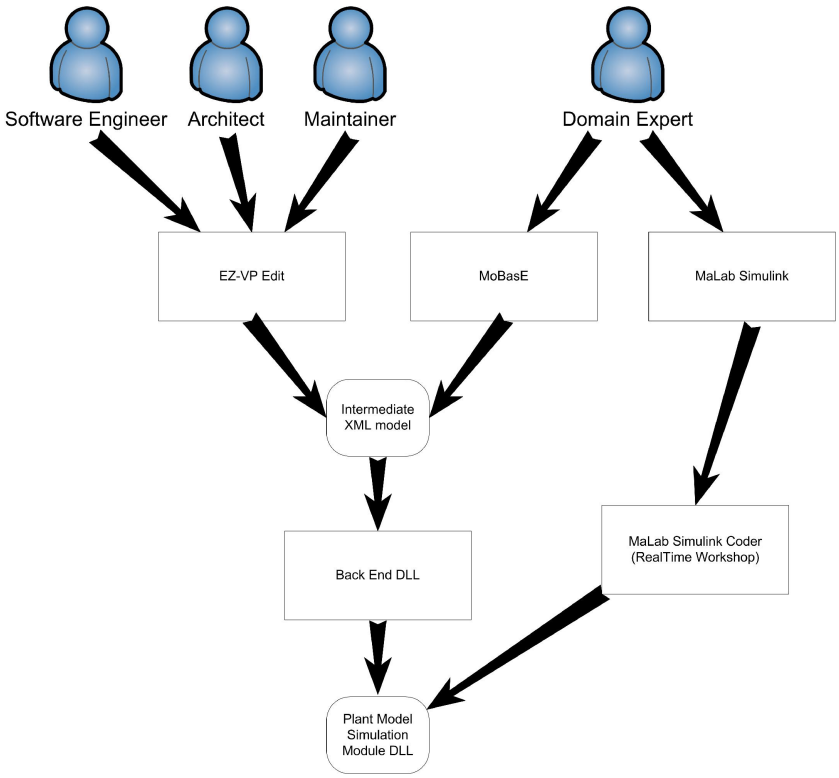

Figure 4.21: Creation of plant model simulation modules with EZ-VP and Matlab Simulink.

the generated editor. The left part of the editor shows the created elements. The shown model represents the model from figure 4.5 (The entity PlantModel and PrintModule are added as overall parent elements). The hierarchy of the elements represents containment relations between the elements. So, PlantModel contains the PrintModule, which is again the parent (Element parent/ child relation in meta model) of the Carriage and the CarriageGuidance. The CarriageGuidance contains the two sensors, the motor and the transmission. The right part of the editor shows the properties of the elements. In figure 4.22, the properties view shows the properties of the Carriage. The Carriage is parameterized according to the example system in figure 4.5. The model that is generated is stored in XML, which can be read in by the back-end to create a SIL simulation module. The XML output is shown in listing 4.1.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <metamodel:PlantModel..>
3    <elements Name="printModule">
4      <child xsi:type="metamodel:MovingElement" p
5                  parent="//@elements.0" Name="Carriage" guidance="//@elements.0/@child.1"
6                  size="200.0" relativePosition="100.0"/>
7      <child xsi:type="metamodel:MechanicalGuidance"
8                  parent="//@elements.0" Name="CarriageGuidance" length="1500.0"
9                  sensorPositions="20.0">
10         <child xsi:type="metamodel:Motor"
11                       parent="//@elements.0/@child.1" Name="DriveMotor"/>
12         <positionSensor parent="//@elements.0/@child.1"
13                       Name="HomeSensor" value="1.0"/>
14         <positionSensor parent="//@elements.0/@child.1"
15                       Name="EndSensor"/>
16         <drive parent="//@elements.0/@child.1" Name="geraBoxAndSlider"
17                       motor="//@elements.0/@child.1/@child.0" ratio="2.0"/>
18     </child>
19   </elements>
```

