Master Thesis

In partial fulfillment of the requirements for the degree

"Master of Science in Engineering"

Master program:
Mechatronics & Smart Technologies
Management Center Innsbruck

Supervisors :
Benjamin Massow and 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 I declare, to the best of my knowledge and belief, that all passages taken from published and unpublished sources or docments have been reproduced whether as original, slightlychanged or in thought, have been mentioned as such at the corresponding placesof the thesis, by citation, where the extend 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. However, implementation can easily become complex.
The behaviour 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 of this rule. This variety of possibilities increases the complexity of the integration process enormously. However, in order for virtual commissioning to be efficient, the integration of the used components should be as simple as possible.
Aiming to solve this problem, an exemplary library structure is developed in this thesis, which consists of the behaviour model, CAD data and code snippets with related documentation. The basis for this are freely available and common interfaces. 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 behaviour model and control are executed on two different machines ensuring real-time capability and forming a HIL system. The behaviour model is generated in *Simulink* and integrated into the PLC run-time.
As shown, using the developed library, the process of virtual commissioning can be simplified without the need for additional expertise. However, there is still room for improvement especially in the exchange of the CAD data with a kinematization and the integration of behaviour models in a PLC.

Keywords: Virtual Commissioning, PLC, Automation, Modularity, Data Library.

Contents

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 needed in a virtual commissioning.

1.2. Aim of this Thesis

The aim of this thesis is the development of a data structure and a concept for the integration of simulation models of a plant to be commissioned. These simulation models include different aspects of product development and consist of the components: CAD model, behaviour model based on physics, control code and documentation. Thereby the virtual commissioning is focused on the level of the PLC control. An existing learning factory is used as an exemplary facility for the development of this concept, where the simulation of the plant should be real-time capable.

1.3. Structure of this Thesis

This thesis is divided into several chapters and describes relevant basics, the development and use of a data structure, as well as the evaluation of the results.
In chapter 2 the state of the art in virtual commissioning of a plant and the process of product development are explained. For the purpose of developing a data structure, necessary information are identified, different methods of implementation are considered and an excerpt from available data formats are described. The data structure for a standardized exchange between suppliers and customers is then described in chapter 3. The structure itself and the used data formats are explained and selected in this chapter. Afterwards, this data structure is used in a real-world example in chapter 4 and its usability is tested in practice. In the end, the results are summarized in chapter 5 and evaluated in context of virtual commissioning. And finally, chapter 6 provides an outlook on the next possible steps and further developments.

This thesis and relevant files can be found in Github using this link: `https://github.com/gd8213/MasterThesis`.

2. State of the Art in Virtual Commissioning

In this chapter, the state of the art of virtual commissioning at 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.

2.1. Foundations of Virtual Commissioning

2.1.1. Required Information for Virtual Commissioning

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 process. 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 usually consisting of mechanical and electrical components. The hardware must be developed in collaboration between both disciplines, since changes in the mechanics, for example, 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 be started already. 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 testing and debugging is meaningful.

The final step in product development is an optional 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.

Information from all phases of product development are required for the successful execution of a virtual or a real 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. The characteristics of the used hardware is particularly important in the creation of behaviour models, where the information is given in data sheets or already integrated in models.

Mechanical Engineering: The general structure of the mechanics and the kinematics of the individual components are defined in this step. The design of the product and the used materials determine the physical behaviour and result in multiple CAD files. This information is often relevant for control systems.

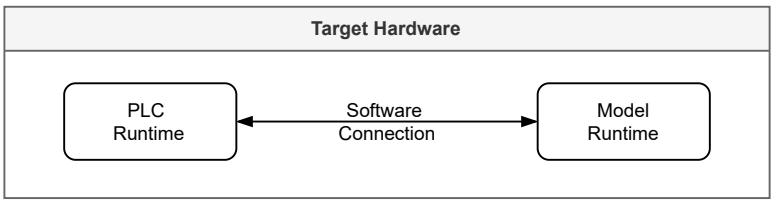
Electrical Engineering: The configuration of the PLC with the used terminals is part of this section. Especially interesting for the development of the software is the mapping between the inputs and outputs of the terminals with the connected devices. Only an exactly documented mapping can avoid wrong connections in the PLC software and the resulting damage of the hardware. The documentation of the mapping and the structure of the PLC can be shown in a tabular form or in a dedicated object in the development environment of the PLC.

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 handling 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 therefore ensure a faster implementation. But also an example PLC code of simpler components can often be advantageous.

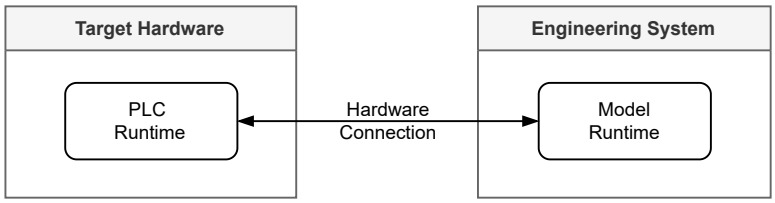
2.1.2. Hardware or Software in the Loop (HIL/SIL)

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. For a better understanding the setup of a HIL and a SIL system is shown in Fig. 2.2
Depending on the complexity of the control system, both methods are suitable for the development and commissioning of PLC software. For example, basic functions and code sections can be tested quickly in a SIL system, thus shortening the development time. For more complex or time-critical controls, however, testing and commissioning is often easier to perform in a HIL system.

The Hardware in the Loop (HIL) method offers advantages especially for complex systems compared to the Software in the Loop (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 im-



(a) SIL system



(b) HIL system

Figure 2.2.: Comparison of hardware setup in a SIL and HIL system. While in a SIL system the model of the plant and the controller are executed on the same hardware, in a SIL system the model and the controller are executed on separate hardware.

plementation of continuous regression testing. The goal of this type of testing is to detect errors and bugs due to newly created sections of code. The major drawback of this method is the additional communication between the controller and the model. This communication should be real-time capable to allow a meaningful simulation.
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. One of the main requirements of the SIL system is the additional demand that the target hardware must be able to run the model of the plant in parallel with the control system.

Testing and commissioning of the PLC software via HIL/SIL can be done in different ways, for which several practices are explained in the following lines.

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. Depending on the used system this information can be processed even in or near to real-time.
In this approach, the human developer imitates the behaviour 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. However, basic code sections and functions can be evaluated during development and in the process of a virtual commissioning. The creation of a Human-machine interface (HMI) is supported by many common PLC manufacturers such as *Siemens* and *Beckhoff*.

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 simple 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. By using the original CAD data, a visually appealing model can be created and controlled via the PLC. This is an important advantage especially in presentations with customers or in sales, in order to be able to address people from the non-technical area as well. But also the start for new employees in the development of the control is easier with a clear and appealing model.

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 $10\,\mathrm{ms}$. The simulation in *Unity* achieves a step time of $5\,\mu\mathrm{s}$.

Digital Twin in a 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 is executed 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 or precautions are often required. This approach can also be used in HIL and SIL systems and is often found in final testing or the virtual commissioning of the software.

Visualization in a CAD Software

The solution with probably the best overview is visualization directly in the CAD software. Through this possibility, a mechanical problem can be quickly identified and solved accordingly. The company *Beckhoff*, for example, has recognized this opportunity and is currently developing its own product *TE1130* with which the visualization can be done in a common CAD software. As said the product is still in development with the planned release date in end of 2022 [2]. In any case, for each CAD tool used, a different interface depending on the manufacturer must be used to implement this possibility in practice. However, in this approach, only the visualization of the current state of the PLC is done in the CAD software and the model of the plant is not calculated.

2.2. Relevant Data Formats

This chapter describes and compares a selection of the most common data formats for relevant virtual commissioning information according to Sec. 2.1.1.

2.2.1. Geometry Data and Kinematization

The branches of industry and their products are very diverse and with them the requirements for the used 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 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 attached components move with them. This is particularly advantageous in the dynamic area with moving parts for example in mechanical engineering.

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. In many areas and also in virtual commissioning, kinematization and thus its exchange is, however, an important prerequisite. The industry has recognized this problem of the missing interface of the kinematization and is therefore working on different solutions.
For example, existing CAD data formats can be extended to support features and kinematization. For this purpose, this paper [3] proposes an possible extension of the *.step* format in order to include the kinematization. However, this extension is not yet part of the standard and therefore not implemented in commercial CAD tools.
A second possibility is the development of a new and neutral data format, with the goal to be able to store the geometry and the kinematization. In order to be able to use this format, an integration into existing CAD tools can be implemented or, alternatively, a conversion of the native formats with the help of translators is also an option. However, both methods are feasible but require a major effort in technical and organizational terms. This paper [**?**] shows the conversion of the native features in the design of a part into a neutral format and back into a second CAD software. The same approach could be used to export the kinematics as well as the features, as shown in this paper [**?**].
As an alternative, the kinematics could be saved in an additional file and linked to the original CAD data. In this case, the CAD software would have to provide a way to save and read the kinematic separately from the assemblies. An example for this implementation is shown in this paper [**?**]. Again, this method is feasible but requires a lot of effort, since a common agreement on the used data format has to be achieved and then implemented.
A promising solution is the *COLLADA* format (Collaborative Design Activity), which supports kinematization starting from version 1.5 [4]. 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 a large number of 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 the *COLLADA* format to describe geometry data per default.

2.2.2. Behaviour Models

Similar to the geometry data, there is a variety of possible file formats for the description of a physical behaviour model. This is mostly dependent on the discipline and the simulation software used. For example, *Simulink* and *SimulationX* are available tools for describing physical systems in a wide range of possible application areas, whereby both tools use a native data format to describe a physical model. Alternatively, the *Modelica* language can also be used for the description of a system as it is universal and offers the possibility to be integrated in a wide range of different tools.

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 and is currently in fact the default format for exchanging models. 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 more than 170 different tools [5]. This great popularity is also the result of the publicly available tools for checking the compatibility of *FMI* objects save as **.fmu!** (**.fmu!**) files and a open source distribution of the sources.

2.2.3. 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.

2.2.4. 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 easy handling. 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.

2.2.5. 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 behaviour models as FMI files. This approach is for example seen in [6]. For this reason, *AutomationML* represents an appropriate exchange format [7]. However, as mentioned above, the lack of support for the *COLLADA* format from the CAD software side is still a problem.

2.3. Criteria for a Data Structure in Virtual Commissioning

Based on these fundamentals of a virtual commissioning, criteria for a data structure can be identified. With the help of these criteria, the efficiency of a data structure can be determined, whereby these should be fulfilled as good as possible. The identified criteria are:

- Independent of used hardware setup (HIL/SIL).
- Contain all needed information.
- Modular layout in order to represent components of a real plants.
- High compatibility with common software tools (Data formats).
- Low level of additional knowledge required / Easy to use.

2.4. Summary

This chapter describes the current state of the art in the field of virtual commissioning of a PLC. With that in mind, the basics of virtual commissioning, consisting of the physical structure and the required information, are described at the beginning. This information results from all stages of product development and are available in various data formats. In the second part, relevant data formats for the identified information are described. These data formats include: CAD, behaviour model, source code of the PLC, documentation and existing exchange formats. Based on these conclusions, the development of a data structure for a modular virtual commissioning is done.
Finally, criteria are defined by which a data structure for virtual commissioning may be evaluated.

3. Proposed Data Structure for Modular Virtual Commissioning

Based on the previous findings, a data structure for a virtual commissioning is presented in this chapter. The main objective of this data structure is a high compatibility and a modular design in order to be able to represent the individual sub-components of a real plant. The data structure itself should contain all necessary information required for a virtual commissioning.

3.1. Layout of the Data Structure

The actual layout of the data structure is kept simple and consists of a root folder with several directories for the relevant information as shown in Fig. 3.1. The sub-directories contain the information of kinematization and CAD, physical model, PLC source code and documentation. An optional *ReadMe* file is used for general information and revision control of the structure itself. Finally the root folder is compressed into a *.zip* file for distribution.

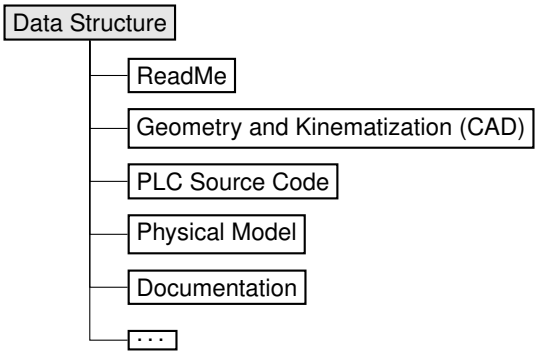
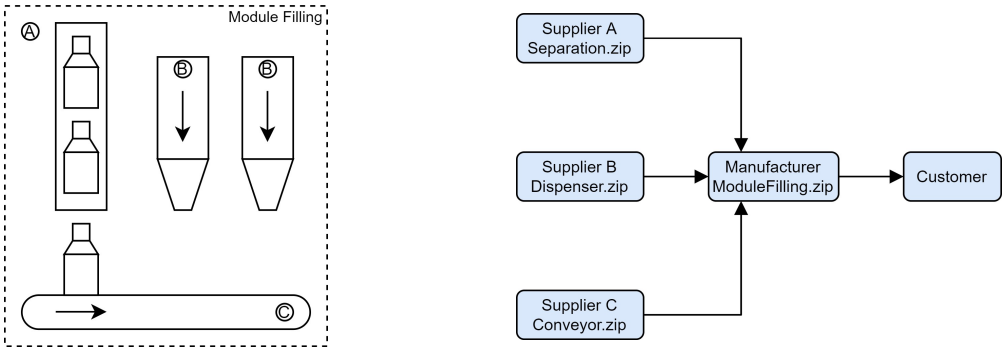


Figure 3.1.: Layout of proposed data structure. The structure consists of a *ReadMe* file with general information of the structure itself and several folders with the needed information of a virtual commissioning.

Due to the plain and simple structure, it can be easily extended and modular exchanged. In order to show this characteristic, an example of a filling stations as part of a larger plant is considered. The design of this station is shown in Fig. 3.2a and consists of a separation (A), two identical dosing units (B) and a conveyor belt (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 represented in just one data structure in planning of the remaining plant. This general procedure is shown in Fig. 3.2b.



(a) Product overview

(b) File components

Figure 3.2.: Use case to show modularity of a data structure. In this case a filling station is considered consisting of a separation (A), dosing units (B) and a conveyor belt (C). The components are bought and merged into a single module.

3.2. Methods for the Selection of Data Formats

In this section, different methods for the virtual commissioning of a simple component are investigated. For this purpose, the different aspects CAD, physical behaviour model and PLC source code are considered and the suitability of different approaches are evaluated. Based on these results, the selection of data formats in the proposed data structure is afterwards made. The used software for this investigation is listed in Tab. 3.1.

Table 3.1.: Used software for selection of data formats.

Area	Used Software
CAD	Autodesk Inventor Professional 2022 Autodesk 3ds Max 2022
PLC	Beckhoff TwinCAT 3 Version 3.1.4024.25
Modelling	Matlab R2020b, Simulink

The assembly from Fig. 3.3 is used as an example for a real plant. Here, a PLC is controlling the speed of the disk in the left section and thus the position of the piston in the right section. The CAD assembly is kinematized and the model of the system describes the angular position of the disc and the position of the piston as a function of the current velocity.

3.2.1. Geometry Data and Kinematization

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. 3.3. 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.

