

Modular Virtual Commissioning - A learning factory approach focussing on modularity and integration of component models.



Master Thesis

In partial fulfillment of the requirements for the degree

"Master of Science in Engineering"

Master program:
Mechatronics & Smart Technologies
Management Center Innsbruck

Supervisor 1: 
Benjamin Massow

Supervisor 2:
Thomas Hausberger

Author:
Dominique Mathäus Geiger
2010620012

Declaration in Lieu of Oath

„I hereby declare, under oath, that this master thesis has been my independent work and has not been aided with any prohibited means. I declare, to the best of my knowledge and belief, that all passages taken from published and unpublished sources or documents have been reproduced whether as original, slightly changed or in thought, have been mentioned as such at the corresponding places of the thesis, by citation, where the extent of the original quotes is indicated.“

Place, Date

Signature

Abstract

In this master thesis a concept for component libraries is created consisting of CAD data, simulation models, example control code and documentation.

Virtual commissioning is becoming increasingly important in the engineering process of plants and machines. This is the result of time savings and thus reduced costs in product development.

The behavior modeling of the different components can be done in several different tools, mostly depending on the manufacturer. The CAD data can also be available in different file formats and are thus no exception. This variety of possibilities increases the complexity of the integration process enormously. However, in order for virtual commissioning to be as efficient as possible, the integration of the used components should be as simple as possible.







In order to solve this problem, an exemplary library structure is developed in this thesis, which consists of the behavior model, CAD data and code snippets with related documentation. The basis for this are freely available and common interfaces, like Functional Mock-up Interface (FMI) for the behavior model and Collada for kinematic CAD data. This ensures that the widest possible range of applications is achieved. The practical use of this library is further demonstrated in a practical application.

In the example shown, the behavior model is executed in a separate task of the *Beckhoff* PLC, ensuring real-time capability. Visualization is done directly in the CAD software *Autodesk Inventor* based on the current state of the behavioral model. The behavior model is generated in *Matlab Simulink* and integrated using *FMI*.



As shown, using the developed library, the process of virtual commissioning can be simplified without the need for additional expertise.

Keywords: Virtual Commissioning, PLC, Automation, Modularity, Development.

Contents

| | | |
|--|-----------|---|
| 1. Introduction | 1 | |
| 1.1. Motivation | 1 | |
| 1.2. Aim of this Thesis | 1 | |
| 1.3. Structure of this Thesis | 1 | |
| 2. Product Development Process | 3 |  |
| 2.1. Overview | 3 | |
| 2.2. Required Information for Virtual Commissioning | 3 | |
| 2.3. Summary | 4 | |
| 3. State of the Art in Virtual Commissioning | 5 | |
| 3.1. Hardware/Software in the Loop (HIL/SIL) | 5 |  |
| 3.2. Different Approaches using Hard-/Software in the Loop | 6 | |
| 3.2.1. Visualization in the PLC Development Environment | 6 | |
| 3.2.2. Digital Twin using an Physics Engine | 6 |  |
| 3.2.3. Digital Twin in Simulation Software | 6 | |
| 3.3. Available Data Formats | 6 | |
| 3.3.1. Geometry Data | 7 |  |
| 3.3.2. Behavior Models | 7 | |
| 3.3.3. Source Code of PLC Projects | 8 | |
| 3.3.4. Documentation | 8 | |
| 3.3.5. Exchange Libraries | 8 | |
| 3.4. Summary | 9 | |
| 4. Proposed Data Structure for Modular Virtual Commissioning | 10 | |
| 4.1. Modular Structure | 10 | |
| 4.2. Used File Formats | 11 | |
| 4.2.1. Geometry and CAD | 11 | |
| 4.2.2. Behaviour Model | 12 | |
| 4.2.3. PLC Code | 12 |  |
| 4.2.4. Documentation | 12 | |
| 4.3. Methods for the Selection of Data Formats | 12 | |
| 4.3.1. Geometry | 13 |  |
| 4.3.2. Behaviour Model | 15 | |
| 4.3.3. PLC Code | 18 | |
| 4.4. Workflow of the proposed Data Structure | 19 | |
| 4.5. Summary | 20 | |
| 5. Example Application using the proposed Data Structure | 21 | |
| 5.1. Introduction | 21 | |
| 5.2. Visualization | 22 | |



| | |
|--|-----------|
| 5.3. Module <i>Seperation</i> | 23 |
| 5.3.1. Introduction | 23 |
| 5.3.2. Generate Data Structure  | 23 |
| 5.3.3. Use Data Structure | 27 |
| 5.3.4. Result | 30 |
| 5.4. Module <i>Conveyor Belt</i> | 31 |
| 5.4.1. Introduction | 31 |
| 5.4.2. Generate Data Structure | 31 |
| 5.4.3. Use Data Structure | 35 |
| 5.4.4. Result  | 39 |
| 5.5. Summary | 39 |
| 6. Results and Evaluation Ausschreiben | 41 |
| 7. Summary and Outlook Translate | 42 |
| Bibliography | VI |
| List of Figures | IX |
| List of Tables | X |
| List of Code | XI |
| List of Acronyms | A1 |
| A. Appendix | A1 |
| A.1. PLC Source Code | A1 |

1. Introduction

1.1. Motivation

The relevance of a virtual commissioning of a new facility is successively increasing in plant engineering. One of the main advantages is the possibility to detect and correct errors in the control system as well as in the hardware at an early stage. This minimizes the costs of a possible change and avoids shutdown times. However, one problem in performing virtual commissioning is the generation and integration of simulation models representing the real plant. While special software such as *Simulink* can be used for modeling, integration is a major challenge in contrast. A dedicated workflow for the integration with a defined data structure for the exchange between customers and suppliers can reduce the effort for a virtual commissioning.

1.2. Aim of this Thesis

The aim of this thesis is the development of a concept for the integration of simulation models in a plant to be commissioned. Thereby the virtual commissioning is focused on the level of the PLC control. These simulation models include different aspects of product development and consist of the components: CAD model, behavioral model based on physics, control code and documentation. An existing learning factory is used as an exemplary facility for the development of this concept. Using this learning factory, students shall learn the handling and the characteristics of a PLC and the corresponding automation. The simulation of the plant should be real-time capable and the visualization should be done in a standard CAD software like *Inventor*.

1.3. Structure of this Thesis

This thesis is divided into several chapters and describes the basics, the development and use of a data structure, as well as the evaluation of the results.

In chapter 2 the process of product development is described and necessary information for a virtual commissioning are identified. This chapter provides the basis for the developed data structure.

In chapter 3 the state of the art in virtual commissioning of a plant is explained. For this purpose, different methods of implementation are considered and the available data formats are described.

The data structure for a standardized exchange between suppliers and customers is described in chapter 4. The structure itself and the used data formats are explained and selected.

This data structure is then used in a real-world example in chapter 5 and its usability is tested in practice.

In the end, the results are summarized in chapter 6 and evaluated in the context of virtual

commissioning. And finally, chapter 7 provides an outlook on the next possible steps and further developments.

2. Product Development Process

2.1. Overview

The development of a product includes several stages in different engineering disciplines as seen in Fig. 2.1. These stages can partly be developed in parallel to save time and costs in the development. However, special attention must be paid to the dependencies between the stages.

The first phase of product development includes the general process of defining the objectives, constraints and interfaces. It is part of project management and must be worked out in close cooperation with the customer and possible suppliers.

The next step is to develop the product's hardware. This usually consists of mechanical and electrical components. The hardware must be developed in conjunction with the two disciplines, since changes in the mechanics, for example, can often also lead to changes in the electrics. This principle also applies in the opposite way.

Parallel to the development of the hardware, the process of software development can already be started. In this way, possible limitations of the software can be identified at an early stage and possible changes to the finished hardware can be avoided. For the completion of the software, however, the hardware must already have been finalized. Only in this case is testing and debugging useful.

The final step in product development is virtual commissioning followed by actual commissioning. Depending on the product, the start of series production or delivery to the customer takes place here.



Figure 2.1.: Steps in product development.

2.2. Required Information for Virtual Commissioning



Information from all phases of product development are required for the successful execution of a virtual commissioning:

Process Design: Interfaces to other parts of the plant, such as transfer points of raw and finished parts, but also maximum time limits and throughput quantities. The selection and characteristics of used sensors and actuators also belong to this area.

Mechanical Engineering: The general structure of the mechanics and the kinematics of the individual components. The design of the product and the used materials determine the physical behavior. This information is often relevant for control systems.

Electrical Engineering: The configuration of the PLC with the used modules is part of this section. Especially interesting for the development of the software is the mapping between the inputs and outputs of the modules with the connected devices. Only an exactly documented mapping can avoid wrong connections in the PLC software and the resulting damage of the hardware.

PLC Coding: If more complex components are used in the plant, often these are addressed via their own interface. The documentation and the knowledge of the handling of these interfaces is important for the creation of the software but also for the later commissioning. In the best case examples exist, which explain the handling and thus ensure a simpler implementation. But also of simpler components an example can often be advantageous.

2.3. Summary

This chapter provides an overview of the individual steps in the development of a product. This process begins with basic project management with the specification of requirements and interfaces. After that, the hardware consisting of mechanics and electrics is developed and then the software is written. The final step is commissioning.

The second part of this chapter covers the information required for virtual commissioning. It should be remembered that information from all steps of product development are required.

3. State of the Art in Virtual Commissioning

In this chapter, the state of the art of virtual commissioning at the PLC level is briefly listed and explained. For this purpose, the general procedure in a virtual commissioning is explained at the beginning and differences are compared. Then commonly used file formats for the required steps are shown.

3.1. Hardware/Software in the Loop (HIL/SIL)



In general, virtual commissioning at PLC level consists of two parts: the control system to be optimized and the simulation model that represents the real plant. The control itself is often tested on the real target hardware, but depending on the manufacturer, it can also be executed directly in the development environment on an engineering system. If the model instance is executed on a separate hardware, this is called Hardware in the Loop (HIL). In comparison, with Software in the Loop (SIL) the model instance is also executed on the target hardware.

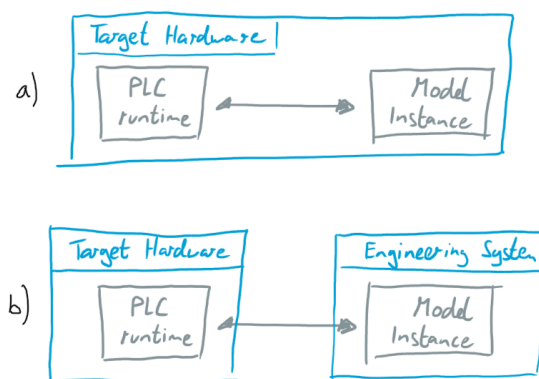


Figure 3.1.: Comparison of structure of SIL (a) and HIL (b) systems.

The HIL method offers advantages especially for complex systems compared to the SIL method. The second hardware avoids performance problems due to the additional computations on the target hardware. Furthermore, this method is the better choice for automated testing, due to a simplified implementation of continuous regression testing. The goal of this type of testing is to detect errors and bugs due to newly created sections of code.

One of the advantages of the SIL method is the avoidance of additional hardware for the model instance in comparison with the HIL method. Thus, in addition to the reduced cost of hardware, space in laboratories can also be saved. Finally, this results in a reduced planning effort for the laboratories and the time required for testing is shortened.

3.2. Different Approaches using Hard-/Software in the Loop

Testing the PLC software via HIL/SIL can be done in several ways, for which several practices are explained in this chapter.

3.2.1. Visualization in the PLC Development Environment



The Human-machine interface (HMI) offers a quick possibility for testing the control software. In most cases, a graphical user interface can be created directly in the development environment in which current values and outputs of the control software can be displayed and inputs can be set.

In this approach, the human developer imitates the behavior of the real plant and therefore tries to identify potential problems in the software. Due to the human interaction, fast processes and reactions can only be tested to a limited extent. The creation of a HMI is supported by many common PLC manufacturers such as *Siemens* and *Beckhoff*.

3.2.2. Digital Twin using an Physics Engine

The next step in virtual commissioning is the use of an external physics engine such as *Unity*. Especially by using existing libraries and well documented interfaces a model can be created in a short time. In addition, the handling of the model can be simplified by using the original geometry data. A disadvantage of this method compared to the direct visualization in the PLC development environment is the requirement of an additional communication between the PLC run-time and the physics engine. Nevertheless, depending on the implementation, this procedure can also be carried out in real-time. The area of application includes HIL and also SIL systems.

An example for the implementation in *Unity* and *TwinCAT* is [1]. Here, a digital twin is tested and used in a HIL system, achieving a communication time of 10 ms. The simulation in *Unity* achieves a step time of 5 μ s.

3.2.3. Digital Twin in Simulation Software

The use of simulation software such as *Matlab/Simulink* is particularly advantageous for complex systems or in control engineering. Complex systems can be easily created using mathematics and a graphical user interface. Furthermore, it improves the overview and minimizes the time needed to maintain the models. In this method, the model runs in the simulation software and for this reason also requires additional communication to the PLC run-time. If the system is supposed to be real-time capable, special hardware is often required. This approach can also be used in HIL and SIL systems.

3.3. Available Data Formats

This chapter describes and compares a selection of the most common data formats for relevant virtual commissioning information according to Sec. 2.2.

3.3.1. Geometry Data



The branches of industry and their products are very diverse and with them the requirements for the CAD software. For this reason, depending on the field of application, one specific CAD software may be more advantageous than another. Major CAD software vendors include *Creo Elements*, *Autodesk Inventor*, *Solidworks* or *Siemens NX*.

A key feature in the design of components using CAD software is the handling with *direct modeling* compared to *parameterized modeling*.

In *direct* modeling, the geometry is generated using constant values. Through this static processing, the different elements of the geometry remain independent of each other and the model is simplified. This type of CAD software is mainly used in the static field, such as in the structural engineering.

In contrast, in *parameterized* modeling, the geometry is generated using dependencies and features. This results in a chronic listing of the steps that lead to the desired geometry. The individual components are then dynamically connected in an assembly, whereby geometric dependencies become visible. Due to this kinematization, the components can be easily moved in the software and adjacent components move with them. This is particularly advantageous in the dynamic area with moving parts.

The exchange of CAD data between different tools is usually done via neutral formats such as *.step* (Standard for the Exchange of Product model data) or *.iges* (Initial Graphics Exchange Specification). However, only the pure geometry is supported and transferred via these formats and an information loss of the kinematization and features occurs. If necessary, in this case the customer has to recreate the kinematization or the CAD exchange has to be done via native formats. The industry has recognized this problem of the missing interface of the kinematization and currently different solutions are worked out. For example, the *.step* format can be extended to support features and kinematization as seen in [2].

A promising solution is the *COLLADA* format (Collaborative Design Activity), which supports kinematization starting from version 1.5 [3]. The *COLLADA* format, released back in 2004, is based on XML documents and is mainly used in the entertainment and gaming industry suffering from the same problem with multiple incompatible tools. The use in the manufacturing industry is currently not attractive, because suitable tools for the conversion to and from native formats are still missing. However, the already widely used exchange format *AutomationML* relies on *COLLADA* format to describe geometry data per default.

