ENTOC
engineering tool chain

for efficient and iterative
development of smart factories

ITEA 3 – 15015

Work package 3
Creation and Distribution of Component Models

Deliverable D3.1
Package Format specification

Document type	: Deliverable
Document version	: Nr.1
Document Preparation Date	: 30.04.2018
Classification	: private
Contract Start Date	: 01.09.2016
Contract End Date	: 31.08.2019

ITEA3

Final approval	**Name**	**Partner**
Review Task Level	Manuel Paul	Festo AG & Co.KG
Review WP Level	Felix Auris	Daimler AG
Review Board Level	Matthias Riedl	ifak e.V.

Executive Summary

This Deliverable 3.1 contains different topics regarding the package format of component models which will become necessary in the ENTOC tool chain. Aspects of the description and explanation of the available package formats are important for the intended extension “Engineering Model Store” of tools to be used in ENTOC tool chain.

In this Deliverable the focus lays on two aspects which are important to distinguish. On the one hand the AutomationML based component package format itself, together with e.g. interface descriptions and standardization activities. The other part comprehends the superior model store package format for common dependency management systems which will be the basic infrastructure for the model store to handle the component models. Based on AutomationML the package format will fit into the ENTOC tool chain and enables the entirely way of engineering.

Contents

List of Figures

List of Tables

1 Introduction

The content of Work Package 3 is the creation and distribution of component models. From the view of Task 3.1 it is necessary to define a way how these component models must look like – based on common industrial standards and independent from the component. Following, the developed topics of Task 3.1 (Specification of component models) will be described in detail. For the package format of the component model itself and the dependencies of a package format used for the model store the following content includes different possibilities and solutions which will be presented in the following chapters.

The results of Task 3.1 are the basic to create the Component Model Store – which is a part of the ENTOC toolchain shown in Figure 1.

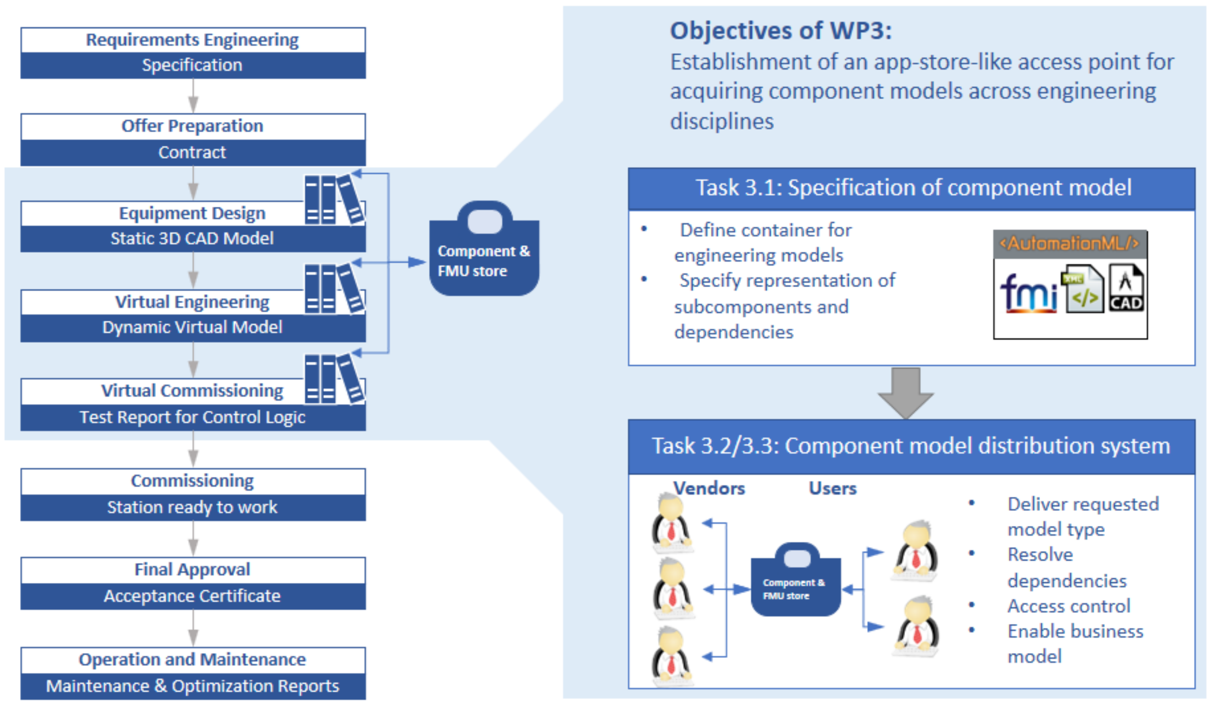
Requirements Engineering
Specification
Offer Preparation
Contract
Equipment Design
Static 3D CAD Model
Virtual Engineering
Dynamic Virtual Model
Virtual Commissioning
Test Report for Control Logic
Commissioning
Station ready to work
Final Approval
Acceptance Certificate
Operation and Maintenance
Maintenance & Optimization Reports
Component & FMU store
Objectives of WP3:
Establishment of an app-store-like access point for acquiring component models across engineering disciplines
Task 3.1: Specification of component model
Define container for engineering models
Specify representation of subcomponents and dependencies
<AutomationML/>
fmi
CAD
Task 3.2/3.3: Component model distribution system
Vendors
Users
Component & FMU store
Deliver requested model type
Resolve dependencies
Access control
Enable business model