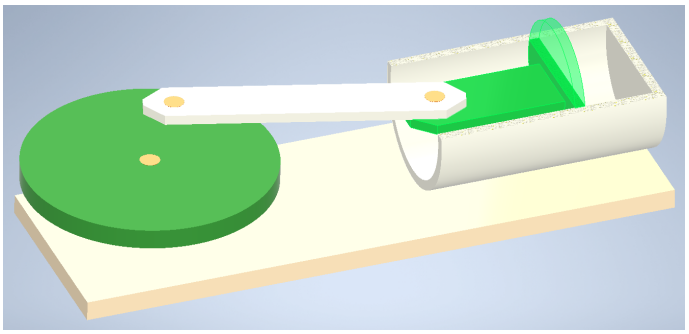


Figure 3.3.: Exemplary assembly for selection of suitable CAD formats. For this a movable piston is considered as a plant. The position of this piston is controlled using the disk on the left side.

The selection criterion for a suitable exchange format is the ability to save the geometry and kinematization of an assembly in combination with a high compatibility with different tools. As a result of these criteria, a neutral exchange format should provide an optimal solution.
However, due to the fact that no default exchange format for kinematization exist, different approaches have to be tested. For this purpose, different methods of CAD data exchange are investigated and described in the following sections. An overview of the tested methods is provided in Fig. 3.4.

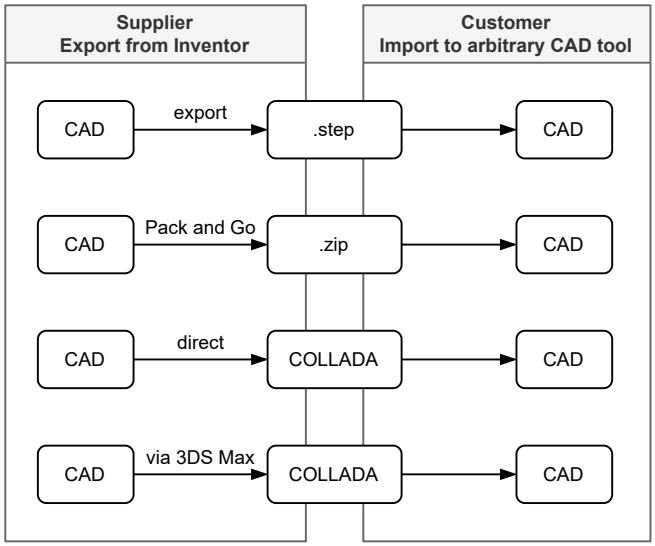


Figure 3.4.: Overview of tested approaches for exchange of geometry with kinematization. In total four approaches are tested, with the neutral *.step* format, the native *Pack and Go* tool and the neutral *COLLADA* format being investigated.

a) *Inventor* to *.step* and back

The first approach is using the *.step* format. This neutral format is chosen because it is already widely used for CAD data exchange.
The export of the assembly into a *.step* file is done according to the instruction [8] via the menu *File-Export-CAD Format*. In the appearing window, the target format must finally be set to *.step*. Additional options are not needed.
The import is also very simple and can be done either in a new file or in an existing assembly, where instructions can be found in [9]. 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 in the distribution of the final data.
The export of the assembly into a bundled *.zip* file is done according to the instruction [10] via *File-Save as-Pack and Go*. Here, in addition to the individual components, related project data is also exported.
For import this *.zip*-file only has to be extracted and afterwards can be opened via *File-Open*. In this case the main assembly of the project can be opend afterwards and no information is lost.
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 high compatibility to 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 [11] in form of a *Inventor* plugin. Importing *COLLADA* back into the native formats of *Inventor* is still 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 software *3DS Max*. The advantage of this compared to the previous approach is the native tool chain of *Autodesk* with the avoidance of 3rd party

tools, but with the limitation of an export of the pure geometry. However, the problem remains similar with feasible support for export, but lack of possibilities for import of *COLLADA* files.
For example, opening *Inventor* files in *3DS Max* is done using the guide [12]. 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.

3.2.2. Behaviour Model

Since the *FMI* format is supported by a large number of tools, it has already been established as the unofficial standard format for data exchange of models in the industry. For this reason, no other data format for describing behaviour models will be further investigated in this thesis.

What is tested, however, is the practical use by means of an example with the given software. For this purpose a model of the moving piston is created in *Simulink* and is afterwards exported to the *FMI* format. The final goal of this evaluation is the integration of this model into the PLC run-time.
The model describes the position of the piston and the angle of the disk depending on the velocity of the disk. The rotation speed of the disc is the output of the control on the PLC, where the position of the piston should be controlled. This model in *Simulink* is shown in Fig. 3.5.

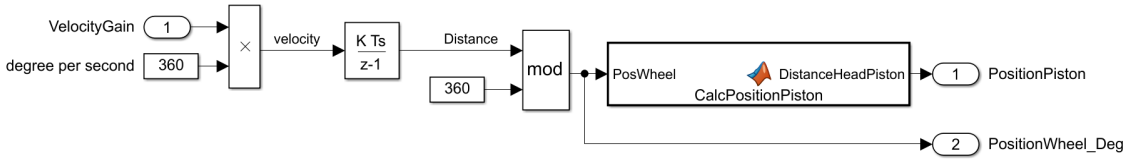


Figure 3.5.: Example for testing the behaviour model.

An overview of the methods to integrate the model in the PLC can be found in Fig. 3.6.

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-time capability is given without any additional steps. Furthermore, possible restrictions and problems can be avoided when using 3rd party tools. In order to integrate the *FMI* model, *Beckhoff* offers the product *TE1420* containing the *FMI* interface to the run-time of a PLC [13]. With this interface it should be possible to create an object for *TwinCAT* directly from the *FMI* file. This object would then have the defined inputs and outputs and could be integrated and linked in the software of the PLC. This would allow to directly use the current values of the behaviour model in the software of the PLC.

Since this interface is still under development at the time of writing this thesis, alternative approaches for integrating an *FMI* file in a virtual commissioning have to be found. This

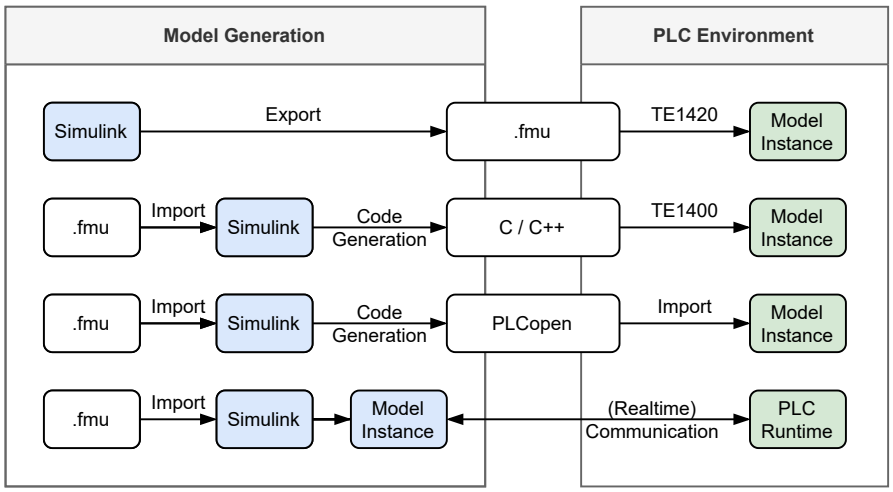


Figure 3.6.: Overview of tested methods for *FMI* handling. In total four approaches are tested in order to use a *.fmu* model in a PLC run time. The blue blocks are required steps in *Simulink* and the green ones are located in the *PLC*.

method is currently rejected due to the missing interface, 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 [14]. 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*. [15].
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 this evaluation, first the original model of *Simulink* is exported and then in the same way the *FMI* object.

The export of the original *Simulink* model works without problems and a *TwinCAT* object can be created. In the same way, the model with the *FMI* object is processed, but In contrast the export of the object fails, due to missing support of the special *FMI* block in the code generation of *Simulink*. As a result, this method is not a suitable solution for the given software in virtual commissioning. However, if a supported model is available in *Simulink* the code generation can be a useful way to include it in *TwinCAT*.

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 original 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* [16] also first for the original model and then for the model with the *FMI* object. When using the PLC coder, attention must be paid to the supported blocks used in the model. For example, continuous blocks can only be used in some cases and must therefore be replaced by discrete alternatives. An example of this is a continuous transfer function which is not supported and must therefore first be discretized.

The used time-discrete model can be exported without major issues similar to 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*, which can also be seen in the list of supported blocks [17]. 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 target 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* [18]. 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 [19].
After the configuration of the *interface* block the PLC variables are available as source/sink in *Simulink* and can be connected to the model. 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. In other words, this method is not an ideal or simple solution, but it provides a feasible way to integrate *FMI* objects into *TwinCAT* and thus perform a virtual commissioning.

3.2.3. PLC Code

The *PLCopen* format already represents a standardized interface for the exchange of PLC code, since the format is well established in the industry and defined by an international standard. For this reason, the review of alternative formats for data exchange can be skipped and only the implementation of the *PLCopen* format in *TwinCAT* needs to be examined.

The export and import is tested using again the moving piston. The aim of the PLC is to set the velocity of the disc in order to control the position of the piston. This is done by defining a variable for the speed and a variable for the position of the piston. Additionally, a variable for the angular position of the disk is defined, which can be used in a visualization. Depending on the set velocity, the program then calculates the position of the piston. The source code is listed in the *PLCopen* format in Sourcecode 3.2.1.

Sourcecode 3.2.1: Example Piston as *PLCopen*-xml

```
<?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://www.plcopen.org/xml/tc6_0200">
  <fileHeader companyName="Beckhoff Automation GmbH" productName="TwinCAT PLC Control" productVersion="3.5.13.21"
      creationDateTime="2022-08-21T13:19:37.4859706" />
  <contentHeader name="mainPLC" modificationDateTime="2022-08-21T13:19:37.487966">
    <coordinateInfo>
      <fbd>
        <scaling x="1" y="1" />
      </fbd>
      <ld>
        <scaling x="1" y="1" />
      </ld>
      <sfc>
        <scaling x="1" y="1" />
      </sfc>
    </coordinateInfo>
    <addData>
      <data name="http://www.3s-software.com/plcopenxml/projectinformation" handleUnknown="implementation">
        <ProjectInformation />
      </data>
    </addData>
  </contentHeader>
  <types>
    <dataTypes />
    <pous>
      <pou name="MAIN" pouType="program">
        <interface>
          <localVars>
            <variable name="PosDegree" address="%I*">
              <type>
                <REAL />
              </type>
              <initialValue>
                <simpleValue value="0" />
              </initialValue>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml"> in deg</xhtml>
              </documentation>
            </variable>
            <variable name="PosPiston" address="%I*">
              <type>
                <REAL />
              </type>
              <initialValue>
                <simpleValue value="0" />
              </initialValue>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml"> in mm</xhtml>
              </documentation>
            </variable>
            <variable name="velocity" address="%Q*">
              <type>
                <REAL />
              </type>
              <initialValue>
                <simpleValue value="0" />
              </initialValue>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml"> in deg/s</xhtml>
              </documentation>
            </variable>
            <variable name="cycleTime">
              <type>
                <REAL />
```

```
                </type>
                <initialValue>
                  <simpleValue value="0.01" />
                </initialValue>
                <documentation>
                  <xhtml xmlns="http://www.w3.org/1999/xhtml"> in sec</xhtml>
                </documentation>
              </variable>
            </localVars>
          </interface>
          <body>
            <ST>
              <xhtml xmlns="http://www.w3.org/1999/xhtml">// Position of Disc
PosDegree := PosDegree + velocity*cycleTime;
WHILE PosDegree &gt; 360 DO
  PosDegree := PosDegree -360;
END_WHILE


// Position of Piston
PosPiston := 13 + COS(PosDegree*pi/180)*9 + SQRT(EXPT(25,2)-EXPT((SIN(PosDegree*pi/180 )*9),2));

</xhtml>
            </ST>
          </body>
          <addData>
            <data name="http://www.3s-software.com/plcopenxml/interfaceasplaintext" handleUnknown="implementation">
              <InterfaceAsPlainText>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">PROGRAM MAIN
VAR
  PosDegree AT %I*  : REAL  := 0;   // in deg
  PosPiston AT %I*  : REAL  := 0;   // in mm
  velocity AT %Q*   : REAL  := 0;   // in deg/s

  cycleTime       : REAL := 0.01;   // in sec
END_VAR
</xhtml>
              </InterfaceAsPlainText>
            </data>
            <data name="http://www.3s-software.com/plcopenxml/objectid" handleUnknown="discard">
              <ObjectId>05518f30-63f5-43e5-a500-2e4c842868d1</ObjectId>
            </data>
          </addData>
        </pou>
      </pous>
    </types>
    <instances>
      <configurations />
    </instances>
    <addData>
      <data name="http://www.3s-software.com/plcopenxml/projectstructure" handleUnknown="discard">
        <ProjectStructure>
          <Object Name="MAIN" ObjectId="05518f30-63f5-43e5-a500-2e4c842868d1" />
        </ProjectStructure>
      </data>
    </addData>
</project>
```

TwinCAT 3 to *PLCopen*

The export and import of *PLCopen* objects is supported in *TwinCAT* and is done without difficulty, whereby instructions for this can be found in [20]. Therefore, using the *PLCopen* format is a suitable solution for data exchange in this case.

3.3. Selected File Formats

As already mentioned in Sec. 2.2, 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 selection of suitable data formats is done.

3.3.1. Geometry Data and Kinematization

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 an ideal 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 if the kinematization should be preserved. 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, *Inventor* offers a quick way to exchange data via the *Pack and Go* tool [21]. 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 *Solidworks* [22]. It should be noted, that although the two functions have the same name, they are not compatible with each other.

An overview of the evaluation of the different approaches can be found in Tab. 3.2.

Table 3.2.: Results of tested methods in CAD exchange evaluated for their usability from bad (○○○) to good (●●●). This table shows the characteristics of the tested methods with respect to save geometry in a CAD file, kinematization of an assembly, a wide support for different tools and an overall evaluation of the usability for a virtual commissioning.

Method	Geometry	Kinematization	wide software support	Usability
a) Using *.step*	yes	no	yes	●●○
b) Using native *Pack and Go*	yes	yes	no	●●○
c) Using *COLLADA* directly	-	-	-	○○○
d) Using *COLLADA* indirectly	yes	no	no	●○○

3.3.2. Behaviour Model

The modeling of the physical behaviour 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, *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 [23] and for the import in [24].
The summary of the evaluation of the tested approached can be found in Tab. 3.3.

Table 3.3.: Results of tested methods in integrating behaviour model evaluated for their usability from bad (○○○) to good (●●●). This table shows the results of different approaches to include the model from *Simulink* or as *FMI* file in *TwinCAT* and afterwards using it in a virtual commissioning. The evaluation is based on the feasibility of the approaches.

Method	Working with *Simulink* model	Working with *FMI* model	Real time	Usability
a) Model directly in *TwinCAT*	-	(yes)	(yes)	(●●●)
b) Model to *TcCom*-Object	yes	no	yes	●○○
c) Model to *PLCopen*	yes	no	yes	●○○
d) Run Model in *Simulink*	yes	yes	no	●●○

3.3.3. 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.

3.3.4. 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.

3.3.5. Overview of Selected Data Formats

An overview of the selected data formats is found in Tab. 3.4.

Table 3.4.: Chosen file formats for the proposed data structure.

Discipline	Information	Choosen File Format
Mech. Engineering	CAD (pure geometry)	Neutral formats like *.step*
	CAD (with kinematics)	Native formats like *.iam* or *COLLADA*
	Physical Behavior	*FMI* as *.fmu*
Elec. Engineering	PLC layout	Native formats or *.pdf*
	List of used inputs and outputs	Native formats or *.pdf*
Coding	PLC Code	*PLCopen* as *.xml*
General	Documentation	*.pdf*

3.4. 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. 3.7.