3.3.2. Behavior Models



Similar to the geometry data, there is a variety of possible file formats for the description of the physical behavior model. This is mostly dependent on the discipline and the simulation software used. However, especially in the field of co-simulation a universal interface is required to simplify data exchange. This is needed due to the combination of multiple engineering disciplines and tools for a complete simulation of the device under test.

For this reason, the Functional Mockup Interface (FMI) was defined by *Modelica Association* already in 2010. The basis of this interface consists of C-code, which can be used

universally on different devices. Currently this interface is available in version 3 and is already supported by over more than 170 tools [4]. The source code is open source and distributed under the *2-Clause BSD* license. This great popularity is also the result of the publicly available tools for checking the compatibility of *FMI* objects called Functional Mockup Unit (FMU).

3.3.3. Source Code of PLC Projects

In the area of control engineering and automation using PLCs, the basis is mainly the international IEC 61131 standard. In the context of this thesis, Part 3, which defines the programming languages, is of particular interest. This standard is based to a large extent on the organization *PLCopen*, which has set itself the goal of increasing the efficiency in the creation of control software and to be platform-independent between different development environments. To make this possible the PLC project with its code and libraries are saved as *XML* files and therefore can be exchanged without problems. This exchange format was standardized in 2019 in Part 10 of the IEC 61131 standard.

Many PLC manufacturers rely on this standard and offer interfaces for the defined programming languages. As a result, the effort required to exchange the software and thus the costs can be minimized.

3.3.4. Documentation

Only good documentation ensures the proper use of a product. This should be easy to read and understand by humans, thereby avoiding incorrect handling. For this purpose, various file formats are available, such as: *.pdf* (Portable Document Format), *.md* (Markdown) or *.html* (Hypertext Markup Language). The individual file formats all offer advantages and disadvantages, whereby the selection of the suitable format depends thereby strongly on the intended use.

For example, a *.pdf* file offers high compatibility between different devices combined with easy handling. The *.md* format is mainly used by software developers due to its standard integration with Git and relatively low requirements. Formatting texts is very easy and the integration of lists, images and tables is possible. In web-based help pages the *.html* format is often used. It can be displayed in any browser and is therefore, similar to the *.pdf* format, independent of the platform.

In any case, the use of plain text files for more complex documentation should be avoided. The missing possibility to embed images and to link between sections and files results in a documentation that is difficult to understand. However, simple instructions are excluded from this.

3.3.5. Exchange Libraries

When data is exchanged between supplier and customer via different tools, there should ideally be no loss of information. This goal cannot always be achieved, but using existing and especially supported exchange formats like *AutomationML* increases the probability of success.

This format represents thereby different information such as hierarchy, schematics and also code. By default, an object is composed in *AutomationML* from the geometry as *COLLADA* file and the control code in *PLCopen* format. Additional information can be added as desired as in the case of behavioral models as FMI files. This approach is for example seen in [5]. For this reason, *AutomationML* represents an appropriate exchange format [6]. However, as mentioned above, the lack of support for the *COLLADA* format from the CAD software side is still a problem.

3.4. Summary

In this chapter, the state of the art in the field of virtual commissioning was described with the help of PLCs. First, types of simulations and testing were explained and their application in practice was shown. In the second part the file formats for the different applications were discussed.

4. Proposed Data Structure for Modular Virtual Commissioning

Based on the previous chapters, a data structure for a virtual commissioning is presented in this section. The focus lies on the modularity of the individual sub-components of the considered plant.

4.1. Modular Structure

In order to improve understanding, the data structure is explained by means of an example. For this purpose, a filling station is considered as part of a larger plant. The design of this station is shown in Fig. 4.1a and consists of two identical dosing units (A), a conveyor belt (B) and a separator (C). These sub-components are bought-in parts and are assembled in this station. The product for the customer is the filling station as a complete unit.

The starting point for the development of the filling station are the exchange packages of the three suppliers with the respective product as content. These packages are connected and expanded and finally result in a package with the entire filling station as seen by the customer. In a next step, the customer can use the filling station in the planning of the remaining plant. This general procedure is shown in Fig. 4.1b.

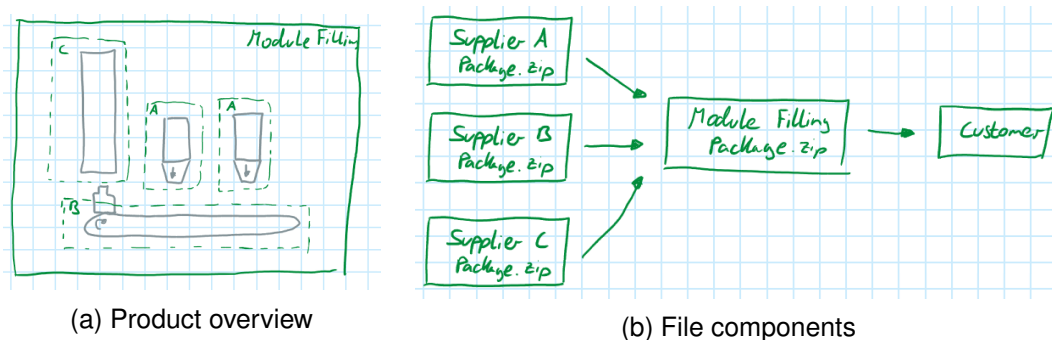


Figure 4.1.: Use case for data structure.

The actual setup of the data structure is kept simple and consists of a root folder with several directories for the relevant information as shown in Fig. 4.2. This root folder is finally compressed into a .zip file for distribution. Due to the plain and simple structure, it can be easily extended and modularly exchanged.

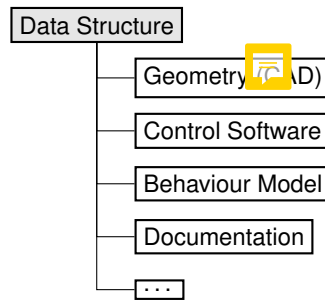


Figure 4.2.: Data structure

4.2. Used File Formats

As already mentioned in Sec. 3.3, a wide range of possible data formats exists to represent the required information for virtual commissioning. The focus in the selection of the used formats, should be on the widest possible support in different tools. Only through this approach the format can become accepted in the industry. In this section, the formats are selected and the result is summarized in Tab. 4.1.

Table 4.1.: Chosen file formats.

| Discipline | Information | Chosen File Format |
|-------------------|---------------------------------|----------------------------------|
| Mech. Engineering | CAD (pure geometry) | <i>.step</i> |
| | CAD (with kinematics) | Native formats or <i>COLLADA</i> |
| | Pyhsical Behavior | <i>FMI</i> |
| Elec. Engineering | PLC layout | native formats or <i>.pdf</i> |
| | List of used inputs and outputs | native formats or <i>.pdf</i> |
| Coding | PLC Code | <i>PLCopen</i> |
| Generel | Documentation | <i>.pdf</i> |
| | Drawings | <i>.pdf</i> |

4.2.1. Geometry and CAD

For a loss-free exchange of CAD data, the information of the kinematization of the assemblies must be preserved in addition to the raw geometry. As mentioned earlier, this is a big problem in practical use and therefore the *COLLADA* format would be a optimal solution. At the moment, suitable tools for conversion are missing or do not support the current version of *COLLADA* implementing the kinematization. As a result, using the *COLLADA* format is the optimal solution in theory but in practice native CAD formats are the better choice. Which CAD tool and further which format is finally used has to be defined between the customer and supplier. Alternatively, neutral formats like *.step* can be used, but with the disadvantage of missing kinematics. For these reasons, although a solution exists for the exchange of CAD data, it is not sufficient in the field of virtual commissioning.

For example, *Autodesk Inventor* offers a quick way to exchange data via the *Pack and*

Go tool [7]. In this case, the entire geometry, dependencies and also materials of an assembly are bundled in one *.zip* file, which also simplifies the exchange. A similar function is also available in *Solidwork* [8]. It should be noted, that although the two functions have the same name, they are not compatible with each other.

4.2.2. Behaviour Model

The modeling of the physical behavior can also be done using various tools. In contrast to CAD data, a neutral and established format for data exchange already exists here: the *FMI*. A growing number of tools support this format, making it a good choice in a modular virtual commissioning process.

In the academic community, but also in industry, *Matlab/Simulink* is often used for modelling. This tool is very popular mainly because of its flexibility, and supports the import and export of *FMI* files in versions 1 and 2. Instructions for the export can be found in [9] and for the import in [10].

4.2.3. PLC Code

The exchange of PLC code is almost no problem in practice. This is thanks to the standardization of automation engineering using PLCs, which also describes the possible programming languages and the exchange format as *.xml* files. In addition to the language basis, more complex functions such as function blocks for movements are also part of the standard and thus easily exchangeable. Thanks to the international validity, many manufacturers rely on this standardization and offer converters to and from the *PLCopen* format.

4.2.4. Documentation

For the exchange of documentation, the *.pdf* format has already been proven in practice. One of the reasons for this is the exact same layout of the document regardless of the operating system or a printout on paper. This ensures that the document is available to the reader in exactly the same form as it was when it was created.

4.3. Methods for the Selection of Data Formats



This section describes the tested approaches. As a result of this investigation, the choice of suitable file formats was made. For these tests the following software was used:

- CAD: Autodesk Inventor Professional 2022, Autodesk 3ds Max 2022
- PLC: Beckhoff TwinCAT 3
- Modelling: Matlab/Simulink

4.3.1. Geometry

When selecting a suitable format for the CAD data, the support of the kinematics is the main focus. For this purpose, an example assembly of a movable piston is created, as shown in Fig. 4.3. The kinematization consists of a rotating motion of the disk in the left section and a translational motion of the piston in the right section. These two parts are connect by a connecting rod in the middle area, with a rotational axis in both cases. If the left disk rotates, the rotational movement is then transformed into a translational movement of the right piston.

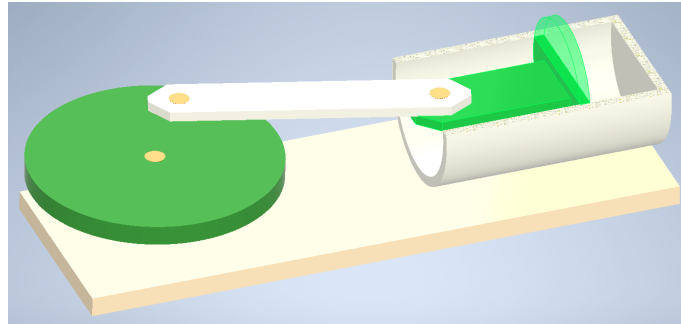


Figure 4.3.: Exemplary assembly for selection of suitable CAD-formats.

The selection criterion for a suitable exchange format is the ability to save the kinematization in combination with a wide software support. As a result of these criteria, a neutral exchange format should provide an optimal solution.

For this purpose, different methods of CAD data exchange are investigated and described in the following section. An overview of the tested methods is provided in Fig. 4.4 and the results of the studies are summarized in Tab. 4.2.

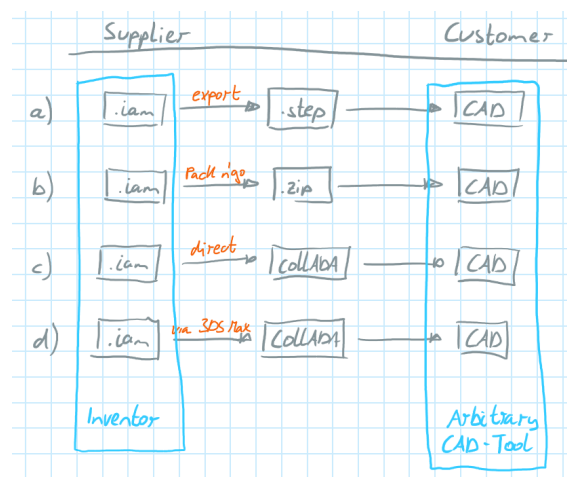


Figure 4.4.: Overview of tested methods for geometry export and import.

Table 4.2.: Results of tested methods in CAD exchange.

| Method | Geometry | Kinematization | wide software support |
|------------------------------------|----------|----------------|-----------------------|
| a) Using <i>.step</i> | yes | no | yes |
| b) Using <i>Pack and Go</i> | yes | yes | no |
| c) Using <i>COLLADA</i> directly | - | - | - |
| d) Using <i>COLLADA</i> indirectly | yes | no | yes |

a) Inventor to *.step* and back

The first test is using the *.step* format. This neutral format was chosen because of its already wide usage for CAD data exchange.

The export of the assembly into a *.step* file is done according to the instruction [11] via the menu *File-Export-CAD Format*. In the appearing window, the target format must finally be set to *.step*.

The import is also very simple and can be done either in a new file or in an existing assembly. Instructions can be found in [12]. In this case the *.step* file is opened in a new file via *File-Open*.

Exporting and importing *.step* files works without any problems, but the missing kinematic is detected when inspecting the imported assembly. However, the pure geometry is imported without errors and even the hierarchy of parts is created by an integrated conversion during import. In any case, the desired criteria are not met and thus this approach does not represent an optimal solution.

b) Inventor to *Pack and Go* and back

In the next attempt, the export and import using the *Pack and Go* tool will be investigated in more detail. Here, the entire assembly is bundled into one *.zip* file, allowing an easier handling.

The export of the assembly into a bundled *.zip* file is done according to the instruction [13] via *File-Save as-Pack and Go*. Here, in addition to the individual components, additional information of the project data is also exported.

For import this *.zip*-file only has to be extracted and afterwards can be opened via *File-Open*.

Because of this simple handling, exporting and importing is done in a short time and without losing any information between two instances of *Inventor*. However, by keeping the native file formats, the wide support in other CAD software is missing and as a result this way is also not an ideal solution for open data exchange.

c) Inventor to *COLLADA* and back

Direct export and import of *COLLADA* files is currently not officially supported by *Inventor*. Freely available but also commercial 3rd party tools partly offer a possibility to export CAD data. An example of a commercial solution for exporting from *Inventor* to *COLLADA* is shown by [14] in form of a *Inventor* plugin. Importing *COLLADA* back into the native formats of *Inventor* is a challenge which can only be solved with great effort. In this case, in practice, often the recreation of the *COLLADA* data in *Inventor* is a faster solution compared to the time-consuming import. A problem of the 3rd party software is the mostly

missing implementation of new versions of *COLLADA* resulting in the missing support of the kinematization during the export.

d) *Inventor* to *3DS Max* to *COLLADA* and back

CAD data from *Inventor* can also be exported to the desired *COLLADA* format via an intermediate step using the *3DS Max* software. The advantage of this compared to the previous approach is the native toolchain of *Autodesk* with the avoidance of 3rd party tools, but with the limitation of an export of the pure geometry. However similar remains the problem with a practicable support for the export but missing possibilities for the import of *COLLADA* files.

For example, opening *Inventor* files in *3DS Max* is done using the guide [15]. Further, a *COLLADA* file can be then exported using *File-Export-Export*.