```
20 </metamodel:PlantModel>
```

Listing 4.1: The model stored in XML.

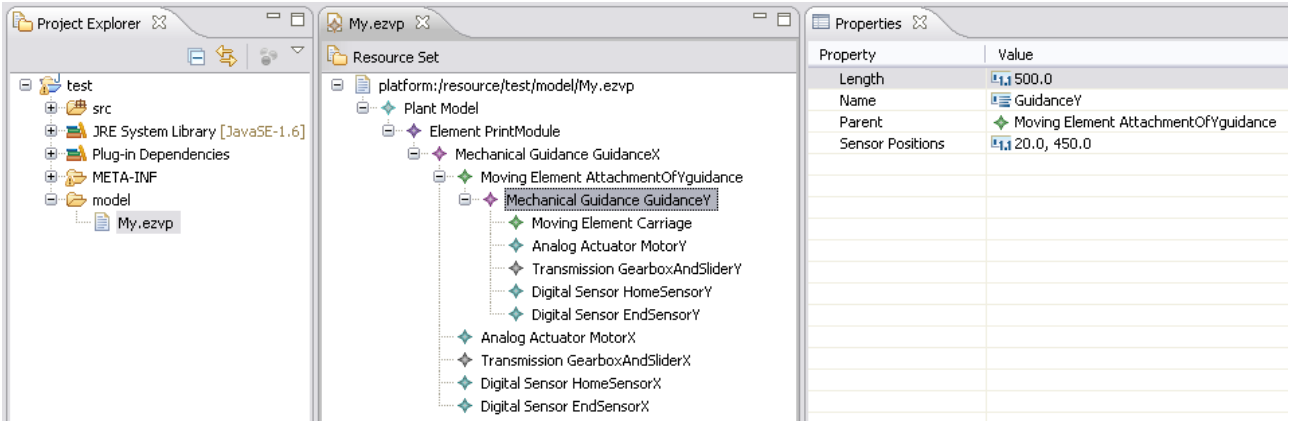

Figure 4.22: EZ-VP-Edit in Eclipse.

MoBasE

The other source of XML models that can be read by the back-end is MoBasE. The data needed by EZ-VP in this project has been added to MoBasE, so that the information from MoBasE can also be used to generate models.

Back-End

The EZ-VP back-end is a DLL created with RoseRT. It reads the generated XML models from EZ-VP-Edit or MoBasE and configures itself with the information from the model. So, the approach is very similar to SheetLogic. This configuration of the DLL is done when it is loaded by SIL. The class structure in the back-end reflects the entities and their relations in the meta model. The difference is that the classes actually implement the behavior of the model that has been presented in figures 4.10 and 4.16 till 4.19.

When a model is read from a XML model file, for each entity in the model an instance of the corresponding class is created and parameterized with the information from the XML model. So the goal of this step is to create the same structure as defined in the XML model with the classes in the back-end. All created instances are stored in a list of elements. Each time a "tick" is received from the SIL core, this list is iterated and every containing element receives a tick.

4.4 Matlab Simulink SIL Plant Model Simulation Modules

The COTS approach that is used in the plant modeling framework is Matlab Simulink. This tool is used within the company in nearly every project and is used by different disciplines to create models to, for example, evaluate electronics, mechanics or control functions. Even though software engineers do not have a lot of experience and therefore can not model complex systems in Matlab Simulink themselves, the models from other disciplines can be modified and used as plant model simulation modules.

To integrate Matlab Simulink models in SIL it is necessary to generate a DLL from a Matlab Simulink model that implements the SIL interface. For this purpose the Matlab Simulink toolbox Simulink Coder (formerly Real-Time Workshop) has been used [3]. This toolbox provides means to generate code from Matlab Simulink models. Matlab Simulink Coder provides a work-flow to generate code from a model and compile it into a DLL. Simulink Coder also provides to add user defined code to the generated code using a so called "custom code block". This custom code block is used to implement the SIL interface and the SILv2 communication approach. The plant model simulation module, create with Matlab Simulink, also has a "tick" function that causes the model to do a computational step. A schema of the DLL generation with Matlab Simulink can be found in figure 4.21.

Figure 4.23 shows an example model that consists of two parts. The left model consists of a counter (counts till ten and starts at 1 again), one input and two outputs. The output *testCounter* always equals the value of the counter, whereas the output *outCounter* is defined as the product of the input *enableCounter* and the counter value. The entities in this model that are defined as inputs and outputs are the signal lines. Those signal lines appear in the generated code as global variables and can therefore be used in the earlier mentioned custom code block that implements the SILv2 communication approach.

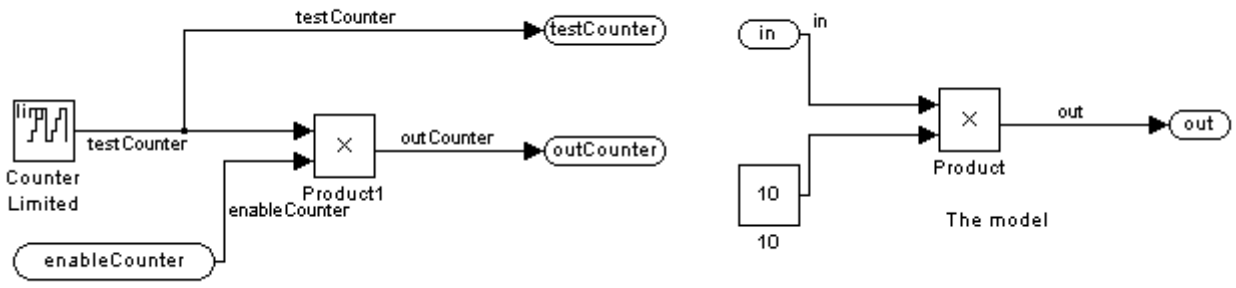

Figure 4.23: Example Matlab Simulink model.

4.5 Implementation

The implementation of the SIL plant modeling framework consists of three main activities:

- Implementing the SILv2 communication approach and the SILv2 interface in the code generation of Matlab Simulink Coder for the creation of plant model simulation modules with Matlab Simulink.
- Implementing the DSML editor EZ-VP-Edit.
- Implementing the EZ-VP back-end.

Nearly everything of these three parts has been implemented in this project. The only detail that has not been implemented is reading in intermediate plant models from the front-end in the back-end. This feature is essential for good usability but is not needed for the proof of concept implementation. The models created in the case studies are made by hand in the back-end and are not created from intermediate models.

Because the feature of reading XML intermediate models is not implemented, there is also no further information about the implementation of the part of the front-end concerning

MoBasE models, since the integration of MoBasE models consists purely of reading intermediate XML models exported from MoBasE. However, to be able to create intermediate models from MoBasE, all needed information (identification of entities, attributes of entities and their relations) are added to MoBasE. This is done in the project presented in [10].

4.5.1 EZ-VP-Edit

The front end of the EZ-VP DSML consists of the editor EZ-VP-Edit and MoBasE. Since the intermediate models from MoBasE are only read in by the back-end, and the feature of reading intermediate models has not been implemented, only the implementation of EZ-VP-Edit is presented.

EZ-VP-Edit is implemented using EMF [4]. EMF has been used to create the EZ-VP meta model as a language definition. This meta model has been used to generate an editor that can be used to create models in the language defined by the meta model. The steps that have to be taken to create a DS(M)L editor with EMF are:

1. Create the meta model in the graphical editor, resulting in a so called "ecore model" and an "ecore diagram", which is the graphical representation of the ecore model.
2. Create a so called "generator model". This entity has capabilities to generate an eclipse plug-in, providing an editor for the DSL defined in the meta model.
3. Generate editor Eclipse plug-in with the "generator model".

Figure 4.24 shows the graphical editor in which the EZ-VP meta model has been created. The toolbox on the left provides different types of entities that can be created. This tool box is divided in "Objects" and "Connections". Objects are again divided in entities like "classes" and "packages", and properties of those entities like "attributes" and "operations". An example of a class can be found in the upper left corner of the meta model, the "class" *MechanicalGuidance*. The "class" *MechanicalGuidance* has a property length, which is an "attribute". Beneath the "Objects", different connection types can be found. Examples are "Inheritance" and "EReference". Examples for the two types are the relation between *MechanicalGuidance* and *Element*, which is an "inheritance" relation, and the relation between *MechanicalGuidance* and *DigitalSensor*, which is an "EReference" relation. At the bottom of the figure, the properties window can be found. In this window the properties of the different entities in the diagram can be edited. In the figure, the properties of the "attribute" *length* can be seen. These comprise, for example, the "name", "upper and lower bound" and the "type" of the attribute.

After the creation of the meta model, a "generator model" has been created. this is used to generate code for the Eclipse plug-in, including the editor. Figure 4.25 shows the drop down menu provided by the generator model, which the option of generating an editor. On the left side of the figure the resulting projects that are created by the generator model are shown. The project "MetaModel.editor" can be started as an eclipse application, which causes a new instance of Eclipse to start. This instance contains the editor plug-in, which enables the creation of models in the language defined by the meta model. Figure 4.26 shows the instance of Eclipse with the created editor plug-in.

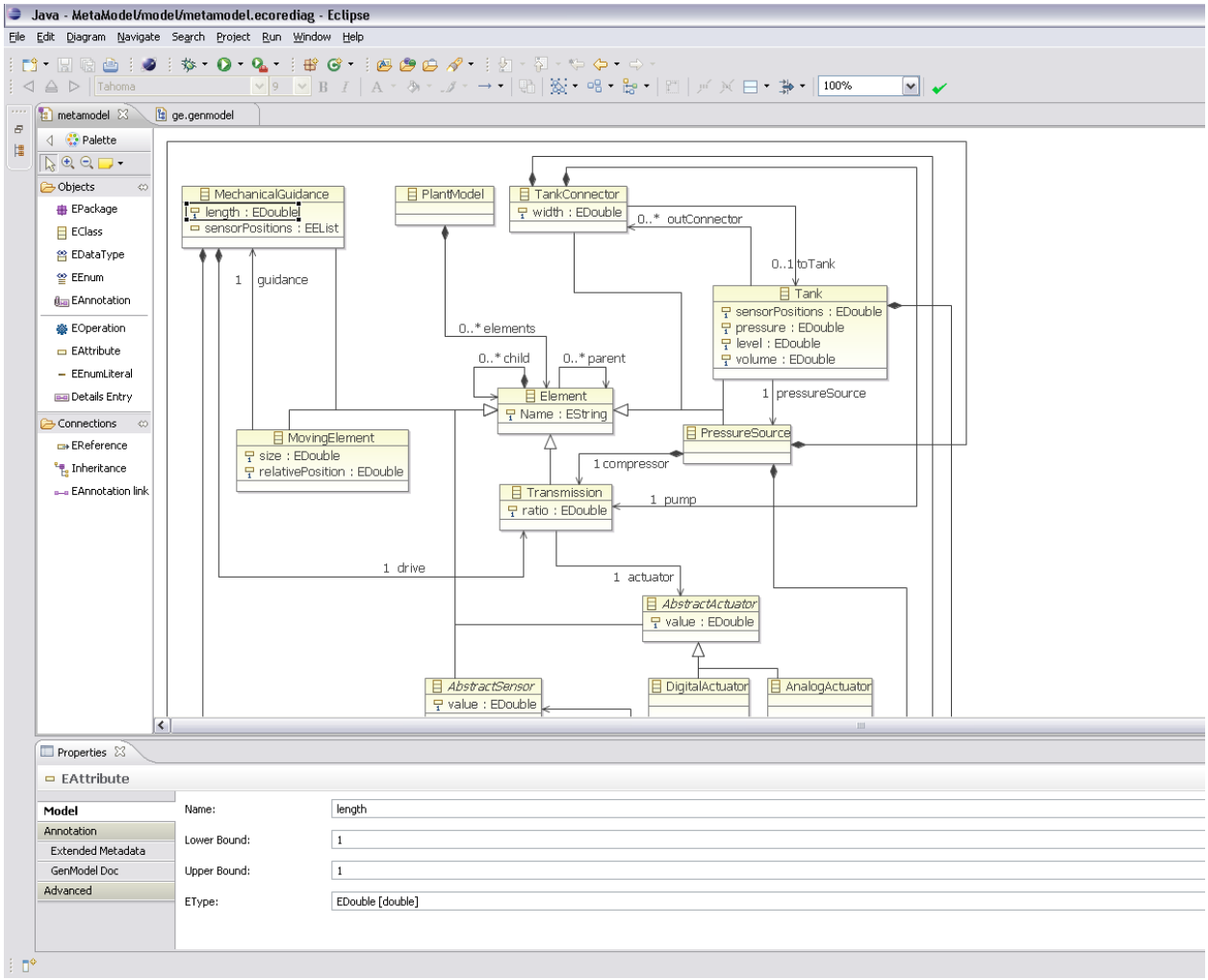

Figure 4.24: The EMF ecore diagram editor.

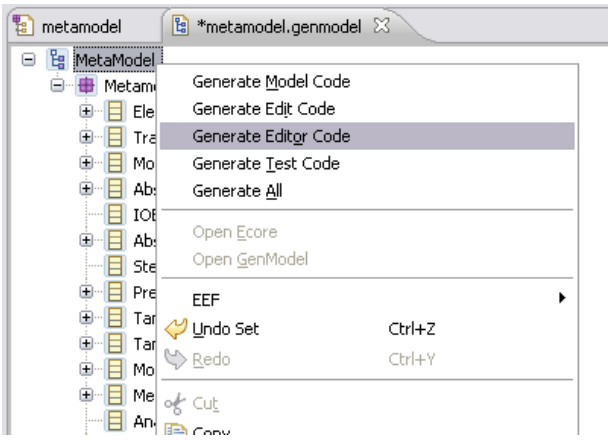

Figure 4.25: The generator model.

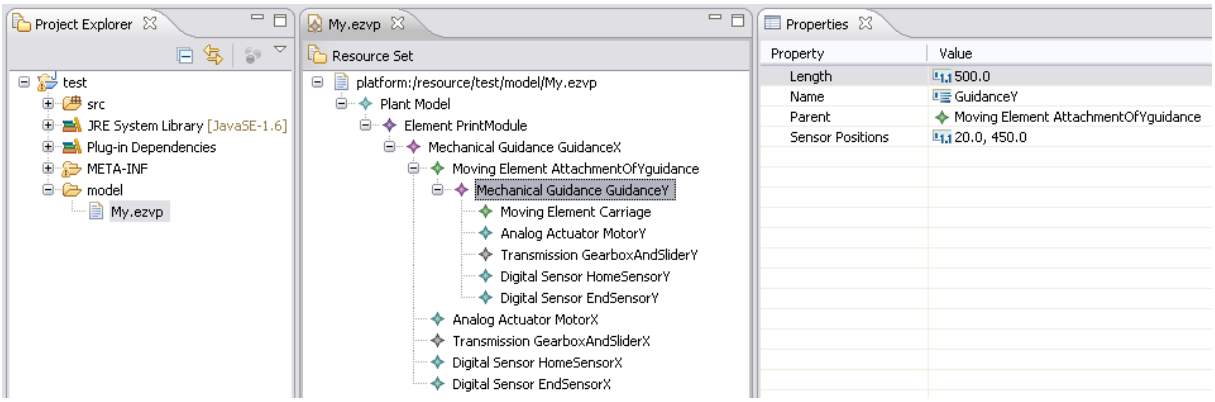

Figure 4.26: EZ-VP-Edit.

In figure 4.26, on the left, in the project explorer, a project is created in which an EZ-VP model is created (extension .ezvp). In the center, the hierarchy of the model can be seen. On the right, the properties (attributes) of the entities can be edited. To create new entities the drop-down menu that appears when performing a right click on an entity can be used. This is shown in figure 4.27 with an example using the *MechanicalGuidance* entity. For each containment relation a child can be created. In the case of the *MechanicalGuidance* it is possible to create every entity of the meta model as a child (inherited child relation from entity *Element*). It is further possible to create a *DigitalSensor* in the role of a position sensor, or a *Transmission* in the role of a drive for the guidance. The drive is grayed out in the menu because for this guidance, there is already a drive created (relation has a cardinality of one).

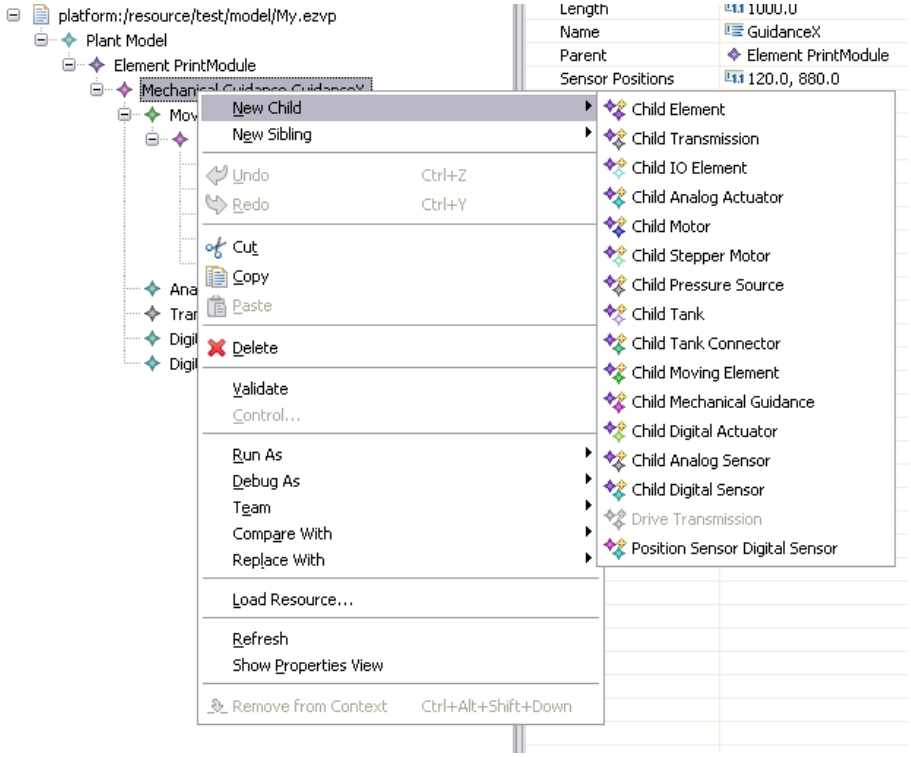

Figure 4.27: EZ-VP-Edit.

When the EZ-VP model is finished it can be opened in an text editor to see the underlying XML file structure. The file for the model shown in figure 4.26 and 4.27, is shown in listing 4.2.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <metamodel:PlantModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="
       http://www.w3.org/2001/XMLSchema-instance" xmlns:metamodel="http://metamodel/1.0">
3    <elements Name="PrintModule">
4      <child xsi:type="metamodel:MechanicalGuidance" parent="//@elements.0" Name="
           GuidanceX" length="1000.0">
5        <child xsi:type="metamodel:MovingElement" parent="//@elements.0/@child.0" Name="
             AttachmentOfYguidance" guidance="//@elements.0/@child.0" size="50.0"
             relativePosition="500.0">
6          <child xsi:type="metamodel:MechanicalGuidance" parent="//@elements.0/@child.0/
               @child.0" Name="GuidanceY" length="500.0">
7            <child xsi:type="metamodel:MovingElement" parent="//@elements.0/@child.0/
                 @child.0/@child.0" Name="Carriage" guidance="//@elements.0/@child.0/@child
                 .0/@child.0" size="200.0" relativePosition="250.0"/>
8            <child xsi:type="metamodel:AnalogActuator" parent="//@elements.0/@child.0/
                 @child.0/@child.0" Name="MotorY"/>
9            <drive parent="//@elements.0/@child.0/@child.0/@child.0" Name="
                 GearboxAndSliderY" actuator="//@elements.0/@child.0/@child.0/@child.0/
                 @child.1"/>
10           <sensorPositions>20.0</sensorPositions>
11           <sensorPositions>450.0</sensorPositions>
12           <positionSensor parent="//@elements.0/@child.0/@child.0/@child.0" Name="
                 HomeSensorY"/>
13           <positionSensor parent="//@elements.0/@child.0/@child.0/@child.0" Name="
                 EndSensorY"/>
14         </child>
15       </child>
16       <child xsi:type="metamodel:AnalogActuator" parent="//@elements.0/@child.0" Name="
             MotorX"/>
17       <drive parent="//@elements.0/@child.0" Name="GearboxAndSliderX" actuator="//
             @elements.0/@child.0/@child.1" ratio="2.0"/>
```

```
18        <sensorPositions>120.0</sensorPositions>
19        <sensorPositions>880.0</sensorPositions>
20        <positionSensor parent="//@elements.0/@child.0" Name="HomeSensorX"/>
21        <positionSensor parent="//@elements.0/@child.0" Name="EndSensorX"/>
22      </child>
23    </elements>
24  </metamodel:PlantModel>
```

Listing 4.2: The created model in XML.

So, the model shown in figures 4.26 and 4.27 is a graphical representation of the XML file shown in listing 4.2. It can be seen that, as in the editor, the containment relationships are represented by nesting. Other relation like the "parent" relation are realized by attributes in the XML tag ($parent$ = "//@$elements$.0/@$child$.0/@$child$.0/@$child$.0"). This XML file is the intermediate model that is further used by the back-end of EZ-VP.