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 or alternatives are 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 represents the import of a *COLLADA* file, due to the lack of conversion tools.
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.

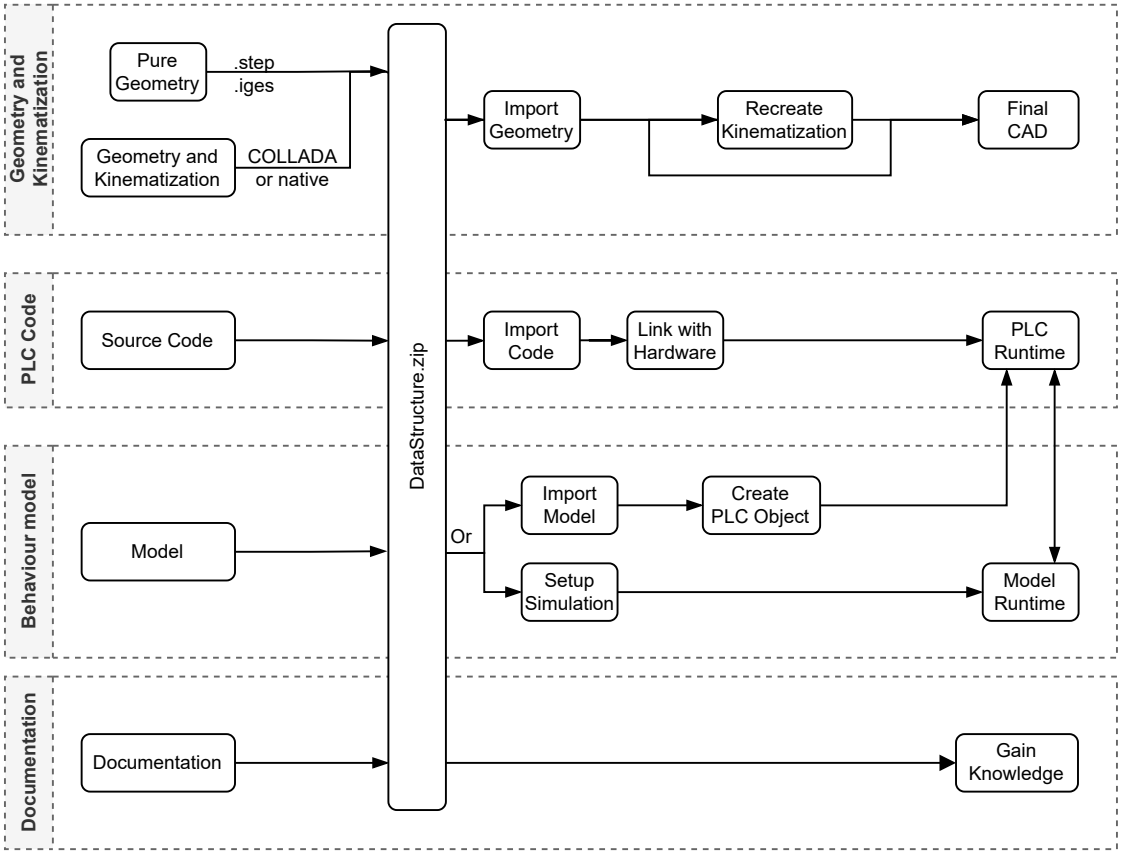


Figure 3.7.: General workflow of proposed data structure. In this picture the most important aspects for the export and import of the data structure are shown and divided into the corresponding categories.

3.5. Summary

This chapter describes the layout 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 behaviour model (*FMI*), the control software (*PLCopen*) and the associated documentation (*.pdf*). Compressing this structure into a *.zip* file simplifies even more the distribution and sharing. In the case of CAD data, no optimal solution has been found yet for the selection of data formats, since kinematization is currently only supported with major problems. However, various approaches to solve this problem are in development but still have to prove their usability in practice. The selection for the format for the behaviour models is made on the already widely used *FMI* interface and the format of the CAD data is even internationally standardized as *PLCopen*. Nevertheless the integration of a *FMI* model in a PLC run-time is still under development and therefore alternatives must be found at the moment.

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.

4. Exemplary Application using the Proposed Data Structure

4.1. Introduction

The practical use and the individual steps in the implementation of the proposed data structure are shown by means of an example. In order to represent a real plant a *Teaching Factory* is used. The aim 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, varying from basic topics like digital and analog signals to also more complex tasks like motor control and serial communication. Furthermore, human interaction and safety are included learning contents. An overview of the whole plant is shown in Fig. 4.1.

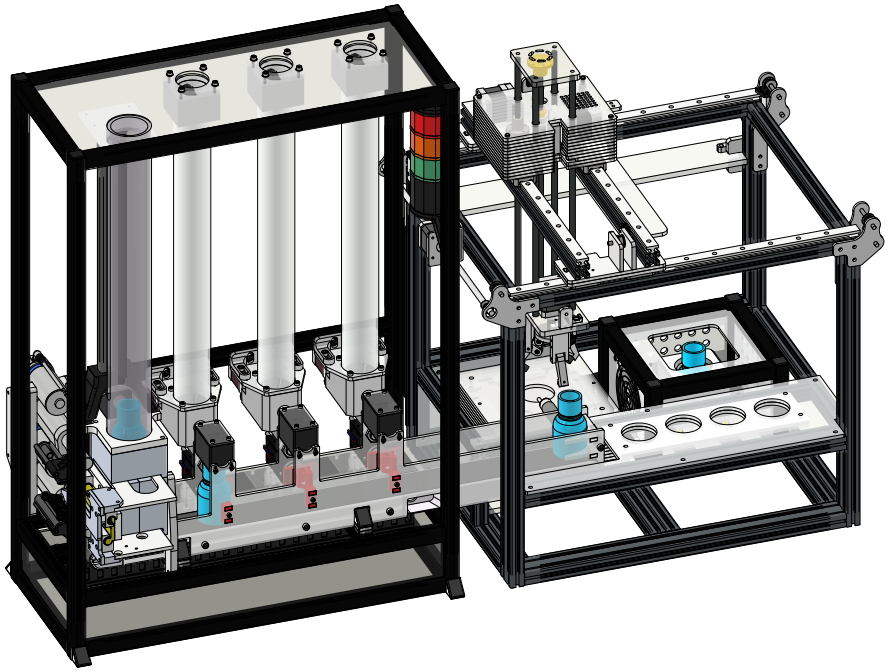


Figure 4.1.: The used *Teaching Factory*. On the left is the module *Separation* followed by the *Conveyor Belt* with three *Dosing Units*. The right half contains the *Cartesian Gripper* with a *Load Cell* (left), *Thermal Processing* (rear right) and an output storage (front right). For a better overview, further attachments and covers are hidden in this picture.

4.1.1. Module *Separation*

The first module offers a fast introduction to the programming of a PLC. Here, the first contact with a PLC can be made and at the same time digital signals for input and output can

be used. The task in relation to the *Teaching Factory* is the separation of container from an input storage to the following conveyor belt. The mechanical design of this module is vertically oriented and consists of the input storage in the upper area and the separation in the lower area. The mechanical layout of the module and relevant components for the control are shown in Fig. 4.2.

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.

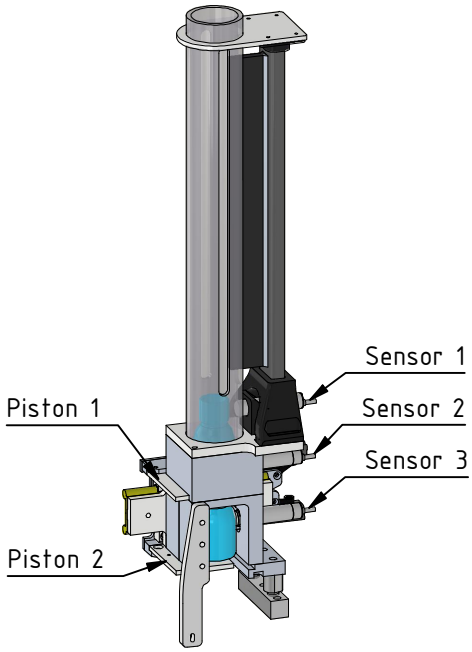


Figure 4.2.: Overview module *Separation* with the three proximity sensors and the two pistons realizing the separation process.

4.1.2. Module *Conveyor Belt*

This module is the user's first introduction to the control of electric drives. Using a DC motor with its comparably simple design allows the theoretical background easier to understand and the main focus can lay on the controlling. In addition, an encoder is attached to the motor axis to determine the actual speed and position of the motor. By this measurement of the position, an optional servo control of the conveyor belt can be implemented in a further step and thus enable an exact positioning. Moreover, through the feedback of the actual speed of the motor, an undesired stop can be detected immediately. In a second step, the knowledge of motor control is further expanded with stepper motors.

The conveyor belt is the main part of the filling process and is divided into three sections: input, filling (three dosing units) and output. At the beginning of the conveyor belt, the containers are passed from module *separation*. After that, the containers go through three identical dosing units used for filling. These dosing units each consist of a helix driven by a stepper motor. In order to simplify the filling process of a container, a stopper and a proximity sensor are located under each dosing unit on the conveyor belt. The stopper normally locks the container and can be retracted via a digital signal. The proximity sensors are identical to those of the module *separation* and provide a digital output signal when a container is detected. This combination of sensors with the stoppers and the conveyor makes it possible to precisely place a container under a dosing unit. At the end of the conveyor belt, the containers are passed on to the next module, where a proximity sensor is also placed at the defined pick-up position.

In this example, the user has to create the control of the conveyor belt in the first step. After that, the software is extended with the digital signals of the stoppers and proximity sensors. Finally, the control for the dosing units has to be implemented, where large parts of the control are identical to the conveyor belt. The mechanical layout and relevant components are shown in Fig. 4.3.

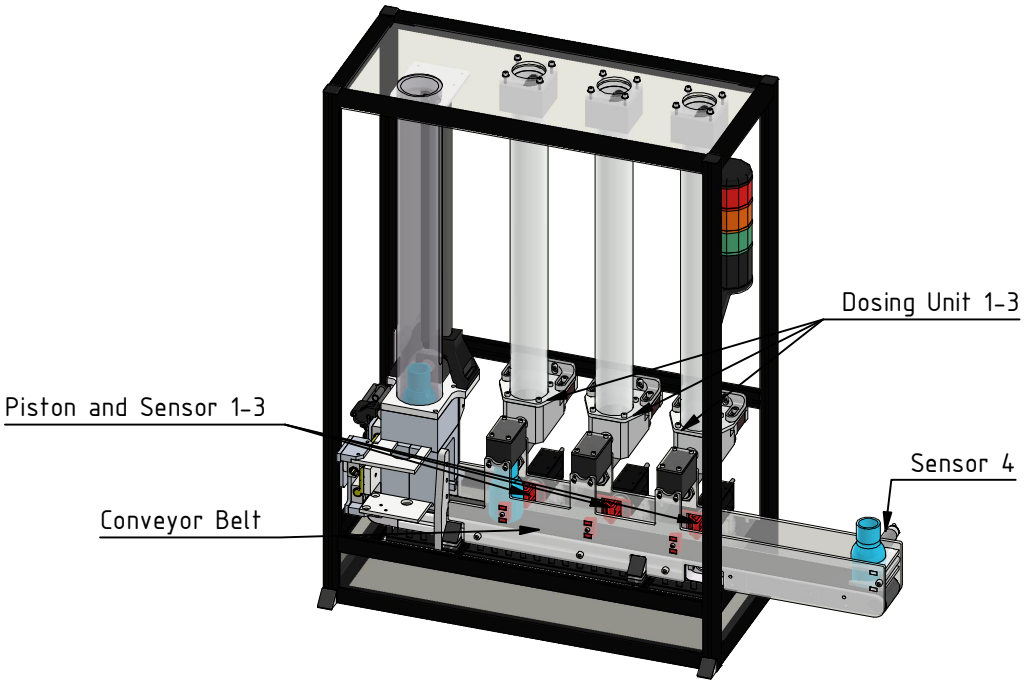


Figure 4.3.: Overview module *Conveyor Belt*. On the left side teh module *Separation* is shown. In the middle the conveyor belt is located with the three dosing units for filling. On the right side the pick-up position to the next module is shown at sensor 4.

4.1.3. Module *Cartesian Gripper*

In this module, the subject of controlling a stepper motor is further discussed and expanded. Topics such as moving to a specific position and the zero point can be covered

in this module. The mechanical layout itself consists of a gripper and three axes for movement in the X, Y and Z direction. The axes are controlled by a stepper motor and have two limit switches for safety purposes. The gripper itself is also driven by a stepper motor including two limit switches. The mechanical layout of the module and relevant components for PLC coding are shown in Fig. 4.4.

The *cartesian gripper* is the main component following the filling of the containers and is responsible for the further transport of these to the final modules. Therefore, the gripper has to pick up the containers from the conveyor belt and place them sequentially in the modules *load cell* and *thermal processing*. Finally, the container has to be placed in the output storage. In order to ensure sufficient positioning of the containers in the modules, the PLC controller must be able to position the gripper precisely. A correctly set reference point for each axis is an essential requirement for this.

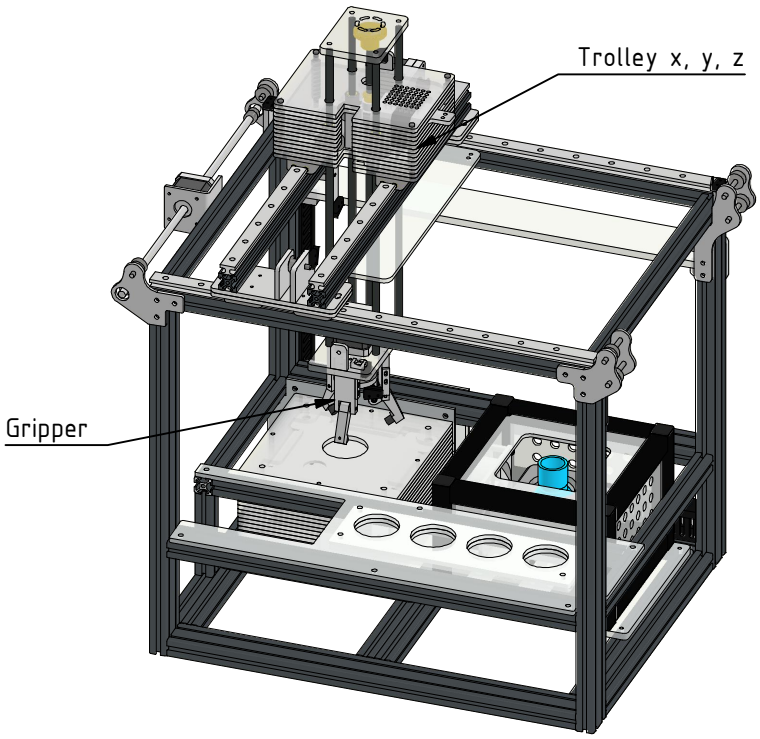


Figure 4.4.: Overview module *Cartesian Gripper* with the following modules and the trolley. In the lower left corner the container are picked up from the conveyor belt.

4.1.4. Module *Load Cell*

In this module the filling level of a single container is determined by means of a weight measurement. A laboratory scale is used as load cell, which sends the current measurement result as text via a serial interface and allows the user to get in touch in communication with external devices. As already mentioned, the container positioning is done by the *cartesian gripper*. The layout of this module is shown in Fig. 4.5.