Figure 1: WP3 classification in the ENTOC toolchain

1.1 AutomationML inside the ENTOC tool chain

Nowadays the demand for automation as well as the complexity of the automated tasks is constantly increasing in the field of production. A possible reason for this increase are demands on the flexibility of the automation system. To guarantee a more cost and time efficient automation solution for an automation task, the whole automation process, especially the data transfer between engineering tools must be optimized. AutomationML is a file format that aims at optimizing the data transfer and make it more seamless. The main goal of AutomationML is a unified exchange file format. Currently there are a lot of software tools that are being used in the planning and engineering stages, each software tool usually stores data in a different format that other software tools may not be directly able to import. Actually, some software providers tend to use their own proprietary file formats that only can be used by their software tools.

AutomationML [1] stands for Automation Markup Language. It’s a neutral file format which is based on XML (Extensible Markup Language). The free and open AutomationML standard can be used free of charge. AutomationML adapts the hierarchical concept of CAEX which includes objects, Interface and compositions. The compositions describe the combination of more than one object with specific relation between them. AutomationML utilizes existing standards for describing essential modelling aspects of an automation project like geometry or behaviour. The linking element in AutomationML, which also represents topologies, is accomplished with the aid of the CAEX (Computer Aided Engineering Exchange) data format. Elements in the CAEX format are either Internal Elements (IE) or interfaces. COLLADA [2] (COLLAborative Design Activity) is an open standard file format that stores kinematics and geometry. Additionally, PLCopenXML [3] is also referenced in AutomationML. It is used for storing the models of behavior of objects, control logic information and could also be used for providing information about the production process like the sequence of process steps. AutomationML is flexible and allows the incorporation of future XML standard formats. AutomationML is used to establish relations and connections between the different objects, which are usually linked to other files that are incorporated in AutomationML. For Example, an object like a cylinder in the CAEX structure can be linked to a COLLADA file representing its geometry.

In order to ease compatibility across all of the software tools, AutomationML adopts the concept of libraries. It offers three different libraries:

- Role Class Library
- Interface Class Library
- System Unit Library (SUC)

The Role Class Library is used to describe the abstract semantics of specific terms. It describes the attributes and interfaces of each role. The Interface Class Library is used to give information about the possible types of relations between different objects in a project. It also includes information about the relation between objects in specific projects to external data. Finally, the System Unit Library stores objects that are usually based on sub objects that are already defined in the Role Class Library, which are related to each other according to the information in the Interface Class Library. In addition, the System Unit Library may also contain objects that are only specific (custom made) for a special use case. The objects in a System Unit Library are usually objects that are often reused in the same engineering use case. That’s why the use of such (already linked objects) can be considered as a time saving practice.

1.2 Understanding of a component

A component is a cultivable object which can be ordered from a manufacturer of your choice. For each component there exists an order code and other dependent files e.g. construction data and documentations. From the virtual engineering and virtual commissioning point of view an additional added physical behavior model is necessary. But it should be declared for which kind of functional behavior the model is prepared. In the ENTOC understanding a component with an associated behavior model is a functional component.

Every functional component may consist of one or more components and may itself be part of a larger composition or aggregation which is shown in Figure 2. The correlated characteristic is derived from the related level of detail.