4.5.2 EZ-VP Back-End

The back-end of EZ-VP is implemented in RoseRT. The back-end consists of a DLL that can be parameterized with a XML file holding an intermediate model from MoBasE or EZ-VP-Edit. So the approach is very similar to SheetLogic. The core of the back-end is the EZ-VP meta model. The meta model, implemented in the class structure of the back-end in RoseRT, is shown in figure 4.28.

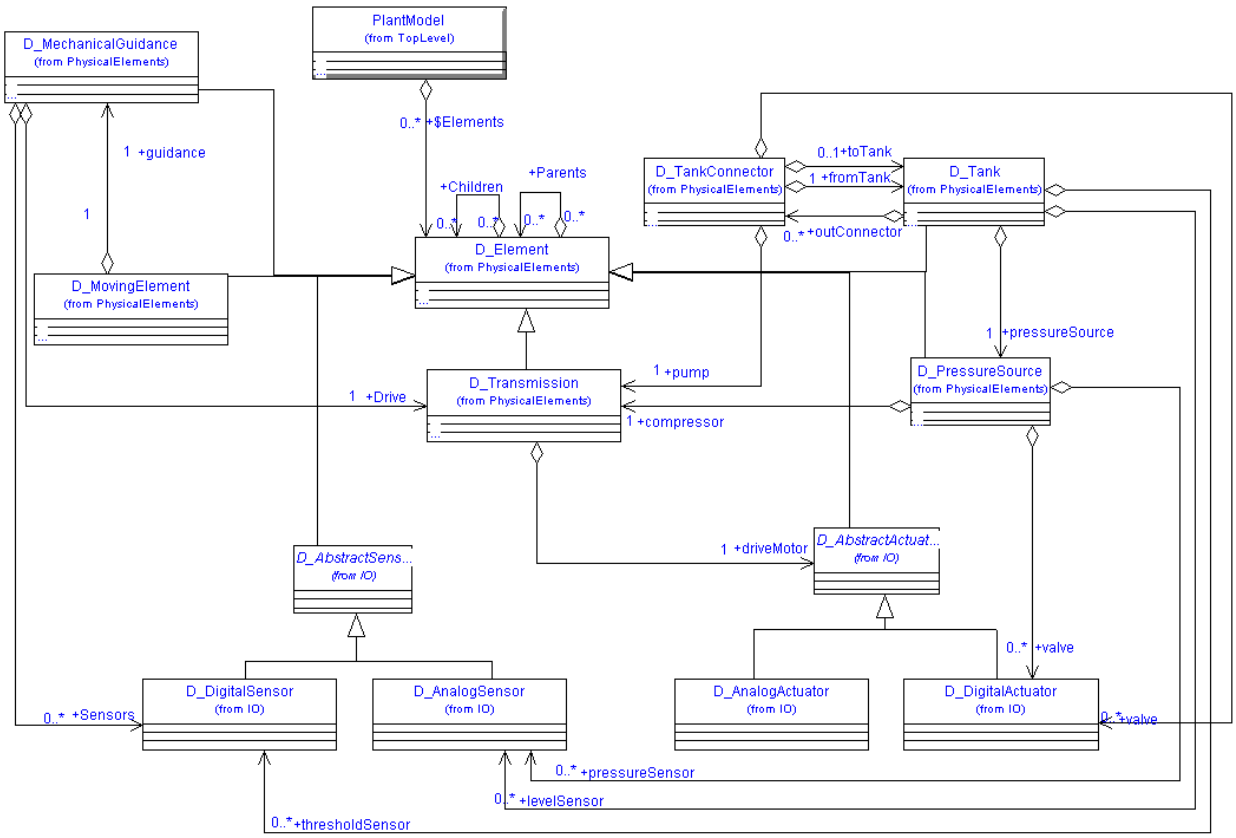

Figure 4.28: EZ-VP meta model implemented in RoseRT in the back-end.

The different types of entities are separated in different packages as shown in figure 4.29. Those packages are:

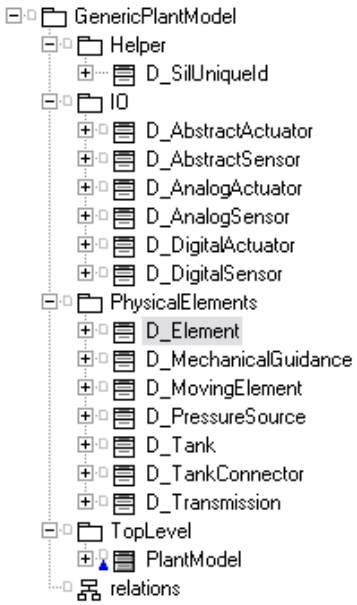

Figure 4.29: Structure of the EZ-VP back-end.

- TopLevel, this package contains the class PlantModel, which is the main entry point of the back-end DLL.
- Helper, this package contains the class *D_SilUniqueId*, which holds the name and moduleId of a plant element. This class is reused from SheetLogic.
- IO, this package contains the classes representing I/O-devices.
- PhysicalElements, this package contains all plant elements of the implemented plant concepts.

I/O-devices

The first parts of the meta model, that are presented, are the I/O-devices. The two super classes of the I/O-devices are *AbstractSensor* and *AbstractActuator*. The abstract class *AbstractSensor* is the superclass of *AnalogSensor* and *DigitalSensor*. It defines the attribute *value*, which is a pointer to a double value in the SIL interface structure. Each I/O-device is represented by a variable in the SIL interface structure of the plant model simulation module. So, the I/O-devices form the interface between the SIL interface structure and the physical elements. The variable, representing the I/O-device, is added to the SIL interface structure, when the I/O-device is created. An example of the creation is:

```
1 D_DigitalSensor* HomeSensorX = new D_DigitalSensor(new D_SilUniqueId("Engine","
      HomeSensorX"), addOutpComponent("HomeSensorX"));
```

Listing 4.3: Adding variables to the SIL interface structure when an I/O-device is created.

In this example a *DigitalSensor* is created with the name "HomeSensorX". The corresponding variable is created in the SIL interface structure by calling the function *addOutpComponent* in the I/O-layer (as described in section 3.6.1). The return value of this function, which is a pointer

to the double value of the created variable is given as a parameter to the created I/O-device. The physical elements use the functions *void setValue(double)* and *double setValue(void)* can be used to access and write the values of the I/O-devices. In the SIL interface structure, Sensors are defined as in- and outputs and actuators as inputs.

PhysicalElements

The common super class in the PhysicalElements package is the class *Element*. It defines the attribute name and the parent and child relation between entities. Everything except the class *PlantModel* is a sub class of *Element*. The other physical elements in this package are implemented as described in the design of EZ-VP (section 4.3.4).

The delivery rate caused by pressure is implemented as a simple formula in the initial implementation. The flow of material equals the pressure in a tank multiplied by a factor. If this equation is fine tuned with measurements from the hardware prototype, it can be used for testing (case studies). However, in the future, it would lead to better results to let a domain expert design a equation that can be used to determine the delivery rate.

TopLevel

The top level class of the back-end is the class *PlantModel*. In this class, the class structure is instantiated to create a plant simulation module. The *PlantModel* holds a list of *Elements*, which is iterated with each "tick" that is received by the plant model simulation module from SIL. Each *Element* in the list receives a tick. The most *Elements* just inherit the function "tick" from the common super class *Element*, which is an empty method. Only the classes *MovingElement* and *Tank* overwrite the "tick" function of the super class to perform the needed actions in each "tick" (as described in the behavior description in section 4.3.4).

4.5.3 Matlab Simulink Plant Model Simulation Module Creation for SIL

The other approach in the SIL plant modeling framework is Matlab Simulink. Since Matlab Simulink is a COTS approach, the implementation consists of developing a work-flow to create plant model simulation modules from models created in Matlab Simulink. For this the code generation capabilites of the Matlab Simulink Coder tool box have been used. A tutorial for the generation of DLL with the Simulink Coder has been used to create this work-flow for generating simulation modules [6].

The main activity in the integration of Matlab Simulink models into SIL has been the implementation of the SIL interface and the SIL communication approach. This is done in a custom code block, which is a model entity, provided by the Simulink Coder tool box.

In the Matlab Simulink models, signal lines can be configured to represent I/O-devices for the communication with SIL. Those signal lines have to be configured as "exported global", which makes them global variables in the generated code. This way it is possible to read and write the variable values to update them from the SIL interface structure or to write the values of them to the SIL interface structure. The custom code block defines the SIL interface structure as shown in section 3.6.1 in listing 3.1. Listing 4.4 shows the part of the custom code block in which the SIL interface structure is created. In addition to the creation of the SIL interface structure, the list of inputs and the list of outputs is created.

```
static struct sil_Interface io;

/*************************************************************************/
// create input and output lists. Order is important and should be equal
// when updating and writing in sil_StartScheduling
static struct sil_Variable inputs[] =  {
                                          { "in", 0 },
                                          {"enableCounter", 0}
                                       };
static struct sil_Variable outputs[] = {
                                          { "outCounter", 0 },
                                          { "out", 0} ,
                                          {"testCounter",0}
                                       };
/*************************************************************************/
```

Listing 4.4: Adding variables to the SIL interface structure.

Listing 4.5 shows the function *struct sil_Interface* sil_GetInterface*(), implemented in the custom code block. In this function, the SIL interface structure *io* is filled with the in- and output lists and is returned.

```
struct sil_Interface* sil_GetInterface()
{
    io.inputs = inputs;
    io.inpCount = sizeof(inputs)/sizeof(struct sil_Variable);
    io.outputs = outputs;
    io.outpCount = sizeof(outputs)/sizeof(struct sil_Variable);
    io.inAndOutputs = 0;
    io.inAndOutpCount = 0;
    return &io;
}
```

Listing 4.5: void sil_GetInterface() in the custom code block.

SIL calls the function *void sil_StartScheduling*() to let the simulation module perform a computational step. This function is shown in listing 4.6.

```
void sil_StartScheduling()
//Periodic update function {
/*************************************************************************/
    //update inputs ( IMPORTANT: the same order as added to the list)
    in = (real_T)io.inputs[0].value;
    enableCounter = (real_T)io.inputs[1].value;

    // call the step function of model
    simple_example_step(); //stepfunction for the model generated by Simulink
    coder

    //update outputs ( IMPORTANT: the same order as added to the list)
    io.outputs[0].value = (double)outCounter;
    io.outputs[1].value = (double)out;
    io.outputs[2].value = (double)testCounter;
/*************************************************************************/
```

Listing 4.6: void sil_StartScheduling() in the custom code block.

First, the inputs are updated by writing the values of the input variables from the SIL interface structure to the global variables (*in*, *enableCounter*). After this, the Simulink model is executed. After the execution, the values of the global variables, that are outputs, are written to the corresponding variables in the SIL interface structure. The whole custom code block can be found in appendix B.

4.6 Case Studies

To evaluate the EZ-VirtualPrinter approach of the modeling framework, two case studies are carried out. A case study for both, the moving elements concept and the material transport concept, of the EZ-VP approach is presented.