The task of the user is to establish communication with the load cell via an RS-232 inter-

face. Furthermore, the scale must be calibrated by sending a defined command and the received messages must be interpreted. Only if the filling level is correct, the containers are passed to the module *thermal processing*.

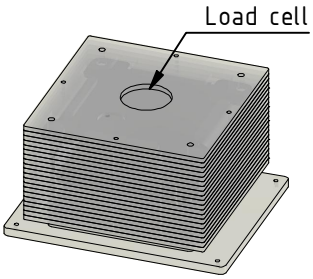


Figure 4.5.: Overview module *Load Cell*. The containers are placed in the middle of the platform.

4.1.5. Module *Thermal Processing*

After checking the filling level, a thermal treatment of the containers is done in this module, using four heating cartridges and two thermocouples. These components are used to raise the temperature of the containers to a defined value. The interface for the heating cartridges and the thermocouples are analog signals and thus the user is able to work on this module at an early stage, although it is the last step in the manufacturing process. The structure and the relevant components are shown in Fig. 4.6.

As mentioned earlier, the task of the user here is to use the analog signals to finally establish a temperature control. In this context, *Timer* components may also be used, for example, to maintain the temperature for a defined period of time.

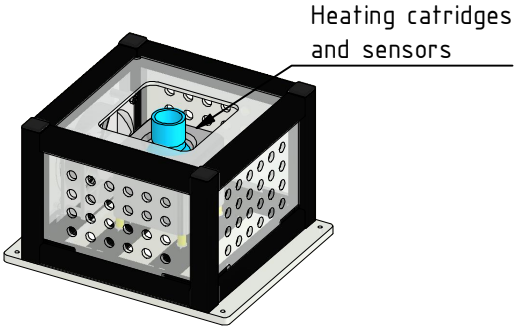


Figure 4.6.: Overview module *Thermal Processing*. The heating cartridges, thermocouples and the container are located in the central block. For safety purpose and in order to cool the block down fans are mounted on the side.

4.1.6. Setup for Testing

A list of the used software for this example can be found in Tab. 4.1.

Table 4.1.: 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 physical setup for the virtual commissioning consists of two real-time capable industrial PCs (IPC) on which the *TwinCAT* run-time is executed. The control software to be tested is executed on the first IPC and the model of the plant is calculated on the second, thereby creating an HIL system. The communication between the two IPCs takes place in real-time via the EtherCAT Automation Protocol (EAP) [25]. The *TwinCAT* projects for both IPCs are written on an engineering PC and then loaded onto the two IPCs. For the visualisation the product *TE1130* is used, which shows the current state of the PLC directly in *Inventor* [2]. To do this, the value of a PLC variable is written to a constraint in the CAD assembly and thus the position of a component is set. This physical setup is shown in Fig. 4.7.

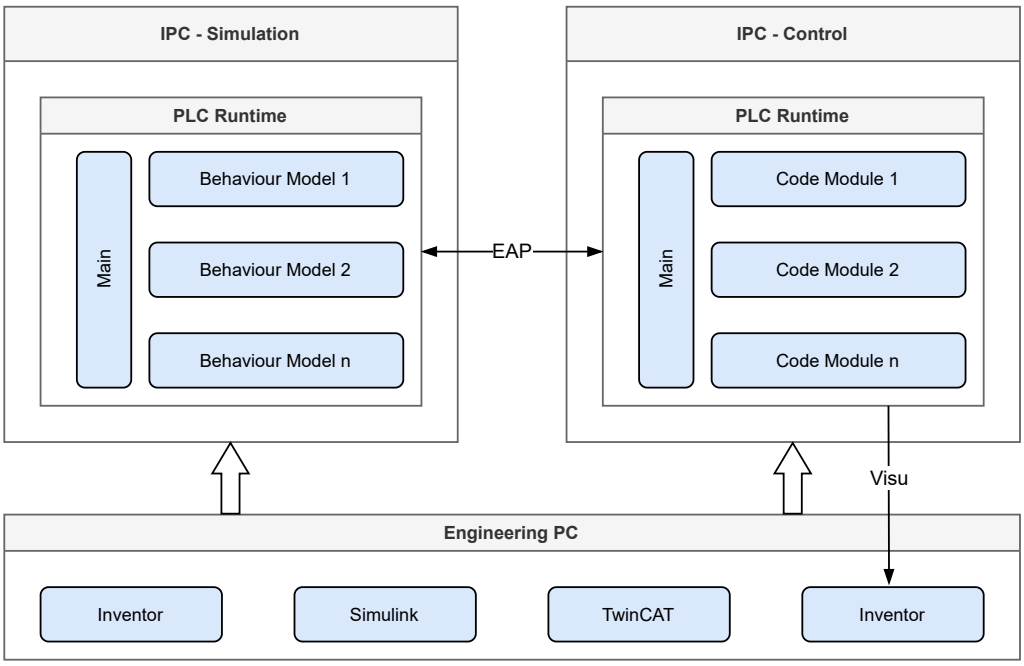


Figure 4.7.: Hardware setup for this example as HIL system. It consists of two real-time capable IPCs running the plant model and the PLC control and an engineering PC for the visualization and the development of the PLC code. The communication between the IPCs is done vie EAP in real-time.

The integration of the model description from a *.fmu* file into a PLC project is done using a own product called *TE1420* for a *Beckhoff* PLC. With the help of this product a *TcCom* object is created, which can then be integrated in TwinCAT. The inputs and outputs from this object are the same as the description from the original *.fmu* file.
At the time of writing this thesis, the *TE1420* product is still in development and therefore cannot be used for this example. As an alternative, code generation from *Simulink* and the product *TE1400* is used, in order to create a *TcCom* object directly from the source

model in *Simulink*. Thus, the generation of this *TcCom* object is slightly different in the alternative, but the further use is identical. A limitation of this alternative is the mandatory use of *Simulink* for the model description. The comparison between the ideal way and the alternative code generation is found in Fig. 4.8.

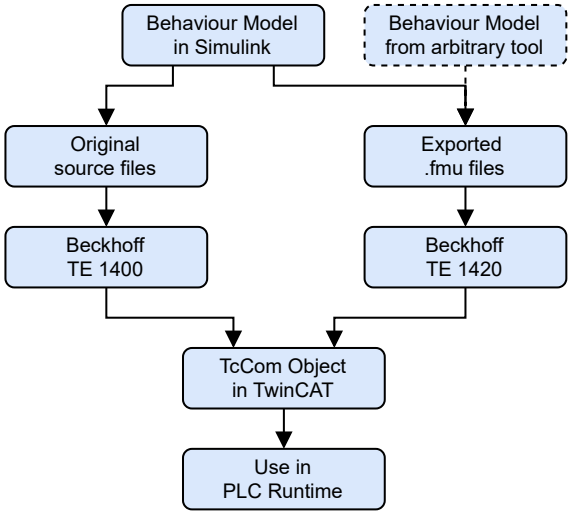


Figure 4.8.: Comparison of the integration of the physical models in *TwinCAT*. Instead of using a *.fmu* file to create a *TcCom* object, the original source from *Simulink* is used.

4.2. Creation of the Data Structure - Workflow of a Manufacturer

In this section the data structure for the exchange is generated. This structure consists of the CAD data, the behaviour model and source code for a PLC as described in chapter 3. The export of the data is done from the view of a supplier or transmitter.

For exporting the CAD data, this example uses the native format of *Inventor* for kinematized models. That is based on the assumption that the transmitter and receiver of the data structure both use *Inventor*. The structure of the assembly corresponds to the desired kinematization, taking into account required degrees of freedom. If the assembly is purely static and thus no kinematicization is required, the *.step* format is used for exchange. This also simplifies the structure of the assembly, whereby logically linked components are bundled into individual assemblies. The export to the *.step* format is done using the instruction [8] and to the native exchange format using the instruction [10].

As already mentioned, in this example the original models from *Simulink* are used as an alternative for the integration into *TwinCAT*. However, the behaviour model is exported as intended as a *.fmu* file and included in the data structure. This is done according to the instructions [23], using a *fixed-step* solver with the step size of $10\,\mathrm{ms}$. The step size corresponds to the cycle time of the PLC.
The interface of the models reproduces the signals of the real hardware and describes the given system. In addition, some purely virtual variables are used, which are useful for higher-level control. The level of detail can be arbitrarily precise, whereby the increasing

complexity of the required calculations must be taken into account.

The source code of the PLC is exported to the *PLCopen* format for exchange. Instructions for this are provided by [20].

4.2.1. Module *Separation*

CAD Data

The CAD model is built based on the desired kinematics of the pistons. In this case, the two pistons each represent a separate assembly to allow easier maintenance and use. As mentioned earlier, the assembly is exported in the native format of *Inventor* based on the desired kinematics and bundled in a *.zip* file.

Behaviour Model

The behaviour 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 entire model with the inputs and outputs is shown in Fig. 4.9.

For this module, the interface consists of digital outputs for moving the two pistons and digital inputs for each of the three proximity sensors. Furthermore, 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 module.

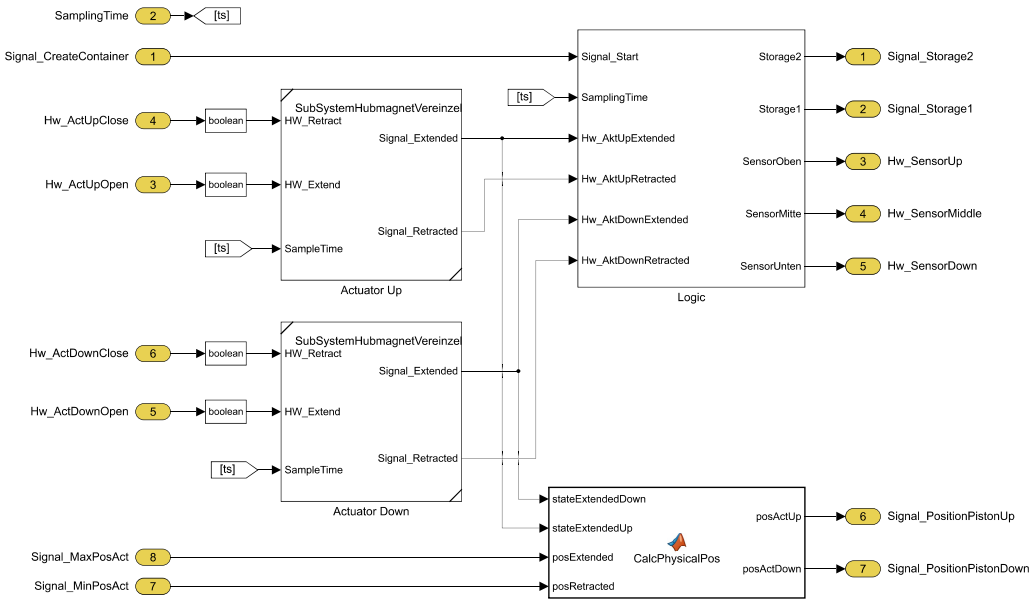


Figure 4.9.: Behaviour model of module *Separation* describing the physical pistons and the logic of the input storage.

PLC Code

For the exemplary control of the *separation* a *function block* is created. 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 resulting *.xml* file is listed in Sourcecode A.1.1.

Resulting File

The generated information of the CAD model, behaviour and control are now bundled in the proposed data structure from chapter 3. The resulting structure for this module is shown in Fig. 4.10.

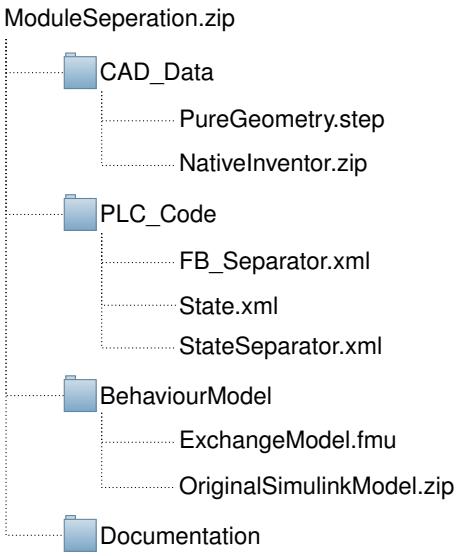


Figure 4.10.: Data structure of module *Separation* consisting of CAD data as pure geometry and including kinematization in a native format, PLC code and the behaviour model as *.fmu* and original *Simulink* files. Additional documentation for this module is not needed resulting in an empty directory.

4.2.2. Module *Conveyor Belt*

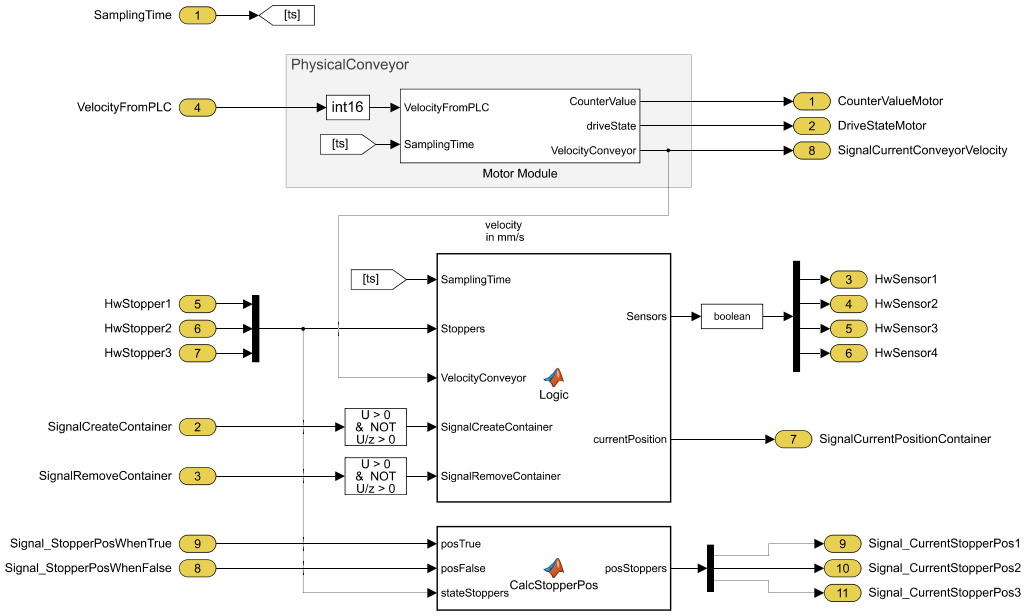
CAD Data

In this module the CAD model of the conveyor belt itself is provided by the manufacturer and integrated in the assembly. The three sliders of the stoppers form again a sub-assembly to simplify the kinematization. The conveyor itself does not need any additional kinematization. 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. The three *dosing units* are also represented as a own sub-assembly.

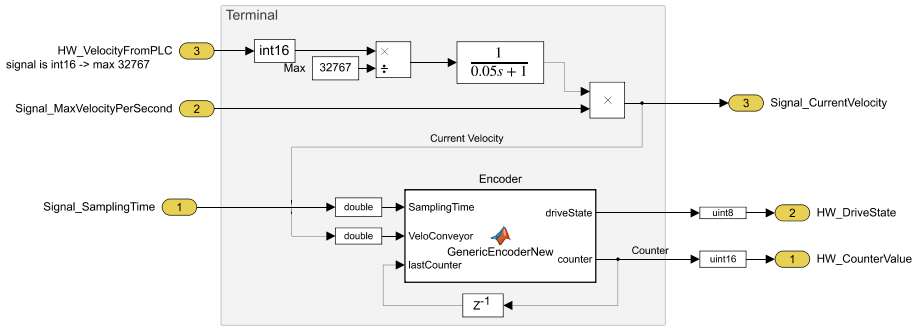
Since the kinematics of the pistons are supposed to be exported in this example as well, the native CAD format is kept. Alternatively, a pure exchange of the geometry with an *.step* file can be done.