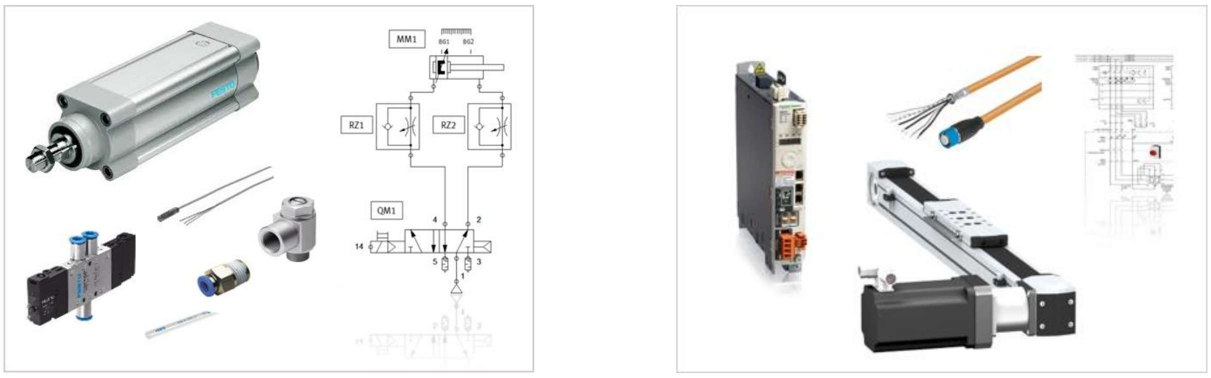

Figure 2: Functional Components Festo pneumatic drive (left) and Schneider electric drive (right)

For the generation of a detailed behavior model component manufacturers need access to internal data and component dependencies – which is normally the know-how of the component manufacturer company. To guarantee a physical behavior of the model with an accuracy of more than 90% a component manufacturer can combine own – well known - components to a functional component and offer the right behavior model for a combination of products.

The example of the functional component pneumatic drive shows a combination of a pneumatic cylinder, valve, throttles, sensors, connectors and tubes. The throttles and tubes can have variable settings you should care about e.g. the opening position of the throttles and the dimension and length of the tube. How these standalone components should fit together is defined in the fluidic plan.

Same for the electric drive. You can´t simulate an accurate behavior just with the component electric motor. You need more information from the gear ratio, axis and motor controller to provide a usable behavior model.

Other topics like dimensioning and the compatibility to guarantee the different kinds of requirements are also important engineering aspects and necessitate a huge knowledge about each component. Typical objects in plant automation comprise information on topology, geometry, kinematics and logic, whereas logic comprises sequencing, behavior and control.

The interface definition depends on the level of detail, which depends on the Use Case. For the ENTOC project we define the interfaces from a PLC programmer point of view and check if this is enough for the ENTOC purpose. The description of each interface is or will be documented in the AutomationML file – independent from the complexity of the used models.

2 Component model package format

Today, it is common for each component model package to be described in a standardized data format used by the engineering software tool. Although only for separated engineering aspects and not tool comprehensive. Especially for behavior modelling there exists a huge number of different software tools with different specifications depending the component model package format.

The ENTOC tool chain prefers AutomationML as integrated engineering format that allows, for example, reference to behavior models for Virtual Engineering and Virtual Commissioning.

2.1 Data formats to describe behavior models

The data format of the behavior models depends on the level of detail. Inside the AutomationML standard PLCopenXML is used for logic descriptions and MathML [4] for behavior description. One additional standard is the FMI (Functional Mock-Up Interface) [5]. FMI is a standard for the interaction of simulation models generated by different tools. The standard was created within the framework of the MODELISAR [6] project, and enjoys great support from simulation program manufacturers - currently 108 programs.

The standard offers great advantages in co-simulation by allowing models from different tools to be simulated together without having access to the native simulation tools. For this, the models (so-called Functional Mockup Units, FMU) have to be exported as stand-alone files with integrated solver as a binary file. This black-box approach protects intellectual property despite detailed modeling. Thus, supplier can provide OEMs with internal engineering models from the design process of their components without fear of losing their know-how. This allows, for example, a plant manufacturer to have early access to realistic models for virtual engineering and virtual commissioning, without having to run their own modeling effort. The FMU standard is not integrated into AutomationML yet – but ENTOC efforts will bring that to standardization.

One major aspect of using behavior descriptions in general is the definition of interfaces. Additionally, different degree of details may require different methods (data formats) for describing. Depending on the level of detail, the decision is to use PLCopenXML data format for a simple behavior description e.g. a time delay for one movement.

For example, an intermediate behavior may be a more detailed movement curve which is calculated with a defined parameter setting for one use case. This behavior curve can be built up in a MathML formula and easily integrated in AutomationML.

The variable parameters are the connection to the simulation environment, e.g. other components or the control logic. With the FMU concept it is possible to calculate each simulation step depending interacting and changing parameters. FMUs are also capable of describing simple behavior, like shown in Table 1. The big advantage of the FMI concept is that the rules and ways how the FMU should interact with each other are defined in the FMI standard.

properties / data format	simple level of detail	intermediate level of detail	complex level of detail	know- how- protection
PLCopenXML	✓	✗	✗	✗
MathML	✓	✓	✗	✗
FMU	✓	✓	✓	✓

Table 1: Data formats depending behavior level of detail

During the start of Virtual Engineering it should be clear which level of detail is needed. It is nearly impossible to derive a complex model from a simple model with comparable behavior accuracy. Alternatively, one can use for specific engineering steps the convenient level of detail with the appropriate data format. Or the complex behavior as FMU can be used through the whole engineering. But the decision depends on the engineering software environment and the needed focus of the engineering aspects.