In the same way, importing a *COLLADA* into *3DS Max* is done via *File-Import-Import* without any major difficulties. However, the next step, consisting of exporting to an *Inventor* file from *3DS Max*, is not supported. For this reason, this method is also not a satisfying solution.

4.3.2. Behaviour Model

Due to the wide support of the *FMI* format and the thus already good establishment in the industry, no alternative formats for the modeling of the physical behavior are tested. What is tested, however, is the practical use by means of an example with the given software. For this, a simple model consisting of a PT_1 element is created in *Simulink* and exported to *FMI* format. This model in *Simulink* is shown in Fig. 4.5.

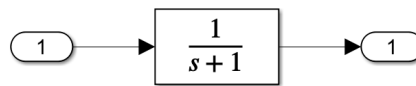


Figure 4.5.: Example for testing behaviour model

The practical implementation of a virtual commissioning using a *FMI* file and the described software is tested under different methods. For this, an overview of the methods can be found in Fig. 4.6 and the results are summarized in Tab. 4.3.

Table 4.3.: Results of tested methods in integrating behaviour model.

| Method | Working with origin model | Working with <i>FMI</i> model |
|-------------------------------------|---------------------------|-------------------------------|
| a) Model directly in <i>TwinCAT</i> | - | (no) |
| b) Model to <i>TcCom-Object</i> | yes | no |
| c) Model to <i>PLCopen</i> | yes | no |
| d) Run Model in <i>Simulink</i> | yes | yes |

a) Run *FMI-Unit* directly in *TwinCAT*

The first approach is also in this case the direct use of the *FMI* object in *TwinCAT*. This offers the advantage that the model can be executed directly on the PLC and thus a real-

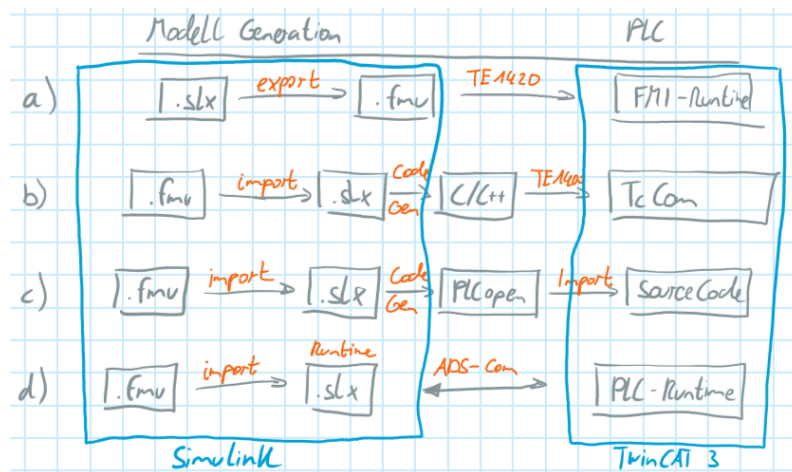


Figure 4.6.: Overview of tested methods for FMI handling.

time capability is given without any additional steps. Furthermore possible restrictions and problems can be avoided when using 3rd party tools. For this purpose *Beckhoff* offers the product *TE1420* to implement the *FMI* interface [16]. At the time of writing this thesis, the product is still under development and there is no documentation available on how to use it.

For this reason, this method is rejected, but should be investigated again in the near future.

b) FMI import in *Simulink* and export to *TwinCAT*

In this approach the target for *Simulink* of *TwinCAT* is analyzed. This target is part of the product *TE1400* and allows the integration of *Simulink* models into *TwinCAT* by using the *Simulink coder*. In this process, C/C++ code is generated from the model in the first step and further transformed into a *TcCom* object in *TwinCAT*. When using the free trial version of *TE1400*, attention must be paid to the limitation of a maximum of five input and five output variables [17]. However, this limitation does not apply to the commercial version.

The integration of the model in *TwinCAT* using this method is done similar to the example *SimpleTemperatureControl* from the documentation of the product *TE1400*. [18].

For this the *System target file* must be set to *TwinCatGrt.tlc* in the code generation settings. The remaining settings of the code generation can be kept, but the solver must be set to the type *fixed-step* with the step size equal to the task time of the PLC. After successful code generation the *TcCom* object must be signed as described in the instructions. Afterwards it can be included and used in *TwinCAT*.

In the same way, the model with the *FMI* object is processed. However, code generation fails due to missing support of the *FMI* block in the part of *code generation* of *Simulink*. The export of the original model works without problems after initial difficulties with the signing and the limitation of the variables. In contrast, the export of the *FMI* object fails. As a result, this method is not a suitable solution for the given software in virtual commissioning.

c) *FMI* import in *Simulink* and export to *PLCopen*

Similar to the previous approach, the code generation of *Simulink* is further investigated here, but with the difference of the target platform. While in the previous approach C/C++ code was generated from the model, here it will be exported directly to the *PLCopen* format. This can be then easily integrated and used in *TwinCAT* in the next step.

The export to *PLCopen* format via *Simulink PLC Coder* is done analog to the example *Generate Structured Text Code for a Simple Simulink Subsystem* [19] also first for the original model and then for the model with the *FMI* object.

For the PLC coder, an additional modification to the original model is required. The continuous transfer function is not supported and must therefore be discretized first with the sampling time $t_s = 0.01$ s. This results in the discrete transfer function $G(z)$:

$$G(s) = \frac{1}{s+1} \quad \Rightarrow \quad G(z) = \frac{0.00995}{z-0.99} \quad (4.1)$$

This time-discrete model can now be exported without major issues, like in the method before. Additional settings besides the *fixed-step* solver are not necessary and the model can now be included and used as *function block* in *TwinCAT*.

As already suspected, the export of the model with the *FMI* object is not possible. As before, the *FMI* block is not supported in the code generation of the *Simulink PLC Coder*. This can also be seen in the list of supported blocks [20].

Like the previous approach, this solution is also rejected for this reason.

d) *FMI* import in *Simulink* and communication with *TwinCAT*

This approach differs from the previous ones by the location of the model run-time. Compared to the other approaches, it is not located directly on the PLC, but remains in *Simulink* on the engineering system. However, this also results in additional communication between the PLC and the used engineering system. For time-critical applications, real-time capability for this engineering system is also recommended by means of using special hardware and software.

In this case the communication to the PLC is done via the *ADS* interface of *Beckhoff*. This communication is integrated via special blocks in *Simulink* and is here done via the block *TwinCAT Symbol Interface* [21]. These blocks are part of the product *TE1410* which contains the interface to *Simulink* and also in this product the limitation of the number of variables in the test version has to be considered [22].

After the configuration of the *interface* block the PLC variables are available as source/sink in *Simulink* and can be connected to the transfer function. The solver must also be set to *fixed-step* again with the same step size as the task time of the PLC.

With this method the model can be used in its original form and exported as *FMI* object. Thus inputs can be read from the PLC and outputs can be written. This means that this method is not optimal and has some disadvantages, but it is a suitable way to integrate *FMI* objects into *TwinCAT* and therefore to perform a virtual commissioning.

4.3.3. PLC Code

Since the *PLCopen* format is well established in the industry and thus a uniform interface for the exchange of PLC code has been created, there is no need to consider alternative possibilities.

The export and import is tested using a small example. This example consists of a counter that is incremented on each call and is listed in the *PLCopen* format in Sourcecode 4.3.1.

Sourcecode 4.3.1: Example counter as *PLCopen-xml*

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <project xmlns="http://www.plcopen.org/xml/tc6_0200">
3    <fileHeader companyName="Beckhoff Automation GmbH" productName="TwinCAT PLC Control" productVersion="3.5.13.21"
4      creationDateTime="2022-06-01T19:40:54.4880245" />
5    <contentHeader name="mainPLC" modificationDateTime="2022-06-01T19:40:54.4900208">
6      <coordinateInfo>
7        <fbid>
8          <scaling x="1" y="1" />
9        </fbid>
10       <ld>
11         <scaling x="1" y="1" />
12       </ld>
13       <sfc>
14         <scaling x="1" y="1" />
15       </sfc>
16     </coordinateInfo>
17     <addData>
18       <data name="http://www.3s-software.com/plcopenxml/projectinformation" handleUnknown="implementation">
19         <ProjectInformation />
20       </data>
21     </addData>
22   </contentHeader>
23   <types>
24     <dataTypes />
25     <pous>
26       <pou name="MAIN" pouType="program">
27         <interface>
28           <localVars>
29             <variable name="increment" address="%I*">
30               <type>
31                 <INT />
32               </type>
33               <initialValue>
34                 <simpleValue value="0" />
35               </initialValue>
36             </variable>
37             <variable name="sum" address="%Q*">
38               <type>
39                 <INT />
40               </type>
41               <initialValue>
42                 <simpleValue value="0" />
43               </initialValue>
44             </variable>
45           </localVars>
46         </interface>
47         <body>
48           <ST>
49             <xhtml xmlns="http://www.w3.org/1999/xhtml">sum := sum + increment;</xhtml>
50           </ST>
51         </body>
52       <addData>
53         <data name="http://www.3s-software.com/plcopenxml/interfaceasplaintext" handleUnknown="implementation">
54           <InterfaceAsPlainText>
55             <xhtml xmlns="http://www.w3.org/1999/xhtml">PROGRAM MAIN
56               VAR
57                 increment AT %I* : INT := 0;
58                 sum AT %Q* : INT := 0;
59               END_VAR
60             </xhtml>
61           </InterfaceAsPlainText>
62         </data>
63         <data name="http://www.3s-software.com/plcopenxml/objectid" handleUnknown="discard">
64           <ObjectId>05518f30-63f5-43e5-a500-2e4c842868d1</ObjectId>
65         </data>
66       </addData>
67     </pous>
68   </types>
69   <instances>
70     <configurations />
71   </instances>
72   <addData>
73     <data name="http://www.3s-software.com/plcopenxml/projectstructure" handleUnknown="discard">
74       <ProjectStructure>
75         <Object Name="MAIN" ObjectId="05518f30-63f5-43e5-a500-2e4c842868d1" />

```