Behaviour Model

The behaviour model in this module consists of the DC motor of the conveyor belt, a generic drive for the dosing units and a logical combination of the pistons and sensors. The final model in *Simulink* is shown in Fig. 4.11. The modeling of the DC motor is similar to this example from the Matlab documentation [26], using the characteristics from the data sheet of the used motor [**?**].
The stepper motors of the dosing units are not modeled as detailed as the DC motor and therefore use a generic description, which also results in a reduction of the required computing power.



(a) Conveyor Belt



(b) Dosing Unit

Figure 4.11.: Behaviour model of module *Conveyor Belt* describing the DC motor of the conveyor belt and the stepper motor of the *dosing units*.

PLC Code

The PLC code in this module consists of two *function blocks*: one block for the *conveyor belt* and one block for the *dosing units*. The functionality of these blocks consists in the initialization of the motors and the movement with a constant speed for the conveyor belt and the traveling of a defined distance for the dosing units. The linking of these blocks remains the customer's task and is not supplied by the manufacturer. The two *function blocks* are listed in *PLCopen* format in Sourcecode A.1.2 and Sourcecode A.1.3.

Resulting File

The collected information is now merged into the proposed data structure. This data structure is shown in Fig. 4.12 and consists of the CAD assembly, the behaviour model and PLC code. In addition some documentation is attached to the data structure.

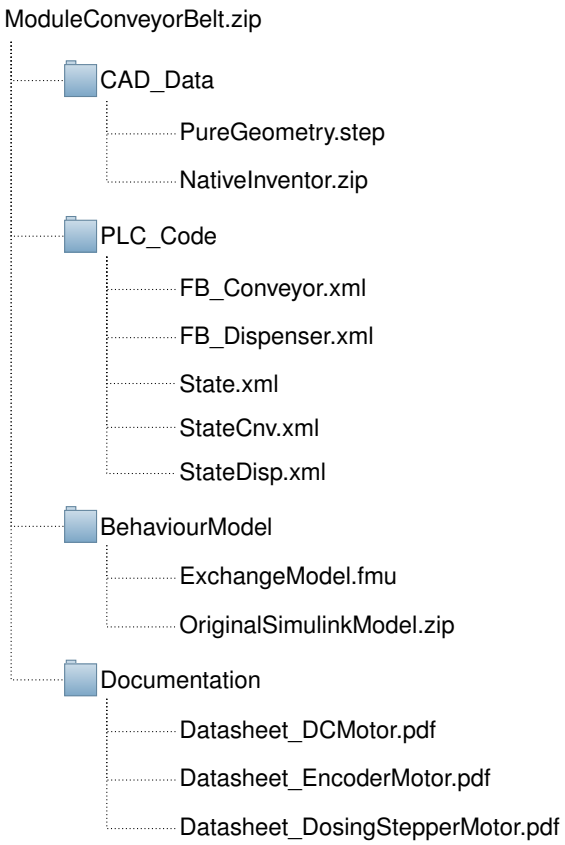


Figure 4.12.: Data structure of module *Conveyor Belt*consisting of CAD data as pure geometry and including kinematization in a native format, PLC code, the behaviour model exported to a *.fmu* file and the original source from *Simulink* and additional documentation.

4.2.3. Module *Cartesian Gripper*

CAD Data

Similar to the previous modules, the structure of the CAD data reflects the desired kinematization. For this reason, the *cartesian gripper* consists of three assemblies for the respective axes and one for the remaining components. The degrees of freedom of the three axes are restricted to the desired direction of motion. Since the CAD data along with the kinematization will be sent to the customer, the native *Inventor* format is used again.

Behaviour Model

The behaviour model of the *cartesian gripper* describe the three axes of motion and the gripper itself. In the process, a stepper motor is merged with the terminal of the PLC into one model and described generically. As a result of the generic model, the level of detail is reduced and so is the complexity of the calculation. Furthermore, the generic model is independent of the selected motor type.

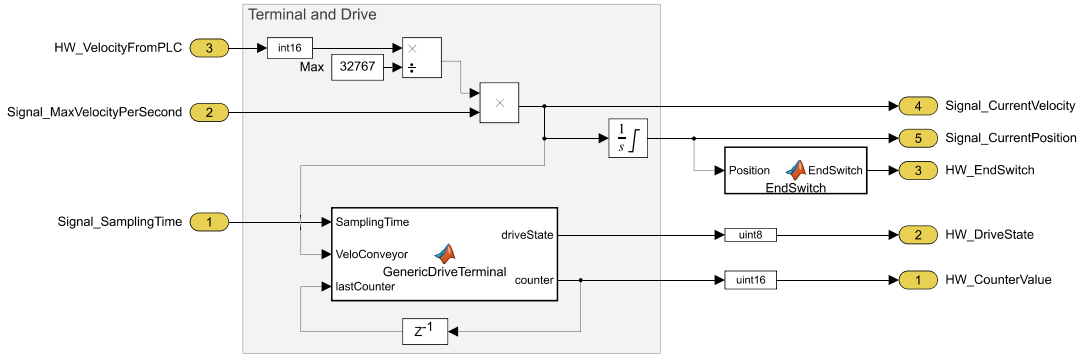


Figure 4.13.: Behaviour model for module *Cartesian Gripper* describing a generic motor merged with a PLC motor terminal.

PLC Code

In this module, a *function block* is provided for controlling a motor axis, where this block controls only one axis and must be instantiated later for each motor. This block thereby independently performs the homing to a reference zero allowing an accurate movement to a target point with a desired speed. Again, the control is exported to the *PLCopen* format and is listed in Sourcecode A.1.5.

Resulting File

The last step is now to merge the data into the proposed data structure from chapter 3. This structure is shown in Fig. 4.14 and consists again of the CAD assembly, the behaviour model and source code for a PLC.

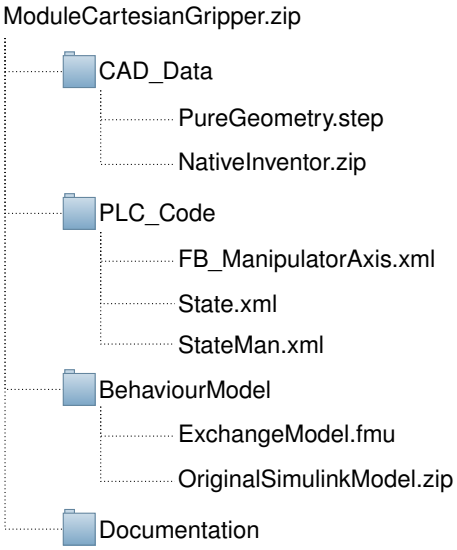


Figure 4.14.: Data structure of module *Cartesian Gripper* consisting of the pure geometry and included kinematization in a native format, PLC code and the behaviour model exported to a *.fmu* file and the original source files from *Simulink*. The folder of additional documentation is also empty in this module.

4.2.4. Module *Load Cell*

CAD Data

Since no kinematization is required for this module, no special steps are necessary in the construction of the CAD assembly. In addition, data exchange is simplified since a *.step* file can be used without losing any relevant information.

Behaviour Model

The generation of the behaviour model of this module is similarly simple, because the serial communication itself is not modeled but only the message as *string*-type for debugging purpose. With this behaviour model shown in Fig. 4.15 a message can be created with a defined prefix and suffix and a desired measured value, which later has to be interpreted by the PLC.

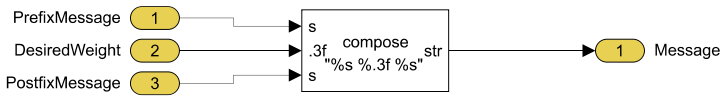


Figure 4.15.: Behaviour model of module *Load Cell* consisting of composing a serial message with the measured weight.

PLC Code

Here the manufacturer provides only the hardware and the documentation of the communication, but no code for the PLC.

Resulting File

Also in this module the last step consists of merging the data into the data structure. The resulting structure is shown in Fig. 4.16 and consists of the CAD assembly, the behaviour model and additional documentation.

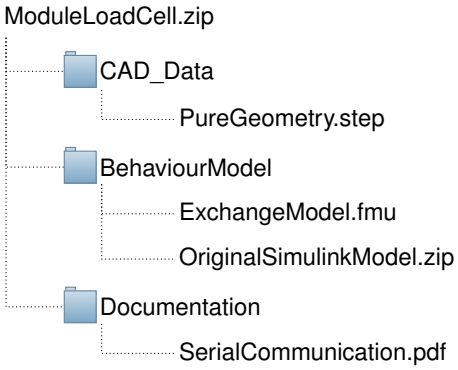


Figure 4.16.: Data structure of module *Load Cell* consisting of the pure geometry, the behaviour model as exported *.fmu* file and the original source in *Simulink* and some documentation.

4.2.5. Module *Thermal Processing*

CAD Data

This module is also based on static components and therefore has no kinematization. This simplifies on the one hand the setup in the CAD software and on the other hand the export of the assembly. As a result, no special requirements for the CAD model are needed and the export can be performed in the *.step* format without the loss of relevant information.

Behaviour Model

However, the behaviour model is slightly more complex with the modeling of the temperature curve. Taking into account the room temperature and the maximum reachable temperature of the cartridge heaters, the temperature of the containers is described with a transfer function. Finally, this temperature has to be converted into the corresponding signals of the sensors. The entire model in *Simulink* is shown in Fig. 4.17.

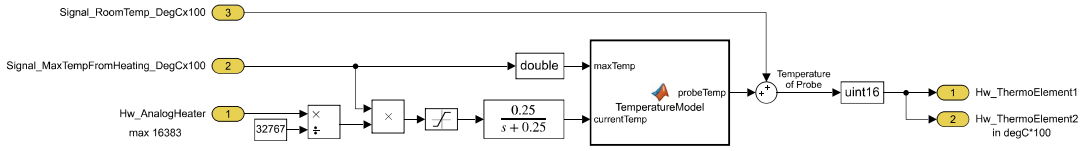


Figure 4.17.: Behaviour model of module *Thermal Processing* describing the temperature curve of a container with the influence of the used heating cartridges and the resulting sensor signals.

PLC Code

With this module the PLC code is again provided in the form of a *function block*. This block is listed in Sourcecode A.1.5 and enables the initialization and operation of the module. In the process, the containers are heated to a defined temperature and then held for a certain time.

Resulting File

Finally, the collected data is merged into the data structure. This structure is shown in Fig. 4.18 and consists of CAD data, behaviour model and PLC code.

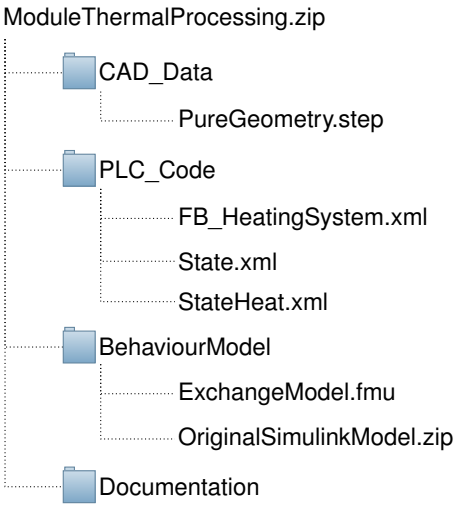


Figure 4.18.: Data structure of module *Thermal Processing* consisting of pure geometry, PLC code and the behaviour model as exported *.fmu* file and the original source from *Simulink*. Also in this module no further documentation is needed.

4.3. Implementation of the Data Structure - Workflow of a Customer

In this section, the data structures shown in Sec. 4.2 are now send to the receiver and used in a virtual commissioning. This is done from the perspective of a customer or receiver.

4.3.1. A Single Module

For the CAD data, as already mentioned, it is assumed that both sides use *Inventor*. This way the native format can be kept and in addition to the geometry also the kinematization is preserved. For purely static modules without kinematization, the *.step* format is used. The import of a *.step* file is done using the instructions [9].

Since in this example the customer uses a PLC from manufacturer *Beckhoff*, the behaviour model is integrated directly from the *Simulink* files into the PLC run-time. As

said, the integration of *.fmu* files is not possible at this time due to ongoing development of the required product. The integration of the *Simulink* models into *TwinCAT* is done analog to this instruction [27].

The *function blocks* of the PLC control can be easily integrated and instantiated in an existing *TwinCAT* project. After assigning the interface variables, the code can be used.

4.3.2. Complete Factory - Combining the Modules

For the mechanical planning of the plant, the individual modules are combined and positioned in an assembly. This process is a standard activity for an average user of *Inventor*.

As already mentioned in Sec. 4.1.6, virtual commissioning takes place in a HIL system. Here, the PLC control and the behaviour models of the plant are executed on two different IPCs, which exchange the respective state via the real-time capable EAP interface. A publisher and subscriber are created on both IPCs and linked to the respective variables of the process image in order to establish communication.

On the IPC with the simulation of the plant first a new *TwinCAT* project is created and the *TcCom* objects of the individual modules are instantiated. These objects are then assigned to a task with $10\,\mathrm{ms}$. The outputs from this model are set as publish variables and the inputs as subscribe variables in the EAP communication.
On the second IPC with the controlling software a new *TwinCAT* project is also created and the clock time is set to $10\,\mathrm{ms}$. In this project now the *function blocks* of the individual modules are imported and called cyclically in the *MAIN* program. Furthermore, in the *MAIN* program the higher-level control between the modules is created and linked to the inputs and outputs of the *function blocks*.
For the used drives several NC axes must be added in *TwinCAT* and parameterized with the corresponding settings from the individual data sheets.
To establish EAP communication with the simulation of the plant, the outputs of the controller (actuators) are set as publish variables and the inputs (sensors) as subscribe variables. Afterwards, they are linked to the according variables in the process image.

As soon as the two IPCs are configured, the runtime is started. From this point on, the plant is simulated on the first IPC and the control on the second IPC can be tested. Finally, a virtual commissioning can be performed with this setup.

4.3.3. Visualization in *Inventor*

As mentioned before, the current state of the PLC is shown directly in *Inventor* in this example. Therefore the desired variables of the PLC are linked with the constraints in the CAD assembly according to the instruction [28]. It should be noted here that in the current beta version of the used product only values of an NC axis can be exported from *TwinCAT*. As a result, for each variable that should be read from *TwinCAT*, an NC axis and a link between the actual value of the NC axis and the variable must be created. This finally makes the variable in *Inventor* available. A screenshot from *Inventor* with the shown state of the PLC can be found in Fig. 4.19.