In general, handling FMUs is very convenient and useful for Virtual Engineering, where you frequently change parameters to optimize the solution. Further you can use the adjusted FMUs for Virtual Commissioning and get very precise results. If the focus is more on Virtual Commissioning it could make more sense to use simpler variants of behavior models like the PLCopenXML or MathML variant. Furthermore, in latter cases there is no need for the FMI environment.

2.2 Practical examples

To show the difference regarding the level of detail from behavior models with different data formats a comparison between PLCopenXML, MathML and FMU behavior models is shown.
For each calculation following parameter settings are used:

- supply pressure: 5 bar (rel.)
- throttle opening: 30%

In the ENTOC understanding we get behavior models from the component manufacturer in the described level of details.

- **SIMPLE** behavior model:

  PLCopenXML function block which integrates a time during a position

- **INTERMEDIATE** behavior model:

  MathML formula which describes the result of the movement curve

- **COMPLEX** behavior model:

  FMU which can be independently used and free parametrized. The movement for example will be calculated at each simulation step

2.2.1 Pneumatic drive use case without load

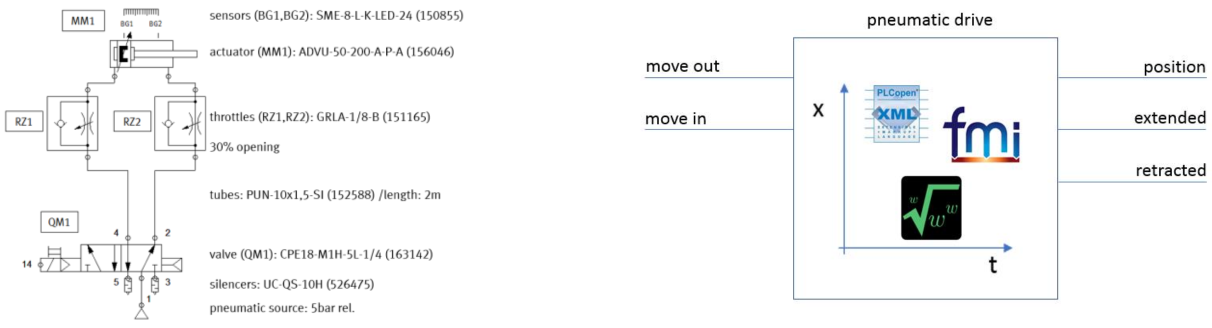
MM1
BG1
BG2
sensors (BG1,BG2): SME-8-L-K-LED-24 (150855)
actuator (MM1): ADVU-50-200-A-P-A (156046)
RZ1
RZ2
throttles (RZ1,RZ2): GRLA-1/8-B (151165)
30% opening
tubes: PUN-10x1,5-SI (152588) /length: 2m
QM1
14
4
2
5
1
3
valve (QM1): CPE18-M1H-5L-1/4 (163142)
silencers: UC-QS-10H (526475)
pneumatic source: 5bar rel.
pneumatic drive
move out
move in
x
PLCopen XML
fmi
t
position
extended
retracted

Figure 3: Fluidic plan of the pneumatic drive example (left) and an example function block (right)

The use case of the pneumatic drive without load compares the different models of the movement behavior from the pneumatic drive for a move out and a move in scene. In Figure 3 the fluidic plan is shown which is the base for the pneumatic drive function block models.

SIMPLE behavior model (without load):

Inside the PLCopenXML model is an integrator interpolating the position as a function of the time during the movement based in calculation results.

$\Delta t_{moveout}$: 2,94s Δt_{movein}: 3,81s

INTERMEDIATE behavior model (without load):

In comparison the MathML model is more detailed and describes the movement as a mathematical function.

$$x(t)=\begin{cases} 0 & t<0{,}02 \\ -0{,}002\times(t-0{,}02)^6+0{,}0191\times(t-0{,}02)^5-0{,}0706\times(t-0{,}02)^4+0{,}1271\times(t-0{,}02)^3-0{,}1138\times(t-0{,}02)^2+0{,}1108\times(t-0{,}02)+0{,}0038 & 0{,}02\le t\le 2{,}94 \\ 0{,}2 & t>2{,}94 \end{cases}$$

Figure 4: MathML formula for the extend movement (without load)

$$x(t)=\begin{cases} 0{,}2 & t<0{,}3 \\ +0{,}000057\times(t-0{,}3)^6-0{,}000613\times(t-0{,}3)^5+0{,}0025\times(t-0{,}3)^4-0{,}0045\times(t-0{,}3)^3+0{,}0035\times(t-0{,}3)^2-0{,}0576\times(t-0{,}3)+0{,}2003 & 0{,}3\le t\le 3{,}81 \\ 0 & t>3{,}81 \end{cases}$$