4.6.1 Case Study 1. Moving Elements

To evaluate the concept of moving elements modeled with EZ-VP, a case study has been performed. The system modeled in this case study is the system presented in figure 4.30. This

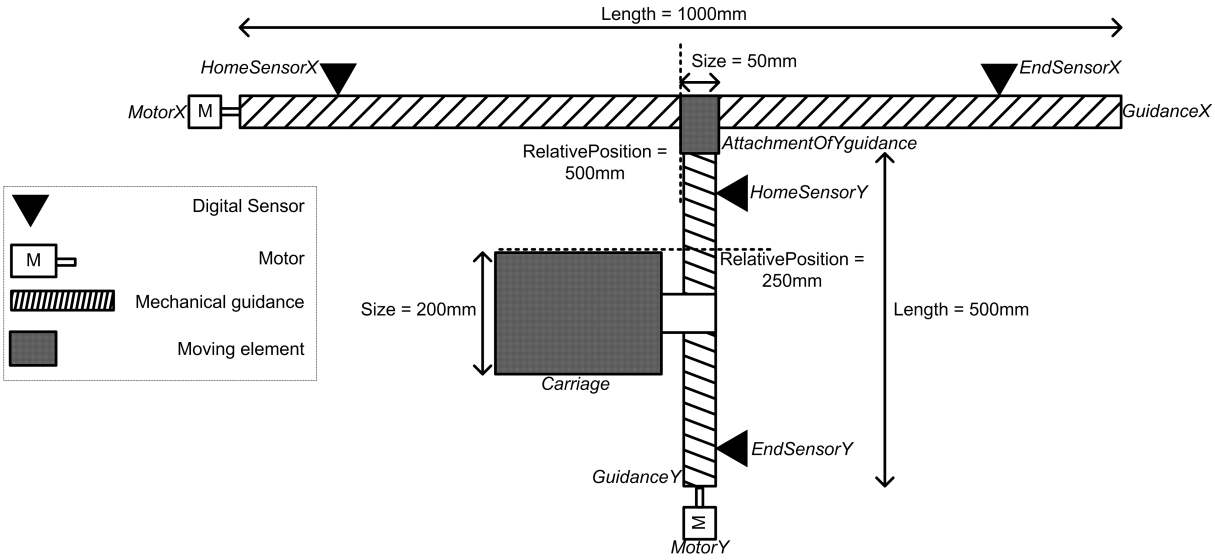

Figure 4.30: The system modeled with EZ-VP in this case study.

system has first been modeled in EZ-VP-Edit. The EZ-VP model is shown in figure 4.31. The

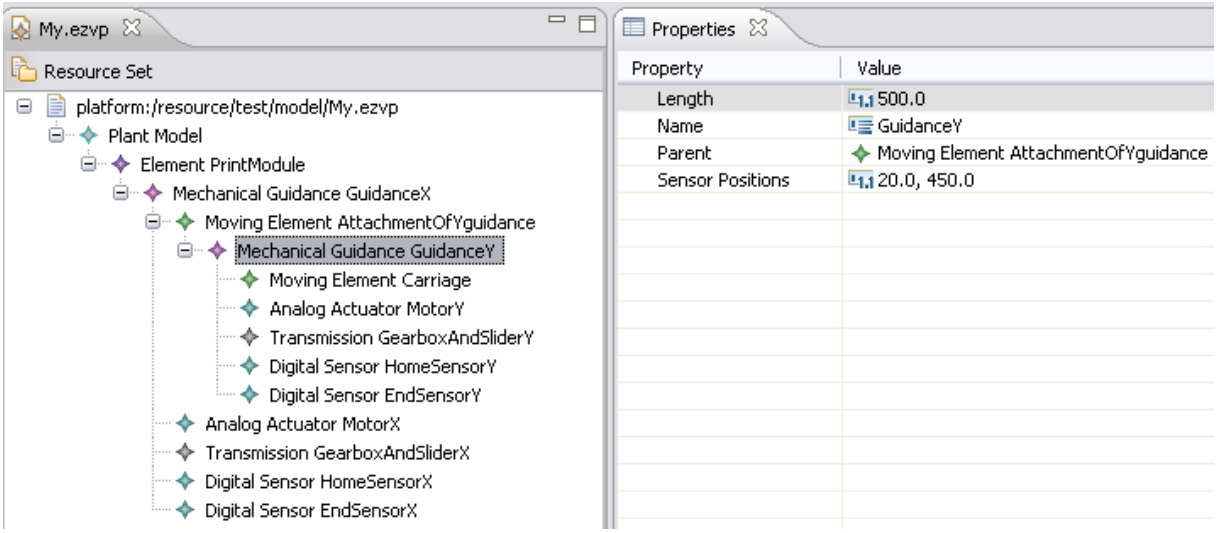

Figure 4.31: The system modeled with EZ-VP in this case study.

entities in the model have been parameterized according to the values in figure 4.30. In addition to this, the instantiation of the meta model for this system can be found in figure 4.9 in section 4.3.4. To control this plant model simulation module, ESW has been implemented. Both moving elements in the plant model are moved from the corresponding home-sensor to the end-sensor and back again. The control software is shown in listing 4.7.

```
1  if(HomeSensorX_status() == SENSOR_ACTIVE){
2    if(!Xstopping){
3      printf("HomeSensorX␣is␣active,␣stopping␣MotorX\n");
4      MotorX_stopNow(true);
5      Xstopping = true;
6    }
7    if(MotorX_stopped() && !Xstarted){
8      printf("MotorX␣stopped,␣begin␣movement␣to␣EndSensorX\n");
9      MotorX_on(1000);
10     Xstarted = true;
11   }
12 }else if(EndSensorX_status() == SENSOR_ACTIVE){
13   if(!Xstopping){
14     printf("EndSensorX␣is␣active,␣stopping␣MotorX\n");
15     MotorX_stopNow(true);
16     Xstopping = true;
17   }
18   if(MotorX_stopped() && !Xstarted){
19     printf("MotorX␣stopped,␣begin␣movement␣to␣HomeSensorX\n");
20     MotorX_on(-1000);
21     Xstarted = true;
22   }
23 }else{
24   Xstopping = false;
25   Xstarted = false;
26 }
27 if(HomeSensorY_status() == SENSOR_ACTIVE){
28   if(!Ystopping){
29     printf("HomeSensorY␣is␣active,␣stopping␣MotorY\n");
30     MotorY_stopNow(true);
31     Ystopping = true;
32   }
33   if(MotorY_stopped() && !Ystarted){
34     printf("MotorY␣stopped,␣begin␣movement␣to␣EndSensorY\n");
35     MotorY_on(1000);
36     Ystarted = true;
37   }
38 }else if(EndSensorY_status() == SENSOR_ACTIVE){
39   if(!Ystopping){
40     printf("EndSensorY␣is␣active,␣stopping␣MotorY\n");
41     MotorY_stopNow(true);
42     Ystopping = true;
43   }
44   if(MotorY_stopped() && !Ystarted){
45     printf("MotorY␣stopped,␣begin␣movement␣to␣HomeSensorY\n");
46     MotorY_on(-1000);
47     Ystarted = true;
48   }
49 }else{
50   Ystopping = false;
51   Ystarted = false;
52 }
```

Listing 4.7: Control software for the moving element.

As actuators for the movement in X and Y direction stepper motors are used. When a stepper motor is stopped or started, it takes some time till the stepper motor is ready with the operations and can receive new commands. Because of this, some additional if statements in the ESW are necessary to prevent that functions of the motor are called during the execution of a start or stop command.

To visualize the movement of the system, simple command line prints are used, since the visualization in Argus is not yet available. The following five figures (4.32 - 4.36) show screen shots of the SIL console window, in which the position of the moving elements and other interesting behavior (sensor activation, motors) is printed.

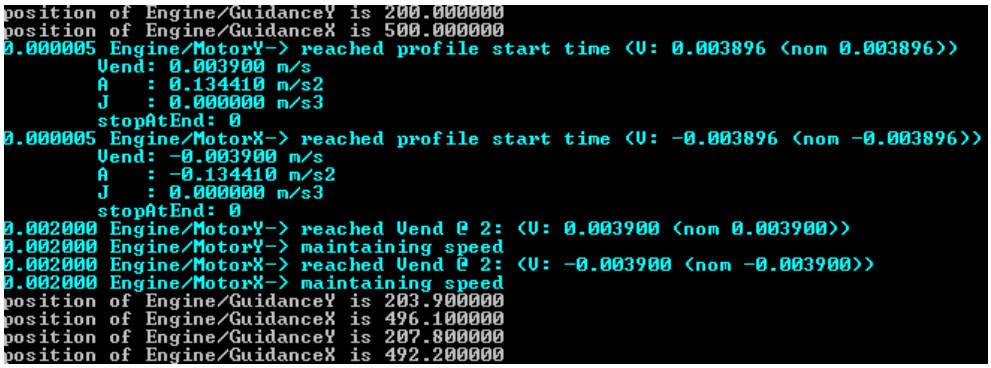

Figure 4.32: Running the System with control software in SIL.

First, both motors (MotorX, MotorY) are started. The prints of the motor are generated by the low level control simulation in SheetLogic. After the motors start, it can be seen that the relative position of the moving elements change. The carriage moves on the Y guidance in the direction of the end-sensor (increasing relative position) whereas the attachment of the Y guidance on the X guidance moves in the direction of the home sensor (decreasing relative position).

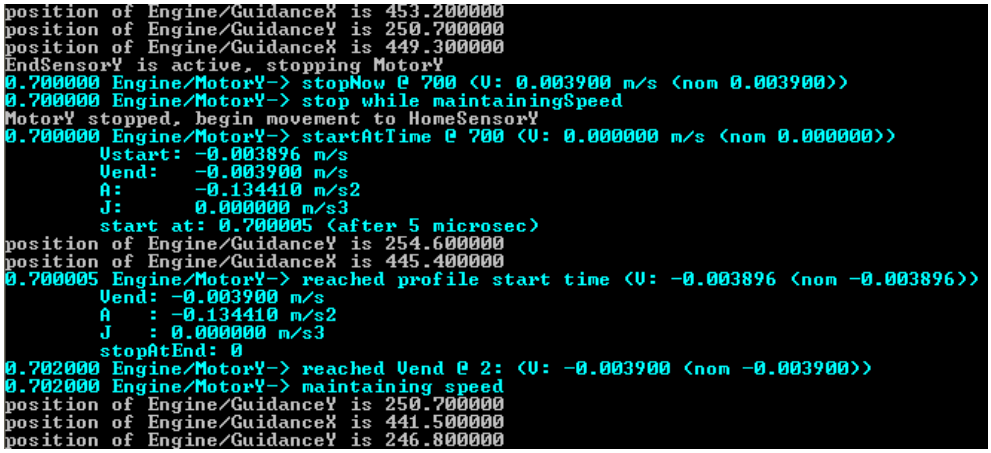

Figure 4.33: Running the System with control software in SIL.

The carriage reaches the end sensor on the Y guidance. The sensor is located at 450mm on the Y guidance and the relative position of the carriage is 250mm. Since the carriage has a size of 200mm the sensor can sense the carriage (position of 250 + size of 200 = 450 = sensor position). MotorY is stopped and started again in the other direction.

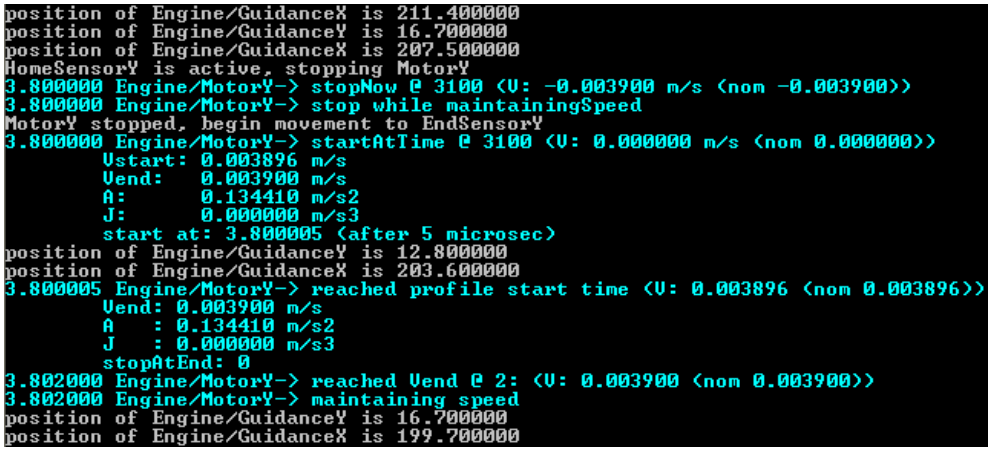

Figure 4.34: Running the System with control software in SIL.

The same happens when the carriage reaches the home sensor on the Y guidance, the MotorY stops and is started in the other direction. This way the carriage moves to and fro on the Y guidance.

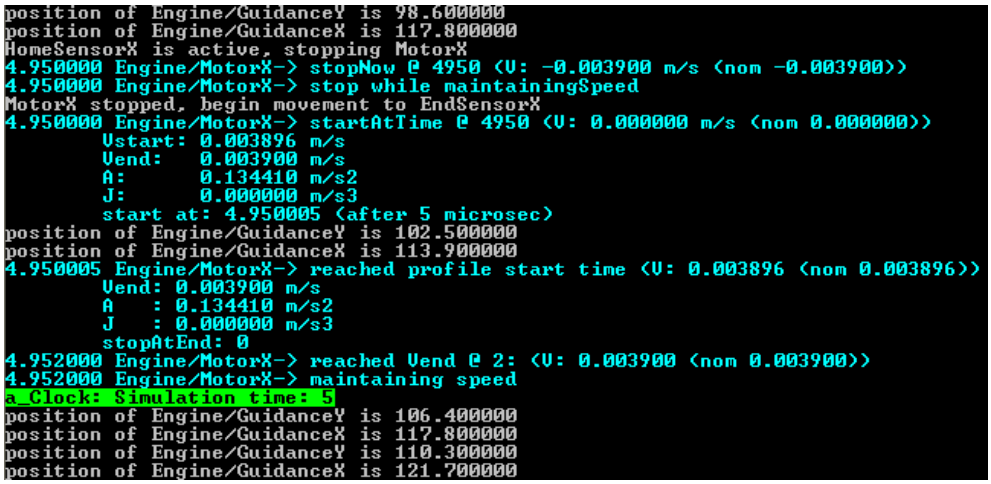

Figure 4.35: Running the System with control software in SIL.