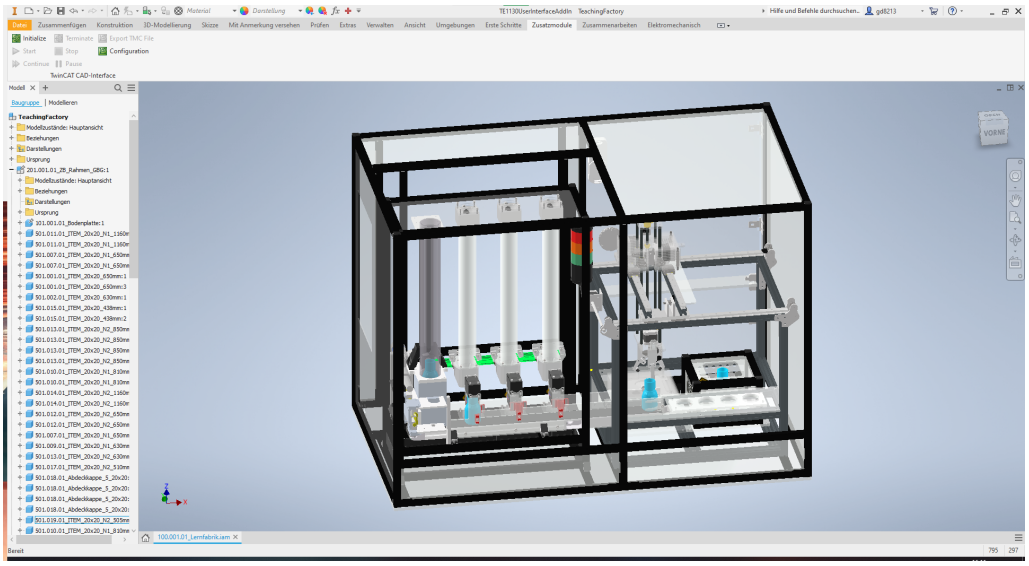


Figure 4.19.: Screenshot of the visualization in *Inventor*, where the position of the pistons and the cartesian gripper is set from the PLC.

4.4. Summary

In this chapter, the proposed data structure is used in a real-life application and tested for its practical usability. An exemplary plant is divided into several modules and a data structure is created for each module and prepared for exchange. For this purpose, the CAD data with and without kinematics are exported, the behaviour model is created and the control code for the PLC is written. In the second step, this information is then imported from that data structure and integrated into a setup for a virtual commissioning. The simulation of the plant is executed in a second IPC in parallel with the control PLC, with the necessary communication taking place in real-time via EAP.

5. Results and Evaluation

The result of this thesis is the proposed data structure shown in chapter 3, bundling information from the CAD design, behaviour model, PLC source code and optional documentation. The structure is modular designed and data is stored in compatible formats. The practical use of the data structure is also investigated in this thesis using an example in chapter 4. To evaluate the benefits of this data structure, this example is evaluated using the criteria from Sec. 2.3.

Through the evaluation it is stated that the data structure can basically be used in a SIL and HIL system, because no dependencies to the used hardware exists. The advantage in a HIL system is the additional computing power for the simulation of the plant with the disadvantage of additional communication.
The data required for virtual commissioning, such as CAD assemblies, behaviour models, PLC code and documentation, can be represented in the data structure without problems. As shown in the example, individual modules of a plant can be reproduced with the data structure and finally integrated into a higher-level system.
However, there is still room for development in the compatibility of the data formats, especially in the area of kinematized CAD data and in the integration of *.fmu* models in a PLC project. While the integration of behaviour models in the case of *Beckhoff* is only a matter of time, the exchange of kinematization in CAD data represents a bigger challenge. In this case, a practicable solution has not yet been established in the industry.
In terms of the requirements for the additional know-how needed, no further skills are required in the case of the CAD data and the pure PLC code. When creating the behaviour models, basic knowledge of programming a PLC is an advantage but not absolutely necessary. However, the integration of the behaviour models into the PLC project requires additional knowledge, which can be acquired in a seminar, for example.
The result of this evaluation is summarized in Tab. 5.1. This evaluation is valid for the used software versions shown in Tab. 3.1, where upward compatibility is likely but downward compatibility is not guaranteed.

Table 5.1.: Evaluation of proposed data structure with respect to the shown example from bad (○○○) to good (●●●). Although the defined information is represented in the data structure, compatibility could be improved, especially for kinematized CAD data.

Criterion	Evaluation
Independent of hardware setup (HIL/SIL)	●●●
Represent all relevant data	●●●
Modular layout to represent components	●●●
High compatibility	●○○
Low level of additional knowledge	●●○

In general, with this data structure, the difficulties of setting up a virtual commissioning can be reduced, which simplifies its implementation. This has the advantage of minimizing the time and cost of commissioning a new plant.

6. Summary and Outlook

6.1. Summary

In the scope of this master thesis, a data structure for the exchange of relevant information for a virtual commissioning is developed. This structure is focused on high compatibility between different tools and has a modular structure. The covered information of the data structure includes the mechanical design represented by the CAD data, physical behaviour models of different used components and source code for the control via a PLC. Additional information such as data sheets can be added to the data structure as required.

The presented data structure is evaluated on the basis of an example for its usability. The result of this evaluation shows potential for improvement, especially in the exchange of kinematized CAD assemblies. Besides the native formats there is no established solution available at the moment. Furthermore, the integration of *.fmu* models into a PLC run-time is also under development in the case of *Beckhoff* but suitable alternatives are found.

6.2. Outlook

The next possible steps of this thesis are on the one hand further research on the exchange of kinematized CAD assemblies based on a neutral data format and on the other hand further evaluation of the data structure with respect to bigger use cases and additional software for the integration of the behaviour models in the PLC. This can also further investigate compatibility of different versions of the software used.

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] Beckhoff Automation GmbH, “TE1130 | TwinCAT 3 CAD Simulation Interface,” Mar. 22 2022, product data sheet.

[3] 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

[4] M. Barnes and E. L. Finch, “COLLADA - Digital Asset Schema Release 1.5.0 Specification.” Sony Computer Entertainment Inc., 04 2008.

[5] Modelica Association Project, *Functional Mock-up Interface Specification, Version 3.0, 2022-05-10*.

[6] S. Faltinski, O. Niggemann, N. Moriz, and A. Mankowski, “AutomationML: From data exchange to system planning and simulation,” 03 2012.

[7] A. Luder, N. Schmidt, and R. Rosendahl, “Data exchange toward PLC programming and virtual commissioning: Is AutomationML an appropriate data exchange format?” in *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*. IEEE, Jul. 2015.

[8] Autodesk Help. Export data to other formats. Visited on May 26 2022. [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

[9] ——. To Import STEP or IGES Data (Construction Environment). Visited on May 26 2022. [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

[10] ——. To Use Pack and Go to Package Files. Visited on May 26 2022. [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

[11] Simlab 3D Plugins. Inventor Collada Exporter Plugin. Visited on Jun. 1 2022. [Online]. Available: https://www.simlab-soft.com/3d-plugins/Inventor/Collada_exporter_for_Inventor-main.aspx

[12] Autodesk Help. Importing Autodesk Inventor Files. Visited on May 26 2022. [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

[13] Beckhoff Automation GmbH, “TE1420 | TwinCAT 3 Target for FMI,” Mar. 22 2022, product data sheet.

[14] Beckhoff Information System. TE1400|TwinCAT 3 Target for Simulink. Visited on May 26 2022. [Online]. Available: https://infosys.beckhoff.com/content/1033/te1400_tc3_target_matlab/index.html?id=7328785815492855617

[15] ——. TE1400|TwinCAT 3 Target for Simulink - Samples. Visited on May 26 2022. [Online]. Available: https://infosys.beckhoff.com/content/1033/te1400_tc3_target_matlab/10825692555.html?id=382902774554596767

[16] Mathworks Help Center. Simulink PLC Coder - Generate Structured Text Code for a Simple Simulink Subsystem. Visited on May 26 2022. [Online]. Available: https://de.mathworks.com/help/plccoder/ug/generating-structured-text-for-a-simple-simulink-subsystem.html

[17] ——. Simulink PLC Coder - Supported Blocks. Visited on May 26 2022. [Online]. Available: https://de.mathworks.com/help/plccoder/ug/supported-simulink-blocks.html

[18] Beckhoff Information System. TE1410|TwinCAT 3 Interface for MATLAB/Simulink - ADS blocks. Visited on May 26 2022. [Online]. Available: https://infosys.beckhoff.com/content/1033/te1410_tc3_interface_matlab/11512422539.html?id=6289303933073219953

[19] ——. TE1410|TwinCAT 3 Interface for MATLAB/Simulink. Visited on May 26 2022. [Online]. Available: https://infosys.beckhoff.com/content/1033/te1410_tc3_interface_matlab/index.html?id=981623218745783434

[20] ——. TwinCAT 3|PLC - Exporting and importing a PLC project. Visited on May 26 2022. [Online]. Available: https://infosys.beckhoff.com/content/1033/tc3_plc_intro/2526208651.html?id=2445988504692268481

[21] Autodesk Help. About Packaging Files with Pack and Go. Visited on May 23 2022. [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

[22] Solidworks Help. Pack and Go - Übersicht. Visited on May 23 2022. [Online]. Available: https://help.solidworks.com/2021/german/SolidWorks/sldworks/c_pack_and_go.htm

[23] Mathworks Help Center. Export Simulink Model to Standalone FMU. Visited on May 23 2022. [Online]. Available: https://de.mathworks.com/help/slcompiler/ug/simulinkfmuexample.html

[24] ——. FMU Importing. Visited on May 23 2022. [Online]. Available: https://de.mathworks.com/help/simulink/in-product-solutions.html

[25] Beckhoff Information System. TE1000|TwinCAT 3 EAP - Manual. Visited on Jul 29 2022. [Online]. Available: https://infosys.beckhoff.com/content/1033/eap/index.html?id=283054218525729603

[26] Mathworks Help Center. Control System Toolbox - DC Motor Control. Visited on May 26 2022. [Online]. Available: https://de.mathworks.com/help/control/ug/dc-motor-control.html

[27] Beckhoff Information System. TE1400|TwinCAT 3 Target for Simulink - Quickstart . Visited on May 26 2022. [Online]. Available: https://infosys.beckhoff.com/content/1033/te1400_tc3_target_matlab/189856267.html?id=3802657758933840306

[28] Beckhoff Automation GmbH, “TE1130 | TwinCAT 3 CAD Simulation Interface,” Jun. 27 2022, manual.

List of Figures

List of Tables

List of Code

List of Acronyms

COLLADA	Collaborative Design Activity
FMI	Functional Mockup Interface
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 *Separation* as *PLCopen-xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://www.plcopen.org/xml/tc6_0200">
  <fileHeader companyName="Beckhoff Automation GmbH" productName="TwinCAT PLC Control" productVersion="3.5.13.21"
      creationDateTime="2022-05-27T15:51:05.3586322" />
  <contentHeader name="main" modificationDateTime="2022-05-27T15:51:05.3606326">
    <coordinateInfo>
      <fbd>
        <scaling x="1" y="1" />
      </fbd>
      <ld>
        <scaling x="1" y="1" />
      </ld>
      <sfc>
        <scaling x="1" y="1" />
      </sfc>
    </coordinateInfo>
    <addData>
      <data name="http://www.3s-software.com/plcopenxml/projectinformation" handleUnknown="implementation">
        <ProjectInformation />
      </data>
    </addData>
  </contentHeader>
  <types>
    <dataTypes />
    <pous>
      <pou name="FB_Separator" pouType="functionBlock">
        <interface>
          <inputVars>
            <variable name="bEnable">
              <type>
                <BOOL />
              </type>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">start/stops the functionblock</xhtml>
              </documentation>
            </variable>
            <variable name="bExecute">
              <type>
                <BOOL />
              </type>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">Ejects 1 flask if true</xhtml>
              </documentation>
            </variable>
            <variable name="bErrorReset">
              <type>
                <BOOL />
              </type>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">Reset Error </xhtml>
              </documentation>
            </variable>
            <variable name="bSensor1">
              <type>
                <BOOL />
              </type>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">Hardware Sensor 1</xhtml>
              </documentation>
            </variable>
            <variable name="bSensor2">
              <type>
                <BOOL />
              </type>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">Hardware Sensor 2</xhtml>
              </documentation>
            </variable>
            <variable name="bSensor3">
              <type>
                <BOOL />
              </type>
              <documentation>
```

```
                <xhtml xmlns="http://www.w3.org/1999/xhtml">Hardware Sensor 3</xhtml>
74            </documentation>
            </variable>
76          </inputVars>
            <outputVars>
78            <variable name="bR1AKT1">
                <type>
80                <BOOL />
                </type>
82              <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>
90              <documentation>
                  <xhtml xmlns="http://www.w3.org/1999/xhtml">Relais 2 Aktor 1</xhtml>
92              </documentation>
              </variable>
94            <variable name="bR1AKT2">
                <type>
96                <BOOL />
                </type>
98              <documentation>
                  <xhtml xmlns="http://www.w3.org/1999/xhtml">Relais 1 Aktor 2 (Schieber unten)</xhtml>
100             </documentation>
              </variable>
102           <variable name="bR2AKT2">
                <type>
104               <BOOL />
                </type>
106             <documentation>
                  <xhtml xmlns="http://www.w3.org/1999/xhtml">Relais 2 Aktor 2</xhtml>
108             </documentation>
              </variable>
110           <variable name="bNoBottle">
                <type>
112               <BOOL />
                </type>
114             <documentation>
                  <xhtml xmlns="http://www.w3.org/1999/xhtml">Keine Flasche vorhanden</xhtml>
116             </documentation>
              </variable>
118           <variable name="bWrongBottle">
                <type>
120               <BOOL />
                </type>
122             <documentation>
                  <xhtml xmlns="http://www.w3.org/1999/xhtml">Falsche oder gedrehte Flasche vorhanden</xhtml>
124             </documentation>
              </variable>
126           <variable name="eStatus">
                <type>
128               <derived name="State" />
                </type>
130             <initialValue>
                  <simpleValue value="State.Off" />
132             </initialValue>
              </variable>
134         </outputVars>
            <localVars>
136           <variable name="timer">
                <type>
138               <derived name="TON" />
                </type>
140             <documentation>
                  <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>
148             <initialValue>
                  <simpleValue value="StateSeparator.Off" />
150             </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