Figure 5: MathML formula for the retract movement (without load)

COMPLEX behavior model (without load):

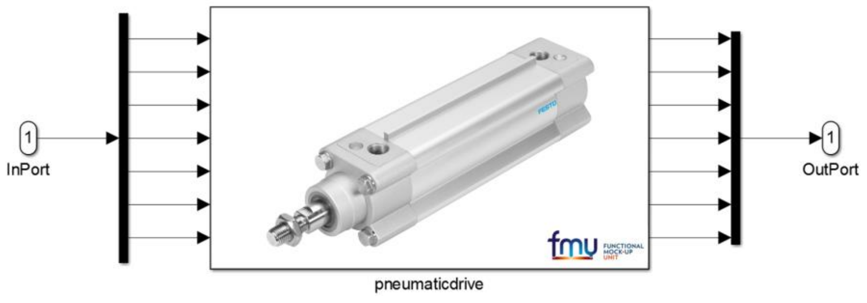

Figure 6: Independent pneumatic drive FMU

The detailed FMU of the pneumatic drive contains a complex physical model and can be used for each parameter setting and is universally applicable (Figure 6). If there is a change of the settings, e.g. a change of the supply pressure or the throttle opening, it is necessary to create a new PLCopenXML and the MathML function.

Results (without load):

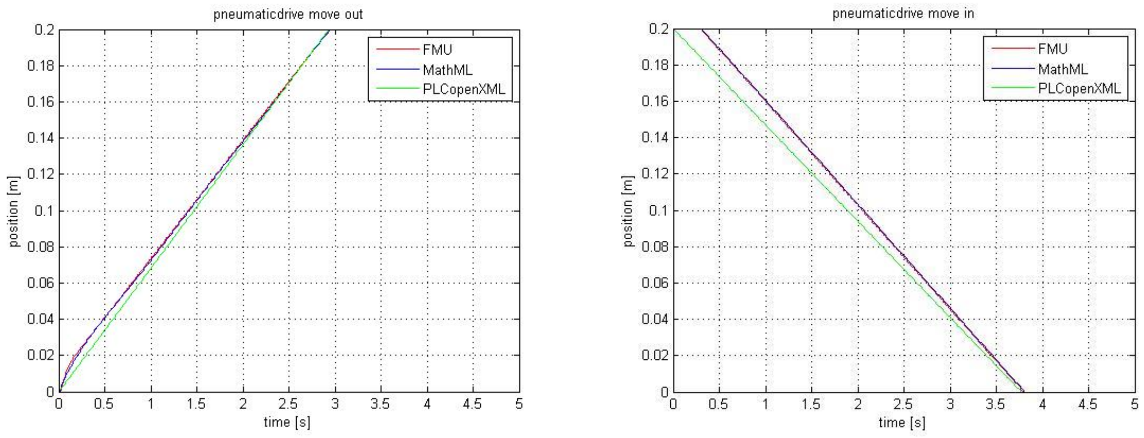

Figure 7: Results of the pneumatic drive movement (extend left/ retract right) without load

As you can see in Figure 7 the results of the different detailed behavior models are very similar to each other. The highest deviation is the not considered switching time delay of the pneumatic valve inside the move in scene from the PLCopenXML model. But in general, the results are appropriate to the movement without load acceptable.

2.2.2 Realistic use case with load curve

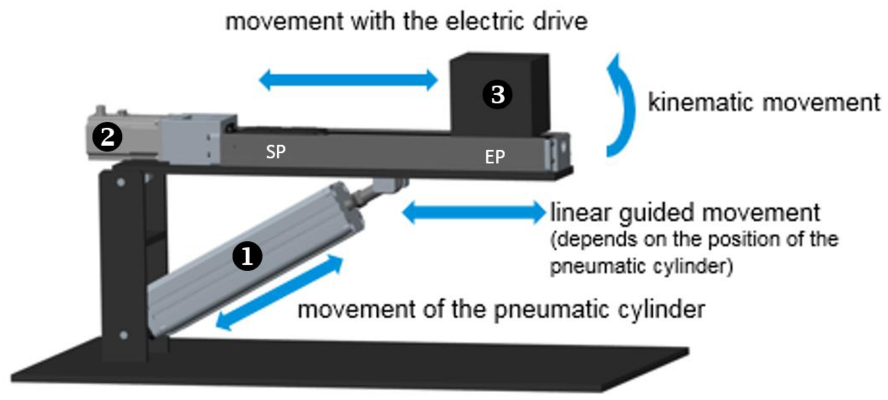
movement with the electric drive
kinematic movement
linear guided movement
(depends on the position of the pneumatic cylinder)
movement of the pneumatic cylinder
SP
EP
1
2
3

Figure 8: ENTOC Component Demonstrator with a pneumatic and an electric drive

The ENTOC component demonstrator contains typical electrical and pneumatic actuators of automated assembly plants: The built-in electric drive (2) is able to linearly move a weight (3) between the start point (SP) and the end point (EP) and is attached to the upper beam. By extending the pneumatic drive (3), this beam can be tilted over the joint, which allows the flow of different processes. The examples rely on the pneumatic part of the ENTOC Component Demonstrator.