```

76 |         </ProjectStructure>
77 |     </data>
78 | </addData>
    | </project>

```

TwinCAT 3 to PLCopen

The export and import of *PLCopen* objects is supported in *TwinCAT* and is done without difficulty. Instructions for this can be found in [23].

4.4. Workflow of the proposed Data Structure

The workflow when using the proposed data structure consists of two main parts: generating the data (export) and using the data in a virtual commissioning (import). The export and import respectively consist of the subcategories of the CAD data, the behaviour model and the control code. The whole workflow is shown in the overview in Fig. 4.7.

Compared to the import, the export of the data is more straightforward and thus easier to accomplish. The data can usually be generated in just a few steps, with CAD data being the exception to this general rule. In principle, the greatest effort is required when using the data structure in the field of CAD data, due to the lack of an interface for saving the kinematics of an assembly. If only the geometry should or can be transferred, a neutral format like *.step* or *.iges* is recommended, which are supported by most of the CAD tools. Until the *COLLADA* format is established in industry, exporting kinematics is done easiest in native CAD formats. The choice of the used software should be made in a close cooperation between the supplier and the customer and thus depends on the specific application.

In comparison, the export of the behaviour model, the control code of the PLC and, if necessary, a technical documentation do not cause major problems and are supported by the majority of available software.

The process of importing the data, in contrast to exporting, is more complex, where the individual steps strongly depend on the used target software. In the case of CAD data, it is necessary to distinguish between three possible approaches: the assembly is not kinematized (for example *.step*), the assembly is kinematized in neutral format (for example *COLLADA*) and the assembly is kinematized in a native format (for example *.iam*). The native format provides the least effort for integration, followed by plain geometry without kinematization (depending on the complexity of the assembly). The greatest effort is in importing a *COLLADA* file, due to the lack of conversion tools. After importing the geometry and recreating the kinematization if needed, communication with the PLC must be established. The goal of this communication is to visualize the current status of the PLC. For this purpose, the plug-in functionality of the various CAD software in combination with existing interfaces of the PLC often provides a suitable solution.

The import of the PLC code is done in the majority of PLC systems without any difficulties thanks to the international standardization of the programming language. As a result sample code can be integrated into an existing project fast and without any difficulties.

However, with the behaviour model, a dependency to the used software is once again recognizable. In the best case, the *FMI* object can already be executed directly in the

PLC and linked to the inputs and outputs of the hardware. Alternatively, the model can be executed in software for simulations, which again communicates with the PLC. If virtual commissioning is performed using this method, special hardware must often be used to ensure real-time capability. However, this requirement only applies to time-critical processes.

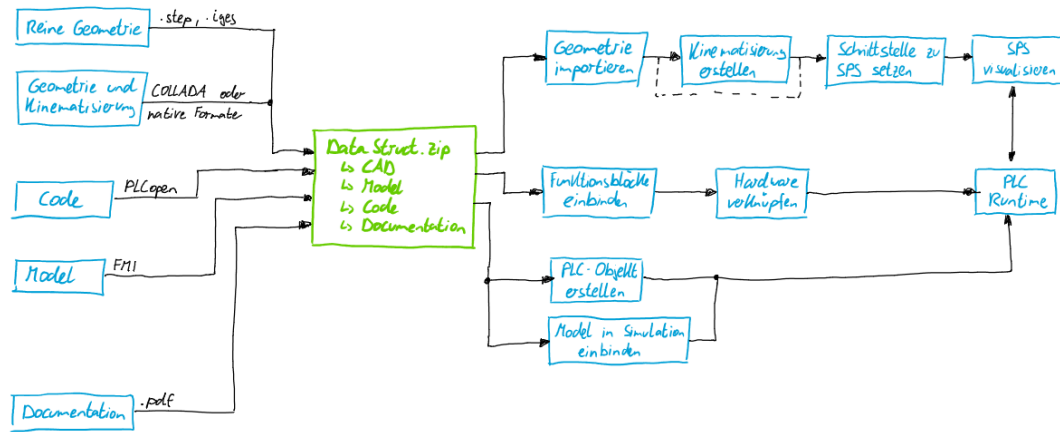


Figure 4.7.: General Workflow

4.5. Summary

This chapter describes the structure and use of a modular data structure. The field of application includes the data exchange for virtual commissioning, where the control is done via a PLC.

The foundation of this structure is a single root folder containing the CAD data (native format or *COLLADA*), the behavior model (*FMI*), the control software (*PLCopen*) and the associated documentation (*.pdf*). Compressing it into a *.zip* file simplifies distribution and sharing.

Finally, the general workflow in using this data structure is explained. Thereby the general steps in exporting the data and the following import are examined and described in more detail.

5. Exemplary Application using the proposed Data Structure

5.1. Introduction

The practical use and the individual steps in the implementation of the proposed data structure are shown by means of an example. For this purpose a section of a *Teaching Factory* is used. The purpose of this plant is to get familiar with automation technology with the help of a PLC and its programming. The plant is modular in order to provide individual stations with defined learning goals. The learning goals include basic topics like digital and analog signals but also more complex tasks like axis control and serial communication. Furthermore, human interaction and safety are included learning contents. An overview of the whole plant is shown in Fig. 5.1 and consists of following modules:

Separation: The sub-module separation covers the area of digital signals. This task involves the controlled positioning of containers from the input storage area onto the conveyor belt.

Filling: In this module, the containers are moved along a conveyor belt and in this process the control of a DC motor can be developed. A dosing unit with stoppers on the conveyor belt and sensors is located at three defined positions along the belt. At these three points the containers can be filled with different or the same granulate. On these three locations the containers can be filled with different or the same granulate. At the end of the conveyor belt, the containers are picked up by the next station.

Gripper: The gripper is responsible for the further transport of the containers and picks them up from the end of the conveyor belt. Here, the main topic of motor control is further investigated. For example, the origin of each axis must be determined in order to ensure that the containers are placed in the exact position.

Load cell: After filling, the weight and thus the filling level of the containers is measured. For this purpose, the container are placed by the gripper on a load cell. After that the load cell transmits the measured value to the PLC via a serial interface. Accordingly, the learning objective in this case is serial communication.

Thermal processing: After the filling level has been determined, heat treatment of the containers is simulated. This is done by activating a heater via an analog signal and controlling the temperature to a defined value. This task includes the learning objective of analog signals and applied control engineering.

Container identification: The containers can be uniquely identified via RFID tags. These tags are attached to the bottom of the containers and have a unique serial number. This module covers the learning goal of component identification and serial communication with the reader of the RFID system.

Output storage: This storage area is the final stage of the process. Here the user can take out the filled containers. This station does not offer any specific learning objectives.

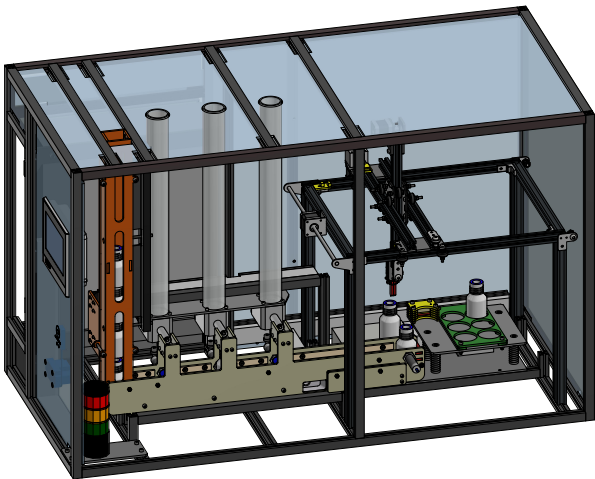


Figure 5.1.: The used Teaching Factory Aktuelles Bild verwenden

In the following section, the *separation* and *filling* stations are discussed in more detail and the virtual commissioning is performed. A list of the used software can be found in Tab. 5.1.

Table 5.1.: Used software in the example *Teaching Factory*.

| Domain | Software | Comments |
|---------------------|--------------------|----------------------------|
| CAD | Autodesk Inventor | Version: Professional 2022 |
| Behaviour Modelling | Matlab / Simulink | Tools: Simulink Coder |
| PLC | Beckhoff TwinCAT 3 | Tools: TE1400, TE1410 |

The setup of the individual components of a virtual commissioning of this example can be found in Fig. 5.2.

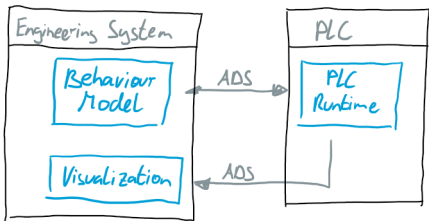


Figure 5.2.: Used setup for a virtual commissioning

5.2. Visualization

The current state of the PLC software and consequently the state of the plant is visualized in this example directly in the CAD software *Inventor*. This implementation is done

as a plugin in *Inventor*, where the objective is a pure visualization of the state. The calculations and the user inputs are done directly in the PLC and in the defined task time. Thus the system as a whole remains real-time capable and, for example, controls can be implemented and tested under realistic conditions.

A disadvantage of this kind of implementation is the strongly limited frame rate of the graphic output in the range of about 1 Hz depending on the complexity of the assembly. This results from the required time for an update of the CAD model, which has to be done for every new value of the PLC. Only in this way the current state is shown and not old values. However, this problem only affects the graphical output and has no influence on the control itself. To avoid this problem, *Beckhoff* is currently developing a target for CAD software with the name *TE1130* [24]. With this target it should be possible to set the position of components in the software depending on a variable in the PLC and the update of the assembly will be optimized. As a result, a faster frame rate should be possible.

The communication with the PLC is done via the *ADS* protocol from *Beckhoff*. In this communication the variables are read from the PLC and written as user parameters in *Inventor*. The variables of the PLC contain the signals to and from the hardware.

If, for example, a piston is extended in the controller, which corresponds to setting a Boolean, the piston should also be extended in the CAD model. For this purpose, the relative position of the piston is linked to a user parameter in the CAD and set accordingly.

5.3. Module Separation

5.3.1. Introduction

This module offers a quick introduction to the programming of a PLC. The main topic here is working with digital inputs and outputs via reading sensor signals and setting simple pistons. The mechanical design of this station is vertically oriented and consists of an input storage in the upper area and the separation in the lower area. A total of six bottles can be picked up and separated as required.

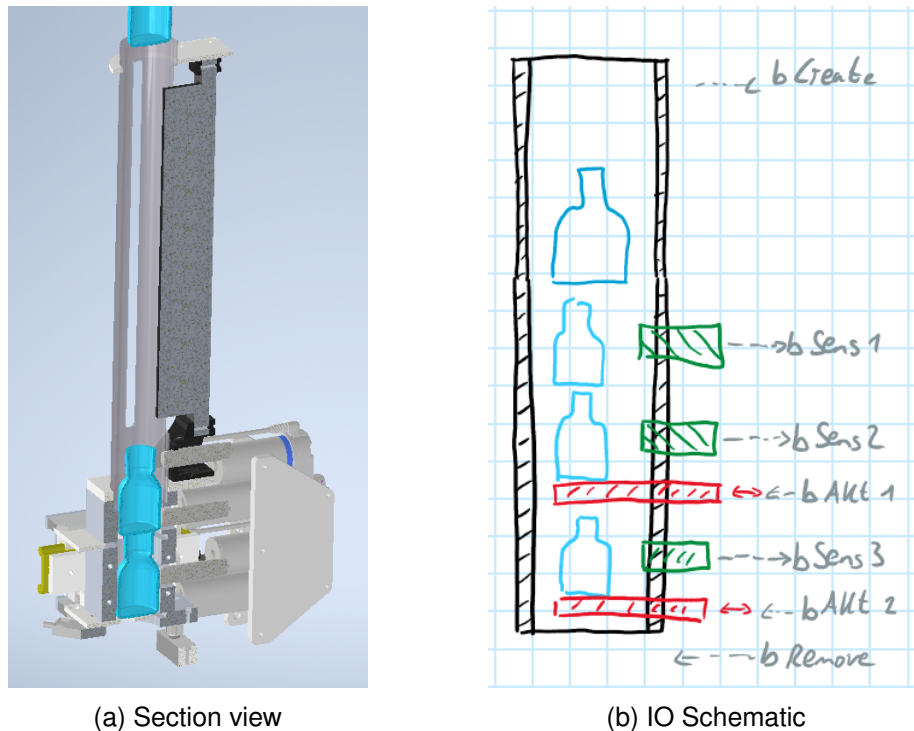
On each of the lower three storage positions a capacitive proximity sensor with a digital output is located. This sensor can be used to detect a container on the corresponding position where in this case the output signal is set.

Above and below the lowest position is a piston attached, which is needed for the actual process of separation. These pistons are moved by a linear motor and controlled by a digital retract or extend signal. In the extended state, the containers can pass the piston undisturbed, while in the retracted state they are stopped at the piston.

The mechanical layout of the station and the scheme of the available signals of the hardware is shown in Fig. 5.3.

5.3.2. Generate Data Structure

In this section the data structure for the exchange is generated. This structure consists of the CAD data, the behavior model and an exemplary control as described in chapter 4. The export of the data is done from the view of the supplier or the transmitter.

Figure 5.3.: Module Separation **In CAD beschriftet**

CAD Data

The design of the separation is done in *Inventor* in this example. The CAD model is built according to the desired kinematics, with the sliders of the two pistons each representing a separate sub-assembly. The remaining components can be grouped in a separate assembly. Thus, the overall assembly *Separation* consists in total of three sub-assemblies, which simplifies the kinematicization significantly. Also possible changes to the design and kinematization can be easily applied with this hierarchy of components.

The *Pack and Go* tool from *Inventor* is now used to export the final CAD data. This tool gathers all relevant components of the assembly and bundles them into one folder. Additionally, this folder can be automatically compressed to a .zip file, as is done in this case.

Behaviour Model

The behavior model in this case is created in *Simulink* and takes into account the influence of gravity on the containers in the incoming storage. Also taken into account is the influence of the actuators on the selected position of the containers in the storage and the resulting signal from the sensors.

The interface of the model consists of the designated inputs and outputs of the PLC to the hardware. This ensures that the software can be tested as close as possible to reality and possible errors in the software can be corrected at an early stage.

For the *separation*, the interface consists of two digital outputs each for moving the two actuators in and out, and one digital input for each of the three proximity sensors. Fur-

thermore, additional signals are needed for the creation and deletion of the containers. These signals are purely virtual and are thus not connected to the hardware, but are required only in the logic of the station.

The creation of the containers is equivalent to the insertion of a container into the input storage and is done in reality by the human user. If a rising edge is now detected by this signal, a container is created in the top storage location. According to the logic of the model, this container then falls down to the lower storage locations and the sensor signals are set.

The removal of the containers corresponds to the transfer to the next module of the *filling* and must be set by a higher-level controller. This means that as soon as a container is placed on the lowest position, the lower piston is opened and a rising edge is detected at this signal, this container is removed in the logic of the *separation*. This signal can be used to simulate the placement of a container on the conveyor belt of the next station, whereby the conveyor belt is not in motion and thus the container is not physically removed.

The model of the *separation* itself consists of the logic that determines the behavior and thus the output signals of the sensors and the models of the two actuators. The logic is created graphically using logic gates. For this purpose, each storage position is represented by a *flip-flop* element. For modeling the actuators, the two signals for retraction and extension are processed and a virtual end switch is set accordingly. These end switches are further used in the logic of the module and do not exist in reality. The entire model with the inputs and outputs is shown in Fig. 5.4.

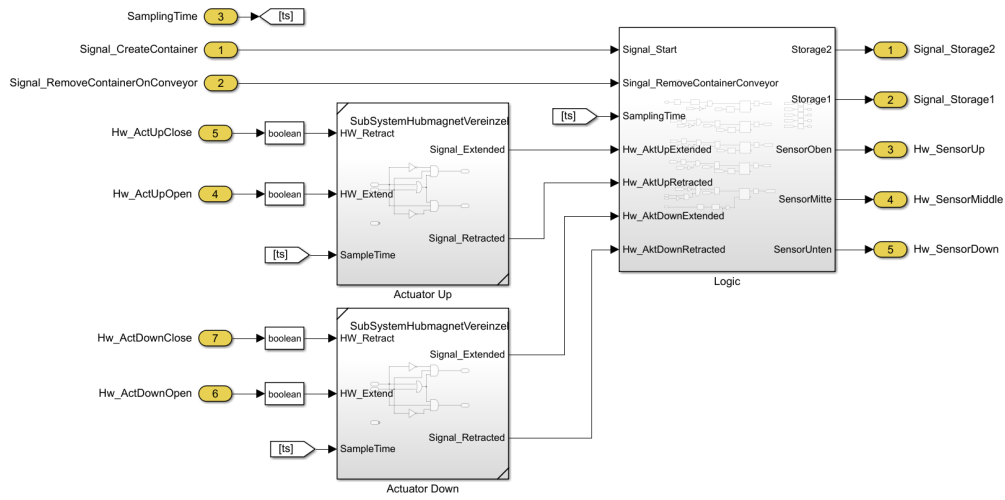
As described in chapter 4, the *Simulink* model must be exported to a *FMI* file, in order to create an interface that is as easily accessible as possible. This export is done according to the instructions [9] and results in a *.fmu* file. For a successful export of the model the solver must be set to *fixed-step* and the step size must correspond to the task time of the PLC or at least be dividable by it.

PLC Code

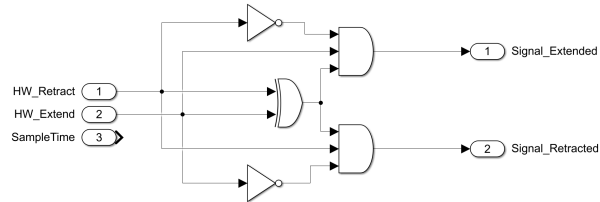
For the exemplary control of the separation a *function block* is created in *TwinCAT*. By structuring the control as a *function block*, an instance of the separation can easily be created and used in a higher-level controller. This code basically consists of a state machine in which the two pistons are controlled depending on the current state and the sensor inputs. The user has the possibility to start the separation process. After the software has been created, the function block is exported to the *PLCopen* format. This is done according to the [23] instructions, with the resulting *.xml* file listed in Sourcecode A.1.1.

Resulting File

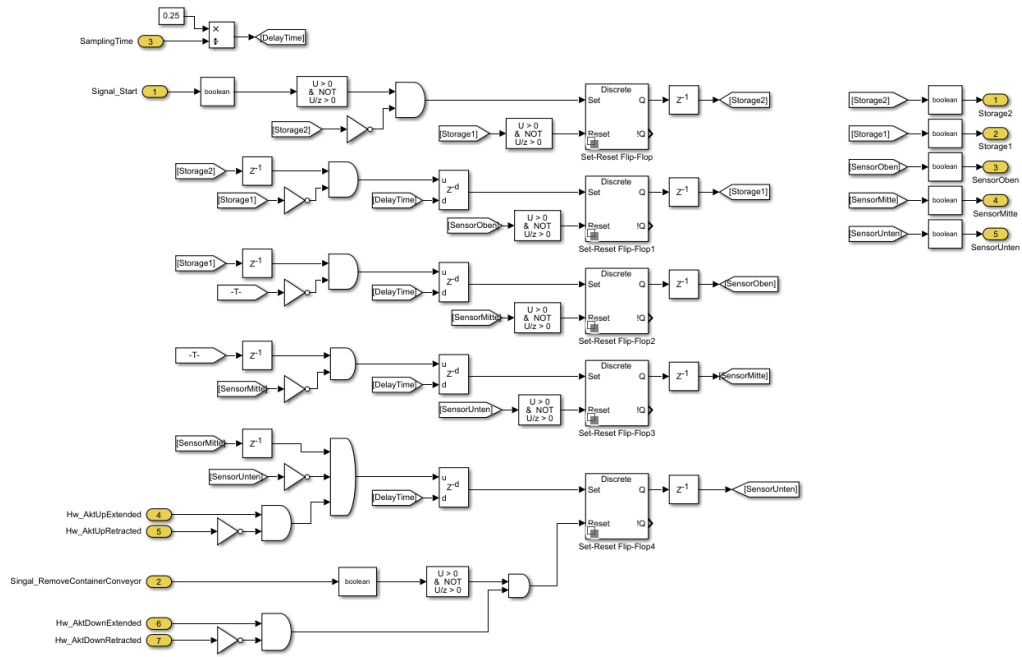
The generated information of the CAD model, behavior and control is now bundled in the proposed data structure from chapter 4. The resulting structure for this example is shown in Fig. 5.5.



(a) Overall model

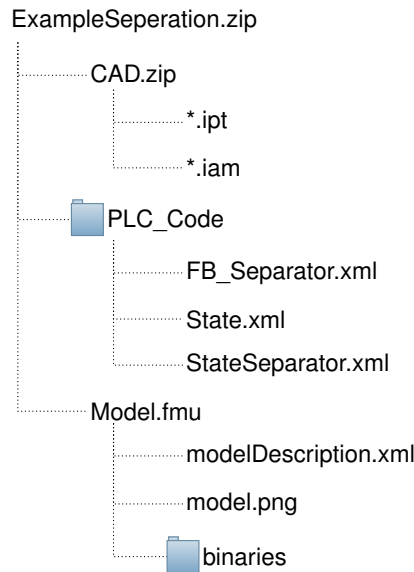


(b) Actuator



(c) Logic

Figure 5.4.: Behaviour model in *Simulink*

Figure 5.5.: Data structure for Example *Seperation*

5.3.3. Use Data Structure

In this section, the data structure shown in Fig. 5.5 is now imported and used in a virtual commissioning. This is done from the perspective of the customer or receiver.

CAD Data

In this example, the receiver is also using the CAD software *Inventor* and therefore the use of native file formats is possible. Since the CAD data was generated via the *Pack and Go* tool, it can easily be imported by the customer. To do this, only the *.zip* folder containing the CAD data needs to be extracted and can then be used immediately to design the remaining plant. By keeping the native formats, the kinematization of the assembly is preserved and as a result, there is no need to recreate the kinematization, due to the possible loss of data when using an unsupported data format.

Behaviour Model

The implementation of the behavior of the station *Separation* for a virtual commissioning is done in *Simulink*, as described before. The *FMI* object is integrated in *Simulink* and communicates with the PLC runtime via the *ADS* interface. The model is executed separately from the PLC on an engineering system.

The final setup in *Simulink* for virtual commissioning is shown in Fig. 5.6. It consists of the *FMI* object with the implementation of the behavior in the middle and the *ADS* communication block in the upper left. In addition, two functions are used to compress the digital signals. For this purpose, the boolean signals to the actuators are combined in one byte by setting corresponding bits. In the same way, the three signals from the sensors are also combined in a second byte, with one bit representing each sensor signal. With this compression, the number of variables for communication and thus the required time can be minimized. A further side effect of this is the possibility of bypassing the limitation

of the test license with respect to the maximum number of variables for communication.

The solver must also be set to type *fixed-step* with the same step size as the task time of the PLC, which in this case is 10 ms.

In the *ADS* block the variables for communication are set according to their data direction. For this purpose the sensor signals are written to the PLC via the command *ADS-Write* and the signals of the actuators are read from the PLC via *ADS-Read*.

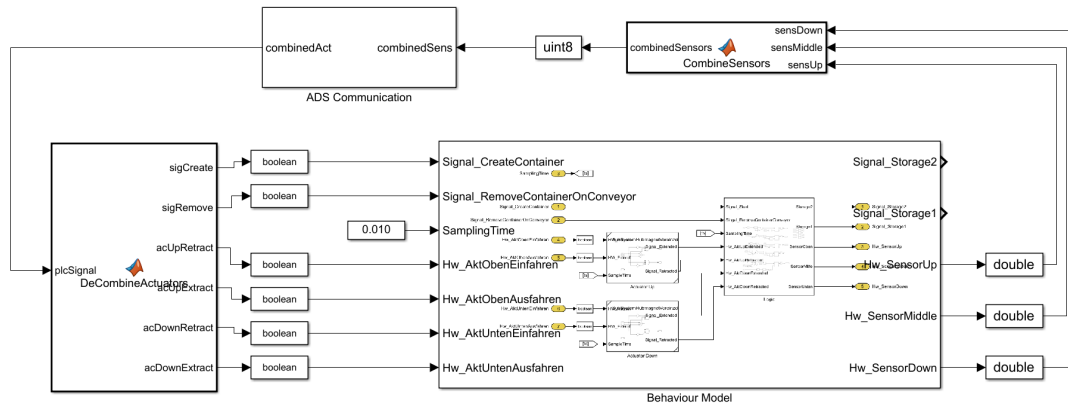


Figure 5.6.: Use of the behaviour model in *Simulink* for example *Separation*

PLC Code

The integration of the exemplary control of the separation in *TwinCAT* takes place in a higher-level program, which is listed in Sourcecode 5.3.1. As part of this control, the input and output variables are defined and linked to the instance of the separation function block. Then this instance must be called periodically to determine the current state. Besides this call, the compression of the input and output variables is also part of this higher-level control. As already described, the signals of the sensors and actuators are each compressed to one byte.

Finally, the virtual commissioning of the station can now be performed by testing the software.

Sourcecode 5.3.1: Main programm with included example code of example *Separation*