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

```
  IF bEnable  THEN
    eState := StateSeparator.Init;
  END_IF

StateSeparator.Init:

  eStatus := State.Busy; //Beschaeftigt Signallampe

  bR1AKT1 := FALSE; //Ausgefahren oben
  bR2AKT1 := TRUE;

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


  IF bSensor1 THEN
    eState := StateSeparator.Init_Wait; //Zustandswechsel sobald Sensor Flasche zwischen Schiebern sieht
  END_IF

StateSeparator.Init_Wait:
  timer(IN := TRUE, PT := T#1000MS); //Timer wartet bis Flasche sicher zum Stehen kommt

  IF timer.Q THEN
    timer(IN := FALSE);
    eState := StateSeparator.Ready; //Zustand Ready nach Timer
  END_IF

StateSeparator.Ready:
  eStatus := State.Ready; //Zustand Ready Signallampe

  bR1AKT1 := TRUE;  //Eingefahren oben
  bR2AKT1 := FALSE; //Eingefahren oben

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

  IF bExecute THEN
    eState := StateSeparator.Eject_Wait_1; //Auswerfen sobald Exceute TRUE
  END_IF

StateSeparator.Eject_Wait_1:  // Warten bis Schieber oben ganz geschlossen ist
    eStatus := State.Busy; //Zustand Busy Signallampe

    timer(IN := TRUE, PT := T#2500MS); //Timer auf 2,5 S eingestellt

  IF timer.Q THEN
    timer(IN := FALSE);
    eState := StateSeparator.Eject; //Zustand bereit nach Timer
  END_IF

StateSeparator.Eject: //Auswerfen der unteren Flasche
  bR1AKT2 := TRUE;  //L sst Flaschen auf Band ab durch  ffnen  des unteren Schiebers
  bR2AKT2 := FALSE;

  IF NOT bSensor1 THEN
    eState := StateSeparator.Eject_Wait_2;
  END_IF

StateSeparator.Eject_Wait_2:  // Warten bis Flasche durch F rderband weit genug von Schieber wegtransportiert
  eStatus := State.Done;
  timer(IN := TRUE, PT := T#4000MS); //Timer auf 4 S eingestellt

  IF timer.Q THEN
    timer(IN := FALSE);
    eState := StateSeparator.Done_Wait;
  END_IF

StateSeparator.Done_Wait: // Schlie en des unteren Schiebers und Warten bis komplettt geschlossen
  bR1AKT2 := FALSE; //Eingefahren unten
  bR2AKT2 := TRUE;  //Eingefahren unten

  timer(IN := TRUE, PT := T#3000MS); //Timer wartet bis Schieber geschlossen ist

  IF timer.Q THEN
    timer(IN := FALSE);
    eState := StateSeparator.Done;
  END_IF

StateSeparator.Done:  //Vereinzelungsvorgang abgeschlossen

  IF NOT bExecute THEN
    eState := StateSeparator.Init;
  END_IF

END_CASE</xhtml>
          </ST>
        </body>
        <addData>
          <data name="http://www.3s-software.com/plcopenxml/interfaceasplaintext" handleUnknown="implementation">
            <InterfaceAsPlainText>
              <xhtml xmlns="http://www.w3.org/1999/xhtml">FUNCTION_BLOCK FB_Separator
VAR_INPUT
  bEnable : BOOL;   // start/stops the functionblock
```

```
  bExecute: BOOL;    // Ejects 1 flask if true
  bErrorReset: BOOL;   //Reset Error

  bSensor1 : BOOL; //Hardware Sensor 1
  bSensor2 : BOOL; //Hardware Sensor 2
  bSensor3 : BOOL; //Hardware Sensor 3
END_VAR
VAR_OUTPUT
  bR1AKT1 : BOOL; //Relais 1 Aktor 1 (Schieber oben)
  bR2AKT1 : BOOL; //Relais 2 Aktor 1
  bR1AKT2 : BOOL; //Relais 1 Aktor 2 (Schieber unten)
  bR2AKT2 : BOOL; //Relais 2 Aktor 2

  bNoBottle    : BOOL;    //Keine Flasche vorhanden
  bWrongBottle   : BOOL; //Falsche oder gedrehte Flasche vorhanden

  eStatus : State := State.Off;
END_VAR
VAR
  timer : TON; //Timer to make sure, that the flask has cleared the slope
  eState : StateSeparator := StateSeparator.Off;
END_VAR
</xhtml>
            </InterfaceAsPlainText>
          </data>
          <data name="http://www.3s-software.com/plcopenxml/objectid" handleUnknown="discard">
            <ObjectId>91946f14-5ef9-4883-993a-962cda20c177</ObjectId>
          </data>
        </addData>
      </pou>
    </pous>
  </types>
  <instances>
    <configurations />
  </instances>
  <addData>
    <data name="http://www.3s-software.com/plcopenxml/projectstructure" handleUnknown="discard">
      <ProjectStructure>
        <Object Name="FB_Separator" ObjectId="91946f14-5ef9-4883-993a-962cda20c177" />
      </ProjectStructure>
    </data>
  </addData>
</project>
```

Sourcecode A.1.2: Code for module *Conveyor Belt* as *PLCopen-xml*.

```
   <?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://www.plcopen.org/xml/tc6_0200">
  <fileHeader companyName="Beckhoff Automation GmbH" productName="TwinCAT PLC Control" productVersion="3.5.13.21"
      creationDateTime="2022-08-13T15:46:57.5102704" />
  <contentHeader name="PlcMain" modificationDateTime="2022-08-13T15:46:57.5102704">
    <coordinateInfo>
      <fbd>
        <scaling x="1" y="1" />
      </fbd>
      <ld>
        <scaling x="1" y="1" />
      </ld>
      <sfc>
        <scaling x="1" y="1" />
      </sfc>
    </coordinateInfo>
    <addData>
      <data name="http://www.3s-software.com/plcopenxml/projectinformation" handleUnknown="implementation">
        <ProjectInformation />
      </data>
    </addData>
  </contentHeader>
  <types>
    <dataTypes />
    <pous>
      <pou name="FB_Conveyor" pouType="functionBlock">
        <interface>
          <inputVars>
            <variable name="bEnable">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="bExecute">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="eDirection">
              <type>
                <derived name="MC_Direction" />
              </type>
              <initialValue>
                <simpleValue value="MC_Positive_Direction" />
              </initialValue>
```

```
            </variable>
            <variable name="nVel">
              <type>
                <INT />
              </type>
              <initialValue>
                <simpleValue value="25" />
              </initialValue>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">mm per s</xhtml>
              </documentation>
            </variable>
          </inputVars>
          <outputVars>
            <variable name="eStatus">
              <type>
                <derived name="State" />
              </type>
              <initialValue>
                <simpleValue value="State.Off" />
              </initialValue>
            </variable>
          </outputVars>
          <inOutVars>
            <variable name="ax">
              <type>
                <derived name="AXIS_REF" />
              </type>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">axis reference</xhtml>
              </documentation>
            </variable>
          </inOutVars>
          <localVars>
            <variable name="fbReset">
              <type>
                <derived name="MC_Reset" />
              </type>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">NC FBs</xhtml>
              </documentation>
            </variable>
            <variable name="fbPower">
              <type>
                <derived name="MC_Power" />
              </type>
            </variable>
            <variable name="fbMove">
              <type>
                <derived name="MC_MoveVelocity" />
              </type>
            </variable>
            <variable name="fbHalt">
              <type>
                <derived name="MC_Halt" />
              </type>
            </variable>
            <variable name="bAxPower">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="bAxMove">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="eState">
              <type>
                <derived name="StateCnv" />
              </type>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">state</xhtml>
              </documentation>
            </variable>
          </localVars>
        </interface>
        <body>
          <ST>
            <xhtml xmlns="http://www.w3.org/1999/xhtml">// update axis
ax.ReadStatus();

// call NC FBs
fbReset(
  Axis:= ax);
fbPower(
  Axis:= ax,
  Enable:= bAxPower,
  Enable_Positive:= bAxPower,
  Enable_Negative:= bAxPower,
  Override:= 100);
fbMove(
  Axis:= ax,
```

```
138     Execute:=bAxMove,
        Velocity:= nVel,
140     Direction:= eDirection);
    fbHalt(
142     Axis:= ax,
        Execute:= NOT bAxMove);
144
    IF NOT bEnable THEN
146     eState := StateCnv.Off;
    END_IF
148
    CASE eState OF
150 StateCnv.Off:
        eStatus := State.Off;
152     bAxPower := FALSE;
        bAxMove := FALSE;
154
        IF bEnable THEN
156         eState := StateCnv.Reset;
        END_IF
158 StateCnv.Reset:
        eStatus := State.Busy;
160     fbReset.Execute := TRUE;

162     IF fbReset.Done THEN
            eState := StateCnv.Power;
164     END_IF
    StateCnv.Power:
166     eStatus := State.Busy;
        bAxPower := TRUE;
168
        IF fbPower.Status THEN
170         eState := StateCnv.Ready;
        END_IF
172 StateCnv.Ready:
        eStatus := State.Ready;
174     bAxMove := FALSE;

176     IF bExecute THEN
            eState := StateCnv.Moving;
178     END_IF
    StateCnv.Moving:
180     eStatus := State.Busy;
        bAxMove := TRUE;
182
        IF NOT bExecute THEN
184         eState := StateCnv.Ready;
        END_IF
186 END_CASE</xhtml>
                </ST>
            </body>
            <addData>
190             <data name="http://www.3s-software.com/plcopenxml/interfaceasplaintext" handleUnknown="implementation">
                  <InterfaceAsPlainText>
192                 <xhtml xmlns="http://www.w3.org/1999/xhtml">FUNCTION_BLOCK FB_Conveyor
    VAR_INPUT
194     bEnable : BOOL;
        bExecute: BOOL;
196     eDirection : MC_Direction := MC_Positive_Direction;
        nVel  : INT := 25; // mm per s
198 END_VAR
    VAR_OUTPUT
200     eStatus : State := State.Off;
    END_VAR
202 VAR_IN_OUT
        ax : AXIS_REF; // axis reference
204 END_VAR
    VAR
206     // NC FBs
        fbReset : MC_Reset;
208     fbPower : MC_Power;
        fbMove  : MC_MoveVelocity;
210     fbHalt  : MC_Halt;

212     bAxPower : BOOL;
        bAxMove : BOOL;
214
        // state
216     eState : StateCnv;
    END_VAR
218 </xhtml>
                  </InterfaceAsPlainText>
220             </data>
                <data name="http://www.3s-software.com/plcopenxml/objectid" handleUnknown="discard">
222               <ObjectId>fbe14543-a9c7-41a4-8387-253afb962875</ObjectId>
                </data>
224           </addData>
          </pou>
226     </pous>
      </types>
228   <instances>
        <configurations />
230   </instances>
```

```
  <addData>
    <data name="http://www.3s-software.com/plcopenxml/projectstructure" handleUnknown="discard">
      <ProjectStructure>
        <Object Name="FB_Conveyor" ObjectId="fbe14543-a9c7-41a4-8387-253afb962875" />
      </ProjectStructure>
    </data>
  </addData>
</project>
```

Sourcecode A.1.3: Code for module *Dosing Unit* as *PLCopen-xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://www.plcopen.org/xml/tc6_0200">
  <fileHeader companyName="Beckhoff Automation GmbH" productName="TwinCAT PLC Control" productVersion="3.5.13.21"
      creationDateTime="2022-08-13T15:46:51.4562744" />
  <contentHeader name="PlcMain" modificationDateTime="2022-08-13T15:46:51.4582743">
    <coordinateInfo>
      <fbd>
        <scaling x="1" y="1" />
      </fbd>
      <ld>
        <scaling x="1" y="1" />
      </ld>
      <sfc>
        <scaling x="1" y="1" />
      </sfc>
    </coordinateInfo>
    <addData>
      <data name="http://www.3s-software.com/plcopenxml/projectinformation" handleUnknown="implementation">
        <ProjectInformation />
      </data>
    </addData>
  </contentHeader>
  <types>
    <dataTypes />
    <pous>
      <pou name="FB_Dispenser" pouType="functionBlock">
        <interface>
          <inputVars>
            <variable name="bEnable">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="bExecute">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="nDispenseGrams">
              <type>
                <INT />
              </type>
              <initialValue>
                <simpleValue value="1" />
              </initialValue>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">Gewunschte Menge in Gramm</xhtml>
              </documentation>
            </variable>
          </inputVars>
          <outputVars>
            <variable name="eStatus">
              <type>
                <derived name="State" />
              </type>
              <initialValue>
                <simpleValue value="State.Off" />
              </initialValue>
            </variable>
          </outputVars>
          <inOutVars>
            <variable name="ax">
              <type>
                <derived name="AXIS_REF" />
              </type>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">Referenz auf die NC-Achse</xhtml>
              </documentation>
            </variable>
          </inOutVars>
          <localVars>
            <variable name="fbReset">
              <type>
                <derived name="MC_Reset" />
              </type>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml">Fur NC-Achse notig</xhtml>
              </documentation>
            </variable>
            <variable name="fbPower">
```

```
80              <type>
                  <derived name="MC_Power" />
82              </type>
              </variable>
84            <variable name="fbMove">
                <type>
86                <derived name="MC_MoveRelative" />
                </type>
88            </variable>
              <variable name="nVelocity">
90              <type>
                  <INT />
92              </type>
                <initialValue>
94                <simpleValue value="720" />
                </initialValue>
96              <documentation>
                  <xhtml xmlns="http://www.w3.org/1999/xhtml">Grad pro Sekunde</xhtml>
98              </documentation>
              </variable>
100           <variable name="fGramPerRev">
                <type>
102               <REAL />
                </type>
104             <initialValue>
                  <simpleValue value="0.245" />
106             </initialValue>
                <documentation>
108               <xhtml xmlns="http://www.w3.org/1999/xhtml">Skalierung auf Gramm pro ganzer Umdrehung (360 Grad)</
      xhtml>
                </documentation>
110           </variable>
              <variable name="eState">
112             <type>
                  <derived name="StateDisp" />
114             </type>
                <documentation>
116               <xhtml xmlns="http://www.w3.org/1999/xhtml">Status der Zustandsmaschine</xhtml>
                </documentation>
118           </variable>
              <variable name="bAxPower">
120             <type>
                  <BOOL />
122             </type>
              </variable>
124           <variable name="bAxMove">
                <type>
126               <BOOL />
                </type>
128           </variable>
            </localVars>
130         <documentation>
              <xhtml xmlns="http://www.w3.org/1999/xhtml">Das ist die Musterlosung zur Steuerung einer Dosiereinheit.
132  Library TC2_MC2 ist notig.</xhtml>
            </documentation>
134       </interface>
          <body>
136         <ST>
              <xhtml xmlns="http://www.w3.org/1999/xhtml">
138 // update axis
    ax.ReadStatus();
140
    // call NC FBs
142 fbReset(
      Axis:= ax);
144 fbPower(
      Axis:= ax,
146   Enable:= bAxPower,
      Enable_Positive:= bAxPower,
148   Enable_Negative:= bAxPower,
      Override:= 100);
150 fbMove(
      Axis:=ax,
152   Execute:=bAxMove,
      Velocity:=nVelocity,
154   Distance:= nDispenseGrams *360 / fGramPerRev);


156


158 IF NOT bEnable THEN
      eState := StateDisp.Off;
160 END_IF

162 CASE eState OF
    StateDisp.Off:
164   eStatus := State.Off;
      bAxMove := FALSE;
166   bAxPower := FALSE;

168   IF bEnable THEN
        eState := StateDisp.Reset;
170   END_IF
    StateDisp.Reset:
```

```
172     eStatus := State.Busy;
        fbReset.Execute := TRUE;
174     IF fbReset.Done THEN
          fbReset.Execute := FALSE;
176       eState := StateDisp.PowerUp;
        END_IF
178   StateDisp.PowerUp:
        eStatus := State.Busy;
180     bAxPower := TRUE;
        IF fbPower.Status THEN
182       eState := StateDisp.Idle;
        END_IF
184   StateDisp.Idle:
        eStatus := State.Ready;
186     bAxMove := FALSE;

188     IF bExecute THEN
          eState := StateDisp.Dispensing;
190     END_IF
      StateDisp.Dispensing:
192     eStatus := State.Busy;
        bAxMove := TRUE;
194
        IF fbMove.Done THEN
196       eState := StateDisp.Done;
        END_IF
198   StateDisp.Done:
        eStatus := State.Done;
200     bAxMove := FALSE;

202     IF NOT bExecute THEN
          eState := StateDisp.Idle;
204     END_IF
      END_CASE</xhtml>
206             </ST>
              </body>
208           <addData>
                <data name="http://www.3s-software.com/plcopenxml/interfaceasplaintext" handleUnknown="implementation">
210               <InterfaceAsPlainText>
                    <xhtml xmlns="http://www.w3.org/1999/xhtml">// Das ist die Musterlosung zur Steuerung einer
        Dosiereinheit.
212   // Library TC2_MC2 ist notig.