For the comparison and further discussions, the pneumatic drive from the Component Demonstrator will be used as functional component.

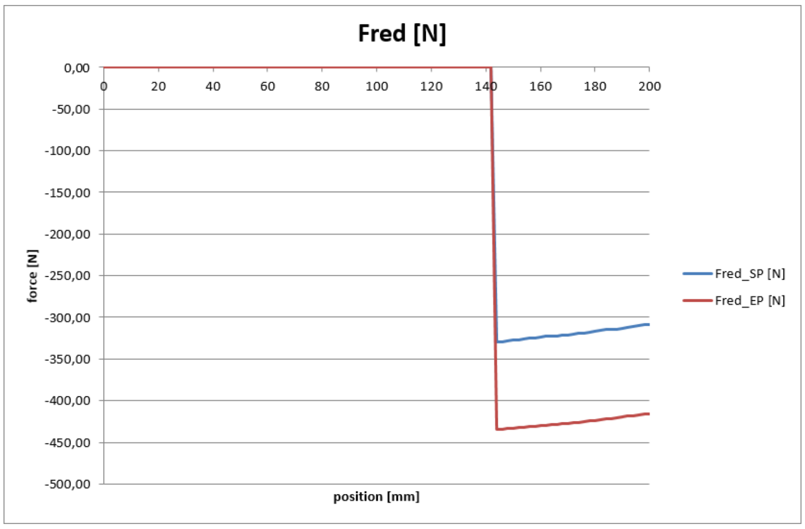

Figure 9: Calculated reduced force consider the kinematic

To keep this example clear, the following results will use the calculated reduced force from mass position of the start point (SP) and the end point (EP). In principle, the Component Demonstrator can vary the reduced force depending the position of the mass between the blue and the red curve which is part of the diagram in Figure 9.

SIMPLE behavior model (load curve SP):

$\Delta t_{moveout}$: 3,27s Δt_{movein}: 3,66s

INTERMEDIATE behavior model (load curve SP):

$$x(t)=\begin{cases} 0 & t<0{,}02 \\ -0{,}0033\times(t-0{,}02)^6+0{,}034\times(t-0{,}02)^5-0{,}1317\times(t-0{,}02)^4+0{,}2341\times(t-0{,}02)^3-0{,}1936\times(t-0{,}02)^2+0{,}1309\times(t-0{,}02)+0{,}000915 & 0{,}02\le t\le 3{,}27 \\ 0{,}2 & t>3{,}27 \end{cases}$$

Figure 10: MathML formula for the extend movement (mass on SP)

$$x(t)=\begin{cases} 0{,}2 & t<0{,}05 \\ +0{,}0029\times(t-0{,}05)^6-0{,}0369\times(t-0{,}05)^5+0{,}1829\times(t-0{,}05)^4-0{,}4409\times(t-0{,}05)^3+0{,}5158\times(t-0{,}05)^2-0{,}2921\times(t-0{,}05)+0{,}2107 & 0{,}05\le t\le 3{,}66 \\ 0 & t>3{,}66 \end{cases}$$

Figure 11: MathML formula for the retract movement (mass on SP)

COMPLEX behavior model (load curve SP):

Independent pneumatic drive FMU.

Results (load curve SP)

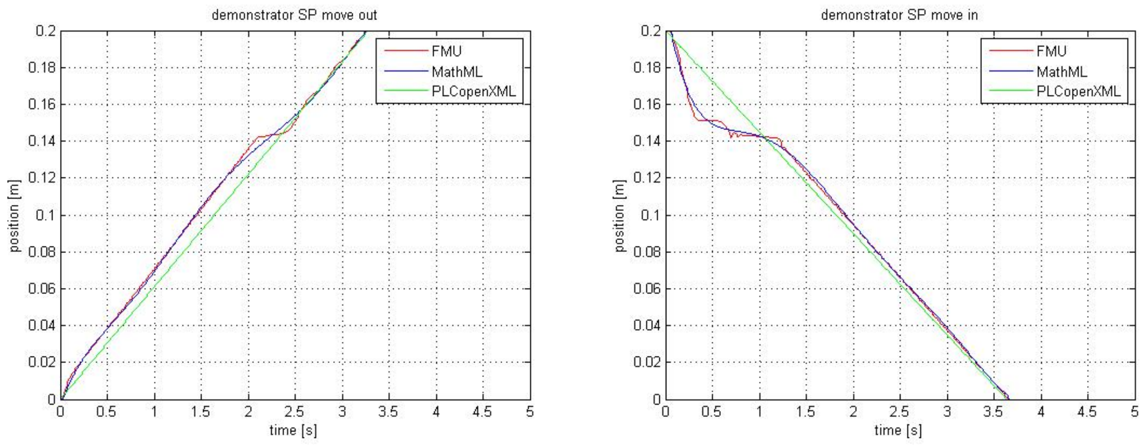

Figure 12: Results of the pneumatic drive movement (extend left/ retract right)