```

2 <?xml version="1.0" encoding="utf-8"?>
3 <project xmlns="http://www.plcopen.org/xml/tc6_0200">
4   <fileHeader companyName="Beckhoff Automation GmbH" productName="TwinCAT PLC Control" productVersion="3.5.13.21"
5     creationDateTime="2022-06-05T16:15:56.4832631" />
6   <contentHeader name="main" modificationDateTime="2022-06-05T16:15:56.4852631">
7     <coordinateInfo>
8       <fbid>
9         <scaling x="1" y="1" />
10      </fbid>
11      <ld>
12        <scaling x="1" y="1" />
13      </ld>
14      <sfc>
15        <scaling x="1" y="1" />
16      </sfc>
17    </coordinateInfo>
18    <addData>
19      <data name="http://www.3s-software.com/plcopenxml/projectinformation" handleUnknown="implementation">
20        <ProjectInformation />
21      </data>
22    </addData>
23  </contentHeader>
24  <types>
25    <dataTypes />
26    <pous>

```

```

26 <pou name="MAIN" pouType="program">
  <interface>
    <localVars>
28      <variable name="ModuleSeperationSensors" address="%I">
        <type>
30          <BYTE />
        </type>
32        <initialValue>
          <simpleValue value="0" />
        </initialValue>
34        <documentation>
          <xhtml xmlns="http://www.w3.org/1999/xhtml"> 0000_0dmu</xhtml>
36        </documentation>
      </variable>
38      <variable name="ModuleSeperationActuators" address="%Q">
        <type>
40          <BYTE />
        </type>
42        <initialValue>
          <simpleValue value="0" />
        </initialValue>
44        <documentation>
          <xhtml xmlns="http://www.w3.org/1999/xhtml"> 0000_00rc</xhtml>
46        </documentation>
      </variable>
50      <variable name="sensUp">
        <type>
52          <BOOL />
        </type>
54      </variable>
56      <variable name="sensMiddle">
        <type>
58          <BOOL />
        </type>
60      </variable>
62      <variable name="sensDown">
        <type>
64          <BOOL />
        </type>
66      </variable>
68      <variable name="sigCreate">
        <type>
70          <BOOL />
        </type>
72      </variable>
74      <variable name="sigRemove">
        <type>
76          <BOOL />
        </type>
78      </variable>
80      <variable name="actUpOpen">
        <type>
82          <BOOL />
        </type>
84      </variable>
86      <variable name="actUpClose">
        <type>
88          <BOOL />
        </type>
90      </variable>
92      <variable name="actDownOpen">
        <type>
94          <BOOL />
        </type>
96      </variable>
98      <variable name="actDownClose">
        <type>
100          <BOOL />
        </type>
102      </variable>
104      <variable name="moduleSeperation">
        <type>
106          <derived name="FB_Separator" />
        </type>
108      </variable>
110      </localVars>
    </interface>
    <body>
      <ST>
114        <xhtml xmlns="http://www.w3.org/1999/xhtml"> // Converting Inputs
116        sensUp := BYTE_TO_BOOL(ModuleSeperationSensors AND 2#0000_0001);
        sensMiddle := BYTE_TO_BOOL(ModuleSeperationSensors AND 2#0000_0010);
        sensDown := BYTE_TO_BOOL(ModuleSeperationSensors AND 2#0000_0100);

```