The same behavior as the carriage on the Y guidance can be seen in the movement of the attachment of the Y guidance on the X guidance. In listing 4.35 the home sensor is reached and the moving element starts to move in the other direction. In figure 4.36, the moving element is sensed at the end sensor of the X guidance at a relative position of 830mm. This is because the sensor is located at 880mm and the attachment of the Y guidance has a size of 50mm (position of 830mm + size of 50mm = 880mm = end sensor position).

```
position of Engine/GuidanceX is 827.600000
position of Engine/GuidanceY is 172.700000
position of Engine/GuidanceX is 831.500000
EndSensorX is active, stopping MotorX
14.200000 Engine/MotorX-> stopNow @ 9250 (V: 0.003900 m/s (nom 0.003900))
14.200000 Engine/MotorX-> stop while maintainingSpeed
MotorX stopped, begin movement to HomeSensorX
14.200000 Engine/MotorX-> startAtTime @ 9250 (V: 0.000000 m/s (nom 0.000000))
        Vstart: -0.003896 m/s
        Vend:   -0.003900 m/s
        A:      -0.134410 m/s2
        J:       0.000000 m/s3
        start at: 14.200005 (after 5 microsec)
position of Engine/GuidanceY is 168.800000
position of Engine/GuidanceX is 835.400000
14.200005 Engine/MotorX-> reached profile start time (V: -0.003896 (nom -0.003896))
        Vend: -0.003900 m/s
        A   : -0.134410 m/s2
        J   : 0.000000 m/s3
        stopAtEnd: 0
14.202000 Engine/MotorX-> reached Vend @ 2: (V: -0.003900 (nom -0.003900))
14.202000 Engine/MotorX-> maintaining speed
position of Engine/GuidanceY is 164.900000
position of Engine/GuidanceX is 831.500000
position of Engine/GuidanceY is 161.000000
```

Figure 4.36: Running the System with control software in SIL.

4.6.2 Case Study 2. Material Transport

To evaluate the material transport concept of EZ-VP, a case study has been carried out. The system used in this case study is shown in figure 4.37. This system uses the two possible ways to transport material, with pressure or with a pump. Again, the system has first been modeled in

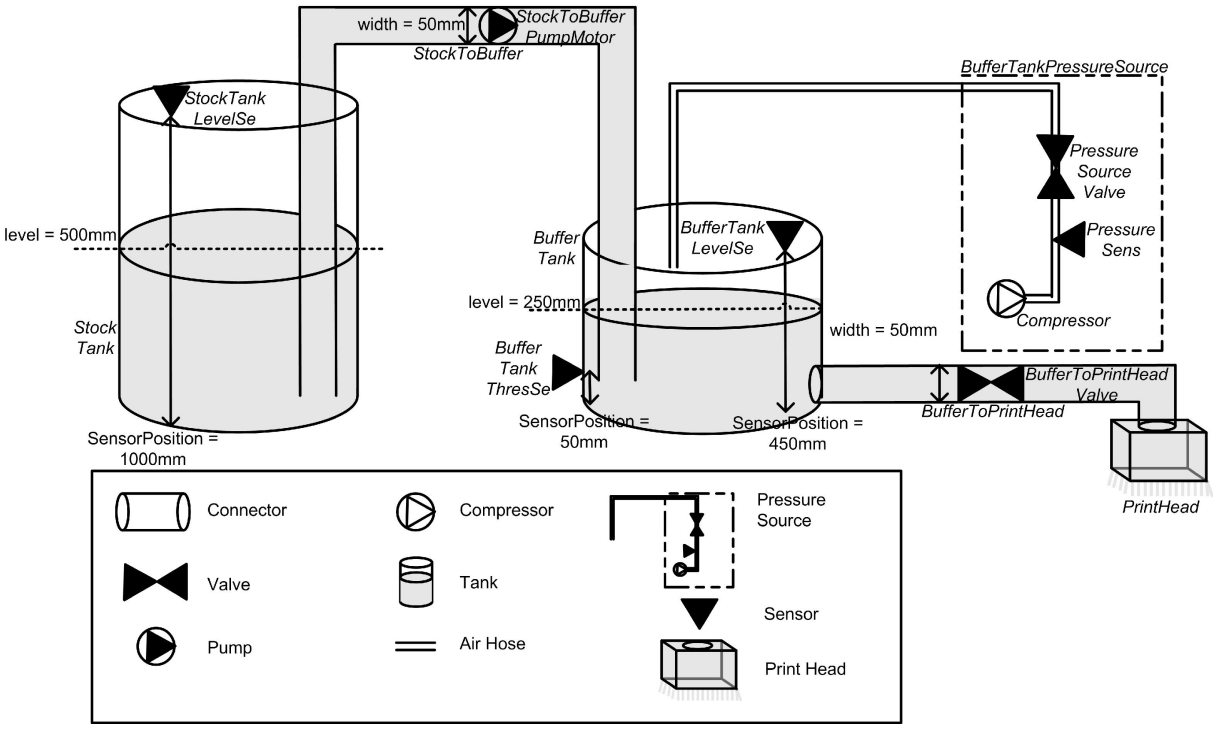

Figure 4.37: An example material transport system in a printer.

EZ-VP-Edit, as can be seen in figure 4.38. The parameter values of the model are set according to the values in figure 4.37. Since the system used in this case study is also mentioned in the design, an instantiation of the EZ-VP meta model can be found in figure 4.13 in section 4.3.5.

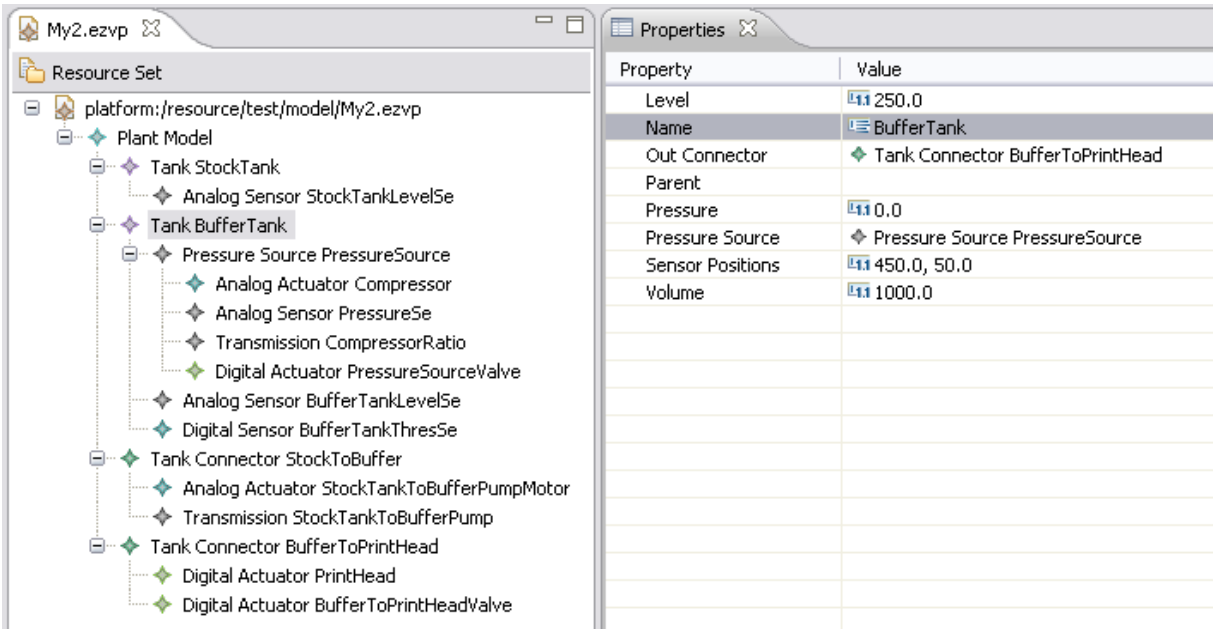

Figure 4.38: The system modeled with EZ-VP in this case study.

The system is controlled by a simple control ESW, which is shown in listing 4.9. In the initialization of the ESW simulation module (listing 4.8), the "bufferToPrintHeadValve" is opened so that material can flow when the print head is active. Further the compressor of the pressure source is turned on and the valve of the pressure source is opened.

```
1 PressureSourceValve_on();
2 Compressor_on(1);
```

Listing 4.8: Init of the control software for the material transport.

After the initialization, the following ESW is executed with every tick. In lines 3 till 8, it is determined if the system is printing, based on a random value. If the system is printing, the print head is turned on and material is flowing out of the system based on the pressure in the buffer tank. Lines 9 till 14 implement a simple pressure control for the buffer tank. Lines 15 till 22 show the control that refills the buffer tank, if the level of it is to low. Pumping from the stock to the buffer is started, if the threshold sensor in the buffer tank is 0 and the system is not yet pumping from the stock to the buffer tank. The pumping is stopped again, when the buffer tank is full or the stock tank is empty.

```
1  printf("stockTankLevelSensor:␣%d,␣bufferTankLevelSensor:␣%d,␣bufferTankThresholdSensor:␣
       %d\n"
2             ,StockTankLevelSe_status(),BufferTankLevelSe_status(),
                  BufferTankThresSe_status());
3  if((rand() % 51) < 25){
4    PrintHead_on();
5    printf("printing\n");
6  }else{
7    PrintHead_off();
8  }
9  if(PressureSens_status() < 10){
10   Compressor_on(++compVal);
11 }else if(PressureSens_status() > 20){
12   Compressor_off();
13   compVal = 0;
14 }
```

```
15 if(BufferTankThresSe_status() == 0 && StockTankToBufferPumpMotor_status() == 0){
16   StockTankToBufferPumpMotor_on(20);
17   printf("Pumping␣from␣stock␣to␣buffer\n");
18 }
19 if(BufferTankLevelSe_status() < 100 || StockTankLevelSe_status() > 999){
20   StockTankToBufferPumpMotor_off();
21   printf("Stop␣Pumping␣from␣stock␣to␣buffer\n");
22 }
```

Listing 4.9: Control software for the material transport.

The following figures show screen shots of the execution of the ESW simulation module and the plant model simulation module, presented in this case study, in SIL. Again, for visualization, interesting behavior is printed in the SIL console.

```
printing
stockTankLevelSensor: 0, bufferTankLevelSensor: 299, bufferTankThresholdSensor: 1
stockTankLevelSensor: 0, bufferTankLevelSensor: 299, bufferTankThresholdSensor: 1
stockTankLevelSensor: 0, bufferTankLevelSensor: 299, bufferTankThresholdSensor: 1
stockTankLevelSensor: 0, bufferTankLevelSensor: 299, bufferTankThresholdSensor: 1
printing
stockTankLevelSensor: 0, bufferTankLevelSensor: 300, bufferTankThresholdSensor: 1
printing
stockTankLevelSensor: 0, bufferTankLevelSensor: 300, bufferTankThresholdSensor: 1
printing
```

Figure 4.39: Running the System with control software in SIL.

In every "tick" of the ESW simulation module, the values of the level sensors of the two tanks and the value of the threshold sensor of the buffer tank are printed. When the print head is active, the output "printing" can be seen. In figure 4.39, it can be seen that the printer is printing sometimes and that therefore material is removed from the buffer tank. When material is taken from the tanks the value of the level sensors becomes bigger (distance from sensor to material surface).

```
stockTankLevelSensor: 0, bufferTankLevelSensor: 399, bufferTankThresholdSensor: 1
printing
stockTankLevelSensor: 0, bufferTankLevelSensor: 399, bufferTankThresholdSensor: 1
printing
stockTankLevelSensor: 0, bufferTankLevelSensor: 400, bufferTankThresholdSensor: 0
printing
Pumping from stock to buffer
stockTankLevelSensor: 0, bufferTankLevelSensor: 399, bufferTankThresholdSensor: 1
printing
stockTankLevelSensor: 0, bufferTankLevelSensor: 398, bufferTankThresholdSensor: 1
stockTankLevelSensor: 1, bufferTankLevelSensor: 397, bufferTankThresholdSensor: 1
printing
stockTankLevelSensor: 1, bufferTankLevelSensor: 396, bufferTankThresholdSensor: 1
printing
stockTankLevelSensor: 1, bufferTankLevelSensor: 396, bufferTankThresholdSensor: 1
stockTankLevelSensor: 2, bufferTankLevelSensor: 395, bufferTankThresholdSensor: 1
```

Figure 4.40: Running the System with control software in SIL.