SIMPLE behavior model (load curve EP):

$\Delta t_{moveout}$: 3,41s Δt_{movein}: 3,65s

INTERMEDIATE behavior model (load curve EP):

$$x(t)=\begin{cases} 0 & t<0{,}03 \\ -0{,}0038\times(t-0{,}03)^6+0{,}0406\times(t-0{,}03)^5-0{,}1594\times(t-0{,}03)^4+0{,}2864\times(t-0{,}03)^3-0{,}2375\times(t-0{,}03)^2+0{,}1449\times(t-0{,}03)-0{,}00008 & 0{,}03\le t\le 3{,}41 \\ 0{,}2 & t>3{,}41 \end{cases}$$

Figure 13: MathML formula for the extend movement (mass on end position)

$$x(t)=\begin{cases} 0{,}2 & t<0{,}03 \\ +0{,}0037\times(t-0{,}03)^6-0{,}0466\times(t-0{,}03)^5+0{,}2271\times(t-0{,}03)^4-0{,}5374\times(t-0{,}03)^3+0{,}6164\times(t-0{,}03)^2-0{,}3335\times(t-0{,}03)+0{,}2121 & 0{,}03\le t\le 3{,}65 \\ 0 & t>3{,}65 \end{cases}$$

Figure 14: MathML formula for the retract movement (mass on end position)

COMPLEX behavior model (load curve SP):

Independent pneumatic drive FMU.

Results (load curve EP)

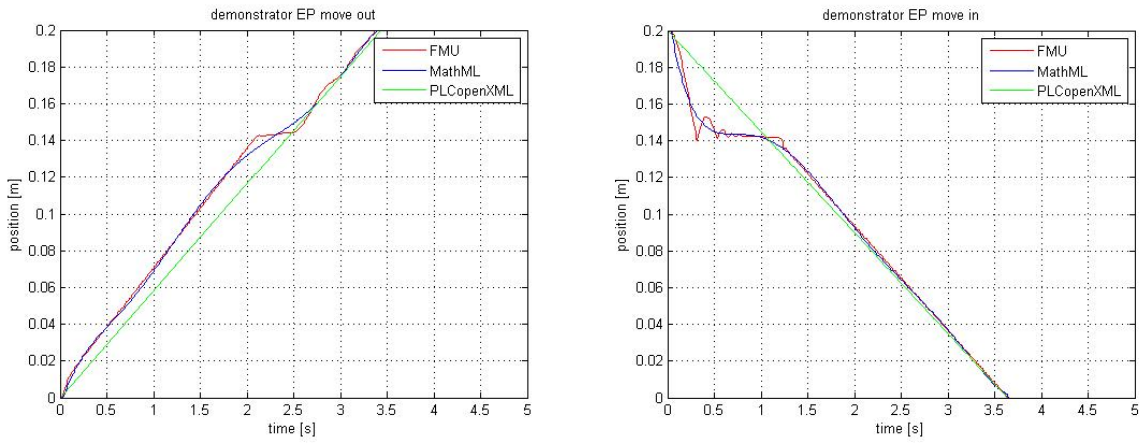

Figure 15: Results of the pneumatic drive movement (extend left/ retract right)

The results show different movement curves depending from the different level of detail (Figure 12 and Figure 15). The largest deviation is from the PLCopenXML model to the MathML and FMU models. Comparing the MathML and the FMU results the deviation is very small – almost insignificant. The huge benefit of the FMU is the independent behavior model, which is used unchanged for each described use case. Compared to the results of the six different PLCopenXML function blocks or six MathML formulas the one used FMU speaks for the easy usability of the FMI concept. For Virtual Engineering, for example, there is not only a change of the reduced forces it is possible that you have to change for example adjustable parameters such as the supply pressure or the throttle opening. With an FMU, these changes are easily realizable without changing or updating the behavior model.