```

118
120 // Instance calls
moduleSeperation(
122   bEnable:= seperationEnable ,
124   bExecute:= seperationExecute ,
126   bErrorReset:= ,
128   bSensor1:= sensDown ,
130   bSensor2:= sensMiddle ,
132   bSensor3:= sensUp ,
134   bR1AKT1=&gt; actUpClose ,
136   bR2AKT1=&gt; actUpOpen ,
138   bR1AKT2=&gt; actDownOpen ,
140   bR2AKT2=&gt; actDownClose ,
142   bNoBottle=&gt; , bWrongBottle=&gt; , eStatus=&gt; );
144
146 // Converting Outputs
ModuleSeperationActuators := 0;
ModuleSeperationActuators := BOOL_TO_BYTE(sigCreate) OR ModuleSeperationActuators;
ModuleSeperationActuators := SHL(BOOL_TO_BYTE(sigRemove),1) OR ModuleSeperationActuators;
ModuleSeperationActuators := SHL(BOOL_TO_BYTE(actUpOpen),7) OR ModuleSeperationActuators;
ModuleSeperationActuators := SHL(BOOL_TO_BYTE(actUpClose),6) OR ModuleSeperationActuators;
ModuleSeperationActuators := SHL(BOOL_TO_BYTE(actDownOpen),5) OR ModuleSeperationActuators;
ModuleSeperationActuators := SHL(BOOL_TO_BYTE(actDownClose),4) OR ModuleSeperationActuators;
144 </xhtml>
146 </ST>
148 </body>
150 <addData>
152   <data name="http://www.3s-software.com/plcopenxml/interfaceasplaintext" handleUnknown="implementation">
154     <InterfaceAsPlainText>
156       <xhtml xmlns="http://www.w3.org/1999/xhtml">PROGRAM MAIN
158       VAR
160         ModuleSeperationSensors   AT%I*   : BYTE := 0;    // 0000_0dmu
162         ModuleSeperationActuators AT%Q*   : BYTE := 0;    // 0000_0orc
164         sensUp, sensMiddle, sensDown : BOOL;
166         sigCreate, sigRemove        : BOOL;
168         actUpOpen, actUpClose       : BOOL;
170         actDownOpen, actDownClose   : BOOL;
172         moduleSeperation            : FB_Separator;
174         seperationEnable, seperationExecute : BOOL;
176       END_VAR
178     </xhtml>
180     </InterfaceAsPlainText>
182   </data>
184   <data name="http://www.3s-software.com/plcopenxml/objectid" handleUnknown="discard">
186     <ObjectId>141fae0c-c78f-4672-9a75-26682aa93051</ObjectId>
188   </data>
190 </addData>
192 </pou>
194 </types>
196 <instances>
198   <configurations />
200 </instances>
202 <addData>
204   <data name="http://www.3s-software.com/plcopenxml/projectstructure" handleUnknown="discard">
206     <ProjectStructure>
208       <Object Name="MAIN" ObjectId="141fae0c-c78f-4672-9a75-26682aa93051" />
210     </ProjectStructure>
212   </data>
214 </addData>
216 </project>

```

5.3.4. Result

This example has shown how the proposed data structure can be used in practice. Therefore the virtual commissioning of a separation was done, where the behavior model is executed in *Simulink*. Via *ADS* communication the signals of the hardware were set in the PLC and read by the PLC.

5.4. Module *Conveyor Belt*

5.4.1. Introduction

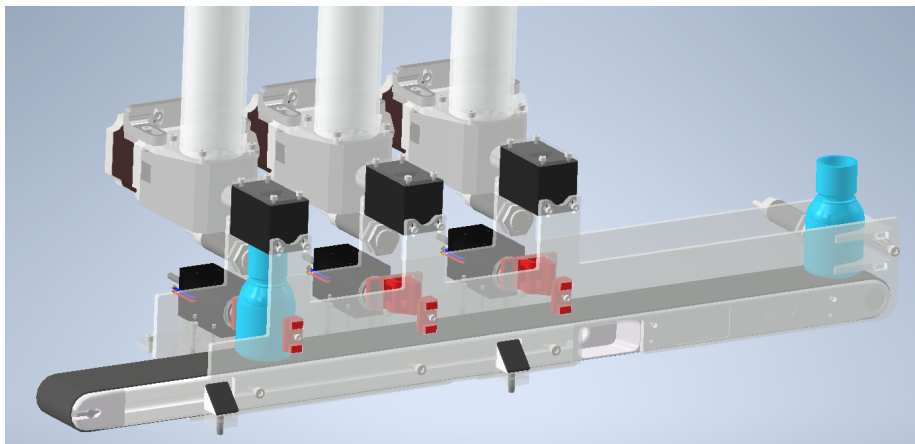
This module is the user's first introduction to the control of electrical drives. In order to provide a quick and easy introduction, a DC motor driving a conveyor belt is controlled. In a simplified approach, the velocity of the motor is dependent on the input voltage and the torque is dependent on the current. This allows a relatively easy implementation of a closed loop control and additionally the theoretical aspect of the motor control becomes simple to understand. An encoder is also attached to the motor axis in order to determine the real velocity and position of the motor. Because of this measurement of the position, a servo control of the conveyor belt can be implemented in an additional step. The difference of a servo control compared to a standard control without encoder is the possibility to move the conveyor belt to an exact position, because the angular position of the motor axis is known. Furthermore, the encoder provides feedback about the real velocity of the motor and therefore an unwanted stop can be detected immediately.

The conveyor belt has a total length of 400 mm and is divided into the following sections: Input, Filling (three times) and Output. At the beginning of the belt conveyor, the containers are taken from the station *separation*, where they are placed on the belt. After that, the containers go through three identical possibilities of filling by means of a dosing unit. The dosing units can be filled individually and as a result use the same or a different granulate. To simplify the process of filling a container, a stopper and a proximity sensor are located below each dosing unit. The stopper is similar to those of the station *separation*, but in contrast only one digital signal is needed for controlling, which retracts the slider. If this signal is not set, the slider is extended and the containers are stopped on the conveyor. The proximity sensors are identical to those of the station *separation* and set a digital output signal if a container is detected. This combination of sensors with the stoppers and the conveyor belt allows a container to be placed precisely under a dosing unit. At the end of the conveyor belt, the containers are passed over to the next station. To ensure the transport of the containers to the next station, again a proximity sensor is located at the defined transfer position.

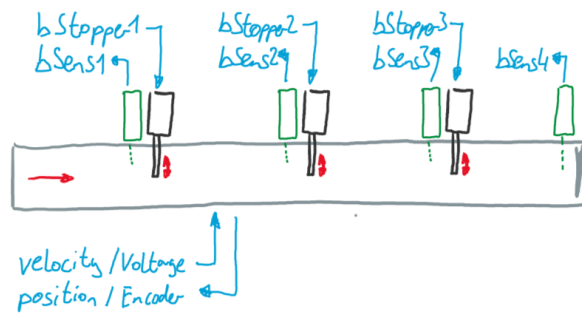
In this example, only the conveyor belt and thus the DC motor is controlled. The task for the user is the motor control consisting of: Starting, moving at a constant velocity and stopping at a defined position. The mechanical structure and the scheme of the available signals of the hardware are shown in Fig. 5.7. The control of the dosing units and the stopper with the related sensors are not covered in this example and are part of a separate learning unit.

5.4.2. Generate Data Structure

In this section, the data structure for exchange described in chapter 4 is again generated and consists of the CAD data and the behavior model. This is done in the same manner from the supplier's or the sender's point of view, who in this case only provides the hardware of the station. The hardware consists of the conveyor belt with the mounting, stoppers and the sensors. In addition, the supplier provides a simplified behaviour model of the DC motor. Since the manufacturer only offers the hardware and the behaviour



(a) Overview



(b) IO Schematic

Figure 5.7.: Module Filling In CAD beschriften

model, no PLC code is exchanged in this example. The development of the PLC code is the responsibility of the customer.

CAD Data

Same as before, the design of the station is created in *Inventor*, with the CAD model of the conveyor provided by the manufacturer as a *.step* file. This *.step* file is included in the assembly after it is imported and converted to a native format of *Inventor*. The three sliders of the stoppers form again a sub-assembly to simplify the kinematization. The conveyor itself does not need any additional kinematization beside the dependencies of the mounting, since the position of the containers on the conveyor are determined by *user-params*. For this reason, the belt does not need to move physically. The remaining components such as the conveyor itself, the attachment, and the sensors are bundled into a second assembly to avoid any obstacles to the kinematics.

In contrast to the previous example, in this station the CAD design is exported without kinematics since the supplier only provides the pure 3D geometry. This is the case, for example, when the customer and the supplier use different CAD tools and thus the native file formats are not compatible. In this situation, the customer has to create the kinematization according to his needs. For the export without the kinematization, the *.step* format is selected due to its wide compatibility and the process is done according to the instruction [11].

Behaviour Model

The behaviour model in this example consists of the DC motor and is again created in *Simulink*. The modeling of the motor in this case is similar to this example from the Matlab documentation [25], using the characteristics from the data sheet of the used motor *2342048CR* [?]. The final model of the DC motor represents a simplified modelling approach and is shown in Fig. 5.8.

The interface for the DC motor consists of the input voltage expressed in the unit V and the velocity on the conveyor belt in mm s^{-1} . Since the angular velocity of the motor axis in rad s^{-1} is used in the model of the motor, a transformation of the units into a linear velocity on the conveyor belt is performed. This transformation represents the gear of the conveyor belt and consists of a constant coefficient in *Simulink*.

The model of the DC motor consists of three parts in a feedback loop: a transfer function of the armature, a transfer function of the load and the feedback to the input voltage as a result of the back electromotive force (back EMF). This approach to modeling is kept simple, but represents the behaviour of the motor with sufficient accuracy for this purpose. However, if needed, the modeling can be further deepened and improved without problems.

The transfer function of the armature $G(s)$ transforms the input voltage V_{in} by means of the torque constant $k_m = 56.1 \text{ mN m A}^{-1}$, the inductance of the windings $L = 1.05 \text{ H}$ and the electrical resistance $R = 31.2 \Omega$ into a torque τ . This torque further acts as an input to the transfer function of the load $H(s)$ with the resulting angular velocity ω of the motor axis. The movement of the motor axis itself generates a back EMF reducing

the input voltage at the armature. This voltage is determined by the back-EMF constant $K_b = 5.87 \text{ mV min}^{-1}$ and the velocity of the motor axis, and is subtracted from the input voltage of the armature.

Using the characteristic values from the data sheet of the motor, the following transfer function of the armature $G(s)_{\text{Armature}}$ is derived and the transfer function of the load $H(s)_{\text{Load}}$ is determined based on the real behavior of the motor with:

$$G(s)_{\text{Armature}} = \frac{k_m}{Ls + R} = \frac{56.1 \cdot 10^{-3}}{1.05s + 31.2} \quad (5.1)$$

$$H(s)_{\text{Load}} = \frac{1}{Js + k_f} = \frac{1}{0.02s + 0.2} \quad (5.2)$$

With these two transfer functions and the information from the data sheet, the model of the motor is created.

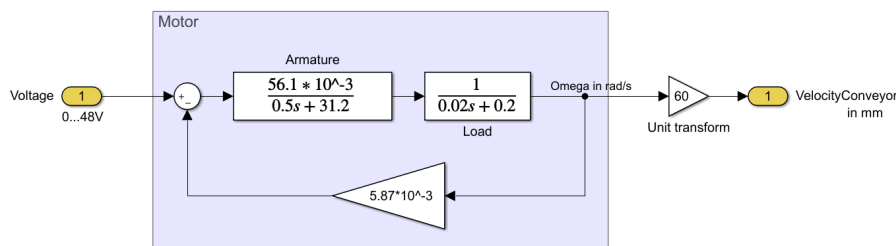


Figure 5.8.: Behaviour model of a DC-drive in *Simulink* for example *Filling*

As already shown in the previous example and described in chapter 4 the model must be exported from *Simulink*. The export is again done according to the instruction [9] and results in a *.fmu* file. Again, the solver must be set to type *fixed-step* and the step size must be dividable by the used task time of the PLC. In this case the step size is equal to the task time with 10 ms.

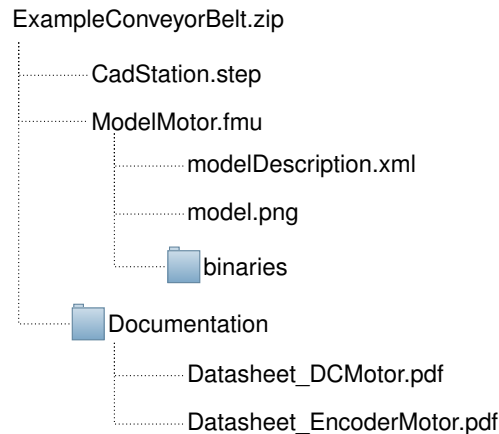
PLC Code

As already mentioned, the control of the motor in this example has to be created by the customer and is not provided by the manufacturer. For this reason, this part of the data exchange remains empty.

This scenario can also occur in practice if the supplier only produces the hardware and the creation of PLC code is not part of his product portfolio, or he does not support the *PLCopen* format. In this case it results in additional work for the customer, but does not cause any additional problems.

Resulting File

The collected information is now merged into the proposed data structure from chapter 4. This data structure is shown in Fig. 5.9 and consists of the data of the CAD assembly, the behaviour model and the documentation consisting of the data sheet of the DC motor with the encoder.

Figure 5.9.: Data structure for Example *Conveyor Belt*

5.4.3. Use Data Structure

In this section, the generated data structure from Fig. 5.9 is now imported and used in a virtual commissioning. This process is again done from the perspective of the customer or recipient of the data.

CAD Data

The CAD data in this example is only available as a non-kinematic *.step* file and as a consequence, the customer has to create the kinematics according to his own needs. The first step is to import the geometry of the station as a *.step* file into *Inventor*. The import with the following conversion into native file formats is done according to the instruction [12] and results in a native assembly in the *.iam* format with the individual parts as *.ipt* files.

After importing the geometry, the assembly must be kinematized again. For this purpose, the slide of each of the three stoppers is again combined in a separate sub-assembly. The remaining individual parts do not require any modifications, but can also be grouped into their own sub-assembly for simpler handling. The kinematics itself consists of the dependencies of the stoppers, where the slides are placed concentric to the stoppers and the position of the slides is given by a *user-param*. These parameters and with them the position will be written by the PLC later during the virtual commissioning.

Behaviour Model

Similarly to the previous example, the behaviour model for the virtual commissioning is created in *Simulink* on an engineering system and communicates with the PLC runtime via the *ADS* interface. The complete setup of the simulation in *Simulink* for the virtual commissioning is shown in Fig. 5.10.

In the simulation for the virtual commissioning, the *.fmu* file with the behaviour model of the motor is embedded. In addition to the motor, the used PLC motor terminal has to be modeled, because it generates the actual signals to the motor and interprets the signals of the encoder. This model is shown in Fig. 5.10b and consists of the conversion of the

reference velocity into a voltage signal in the range from 0 V to 48 V. Since in the real system the supply voltage of the motor is available only up to 24 V, the maximum output voltage in the model is also limited to 24 V. In the lower section, the position of the motor axis is determined by the integral of the actual velocity of the motor over time. From this calculated position a counter variable is determined. This integral calculus represents the real encoder on the motor axis in the model.

The motor control itself is done via an NC axis in *TwinCAT*. The basic principle in the use of an NC axis can be summarized as follows: Based on the configuration of the motor, the NC axis generates a profile for the acceleration and, derived from this, the velocity. With this profile, a current reference velocity is determined for each time step and sent to the PLC motor terminal. The motor terminal transforms the reference velocity into a physical signal to the motor. In this case, the velocity is transformed into a voltage between -24 V and 24 V . The encoder on the motor axis is again connected to the motor terminal and supplies a signal depending on the actual velocity. This signal is interpreted in the motor terminal and converted into the actual position of the axis. This value of the actual position is then returned to the NC axis and is taken into account in the calculation of the next reference value of the velocity. [26]

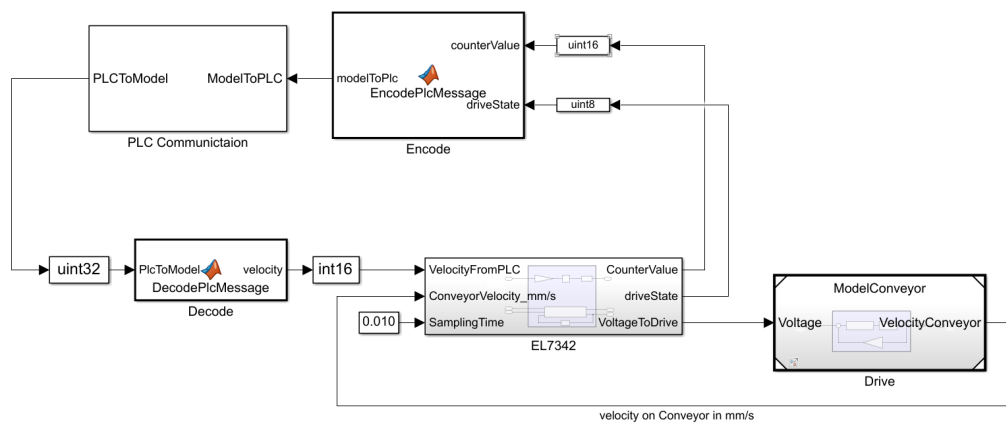
The communication with the PLC is again done via the *ADS* block in the upper left area. Also in this case the necessary signals of the communication are compressed in two variables. This means that only one variable has to be read or written per cycle, which subsequently results in a time saving. The compression and decompression of the variables for the communication is done in two Matlab functions by means of different bit operations. These operations require attention to the correct data type of the different variables, which can be seen in the various *cast* blocks in the simulation. A side effect of the compression is again the minimization of the number of variables and the thereby allowed use of the freely available test license of *TwinCAT*.

The settings of the solver for this simulation consists of setting the type *fixed-step*, where the step size again corresponds to the task time of the PLC with 10 ms . In the *ADS* block, the two variables are configured based on their direction, where the variable *PLCtoModel* is written by the PLC and thus read by the simulation with *ADS-Read* and the variable *modelToPLC* should be written by the simulation and read by the PLC with *ADS-Write*.

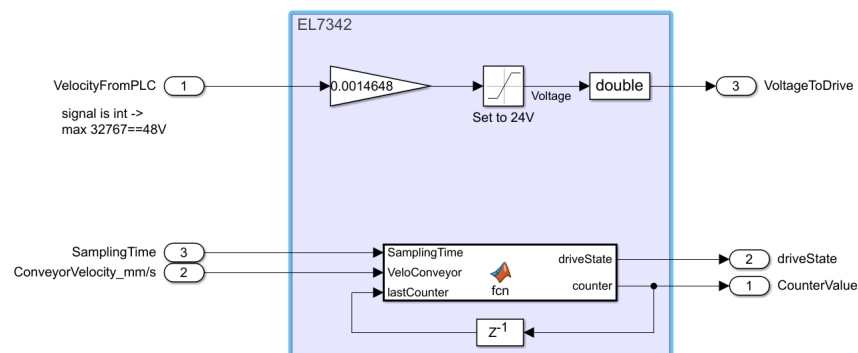
PLC Code

The code for the PLC remains very minimalistic in this example, as a result of the used NC axis. This axis can be controlled via functions in the software, but in this case the graphical interface is sufficient for simple commissioning and the first steps with the motor control. If in the next step the software of the entire station with the additional components, such as the dosing units with the stoppers and the sensors, should be created, the control of the motors in own *function blocks* is recommended. In these blocks, the NC axes are then controlled via normal commands in the source code.

Instead, the PLC software consists of compressing and decompressing the variables for communication with the simulation. For this purpose, the current reference velocity must be compressed and the actual position must be decompressed with the current state of the motor terminal. These three variables are linked to the instance of the NC axis at



(a) Complete setup



(b) Model of the used IO module

Figure 5.10.: Use of the behaviour model in *Simulink* for example *conveyor belt*

the corresponding places. The code for this example can be found in Sourcecode 5.4.1 and with this, the control of the motor can now be achieved by means of the graphical interface.

Sourcecode 5.4.1: Main program of example *Conveyor Belt*

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <project xmlns="http://www.plcopen.org/xml/tc6_0200">
3    <fileHeader companyName="Beckhoff Automation GmbH" productName="TwinCAT PLC Control" productVersion="3.5.13.21"
4      creationDateTime="2022-06-12T18:07:28.7209284" />
5    <contentHeader name="Main" modificationDateTime="2022-06-12T18:07:28.7229285">
6      <coordinateInfo>
7        <fbid>
8          <scaling x="1" y="1" />
9        </fbid>
10       <ld>
11         <scaling x="1" y="1" />
12       </ld>
13       <sfc>
14         <scaling x="1" y="1" />
15       </sfc>
16     </coordinateInfo>
17     <addData>
18       <data name="http://www.3s-software.com/plcopenxml/projectinformation" handleUnknown="implementation">
19         <ProjectInformation />
20       </data>
21     </addData>
22   </contentHeader>
23   <types>
24     <dataTypes />
25     <pous>
26       <pou name="MAIN" pouType="program">
27         <interface>
28           <localVars>
29             <variable name="velocity" address="%I*">
30               <type>
31                 <INT />
32               </type>
33               <documentation>
34                 <xhtml xmlns="http://www.w3.org/1999/xhtml"> nDataOut2[0] . nDataOut2 . Out . Ausg nge . Drive .
35                 ConveyorMotorAxis . ConveyorMotorAxis . Achsen . NC-Task 1 SAF</xhtml>
36               </documentation>
37             </variable>
38             <variable name="counterValue" address="%Q*">
39               <type>
40                 <UINT />
41               </type>
42               <initialValue>
43                 <simpleValue value="0" />
44               </initialValue>
45               <documentation>
46                 <xhtml xmlns="http://www.w3.org/1999/xhtml">nDataIn1[0] . nDataIn1 . In . Eing nge . Enc .
47                 ConveyorMotorAxis . ConveyorMotorAxis . Achsen . NC-Task 1 SAF</xhtml>
48               </documentation>
49             </variable>
50             <variable name="driveInState1" address="%Q*">
51               <type>
52                 <USINT />
53               </type>
54               <documentation>
55                 <xhtml xmlns="http://www.w3.org/1999/xhtml"> State of the hardware motor module</xhtml>
56               </documentation>
57             </variable>
58             <variable name="ModelToPLC">
59               <type>
60                 <DWORD />
61               </type>
62               <initialValue>
63                 <simpleValue value="0" />
64               </initialValue>
65               <documentation>
66                 <xhtml xmlns="http://www.w3.org/1999/xhtml"> empty (1), driveInState1 (1), counterValue (2)</xhtml>
67               </documentation>
68             </variable>
69             <variable name="PLCToModel">
70               <type>
71                 <DWORD />
72               </type>
73               <initialValue>
74                 <simpleValue value="0" />
75               </initialValue>
76               <documentation>
77                 <xhtml xmlns="http://www.w3.org/1999/xhtml"> empty (2), velocity (2)</xhtml>
78               </documentation>
79             </variable>
80           </localVars>
81         </interface>
82         <body>
83           <ST>
84             <xhtml xmlns="http://www.w3.org/1999/xhtml"> // Communication