After some time the material level in the buffer tank becomes so low that the threshold sensor of the buffer tank is 0. Now, the pump in the connector from the stock to the buffer tank is turned on and material is flowing from the stock to the buffer tank.

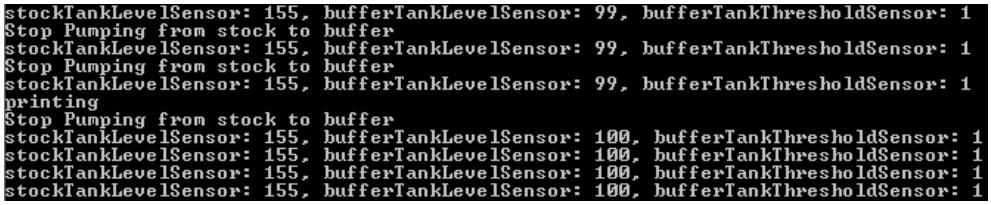

Figure 4.41: Running the System with control software in SIL.

The transport of material from the stock to the buffer tank stops, when the value of the level sensor of the buffer tank is lower than or equal to 100.

4.7 Evaluation

To evaluate the design of the SIL plant modeling framework, it has been determined if and how the requirements of the design are met. To do this, case studies have been performed and evaluated. In addition the modeling framework has been reviewed and evaluated by future users of the framework.

R1, The plant modeling framework shall support short modeling times.

The modeling framework provides three ways to create a plant model simulation module for SIL. This is done to give the modeler the possibility to make a trade-off between high expressiveness (Matlab Simulink), and short modeling times (EZ-VirtualPrinter). With EZ-VP it is possible to generate a plant model simulation module from MoBasE, resulting in virtually no modeling time, assuming the needed information is already in MoBasE.

The modeling using EZ-VP-Edit also offers the possibility to create models in short time. The models created for the case studies (4.6.1, 4.6.2), had a modeling time of about 15 minutes each. This time is pure modeling time. Before it is possible to create a model it is necessary to investigate the real plant to obtain the right parameter values for the model. An important and in some circumstances difficult to determine value is the ratio of a transmission. This is because, in the case of a moving element, it abstracts from the whole mechanic connecting the drive motor to the moving element. The same applies for the transmission, used in the context of material transport.

Compared to the creation of models with the stub mechanism, EZ-VP has a shorter modeling time since the behavior has not to be implemented. So, this requirement is met for the developed DSML EZ-VirtualPrinter. The creation of plant models with Matlab Simulink takes longer than modeling in EZ-VP (dependent on the size of the created model). The time needed for creating plant models with Matlab Simulink can be decreased drastically by reusing earlier created Simulink blocks or by reusing whole models created by people from other disciplines.

R2, It shall be possible for a non-domain expert to create plant model simulation modules.

EZ-VP-edit makes it possible to create plant model simulation modules on a high level of abstraction, which makes modeling much more feasible for software engineers compared to a general purpose modeling language for physical systems. This has also been confirmed by the software engineers, who have been interviewed to evaluate the plant modeling framework. It has been confirmed by them that every software engineer has enough knowledge of the printer

he works on, to model its plant in EZ-VP. However, some properties of the model have to be requested from domain experts (mechanical-, electrical engineers), like the transmission value. When intermediate EZ-VP XML models, for the generation of simulation modules, are taken from MoBasE, no domain knowledge is needed. Even if modeling in Matlab Simulink is not trivial, it is possible to reuse an already created block, to make modeling easier for software engineers.

R3, The modeling framework shall provide capabilities to model a plant in enough detail to properly evaluate the ESW.

During the analysis for the plant modeling framework, it has been realized that most of the time, meaning for the most of the control loops of the ESW, only a high level of abstraction, meaning high level information, is needed to properly evaluate the ESW. So, the concepts included in EZ-VP are abstracted as much as possible while maintaining enough detail to evaluate ESW. Next to the evaluation in the case studies, the meta model of both concepts, the moving elements concept and the material transport concept, has also been evaluated with domain experts and software engineers to determine whether it is possible to model systems using these two concepts in EZ-VP, and if the models have enough detail to properly evaluate the ESW. The result of this has been that it is possible to model the parts of a plant that use these concepts, in EZ-VP.

Since Matlab Simulink is a general purpose modeling language, it is possible to model all parts of the plant of a printer in the desired detail, making it a very expressive but time consuming modeling approach.

R4, The modeling framework shall be well maintainable.

In the case of Matlab Simulink, the maintenance effort is minimal. This is because it is a COTS solution with the associated advantages like updates and support from the vendor.

In the case of EZ-VP, maintenance involves more effort. When a new I/O-device or plant concept is added, the meta model in EZ-VP-Edit and in the back-end, has to be updated. In the back-end, the right semantics for the new I/O-device or concept have to be added. New I/O-devices only have to be added to EZ-VP if the I/O-device can not be abstracted to an analog sensor/actuator or digital sensor/actuator. It has also to be prevented that specific concepts (for example, only for one printer model) are added to EZ-VP, because this would lead to a steadily growing meta model of EZ-VP and therefore would make maintenance very complex. To keep EZ-VP maintainable, it is necessary to chose wisely which concepts are added to EZ-VP.

R5, The models created with the modeling framework shall be well maintainable.

The plant model simulation modules, created in EZ-VP, are created by parameterizing the back-end DLL with intermediate models from EZ-VP-Edit or MoBasE. To update models that are created in EZ-VP-Edit, the models can be altered using EZ-VP-Edit and provided to the back-end DLL to create an updated simulation module. To update models that are taken from MoBasE, MoBasE has to be updated and the new model can be used to create a new simulation module.

The models created with Matlab Simulink have to be updated by modifying the model in the Matlab Simulink editor. The effort of doing this depends on the needed changes in the model. If the interface of the plant model has changed, these changes also have to be performed

in the custom code block implementing the SIL interface structure. After that a new simulation module DLL can be generated.

4.8 Conclusion

A plant modeling framework has been designed that supports easy creation of plant model simulation modules for the purpose of emulating the plant behavior to evaluate the ESW of printers in the SIL simulation environment. The requirements for the plant modeling framework have been identified by interviewing the future users of the framework (section 4.1.2). With the new communication approach and interface of SIL, it has been possible to make use of two plant modeling approaches that are suitable in different situations:

- For the modeling of common plant concepts, the DSML EZ-VirtualPrinter has been designed (section 4.3), implemented (section 4.5) and evaluated (section 4.7). This easy-to-use plant modeling approach enables short modeling times and is well suited for software engineers.
- To integrate plant models, created with Matlab Simulink, into SIL, the SIL communication approach and interface have been implemented in the code generation process of Matlab Simulink Coder, making the generation of SIL simulation modules directly from Matlab Simulink possible (sections 4.4, 4.5.3). The integration of Matlab Simulink in SIL makes it possible to use Matlab Simulink to model plant concepts that are not included in EZ-VirtualPrinter. That are, for example, models of concepts specific to only one printer type. Another reason to use Matlab Simulink is that a plant concept is very complex and therefore Matlab Simulink is more appropriate.

During the design of the SIL plant modeling framework the main trade-off has been between expressiveness and ease-of-use. This trade-off is important, since the plant models must have enough detail to properly evaluate the ESW and the modeling process should be easy enough for a software engineer. The evaluation shows that the identified requirements of the plant modeling framework are met (section 4.7) and that, especially EZ-VP, provides a well suited tool to enable plant modeling for software engineers.

Chapter 5

Discussion and Future Work

In this project, the SIL simulation environment has been evolved and a plant modeling framework has been added. A main driver in the development of the architecture of the SIL simulation environment and the plant modeling framework has been ease-of-use. Ease-of-use has been important since the simulation setup, plant modeling and simulation runs are performed by software engineers, next to their day-to-day work. If, particularly the plant modeling, would be performed by domain experts, for example, mechanical engineers with experience in physical system modeling, at first glance it would not have been necessary to design a DSML. One might argue that, in this case, it would have been enough to design the work flow to include Matlab Simulink models into SIL. However, next to the fact that EZ-VP does not require this detailed domain knowledge and experience in physical system modeling, it is possible to generate models from MoBasE, which is an enormous advantage of EZ-VP. Modeling in EZ-VP is also less error prone than modeling in another general purpose modeling language, since the modeler models the behavior on a high level. So, even if domain experts and not software engineers create plant models, EZ-VP still is a well suited approach.

There are nearly endless possibilities to include concepts in EZ-VP, however it is important in the future that the language is well maintained to prevent that more and more specific concepts are added, because this way the language would become unmaintainable. Concepts that are, for example, specific for one type of printer, can be modeled using Matlab Simulink. There are some concepts that have not been implemented in EZ-VP in the initial design. Those are cross-cutting concepts like energy consumption of I/O-devices or tolerances of certain parts of the system. This would make it possible to not only evaluate the correctness of the ESW but also to optimize the ESW in certain ways (for example, print speed vs. energy consumption). Since the approach of SheetLogic and EZ-VP to generate plant models from MoBasE are very similar, it is also desirable to include the plant concepts of SheetLogic in EZ-VP for better maintainability. This could be an option during the redesign of SheetLogic to implement SILv2.

Globally speaking, from this project can be seen that DS(M)Ls can be regarded very suitable in situations where the domain is very bounded or the concepts of the domain can be abstracted to be more generic. However, abstracting to generic domain concepts has to be done with caution to prevent loosing important detail. In the case of the two implemented concepts of EZ-VP, moving elements and transport of material through the printer (section 4.3), the model has been detailed enough for the purpose of evaluating the ESW. The use of a DSML has been very suitable for the situation since it bridges the implementation gap and no detail of how the physical processes in the model are constructed, is needed.

There are many imaginable future projects in the context of the SIL simulation environment. One important project would be to complete the implementation of SILv2. This comprises:

- Extract SheetLogic from the SIL core and compile it as a simulation module, with the new interface, as a DLL. This way it becomes one of many simulation modules in SIL and can be used in a simulation when needed.
- The low level control simulation has to be removed from SheetLogic. It has to be added as a layer beneath the I/O-layer in the ESW simulation modules as described in figure 3.5 and figure 3.1.

Other future projects are the consequence of the change cases that have been presented in the evaluation of SILv2 (section 3.5). The visualization tool Argus has to be extended to visualize other plant models. For the concepts in EZ-VP, a 3D visualization, similar to the visualization of SheetLogic, should be implemented. For plant models created with Matlab Simulink, or for all plant models in general, it would be helpful to implement the possibility to visualize variable values in Argus. Another possible feature is described in change case "4. Change Settings of Simulation Modules On-The-Fly", in section 3.5.1. This feature makes it possible to load simulation modules by just knowing the place the DLL is stored. This makes the configuration of SIL easier because the SIL core does not need information about the simulation module configuration beforehand. Generally, it would be helpful to improve the usability of the SIL simulation environment. This could, for example, be done by implementing a user interface for the SIL simulation environment. In the current state the SIL simulation (console window), the Remote Control and the visualization have to be started separately. To setup a simulation, several XML files have to be edited by hand. The integration of all steps, from the simulation setup, the simulation run till the evaluation of test cases should be integrated in one tool, which would make SIL much more user-friendly.

There are two major future activities that are needed to make EZ-VP a well usable plant modeling language for the domain of printing systems. These activities are:

- Implement that the EZ-VP back-end can read in intermediate models from the front-end (MoBasE and EZ-VP-Edit).
- Integrate more generic plant concepts to EZ-VP (print process, temperature/humidity control).

To make the plant modeling framework more user-friendly, a user interface should be implemented that leads the user through the creation of models. In the future, it is also imaginable to include another approach, like Modelica, into the SIL plant modeling framework. But if and how another approach should be integrated, depends on the future use of the SIL plant modeling framework. If more and more domain experts create plant models for SIL, it could be advantageous to integrate such a specialized modeling language for modeling physical systems, into the SIL modeling framework.

Chapter 6

Conclusion

This project has been undertaken to evolve the SIL simulation environment, from a dedicated test tool for the ESW, controlling the paper handling, to a test environment for testing the ESW of printers in general. The project has been separated in two major activities; the modularization of the SIL simulation environment and the design and implementation of a plant modeling framework.