214   FUNCTION_BLOCK FB_Dispenser
        VAR_INPUT
216       bEnable           : BOOL;
          bExecute          : BOOL;
218       nDispenseGrams       : INT := 1;    // Gewunschte Menge in Gramm
        END_VAR
220
        VAR_OUTPUT
222       eStatus : State := State.Off;
        END_VAR
224
        VAR_IN_OUT
226       ax     : AXIS_REF;        // Referenz auf die NC-Achse
        END_VAR
228
        VAR
230       // Fur NC-Achse notig
          fbReset      : MC_Reset;
232       fbPower      : MC_Power;
          fbMove       : MC_MoveRelative;
234
          // Parameter
236       nVelocity      : INT := 720;    // Grad pro Sekunde
          fGramPerRev      : REAL := 0.245;     // Skalierung auf Gramm pro ganzer Umdrehung (360 Grad)
238
          // Working variables
240       eState         : StateDisp;     // Status der Zustandsmaschine
          bAxPower       : BOOL;
242       bAxMove        : BOOL;
      END_VAR
244
      // ----- Ende der Definition. Hier folgt der Code ------
246


248
      </xhtml>
250               </InterfaceAsPlainText>
                </data>
252             <data name="http://www.3s-software.com/plcopenxml/objectid" handleUnknown="discard">
                  <ObjectId>45b31251-7302-454e-ad8b-8a17ba762004</ObjectId>
254             </data>
              </addData>
256         </pou>
          </pous>
258     </types>
        <instances>
          <configurations />
        </instances>
262     <addData>
          <data name="http://www.3s-software.com/plcopenxml/projectstructure" handleUnknown="discard">
```

```
      <ProjectStructure>
        <Object Name="FB_Dispenser" ObjectId="45b31251-7302-454e-ad8b-8a17ba762004" />
      </ProjectStructure>
    </data>
  </addData>
</project>
```

Sourcecode A.1.4: Code for module *Cartesian Gripper* as *PLCopen-xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://www.plcopen.org/xml/tc6_0200">
  <fileHeader companyName="Beckhoff Automation GmbH" productName="TwinCAT PLC Control" productVersion="3.5.13.21"
      creationDateTime="2022-08-12T11:34:54.3732365" />
  <contentHeader name="PlcMain" modificationDateTime="2022-08-12T11:34:54.3752427">
    <coordinateInfo>
      <fbd>
        <scaling x="1" y="1" />
      </fbd>
      <ld>
        <scaling x="1" y="1" />
      </ld>
      <sfc>
        <scaling x="1" y="1" />
      </sfc>
    </coordinateInfo>
    <addData>
      <data name="http://www.3s-software.com/plcopenxml/projectinformation" handleUnknown="implementation">
        <ProjectInformation />
      </data>
    </addData>
  </contentHeader>
  <types>
    <dataTypes />
    <pous>
      <pou name="FB_ManipulatorAxis" pouType="functionBlock">
        <interface>
          <inputVars>
            <variable name="bEnable">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="bExecute">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="bEndschalterMin">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="bEndSchalterMax">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="fPosition">
              <type>
                <LREAL />
              </type>
              <documentation>
                <xhtml xmlns="http://www.w3.org/1999/xhtml"> gewunschte Postition der Achse</xhtml>
              </documentation>
            </variable>
            <variable name="fVelocity">
              <type>
                <LREAL />
              </type>
            </variable>
            <variable name="Direction">
              <type>
                <derived name="MC_Direction" />
              </type>
            </variable>
          </inputVars>
          <outputVars>
            <variable name="eOutState">
              <type>
                <derived name="State" />
              </type>
              <initialValue>
                <simpleValue value="State.Off" />
              </initialValue>
            </variable>
            <variable name="fActPos">
              <type>
                <LREAL />
              </type>
            </variable>
          </outputVars>
```

```
82            <inOutVars>
              <variable name="aAchse">
84              <type>
                  <derived name="AXIS_REF" />
86              </type>
                <documentation>
88                <xhtml xmlns="http://www.w3.org/1999/xhtml"> NC-Achse</xhtml>
                </documentation>
90            </variable>
            </inOutVars>
92          <localVars>
              <variable name="Reset_Achse">
94              <type>
                  <derived name="MC_Reset" />
96              </type>
                <documentation>
98                <xhtml xmlns="http://www.w3.org/1999/xhtml"> Fur NC-Achse notig</xhtml>
                </documentation>
100           </variable>
              <variable name="Power_Achse">
102             <type>
                  <derived name="MC_Power" />
104             </type>
              </variable>
106           <variable name="Move_Achse">
                <type>
108               <derived name="MC_MoveAbsolute" />
                </type>
110           </variable>
              <variable name="Move_Vel">
112             <type>
                  <derived name="MC_MoveVelocity" />
114             </type>
              </variable>
116           <variable name="Home_Achse">
                <type>
118               <derived name="MC_Home" />
                </type>
120           </variable>
              <variable name="Halt_Achse">
122             <type>
                  <derived name="MC_Halt" />
124             </type>
              </variable>
126           <variable name="Position_Achse">
                <type>
128               <derived name="MC_ReadActualPosition" />
                </type>
130           </variable>
              <variable name="bAchsePower">
132             <type>
                  <BOOL />
134             </type>
              </variable>
136           <variable name="bAchseMove">
                <type>
138               <BOOL />
                </type>
140           </variable>
              <variable name="eState">
142             <type>
                  <derived name="StateMan" />
144             </type>
              </variable>
146         </localVars>
          </interface>
148       <body>
            <ST>
150           <xhtml xmlns="http://www.w3.org/1999/xhtml">// Einstellungen der Achse
aAchse.ReadStatus();
152 Reset_Achse(Axis:=aAchse);
Power_Achse(Axis:=aAchse, Override := 100, Enable := bAchsePower, Enable_Negative := bAchsePower, Enable_Positive :=
      bAchsePower);
154 Move_Achse(Axis:=aAchse);
Home_Achse(Axis:= aAchse, bCalibrationCam := NOT bEndschalterMin, Position := 0);
156 Move_Vel(Axis:=aAchse);
Halt_Achse(Axis:=aAchse);
158
CASE eState OF
160 StateMan.Off:
  eOutState := State.Off;
162 bAchseMove := FALSE;
  bAchsePower := FALSE;
164
  IF bEnable THEN
166   eState := StateMan.Reset;
  END_IF
168 StateMan.Reset:
  eOutState := State.Busy;
170 Reset_Achse.Execute := TRUE;

172 IF Reset_Achse.Done THEN
    Reset_Achse.Execute := FALSE;
```

```
174       eState := StateMan.PowerUp;
    END_IF
176 StateMan.PowerUp:
    eOutState := State.Busy;
178 bAchsePower := TRUE;

180 IF Power_Achse.Status THEN
      eState := StateMan.Homing;
182 END_IF
    StateMan.Homing:
184 eOutState := State.Busy;
    Home_Achse.Execute := TRUE;
186
    IF Home_Achse.Done THEN
188   Home_Achse.Execute := FALSE;
      eState := StateMan.Ready;
190 END_IF
    StateMan.Ready:
192   eOutState := State.Ready;
      Move_Achse.Execute := FALSE;
194 IF bExecute THEN
      eState := StateMan.Moving;
196 END_IF
    StateMan.Moving:
198 eOutState := State.Busy;
    Move_Achse.Velocity := fVelocity;
200
    Move_Achse.Position := fPosition;
202 Move_Achse.Execute := TRUE;

204 IF Move_Achse.Done OR ( NOT bEndschalterMin  AND (Move_Achse.Position &lt; Position_Achse.Position)) OR (NOT
        bEndschalterMax AND (Move_Achse.Position &gt; Position_Achse.Position)) THEN

206   IF NOT(bEndschalterMax) OR NOT(bEndschalterMin) THEN
        Halt_Achse.Execute := TRUE;
208   END_IF

210   eState := StateMan.Done;
    END_IF
212 StateMan.Done:
    eOutState := State.Done;
214 Move_Achse.Execute := FALSE;

216 IF NOT bExecute THEN
      eState := StateMan.Ready;
218 END_IF
    END_CASE</xhtml>
220           </ST>
            </body>
222         <addData>
              <data name="http://www.3s-software.com/plcopenxml/interfaceasplaintext" handleUnknown="implementation">
224             <InterfaceAsPlainText>
                  <xhtml xmlns="http://www.w3.org/1999/xhtml">FUNCTION_BLOCK FB_ManipulatorAxis
226 VAR_INPUT
      bEnable      : BOOL;
228   bExecute     : BOOL;
      bEndschalterMin  : BOOL;
230   bEndSchalterMax  : BOOL;
      fPosition    : LREAL;       // gewunschte Postition der Achse
232   fVelocity      : LREAL;
      Direction    : MC_Direction;
234 END_VAR

236 VAR_OUTPUT
      eOutState        : State:=State.Off;
238   fActPos          : LREAL;
    END_VAR
240
    VAR_IN_OUT
242   aAchse       : AXIS_REF;    // NC-Achse
    END_VAR
244
    VAR
246   // Fur NC-Achse notig
      Reset_Achse      : MC_Reset;
248   Power_Achse      : MC_Power;
      Move_Achse       : MC_MoveAbsolute;
250   Move_Vel       : MC_MoveVelocity;
      Home_Achse       : MC_Home;
252   Halt_Achse       : MC_Halt;
      Position_Achse     : MC_ReadActualPosition;
254   bAchsePower      : BOOL;
      bAchseMove       : BOOL;
256
      eState       : StateMan;
258 END_VAR</xhtml>
              </InterfaceAsPlainText>
260         </data>
            <data name="http://www.3s-software.com/plcopenxml/objectid" handleUnknown="discard">
262           <ObjectId>e010b4d0-16a4-42c7-b4be-1acb68c0d19a</ObjectId>
            </data>
264       </addData>
        </pou>
```

```
    </pous>
  </types>
  <instances>
    <configurations />
  </instances>
  <addData>
    <data name="http://www.3s-software.com/plcopenxml/projectstructure" handleUnknown="discard">
      <ProjectStructure>
        <Object Name="FB_ManipulatorAxis" ObjectId="e010b4d0-16a4-42c7-b4be-1acb68c0d19a" />
      </ProjectStructure>
    </data>
  </addData>
</project>
```

Sourcecode A.1.5: Code for module *Thermal Processing* as *PLCopen-xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://www.plcopen.org/xml/tc6_0200">
  <fileHeader companyName="Beckhoff Automation GmbH" productName="TwinCAT PLC Control" productVersion="3.5.13.21"
      creationDateTime="2022-08-12T17:36:56.3108408" />
  <contentHeader name="PlcMain" modificationDateTime="2022-08-12T17:36:56.3128405">
    <coordinateInfo>
      <fbd>
        <scaling x="1" y="1" />
      </fbd>
      <ld>
        <scaling x="1" y="1" />
      </ld>
      <sfc>
        <scaling x="1" y="1" />
      </sfc>
    </coordinateInfo>
    <addData>
      <data name="http://www.3s-software.com/plcopenxml/projectinformation" handleUnknown="implementation">
        <ProjectInformation />
      </data>
    </addData>
  </contentHeader>
  <types>
    <dataTypes />
    <pous>
      <pou name="FB_HeatingSystem" pouType="functionBlock">
        <interface>
          <inputVars>
            <variable name="bEnable">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="bExecute">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="nMaxTemp">
              <type>
                <UINT />
              </type>
            </variable>
            <variable name="nTemp">
              <type>
                <UINT />
              </type>
            </variable>
          </inputVars>
          <outputVars>
            <variable name="bVentilator1">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="bVentilator2">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="bVentilator3">
              <type>
                <BOOL />
              </type>
            </variable>
            <variable name="nHeizpatrone">
              <type>
                <UINT />
              </type>
              <initialValue>
                <simpleValue value="0" />
              </initialValue>
            </variable>
            <variable name="eStatus">
              <type>
```

```
                <derived name="State" />
              </type>
              <initialValue>
                <simpleValue value="State.Off" />
              </initialValue>
            </variable>
          </outputVars>
          <localVars>
            <variable name="nHeat">
              <type>
                <REAL />
              </type>
              <initialValue>
                <simpleValue value="0" />
              </initialValue>
            </variable>
            <variable name="timer">
              <type>
                <derived name="TON" />
              </type>
            </variable>
            <variable name="timer1">
              <type>
                <derived name="TON" />
              </type>
            </variable>
            <variable name="eState">
              <type>
                <derived name="StateHeat" />
              </type>
              <initialValue>
                <simpleValue value="StateHeat.Off" />
              </initialValue>
            </variable>
          </localVars>
        </interface>
        <body>
          <ST>
            <xhtml xmlns="http://www.w3.org/1999/xhtml">nHeizpatrone := TO_UINT(F_map(nHeat, 0, 5, 0, 16383));

IF NOT bEnable THEN
  eState:= StateHeat.Off;
END_IF

CASE eState OF
StateHeat.Off:
  eStatus := State.Off;

  bVentilator1 := FALSE;
  bVentilator2 := FALSE;
  bVentilator3 := FALSE;
  nHeat := 0;

  IF bEnable THEN
    eState := StateHeat.Init;
  END_IF

StateHeat.Init:
  eStatus := State.Busy;
  bVentilator1 := TRUE;
  bVentilator2 := TRUE;
  bVentilator3 := TRUE;

  IF nTemp &lt;= nMaxTemp THEN
    eState := StateHeat.Ready;
  END_IF

StateHeat.Ready:
  eStatus := State.Ready;

  nHeat := 0;

  IF bExecute THEN
    eState := StateHeat.Heat;
  END_IF

StateHeat.Heat:
  eStatus := State.Busy;

  nHeat := 5;

  IF nTemp &gt;= 3000 THEN
    nHeat := 0;
    eState := StateHeat.Hold;
  END_IF

StateHeat.Hold:
  eStatus := State.Busy;

  timer(IN := TRUE, PT := T#30S);

  IF nTemp &gt;= 3700 THEN
    nHeat := 0;
```

```
  ELSIF nTemp &lt;= 3500 THEN
    timer1(IN := TRUE, PT:= T#500MS);
    IF timer1.Q THEN
      timer1(IN := FALSE);
      nHeat := nHeat + 0.05;
      IF nHeat &gt;= 5 THEN
        nHeat := 5;
      END_IF
    END_IF

  END_IF

  IF timer.Q THEN
    timer(IN := FALSE);
    nHeat := 0;
    eState := StateHeat.Done;
  END_IF

StateHeat.Done:
  eStatus := State.Done;

  nHeat := 0;

  IF NOT bExecute THEN
    eState := StateHeat.Init;
  END_IF

END_CASE</xhtml>
          </ST>
        </body>
        <addData>
          <data name="http://www.3s-software.com/plcopenxml/interfaceasplaintext" handleUnknown="implementation">
            <InterfaceAsPlainText>
              <xhtml xmlns="http://www.w3.org/1999/xhtml">FUNCTION_BLOCK FB_HeatingSystem
VAR_INPUT
  bEnable : BOOL;
  bExecute : BOOL;

  nMaxTemp : UINT;
  nTemp : UINT;
END_VAR
VAR_OUTPUT
  bVentilator1 : BOOL;
  bVentilator2 : BOOL;
  bVentilator3 : BOOL;
  nHeizpatrone : UINT := 0;

  eStatus : State := State.Off;
END_VAR
VAR
  nHeat : REAL := 0;
  timer : TON;
  timer1 : TON;
  eState : StateHeat := StateHeat.Off;
END_VAR
</xhtml>
            </InterfaceAsPlainText>
          </data>
          <data name="http://www.3s-software.com/plcopenxml/objectid" handleUnknown="discard">
            <ObjectId>7c7b126e-8b79-4d92-8382-02f37357639c</ObjectId>
          </data>
        </addData>
      </pou>
    </pous>
  </types>
  <instances>
    <configurations />
  </instances>
  <addData>
    <data name="http://www.3s-software.com/plcopenxml/projectstructure" handleUnknown="discard">
      <ProjectStructure>
        <Object Name="FB_HeatingSystem" ObjectId="7c7b126e-8b79-4d92-8382-02f37357639c" />
      </ProjectStructure>
    </data>
  </addData>
</project>
```