```

```

84  counterValue := DWORD_TO_UINT(ModelToPLC AND 16#00_00_FF_FF);
    driveInState1 := DWORD_TO_USINT(SHR((ModelToPLC AND 16#00_FF_00_00),16));

86
88  // ToModel
    PLCToModel := 0;
    PLCToModel := SHL((PLCToModel OR velocity),0);</xhtml>
90  </ST>
    </body>
    <addData>
92  <data name="http://www.3s-software.com/plcopenxml/interfaceasplaintext" handleUnknown="implementation">
        <InterfaceAsPlainText>
94  <xhtml xmlns="http://www.w3.org/1999/xhtml">PROGRAM MAIN
96  VAR
    velocity AT%I* : INT; // nDataOut2[0] . nDataOut2 . Out . Ausg nge . Drive . ConveyorMotorAxis .
        ConveyorMotorAxis . Achsen . NC-Task 1 SAF
98  counterValue AT%Q* : UINT := 0; //nDataIn1[0] . nDataIn1 . In . Eing nge . Enc . ConveyorMotorAxis .
        ConveyorMotorAxis . Achsen . NC-Task 1 SAF
    driveInState1 AT%Q* : USINT; // State of the hardware motor module
100
102  ModelToPLC : DWORD := 0; // empty (1), driveInState1 (1), counterValue (2)
    PLCToModel : DWORD := 0; // empty (2), velocity (2)
104  END_VAR
</xhtml>
106  </InterfaceAsPlainText>
    </data>
108  <data name="http://www.3s-software.com/plcopenxml/objectid" handleUnknown="discard">
        <ObjectId>89812e0b-772f-4aaa-8733-9c5b26190d90</ObjectId>
110  </data>
    </addData>
112  </pou>
    </pous>
114  </types>
    <instances>
    <configurations />
    </instances>
116  <addData>
    <data name="http://www.3s-software.com/plcopenxml/projectstructure" handleUnknown="discard">
120  <ProjectStructure>
        <ObjectName>"MAIN" ObjectId="89812e0b-772f-4aaa-8733-9c5b26190d90" />
122  </ProjectStructure>
    </data>
124  </addData>
</project>

```

5.4.4. Result

This example uses the proposed data structure in a virtual commissioning of a DC motor. Since the CAD data was sent as a *.step* file without kinematization, the customer has to create it again in his CAD software. However, the supplier provides the model of the motor as a *.fmu* file, which is then integrated in the simulation for the virtual commissioning. In addition, the PLC terminal is modeled, since it is needed for the correct control of the motor. Again, the simulation runs on an engineering system in parallel with the PLC and communicates with it via the *ADS* interface. In this way, a reference velocity of the motor is given from the PLC and converted into an actual velocity in the simulation. This value is then returned to the PLC and is used for the calculation of the next reference velocity.

5.5. Summary

In this chapter, the data structure from chapter 4 is used in two use cases and its practical usability is tested. In each of these examples, the data structure is first generated and prepared for distribution. For this purpose, the CAD data is exported with and without kinematization, the behaviour model is created and control code for the PLC is written. In the second step, this information is then imported from this data structure and is integrated into a virtual commissioning. The simulation of the plant is executed in an engineering system parallel to the PLC, with the necessary communication using the *ADS* protocol.

The first example deals with a *separation*, where digital signals have to be read from three sensors and have to be written to two pistons. The CAD data with the kinematization is available in the native format of the software *Inventor* where the customer uses the same software. For this reason, there is no need to recreate the kinematization on the customer's side. The behaviour model as *.fmu* file describes the whole station and provides the sensor signals depending on the filling level of the input storage. The PLC code in *PLCopen* format includes a *function block* with a state machine in which the control logic is implemented.

Due to the formats of this data, the information can be integrated in the virtual commissioning without major effort and thus a setup for testing the plant is quickly available. The customer only has to write the main program of the PLC software and include there the provided *function block* and create the simulation in *Simulink* with the communication with the PLC.

The second example examines the virtual commissioning of a conveyor belt in more detail. This conveyor belt is driven by a DC motor, which also has an additional encoder on the motor axis. The PLC software now has to control the motor via a corresponding signal and process the feedback from the encoder. In contrast to the first example, the CAD data is available without kinematization in the neutral *.step* format and, as a result, must be newly created by the customer. In the behaviour model, the DC motor is described in the form of a transfer function, with the voltage signal as input and the velocity on the conveyor belt as output. A PLC code is not supplied by the manufacturer and therefore has to be written by the customer. This data is again bundled in the data structure and sent to the customer.

Setting up the simulation for virtual commissioning and, as part of this, importing the data involves more effort for the customer in this example. First, the CAD assembly must be converted and newly kinematized. Then the model of the motor must be integrated in the simulation in *Simulink* and additionally the motor terminal of the PLC must be modeled. Again, the communication with the PLC is part of the setup in the simulation. Finally, the virtual commissioning of the motor control is done via the graphical user interface in *TwinCAT*.

6. Results and Evaluation Ausschreiben



Das Ergebnis dieser Masterarbeit ist die vorgeschlagene Datenstruktur aus chapter 4. Die Struktur bündelt die Informationen aus der CAD-Konstruktion, des Verhaltensmodells, des Steuerungscode der SPS und optionaler Dokumentation. Die Struktur ist möglichst modular aufgebaut und Daten werden in einem möglichst verbreiteten Format gespeichert. Die Auswahl des Datenformates ist im Fall der CAD-Daten mit einer Kinetisierung noch nicht optimal, da ein neutrales Datenformat noch fehlt bzw. noch nicht breit in der Industrie etabliert ist. Des Weiteren wird in dieser Thesis der Workflow bei der Verwendung der Datenstruktur mit definierter Software aufgezeigt und beschrieben. Die verwendete Software ist in diesem Fall *Inventor* für die CAD-Daten, *Simulink* für die Modellierung und das Erstellen der Simulation für die virtuelle Inbetriebnahme und *TwinCAT* als Entwicklungsumgebung für die Programmierung der SPS. Die Auswahl der Software wurde aufgrund der leichten Verfügbarkeit ausgewählt und kann mit Alternativen noch erweitert werden.

Mit dieser Datenstruktur können die Hürden bei der Erstellung einer Virtuellen Inbetriebnahme gemindert werden, wodurch sie leichtert durchgeführt werden kann. Dadurch entsteht der Vorteil der Zeit- und Kostenminimierung bei der Inbetriebnahme von einer neuen Anlage.

7. Summary and Outlook **Translate**



Diese Thesis beschäftigt sich mit einer Datenstruktur für die virtuelle Inbetriebnahme einer SPS. Diese Struktur ist modular aufgebaut und umfasst die Informationen aus der CAD-Konstruktion, den Verhaltensmodellen, Software für die SPS und der benötigten Dokumentation.

Um den Aufbau der Datenstruktur bestimmen zu können, werden zuerst die nötigen Informationen für die Inbetriebnahme in chapter 2 gesammelt. Dies erfolgt über die verschiedenen Stufen in der Entwicklung einer Anlage oder eines Produkts, wobei das Schnittstellenmanagement, die Hardware und die Software untersucht wird.

Dannach wird in chapter 3 der aktuelle State of the art in der virtuellen Inbetriebnahme untersucht. Der Fokus dabei liegt auf der Identifikation von verfügbaren Datenformate und ihre Kompatibilität zu anderer Software.

Die eigentliche Datenstruktur wird schließlich in chapter 4 vorgestellt und die ausgewählten Datenformate beschrieben. Bei der Auswahl der Datenformate stellt sich vor allem der Austausch der Kinematisierung der CAD-Baugruppe als Problem dar. Zwar existiert das Austauschformat *COLLADA*, welches die Kinematisierung unterstützt, jedoch fehlt es an Möglichkeiten und Tools für die Konvertierung der Daten. Aus diesem Grund ist zur Zeit der Erstellung dieser Thesis die praktikabelste Lösung native Formate zu verwenden oder die Kinematisierung neu zu erstellen. Zu einem späteren Zeitpunkt sollte das *COLLADA*-Format nochmals auf seine Tauglichkeit untersucht werden.

Abschließend wird die Datenstruktur in chapter 5 in zwei Beispielen getestet. In diesen Beispielen wird für die Konstruktion die Software *Inventor*, für die Modellierung *Simulink* und für die Steuerung der SPS *TwinCAT* verwendet. Auch in diesem Bereich kann weiter Erfahrung im Handling von anderer Software gesammelt werden. Die Entwicklung dieser Software bleibt auch nicht stehen und daher sollten gerade im Bereich von *TwinCAT* in naher Zukunft Produkte zur Verfügung stehen, die den Prozess der virtuellen Inbetriebnahme vereinfachen sollen.

Bibliography

- [1] H. Sangvik, "Digital Twin of 3d Motion Compensated Gangway Use of Unity Game Engine and TwinCAT PLC Control for Hardware-in-the-Loop Simulation," Master's thesis, University of Agder, 2021.
- [2] J. Kim, M. J. Pratt, R. G. Iyer, and R. D. Sriram, "Standardized data exchange of CAD models with design intent," *Computer-Aided Design*, vol. 40, no. 7, pp. 760–777, 2008, current State and Future of Product Data Technologies (PDT). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0010448507001625>
- [3] M. Barnes and E. L. Finch, "COLLADA - Digital Asset Schema Release 1.5.0 Specification." Sony Computer Entertainment Inc., 04 2008.
- [4] *Functional Mock-up Interface Specification*. Modelica Association Project.
- [5] S. Faltinski, O. Niggemann, N. Moriz, and A. Mankowski, "AutomationML: From data exchange to system planning and simulation," 03 2012.
- [6] A. Lüder, N. Schmidt, and R. Rosendahl, "Data exchange toward PLC programming and virtual commissioning: Is AutomationML an appropriate data exchange format?" *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, 2015.
- [7] Autodesk Help. About Packaging Files with Pack and Go. Visited on 2022-05-23. [Online]. Available: <https://knowledge.autodesk.com/support/inventor/learn-explore/caas/CloudHelp/cloudhelp/2019/ENU/Inventor-Help/files/GUID-018371A9-B60D-44CB-B70C-8618155CC598-htm.html>
- [8] Solidworks Help. Pack and Go - Übersicht. Visited on 2022-05-23. [Online]. Available: https://help.solidworks.com/2021/german/SolidWorks/sldworks/c_pack_and_go.htm
- [9] Mathworks Help Center. Export Simulink Model to Standalone FMU. Visited on 2022-05-23. [Online]. Available: <https://de.mathworks.com/help/slcompiler/ug/simulinkfmuexample.html>
- [10] —. FMU Importing. Visited on 2022-05-23. [Online]. Available: <https://de.mathworks.com/help/simulink/in-product-solutions.html>
- [11] Autodesk Help. Export data to other formats. Visited on 2022-05-26. [Online]. Available: <https://knowledge.autodesk.com/support/inventor-products/learn-explore/caas/CloudHelp/cloudhelp/2015/ENU/Inventor-Help/files/GUID-A693B9CD-63FA-4E98-92AD-FDA3E17BA298-htm.html>
- [12] —. To Import STEP or IGES Data (Construction Environment). Visited on 2022-05-26. [Online]. Available: <https://knowledge.autodesk.com/support/inventor/learn-explore/caas/CloudHelp/cloudhelp/2021/ENU/Inventor-Help/files/GUID-0F475FF0-0B1D-46B2-9F0F-7F7E211925EF-htm.html>

- [13] ——. To Use Pack and Go to Package Files. Visited on 2022-05-26. [Online]. Available: <https://knowledge.autodesk.com/support/inventor/learn-explore/caas/CloudHelp/cloudhelp/2019/ENU/Inventor-Help/files/GUID-730304AA-13BD-467B-9351-C7C1362876BD-htm.html>
- [14] Simlab 3D Plugins. Inventor Collada Exporter Plugin. Visited on 2022-06-01. [Online]. Available: https://www.simlab-soft.com/3d-plugins/Inventor/Collada_exporter_for_Inventor-main.aspx
- [15] Autodesk Help. Importing Autodesk Inventor Files. Visited on 2022-05-26. [Online]. Available: <https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2020/ENU/3DSMax-Data-Exchange/files/GUID-CCF94205-D5DA-4EA0-A3F1-F2281EE1FC01-htm.html>
- [16] Beckhoff Automation GmbH, *TE1420 / TwinCAT 3 Target for FMI*, product data sheet.
- [17] Beckhoff Information System. TE1400|TwinCAT 3 Target for Simulink. Visited on 2022-05-26. [Online]. Available: https://infosys.beckhoff.com/content/1033/te1400_tc3_target_matlab/index.html?id=7328785815492855617
- [18] ——. TE1400|TwinCAT 3 Target for Simulink - Samples. Visited on 2022-05-26. [Online]. Available: https://infosys.beckhoff.com/content/1033/te1400_tc3_target_matlab/10825692555.html?id=382902774554596767
- [19] Mathworks Help Center. Simulink PLC Coder - Generate Structured Text Code for a Simple Simulink Subsystem. Visited on 2022-05-26. [Online]. Available: <https://de.mathworks.com/help/plccoder/ug/generating-structured-text-for-a-simple-simulink-subsystem.html>
- [20] ——. Simulink PLC Coder - Supported Blocks. Visited on 2022-05-26. [Online]. Available: <https://de.mathworks.com/help/plccoder/ug/supported-simulink-blocks.html>
- [21] Beckhoff Information System. TE1410|TwinCAT 3 Interface for MATLAB/Simulink - ADS blocks. Visited on 2022-05-26. [Online]. Available: https://infosys.beckhoff.com/content/1033/te1410_tc3_interface_matlab/11512422539.html?id=6289303933073219953
- [22] ——. TE1410|TwinCAT 3 Interface for MATLAB/Simulink. Visited on 2022-05-26. [Online]. Available: https://infosys.beckhoff.com/content/1033/te1410_tc3_interface_matlab/index.html?id=981623218745783434
- [23] ——. TwinCAT 3|PLC - Exporting and importing a PLC project. Visited on 2022-05-26. [Online]. Available: https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2526208651.html?id=2445988504692268481
- [24] Beckhoff Automation GmbH, *TE1130 / TwinCAT 3 CAD Simulation Interface*, product data sheet.
- [25] Mathworks Help Center. Control System Toolbox - DC Motor Control. Visited on 2022-05-26. [Online]. Available: <https://de.mathworks.com/help/control/ug/dc-motor-control.html>