To get a clear view on the short and long term goals of SIL, several users of SIL have been interviewed (section 2.1). From those goals, requirements for the SILv2 simulation environment have been derived (section 3.1). These requirements are:

- R1. ESW shall be encapsulated within simulation modules.
- R2. Plant models shall be encapsulated within simulation modules.
- R3. A simulation shall be composed of multiple simulation modules.
- R4. Plant models shall be implemented using different plant modeling technologies.
- R5. Communication between ESW simulation modules and plant model simulation modules shall be based on a generic and well maintainable interface.

After identifying those requirements, the old situation of the SIL simulation environment has been reviewed and several problems, regarding the requirements, have been identified in its structure (section 3.2). To meet the requirements, a new communication approach between the SIL core and the simulation modules as well as a corresponding interface has been designed, implemented and evaluated (chapter 3). After the redesign of the overall structure of SIL by developing a new communication approach between simulation modules and SIL core, SIL is well prepared for the future use of different plant model simulation modules. The evaluation (3.5) shows that SILv2 meets the qualities that have been the key drivers in the development (Modular, Generic, Extensible, Composable, Efficient) and the identified requirements for the evolution of SIL.

The research question, identified in section 2.1:

- *How to develop a modular and extensible Software-In-The-Loop simulation environment that enables the execution of multiple embedded software modules and multiple plant models with the goal of evaluating the behavior of the embedded software?*

can be answered as follows. It is important to separate unrelated entities in the simulation environment (for example, plant models, ESW and a simulation core). The resulting modularity, and the use of a generic way to set up connections and to communicate between ESW and plant models, leads to an environment in which multiple modules can communicate with each other. Together with a minimalistic and generic interface between modules and simulation core, which can be implemented with low effort for the ESW and plant model approaches, this results in a modular and extensible environment.

In addition, a plant modeling framework has been designed that supports easy creation of plant model simulation modules for the SIL simulation environment (Chapter 4). The requirements for the plant modeling framework have been identified by interviewing the future users of the framework (section 4.1.2). The requirements are:

- R1. The plant modeling framework shall support short modeling times.
- R2. It shall be possible for a non-domain expert to create plant model simulation modules.
- R3. The modeling framework shall provide capabilities to model a plant in enough detail to properly evaluate the ESW.
- R4. The modeling framework shall be well maintainable.
- R5. The models created with the modeling framework shall be well maintainable.

With the new communication approach and interface of SIL, it has been possible to make use of two plant modeling approaches that are suitable in different situations:

- For the modeling of common plant concepts, the DSML EZ-VirtualPrinter has been designed (section 4.3), implemented (section 4.5) and evaluated (section 4.7). This easy-to-use plant modeling approach enables short modeling times and is well suited for software engineers.
- To integrate plant models, created with Matlab Simulink, into SIL, the SIL communication approach and interface have been implemented in the code generation process of Matlab Simulink Coder, making the generation of SIL simulation modules directly from Matlab Simulink possible (sections 4.4, 4.5.3). The integration of Matlab Simulink in SIL makes it possible to use Matlab Simulink to model plant concepts that are not included in EZ-VirtualPrinter. That are, for example, models of concepts specific to only one printer type. Another reason to use Matlab Simulink is that a plant concept is very complex and therefore Matlab Simulink is more appropriate.

During the design of the SIL plant modeling framework the main trade-off has been between expressiveness and ease-of-use. This trade-off is important, since the plant models must have enough detail to properly evaluate the ESW and the modeling process should be easy enough for a software engineer. The evaluation shows that the identified requirements of the plant modeling framework are met (section 4.7) and that, especially EZ-VP, provides a well suited tool to enable plant modeling for software engineers.

The research question, identified in section 2.1:

- *How to design a plant modeling framework for the creation of plant models for simulation in a Software-In-The-Loop simulation environment that is expressive enough to model a multitude of domain concepts of high performance printers, but also uses an adequate level of abstraction to make the approach usable for non-domain experts?*

can be answered as follows. The use of a DSML for plant modeling, is very suitable in a case in which the modeler has no detailed domain knowledge or experience with modeling physical systems. The steps that have to be taken in order to design a plant modeling DSML are:

- Analyze the domain and identify common domain concepts.
- Analyze the structure and behavior of parts in the identified domain concepts.
- Abstract the domain concepts to make them applicable in many situations.
- Create a language definition (meta model).
- Create an editor for model creation.
- Create a model interpreter or compiler to create an executable version of the model.

In this project, SIL has evolved from a dedicated test tool to test the ESW for controlling the paper transport, to a test framework to evaluate the behavior of ESW in general. The partial implementation of SILv2, presented in 3.6, is used in three projects. Together with the

novel SIL plant modeling framework, it is now possible to use SIL for evaluating the ESW of the whole printer. This means that the advantages identified for the ESW development, using SIL, apply now for the whole ESW development process. Those advantages are, testing without hardware and therefore starting the development earlier in the development of a printer, easy error injection and direct feedback for software engineers. This project has been an important step in the direction of developing printers with virtual prototypes rather than hardware prototypes, which is a long term goal for the development process of printers. With virtual prototype based development, the number of needed hardware prototypes in the development process can be decreased, which leads to lower cost and a short time-to-market.

Bibliography

[1] M. Andersson, D. Henriksson, and A. Cervin. Truetime 1.3 reference manual, june 2005, www.control.lth.se/ dan/truetime/.

[2] Modelica Association. "modelica.org/".

[3] Matlab Simulink Coder. "www.mathworks.nl/products/simulink-coder/index.html".

[4] Eclipse Modeling Framework Project (EMF). "www.eclipse.org/modeling/emf/".

[5] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe, 2003.

[6] Matlab Simulink Coder Tutorial for DLL Generation. "www.mathworks.nl/support/solutions/en/data/1-2f3i19/index.html?product=rtsolution=1-2f3i19".

[7] V.A. Kalloe. Simulator for verifying embedded control software of printing modules. Sai technical report, Eindhoven University of Technology, September 2009.

[8] Siu Hong Li. Interactive printer simulation visualization, June 2010.

[9] MathWorks. Simulink - simulation and model-based design, http://www.mathworks.nl/products/simulink/.

[10] C.G.U. Okwudire. Modelling variability in a simulated printer for improved robustness testing of embedded control software. Masters thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, August 2010.

[11] Karl Pfleger. An introduction to blackboard-style systems organization. Technical report, 1998.

[12] Christian Terwellen. A Survey of Plant Simulation Models for Testing Embedded Control Software of Mechatronic Systems. Research Topic Course, May 2012.

[13] S. van der Hoest. The development of a software-in-the-loop simulation framework for testing real-time control software. Sai technical report, Eindhoven University of Technology, August 2006.

[14] P.F.A. van den Bosch Z. Yuan B. van der Wijst A. van den Brand G.J. Muller W.P.M.H. Heemels, L.J.A.M. Somers. The key driver method, 2006. In Boderc: Model-based design of high-tech systems: a collaborative research project for multi-disciplinary design analysis of high-tech systems / Ed. W.P.M.H. Heemels, G.J. Muller. - Eindhoven : Embedded Systems Institute, 2006. - ISBN 9-78679-01-8. - p. 27-42.