- [26] Beckhoff Information System. TwinCAT NC - General. Visited on 2022-05-26. [Online]. Available: <https://infosys.beckhoff.com/content/1031/tcncgeneral/html/note.htm?id=5419198730788984622>

List of Figures

| | | |
|-------|--|----|
| 2.1. | Steps in product development. | 3 |
| 3.1. | Comparison of structure of SIL (a) and HIL (b) systems. | 5 |
| 4.1. | Use case for data structure. | 10 |
| 4.2. | Data structure | 11 |
| 4.3. | Exemplary assembly for selection of suitable CAD-formats. | 13 |
| 4.4. | Overview of tested methods for geometry export and import. | 13 |
| 4.5. | Example for testing behaviour model | 15 |
| 4.6. | Overview of tested methods for <i>FMI</i> handling. | 16 |
| 4.7. | General Workflow | 20 |
| 5.1. | The used Teaching Factory Aktuelles Bild verwenden | 22 |
| 5.2. | Used setup for a virtual commissioning | 22 |
| 5.3. | Module Separation In CAD beschriften | 24 |
| 5.4. | Behaviour model in <i>Simulink</i> | 26 |
| 5.5. | Data structure for Example <i>Seperation</i> | 27 |
| 5.6. | Use of the behaviour model in <i>Simulink</i> for example <i>Seperation</i> | 28 |
| 5.7. | Module Filling In CAD beschriften | 32 |
| 5.8. | Behaviour model of a DC-drive in <i>Simulink</i> for example <i>Filling</i> | 34 |
| 5.9. | Data structure for Example <i>Conveyor Belt</i> | 35 |
| 5.10. | Use of the behaviour model in <i>Simulink</i> for example <i>conveyor belt</i> | 37 |

List of Tables

| | | |
|------|---|----|
| 4.1. | Chooosen file formats. | 11 |
| 4.2. | Results of tested methods in CAD exchange. | 14 |
| 4.3. | Results of tested methods in integrating behaviour model. | 15 |
| 5.1. | Used software in the example <i>Teaching Factory</i> | 22 |

List of Code

- 4.3.1. Example counter as *PLCopen-xml* 18
- 5.3.1. Main programm with included example code of example *Seperation* . . . 28
- 5.4.1. Main program of example *Conveyor Belt* 38
- A.1.1. Code for module *seperation* as *PLCopen-xml* A1

List of Acronyms

| | |
|----------------|---|
| COLLADA | Collaborative Design Activity |
| FMI | Functional Mockup Interface |
| FMU | Functional Mockup Unit |
| HIL | Hardware in the Loop |
| HMI | Human-machine interface |
| HTML | Hypertext Markup Language |
| IGES | Initial Graphics Exchange Specification |
| MD | Markdown |
| PDF | Portable Document Format |
| SIL | Software in the Loop |
| STEP | Standard for the Exchange of Product model data |

A. Appendix

A.1. PLC Source Code

Sourcecode A.1.1: Code for module *seperation* as *PLCopen-xml*

```
2 <?xml version="1.0" encoding="utf-8"?>
3 <project xmlns="http://www.plcopen.org/xml/tc6_0200">
4   <fileHeader companyName="Beckhoff Automation GmbH" productName="TwinCAT PLC Control" productVersion="3.5.13.21"
5     creationDateTime="2022-05-27T15:51:05.3586322" />
6   <contentHeader name="main" modificationDateTime="2022-05-27T15:51:05.3606326">
7     <coordinateInfo>
8       <fbid>
9         <scaling x="1" y="1" />
10      </fbid>
11      <ld>
12        <scaling x="1" y="1" />
13      </ld>
14      <sfc>
15        <scaling x="1" y="1" />
16      </sfc>
17    </coordinateInfo>
18    <addData>
19      <data name="http://www.3s-software.com/plcopenxml/projectinformation" handleUnknown="implementation">
20        <ProjectInformation />
21      </data>
22    </addData>
23  </contentHeader>
24  <types>
25    <dataTypes />
26    <pous>
27      <pou name="FB_Separator" pouType="functionBlock">
28        <interface>
29          <inputVars>
30            <variable name="bEnable">
31              <type>
32                <BOOL />
33              </type>
34              <documentation>
35                <xhtml xmlns="http://www.w3.org/1999/xhtml"> start/stops the functionblock </xhtml>
36              </documentation>
37            </variable>
38            <variable name="bExecute">
39              <type>
40                <BOOL />
41              </type>
42              <documentation>
43                <xhtml xmlns="http://www.w3.org/1999/xhtml"> Ejects 1 flask if true </xhtml>
44              </documentation>
45            </variable>
46            <variable name="bErrorReset">
47              <type>
48                <BOOL />
49              </type>
50              <documentation>
51                <xhtml xmlns="http://www.w3.org/1999/xhtml">Reset Error </xhtml>
52              </documentation>
53            </variable>
54            <variable name="bSensor1">
55              <type>
56                <BOOL />
57              </type>
58              <documentation>
59                <xhtml xmlns="http://www.w3.org/1999/xhtml">Hardware Sensor 1</xhtml>
60              </documentation>
61            </variable>
62            <variable name="bSensor2">
63              <type>
64                <BOOL />
65              </type>
66              <documentation>
67                <xhtml xmlns="http://www.w3.org/1999/xhtml">Hardware Sensor 2</xhtml>
68              </documentation>
69            </variable>
70            <variable name="bSensor3">
71              <type>
72                <BOOL />
73              </type>
74              <documentation>
```

```

74         <xhtml xmlns="http://www.w3.org/1999/xhtml">Hardware Sensor 3</xhtml>
        </documentation>
        </variable>
76    </inputVars>
    <outputVars>
78        <variable name="bR1AKT1">
            <type>
80                <BOOL />
            </type>
            <documentation>
            <xhtml xmlns="http://www.w3.org/1999/xhtml">Relais 1 Aktor 1 (Schieber oben)</xhtml>
84            </documentation>
        </variable>
86        <variable name="bR2AKT1">
            <type>
88                <BOOL />
            </type>
            <documentation>
            <xhtml xmlns="http://www.w3.org/1999/xhtml">Relais 2 Aktor 1</xhtml>
90            </documentation>
        </variable>
92        <variable name="bR1AKT2">
            <type>
94                <BOOL />
            </type>
            <documentation>
96            <xhtml xmlns="http://www.w3.org/1999/xhtml">Relais 1 Aktor 2 (Schieber unten)</xhtml>
            </documentation>
        </variable>
102        <variable name="bR2AKT2">
            <type>
104                <BOOL />
            </type>
            <documentation>
106            <xhtml xmlns="http://www.w3.org/1999/xhtml">Relais 2 Aktor 2</xhtml>
            </documentation>
        </variable>
110        <variable name="bNoBottle">
            <type>
112                <BOOL />
            </type>
            <documentation>
114            <xhtml xmlns="http://www.w3.org/1999/xhtml">Keine Flasche vorhanden</xhtml>
            </documentation>
        </variable>
118        <variable name="bWrongBottle">
            <type>
120                <BOOL />
            </type>
            <documentation>
122            <xhtml xmlns="http://www.w3.org/1999/xhtml">Falsche oder gedrehte Flasche vorhanden</xhtml>
            </documentation>
        </variable>
126        <variable name="eStatus">
            <type>
128                <derived name="State" />
            </type>
            <initialValue>
130                <simpleValue value="State.Off" />
            </initialValue>
        </variable>
134    </outputVars>
    <localVars>
136        <variable name="timer">
            <type>
138                <derived name="TON" />
            </type>
            <documentation>
140            <xhtml xmlns="http://www.w3.org/1999/xhtml">Timer to make sure, that the flask has cleared the slope</
xhtml>
142            </documentation>
        </variable>
144        <variable name="eState">
            <type>
146                <derived name="StateSeparator" />
            </type>
            <initialValue>
148                <simpleValue value="StateSeparator.Off" />
            </initialValue>
        </variable>
152    </localVars>
    </interface>
154    <body>
        <ST>
156            <xhtml xmlns="http://www.w3.org/1999/xhtml">CASE eState OF
StateSeparator.Off:
158            eStatus := State.Off;
            bR1AKT1 := TRUE; //Eingefahren oben
160            bR2AKT1 := FALSE; //Eingefahren oben

            bR1AKT2 := FALSE; //Eingefahren unten
162            bR2AKT2 := TRUE; //Eingefahren unten
164

```



```

166     IF bEnable THEN
167         eState := StateSeparator.Init;
168     END_IF
169
170 StateSeparator.Init:
171
172     eStatus := State.Busy; //Beschaeftigt Signallampe
173
174     bR1AKT1 := FALSE; //Ausgefahren oben
175     bR2AKT1 := TRUE;
176
177     bR1AKT2 := FALSE; //Eingefahren unten
178     bR2AKT2 := TRUE;
179
180     IF bSensor1 THEN
181         eState := StateSeparator.Init_Wait; //Zustandswechsel sobald Sensor Flasche zwischen Schiebern sieht
182     END_IF
183
184 StateSeparator.Init_Wait:
185     timer(IN := TRUE, PT := T#1000MS); //Timer wartet bis Flasche sicher zum Stehen kommt
186
187     IF timer.Q THEN
188         timer(IN := FALSE);
189         eState := StateSeparator.Ready; //Zustand Ready nach Timer
190     END_IF
191
192 StateSeparator.Ready:
193     eStatus := State.Ready; //Zustand Ready Signallampe
194
195     bR1AKT1 := TRUE; //Eingefahren oben
196     bR2AKT1 := FALSE; //Eingefahren oben
197
198     bR1AKT2 := FALSE; //Eingefahren unten
199     bR2AKT2 := TRUE; //Eingefahren unten
200
201     IF bExecute THEN
202         eState := StateSeparator.Eject_Wait_1; //Auswerfen sobald Exceute TRUE
203     END_IF
204
205 StateSeparator.Eject_Wait_1: // Warten bis Schieber oben ganz geschlossen ist
206     eStatus := State.Busy; //Zustand Busy Signallampe
207
208     timer(IN := TRUE, PT := T#2500MS); //Timer auf 2,5 S eingestellt
209
210     IF timer.Q THEN
211         timer(IN := FALSE);
212         eState := StateSeparator.Eject; //Zustand bereit nach Timer
213     END_IF
214
215 StateSeparator.Eject: //Auswerfen der unteren Flasche
216     bR1AKT2 := TRUE; //Lsst Flaschen auf Band ab durch ffnen des unteren Schiebers
217     bR2AKT2 := FALSE;
218
219     IF NOT bSensor1 THEN
220         eState := StateSeparator.Eject_Wait_2;
221     END_IF
222
223 StateSeparator.Eject_Wait_2: // Warten bis Flasche durch F rderband weit genug von Schieber wegtransportiert
224     eStatus := State.Done;
225     timer(IN := TRUE, PT := T#4000MS); //Timer auf 4 S eingestellt
226
227     IF timer.Q THEN
228         timer(IN := FALSE);
229         eState := StateSeparator.Done_Wait;
230     END_IF
231
232 StateSeparator.Done_Wait: // Schlie en des unteren Schiebers und Warten bis komplett geschlossen
233     bR1AKT2 := FALSE; //Eingefahren unten
234     bR2AKT2 := TRUE; //Eingefahren unten
235
236     timer(IN := TRUE, PT := T#3000MS); //Timer wartet bis Schieber geschlossen ist
237
238     IF timer.Q THEN
239         timer(IN := FALSE);
240         eState := StateSeparator.Done;
241     END_IF
242
243 StateSeparator.Done: //Vereinzelungsvorgang abgeschlossen
244
245     IF NOT bExecute THEN
246         eState := StateSeparator.Init;
247     END_IF
248
249 END_CASE</xhtml>
250 </ST>
251 </body>
252 <addData>
253     <data name="http://www.3s-software.com/plcopenxml/interfaceasplaintext" handleUnknown="implementation">
254         <InterfaceAsPlainText>
255             <xhtml xmlns="http://www.w3.org/1999/xhtml">FUNCTION_BLOCK FB_Separator
256 VAR_INPUT
257     bEnable : BOOL; // start/stops the functionblock

```

```

258 bExecute: BOOL; // Ejects 1 flask if true
    bErrorReset: BOOL; // Reset Error
260
    bSensor1 : BOOL; // Hardware Sensor 1
262 bSensor2 : BOOL; // Hardware Sensor 2
    bSensor3 : BOOL; // Hardware Sensor 3
264 END_VAR
VAR_OUTPUT
266 bR1AKT1 : BOOL; // Relais 1 Aktor 1 (Schieber oben)
    bR2AKT1 : BOOL; // Relais 2 Aktor 1
268 bR1AKT2 : BOOL; // Relais 1 Aktor 2 (Schieber unten)
    bR2AKT2 : BOOL; // Relais 2 Aktor 2
270
    bNoBottle : BOOL; // Keine Flasche vorhanden
272 bWrongBottle : BOOL; // Falsche oder gedrehte Flasche vorhanden
274
    eStatus : State := State.Off;
END_VAR
276 VAR
    timer : TON; // Timer to make sure, that the flask has cleared the slope
278 eState : StateSeparator := StateSeparator.Off;
END_VAR
280 </xhtml>
    </InterfaceAsPlainText>
282 </data>
    <data name="http://www.3s-software.com/plcopenxml/objectid" handleUnknown="discard">
284 <ObjectId>91946f14-5ef9-4883-993a-962cda20c177</ObjectId>
    </data>
286 </addData>
</pou>
288 </pous>
</types>
290 <instances>
    <configurations />
292 </instances>
<addData>
294 <data name="http://www.3s-software.com/plcopenxml/projectstructure" handleUnknown="discard">
    <ProjectStructure>
296 <Object Name="FB_Separator" ObjectId="91946f14-5ef9-4883-993a-962cda20c177" />
    </ProjectStructure>
298 </data>
</addData>
300 </project>

```