Appendix A

I/O-layer

Listing A.1: changed parts of ln_iolayerTargetMacros.c

```
1  /************************************************************** *
2   *              %name: ln_iolayerTargetMacros.c %
3   *            %version: 41 %
4   *         %created_by: hhun %
5   *       %date_created: Thu May 31 11:25:28 2012 %
6   *      %full_filespec: ln_iolayerTargetMacros.c~41:csrc:ve020#12 %
7   *
8   ************************************************************/
9  #include "ln_iolayertypes.h"
10
11 #include <stdio.h>
12 #include <string.h>
13 #include <math.h>
14
15 #define MAX_STRING_LEN 80
16
17 #define EPSILON 0.001
18 #define MAX_IN_OUT_PUT 500
19 #include "SILSimulationItf.h"
20
21 int inpCompCount = 0;
22 int outpCompCount = 0;
23 int inAndOutpCompCount = 0;
24 static struct sil_Variable inputs[MAX_IN_OUT_PUT];
25 static struct sil_Variable outputs[MAX_IN_OUT_PUT];
26 static struct sil_Variable inAndOutputs[MAX_IN_OUT_PUT];
27 static struct sil_Interface io;
28 static int initialized = 0;
29
30 #ifdef __cplusplus
31 extern "C" {
32 #endif
33
34 extern char g_IoModuleId[100];
35
36 #ifdef __cplusplus
37 }
38 #endif
39
40 //helper for double compare
41 bool doubleEqual(double a, double b)
42 {
43     return fabs(a - b) < EPSILON;
44 }
45
46 // create the sil_Interface from the IO lists and return it
47 struct sil_Interface* sil_GetInterface()
48 {
49         io.inputs = inputs;
```

```
50          io.inpCount = inpCompCount;
51          io.outputs = outputs;
52          io.outpCount = outpCompCount;
53          io.inAndOutputs = inAndOutputs;
54          io.inAndOutpCount = inAndOutpCompCount;
55          initialized = 1;
56          return &io;
57  }
58
59  // for SIL plant models the direction of IO is inverted.
60  // So outputs of the ESW as actuators are inputs in a plant model.
61  // ESW inputs become inAndOutputs and not only outputs because
62  // the plant model still needs to read them occasionally. So,
63  // outputs become inputs and inputs become inAndOutputs.
64  #ifdef SILPLANTMODEL
65  // adds an component to the outputs list and returns
66  // pointer to the variables value
67  static double* addOutpComponent(const char* compName)
68  {
69          struct sil_Variable temp={compName, 0};
70          inputs[inpCompCount] = temp;
71          return &inputs[inpCompCount++].value;
72  }
73
74  // adds an component to the inAndOutputs list and returns
75  // pointer to the variables value
76  static double* addInpComponent(const char* compName)
77  {
78          struct sil_Variable temp={compName, 0};
79          inAndOutputs[inAndOutpCompCount] = temp;
80          return &inAndOutputs[inAndOutpCompCount++].value;
81  }
82  #endif /* SILPLANTMODEL */
83
84  #ifndef SILPLANTMODEL
85  // adds an component to the inputs list and returns
86  // pointer to the variables value
87  static double* addOutpComponent(const char* compName)
88  {
89          struct sil_Variable temp={compName, 0};
90          inputs[inpCompCount] = temp;
91          return &inputs[inpCompCount++].value;
92  }
93
94  // adds an component to the outputs list and returns
95  // pointer to the variables value
96  static double* addInpComponent(const char* compName)
97  {
98          struct sil_Variable temp={compName, 0};
99          outputs[outpCompCount] = temp;
100         return &outputs[outpCompCount++].value;
101 }
102
103 // adds an component to the inoutputs list and returns
104 // pointer to the variables value
105 static double* addInAndOutpComponent(const char* compName)
106 {
107         struct sil_Variable temp={compName, 0};
108         inAndOutputs[inAndOutpCompCount] = temp;
109         return &inAndOutputs[inAndOutpCompCount++].value;
110 }
111 #endif /* SILPLANTMODEL */
112
113 #define SILSIMULATION_SET_VARIABLE(symbolicName, deviceId, IoId)\
114         \
115         double* symbolicName##_value;\
116         \
117         void symbolicName##_init(void)\
118         {\
119                 if(initialized != 1){\
```

```
120                         symbolicName##_value = addOutpComponent(#symbolicName);\
121                 }\
122         }\
123         \
124         void symbolicName##_setValue(double a_Value)\
125         {\
126                 *symbolicName##_value = a_Value;\
127         }\
128 
129 #define SILSIMULATION_GET_VARIABLE(symbolicName, deviceId, IoId)\
130         \
131         double* symbolicName##_value;\
132         \
133         void symbolicName##_init(void)\
134         {\
135                 if(initialized != 1)\
136                 {\
137                         symbolicName##_value = addInpComponent(#symbolicName);\
138                 }\
139         }\
140         \
141         double symbolicName##_getValue(void)\
142         {\
143                 return *symbolicName##_value;\
144         }\
145 
146 // Simple sensor is used as an example to show what the general
147 // steps are to use the new interface
148 #define SILSIMULATION_SIMPLE_SENSOR(symbolicName, deviceId, IoId)\
149         \
150         /*create a double pointer that points to the variables value*/\
151         double* symbolicName##_state;\
152         \
153         void symbolicName##_init(void)\
154         {\
155                 if(initialized != 1)\
156                 {\
157                         /*call addInpComponent at init and retrieve*/\
158                         /* pointer that points to the variables value.*/\
159                         /* Values of inputs are automatically update by SIL before */\
160                         /* the software module gets a tick. Outputs are */\
161                         /* automatically read by SIL after a tick */\
162                         symbolicName##_state = addInpComponent(#symbolicName);\
163                 }\
164         }\
165         \
166         E_SimpleStatus symbolicName##_status(void)\
167         {\
168                 /*use the pointer instead of the sheetlogic functions*/\
169                 if (doubleEqual(*symbolicName##_state, 0.0))\
170                 {\
171                         return SENSOR_INACTIVE;\
172                 }\
173                 else\
174                 {\
175                         return SENSOR_ACTIVE;\
176                 }\
177         }\
178         void symbolicName##_on(void)\
179         {\
180                 *symbolicName##_state = 1.0;\
181         }\
182         void symbolicName##_off(void)\
183         {\
184                 *symbolicName##_state = 0.0;\
185         }\
186 
187 
188 #define SILSIMULATION_SIMPLE_ACTUATOR(symbolicName, deviceId, IoId)\
189         \
```

```
190         double* symbolicName##_state;\
191         \
192         void symbolicName##_init(void)\
193         {\
194                 if(initialized != 1)\
195                 {\
196                         symbolicName##_state = addOutpComponent(#symbolicName);\
197                 }\
198         }\
199         \
200         void symbolicName##_on(void)\
201         {\
202                 *symbolicName##_state = 1.0;\
203         }\
204         \
205         void symbolicName##_off(void)\
206         {\
207                 *symbolicName##_state = 0.0;\
208         }\
209         \
210         /* status function can be used by external simulation packages*/\
211         E_SimpleStatus symbolicName##_status(void)\
212         {\
213                 if (doubleEqual(*symbolicName##_state, 0.0))\
214                 {\
215                         return SENSOR_INACTIVE;\
216                 }\
217                 else\
218                 {\
219                         return SENSOR_ACTIVE;\
220                 }\
221         }\
222 
223 
224 #define SILSIMULATION_ANALOG_SENSOR(symbolicName, deviceId, IoId)\
225         \
226         double* symbolicName##_state;\
227         \
228         void symbolicName##_init(void)\
229         {\
230                 if(initialized != 1){\
231                         symbolicName##_state = addInpComponent(#symbolicName);\
232                 }\
233         }\
234         \
235         uint16_t symbolicName##_status(void)\
236         {\
237                 return (uint16_t)*symbolicName##_state;\
238         }\
239         /* on function can be used by external simulation packages*/\
240         void symbolicName##_on(uint16_t a_Value)\
241         {\
242                 *symbolicName##_state = (double)a_Value;\
243         }\
244         void symbolicName##_setValue(uint16_t a_Value)\
245         {\
246                 *symbolicName##_state = (double)a_Value;\
247         }\
248 
249 
250 #define SILSIMULATION_ANALOG_ACTUATOR(symbolicName, deviceId, IoId)\
251   \
252         double* symbolicName##_state;\
253         \
254         void symbolicName##_init(void)\
255         {\
256                 if(initialized != 1)\
257                 {\
258                         symbolicName##_state = addOutpComponent(#symbolicName);\
259                 }\
```

```
260         }\
261         \
262         void symbolicName##_on(int16_t a_Value)\
263         {\
264                 *symbolicName##_state = (double)a_Value;\
265         }\
266         void symbolicName##_off(void)\
267         {\
268                 *symbolicName##_state = 0.0;\
269         }\
270         /* statusd function can be used by external simulation packages*/\
271         uint16_t symbolicName##_status(void)\
272         {\
273                 return (uint16_t)*symbolicName##_state;\
274         }\
275
276 #define SILSIMULATION_FPGA_WRITE_REGISTER16(symbolicName, deviceId, IoId)\
277   \
278         double* symbolicName##_state;\
279         \
280         void symbolicName##_init(void)\
281         {\
282                 if(initialized != 1)\
283                 {\
284                         symbolicName##_state = addOutpComponent(#symbolicName);\
285                 }\
286         }\
287         \
288         void symbolicName##_set(uint16_t a_Value)\
289         {\
290                 *symbolicName##_state = (double)a_Value;\
291         }\
292         /* statusd function can be used by external simulation packages*/\
293         uint16_t symbolicName##_status(void)\
294         {\
295                 return (uint16_t)*symbolicName##_state;\
296         }
297
298
299 #define SILSIMULATION_PWM_ACTUATOR(symbolicName, deviceId, IoId)\
300   \
301         double* symbolicName##_state;\
302         \
303         void symbolicName##_init(void)\
304         {\
305                 if(initialized != 1)\
306                 {\
307                         symbolicName##_state = addOutpComponent(#symbolicName);\
308                 }\
309         }\
310         \
311         void symbolicName##_on(int16_t a_Value)\
312         {\
313                 /* duty cycle is int, but is between 0 and 1000 */\
314                 *symbolicName##_state = (double)a_Value;\
315         }\
316         void symbolicName##_off(void)\
317         {\
318                 *symbolicName##_state = 0.0;\
319         }\
320         /* status function can be used by external simulation packages*/\
321         uint16_t symbolicName##_status(void)\
322         {\
323                 return (uint16_t)*symbolicName##_state;\
324         }\
325         void symbolicName##_maskExtHwEnaLine(int a_Value)\
326         {\
327         }\
328
329
```

```
330 #define SILSIMULATION_INVERTED_PWM_ACTUATOR(symbolicName, deviceId, IoId)\
331   \
332         double* symbolicName##_state;\
333         \
334         void symbolicName##_init(void)\
335         {\
336                 if(initialized != 1)\
337                 {\
338                         symbolicName##_state = addOutpComponent(#symbolicName);\
339                 }\
340         }\
341         \
342         void symbolicName##_on(int16_t a_Value)\
343         {\
344                 /* duty cycle is int, but is between 0 and 1000 */\
345                 *symbolicName##_state = 1000 -(double)a_Value;\
346         }\
347         void symbolicName##_off(void)\
348         {\
349                 *symbolicName##_state = 1000;\
350         }\
351         /* status function can be used by external simulation packages*/\
352         uint16_t symbolicName##_status(void)\
353         {\
354                 return (uint16_t)*symbolicName##_state;\
355         }\
356         void symbolicName##_maskExtHwEnaLine(int a_Value)\
357         {\
358         }\
359 
360 
361 #define SILSIMULATION_PWM_ACTUATOR_WITH_ENABLE(symbolicName, deviceId, IoId,
        symbolicNameEnable)\
362   \
363         double* symbolicName##_state;\
364         \
365         void symbolicName##_init(void)\
366         {\
367                 if(initialized != 1)\
368                 {\
369                         symbolicName##_state = addOutpComponent(#symbolicName);\
370                 }\
371         }\
372         \
373         void symbolicName##_on(int16_t a_Value)\
374         {\
375                 *symbolicName##_state = (double)a_Value;\
376                 symbolicNameEnable##_on();\
377         }\
378         void symbolicName##_off(void)\
379         {\
380                 symbolicNameEnable##_off();\
381                 *symbolicName##_state = 0.0;\
382         }\
383         /* status function can be used by external simulation packages*/\
384         uint16_t symbolicName##_status(void)\
385         {\
386                 return (uint16_t)*symbolicName##_state;\
387         }\
388         void symbolicName##_maskExtHwEnaLine(int a_Value)\
389         {\
390         }\
391 
392 #define SILSIMULATION_PWM_ACTUATOR_WITH_DIRECTION(symbolicName, deviceId, IoId,
        symbolicNameDirection)\
393   \
394         double* symbolicName##_state;\
395         \
396         void symbolicName##_init(void)\
397         {\
```

```
398                 if(initialized != 1)\
399                 {\
400                         symbolicName##_state = addOutpComponent(#symbolicName);\
401                 }\
402         }\
403         \
404         void symbolicName##_on(int16_t a_Value)\
405         {\
406                 if( a_Value < 0 )\
407                 {\
408                         symbolicNameDirection##_off();\
409                         /* do not make value positive, SIL derives direction from
                                polarity */\
410                 }\
411                 else\
412                 {\
413                         symbolicNameDirection##_on();\
414                 }\
415                 *symbolicName##_state = (double)a_Value;\
416         }\
417         void symbolicName##_off(void)\
418         {\
419                 symbolicNameDirection##_off();\
420                 *symbolicName##_state = 0.0;\
421         }\
422         /* statusd function can be used by external simulation packages*/\
423         uint16_t symbolicName##_status(void)\
424         {\
425                 return (uint16_t)*symbolicName##_state;\
426         }\
427         void symbolicName##_maskExtHwEnaLine(int a_Value)\
428         {\
429         }\
430 
431 #define SILSIMULATION_PWM_ACTUATOR_WITH_ENABLE_AND_DIRECTION(symbolicName, deviceId,
        IoId, symbolicNameEnable, symbolicNameDirection)\
432   \
433         double* symbolicName##_state;\
434         \
435         void symbolicName##_init(void)\
436         {\
437                 if(initialized != 1)\
438                 {\
439                         symbolicName##_state = addOutpComponent(#symbolicName);\
440                 }\
441         }\
442         \
443         void symbolicName##_on(int16_t a_Value)\
444         {\
445                 if( a_Value < 0 )\
446                 {\
447                         symbolicNameDirection##_off();\
448                         /* do not make value positive, SIL derives direction from
                                polarity */\
449                 }\
450                 else\
451                 {\
452                         symbolicNameDirection##_on();\
453                 }\
454                 *symbolicName##_state = (double)a_Value;\
455                 symbolicNameEnable##_on();\
456         }\
457         void symbolicName##_off(void)\
458         {\
459                 symbolicNameEnable##_off();\
460                 symbolicNameDirection##_off();\
461                 *symbolicName##_state = 0.0;\
462         }\
463         /* statusd function can be used by external simulation packages*/\
464         uint16_t symbolicName##_status(void)\
```

```
465         {\
466                 return (uint16_t)*symbolicName##_state;\
467         }\
468         void symbolicName##_maskExtHwEnaLine(int a_Value)\
469         {\
470 }\
```

Appendix B

Simulink to SIL module custom code block code

Listing B.1: The custom code block

```
struct sil_Variable
{
    const char* name;
    double value;
};

struct sil_Interface
{
    struct sil_Variable *inputs;
    int inpCount;
    struct sil_Variable *outputs;
    int outpCount;
    struct sil_Variable *inAndOutputs;
    int inAndOutpCount;
};

static struct sil_Interface io;

/*************************************************************************/
// create input and output lists. Order is important and should be equal
// when updating and writing in sil_StartScheduling
static struct sil_Variable inputs[] =  {
                                          { "in", 0 },
                                          {"enableCounter", 0}
                                       };
static struct sil_Variable outputs[] = {
                                          { "outCounter", 0 },
                                          { "out", 0} ,
                                          {"testCounter",0}
                                       };
/*************************************************************************/

void sil_StartScheduling() // Periodic update function
{
/*************************************************************************/
    //update inputs ( IMPORTANT: the same order as added to the list)
    in = (real_T)io.inputs[0].value;
    enableCounter = (real_T)io.inputs[1].value;

    // call the step function of model
    simple_example_step(); //stepfunction for the model generated by simulink coder

    //update outputs ( IMPORTANT: the same order as added to the list)
    io.outputs[0].value = (double)outCounter;
    io.outputs[1].value = (double)out;
    io.outputs[2].value = (double)testCounter;
```

```
47 /*****************************************************************************/
48 }
49 void sil_GetVersionInfo(unsigned char * ap_VersionV, unsigned char * ap_VersionR,
50                         unsigned char * ap_VersionL, unsigned char *
51                         ap_VersionP) {
52     /*************************************************************************/
53     simple_example_initialize(true);
54     /*************************************************************************/
55     *ap_VersionV = (unsigned char)versionVersion;
56     *ap_VersionR = (unsigned char)versionRelease;
57     *ap_VersionL = (unsigned char)versionLevel;
58     *ap_VersionP = (unsigned char)versionPatch;
59 }
60 /********************** Generic code below DON'T TOUCH ***********************/
61 struct sil_Interface* sil_GetInterface()
62 {
63     io.inputs = inputs;
64     io.inpCount = sizeof(inputs)/sizeof(struct sil_Variable);
65     io.outputs = outputs;
66     io.outpCount = sizeof(outputs)/sizeof(struct sil_Variable);
67     io.inAndOutputs = 0;
68     io.inAndOutpCount = 0;
69     return &io;
70 }
71
72 void sil_StartNode(const char *apc_ModuleId, const char * apc_Parameters) {}
73 void sil_StopNode(){}
```