2.3 Kinematic behavior modelling with FMU

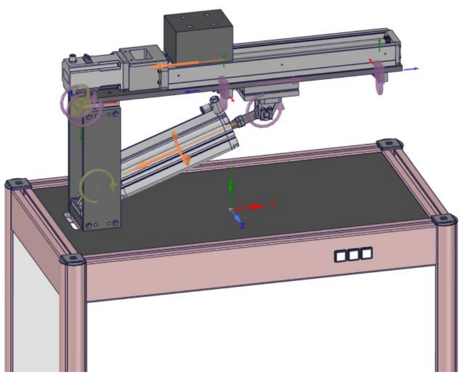
Figure 16: Mechanics FMU in Algoryx Momentum

The idea of the behavior kinematic FMU is that in one simulation environment all FMUs, consisting of different engineering disciplines, can be connected to each other.

A mechanics FMU of the Component Demonstrator, using articulated rigid body dynamics, was created with Algoryx Momentum. The basis for this was the CAD-data provided by FESTO. The FMUs for the pneumatic cylinder and the electric motor which was to be co-simulated together with the mechanical system.
For co-simulating the complete system, the FMUs were loaded inside the Simulink environment in MATLAB, where the pneumatic FMU was connected to the Algoryx Momentum-generated mechanics FMU using a PID controller, synchronizing the pneumatic cylinder position with the corresponding mechanical joint of the cylinder position in Algoryx Momentum. A simple forwards and backwards movement was conducted, where the pneumatic cylinder drove the mechanics FMU.
Some issues appeared when simulating the system in MATLAB 2016b as the default FMU support were not properly able to simulate the FMUs. The main tool used to simulate the system was thus FMU Toolbox from Modelon, which proved to be able to successfully couple the system when tested by Algoryx.
The system was successfully simulated (Figure 17) in real time with an update frequency of 100 Hz. Future work could be spent on including force feedback from the mechanical system to the pneumatic cylinder FMU, as well as including the electric motor FMU in the simulation which would be done in a similar way as the pneumatic cylinder.

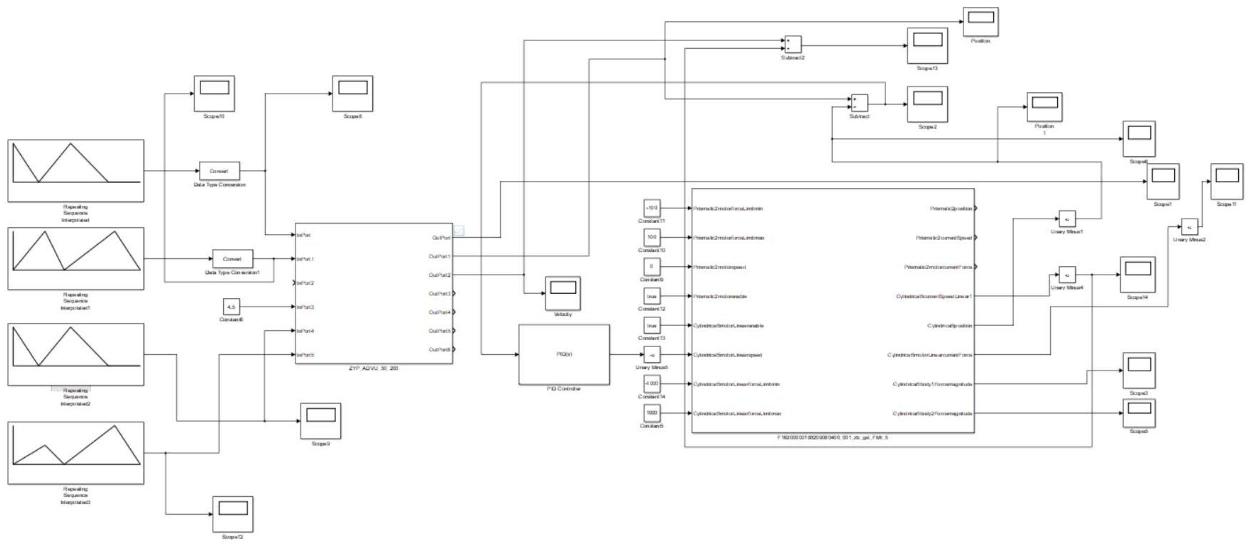
Figure 17: Simulink co-simulation of pneumatics and mechanics FMUs

2.4 Independent standardized interface for component models

The FMI/FMU standard is today no part of the AutomationML description. Ongoing standardization activities driven from the ENTOC project will change that. The main part of the activities is to describe a neutral way to reference from an AutomationML file to a FMU and the appropriate interfaces.

- For the FMI/FMU standard there are some assumptions:
  - It is assumed, that many CAEX-tools will support AutomationML for file transfer in near future.
  - It would be efficient to assign FMUs to components during engineering with CAEX tools and thus use AutomationML for adding FMU-based co-simulation configuration data.
  - Alternatively, AutomationML files generated by CAEX tools may be enhanced with assignments to FMUs afterwards by using a separate tool.

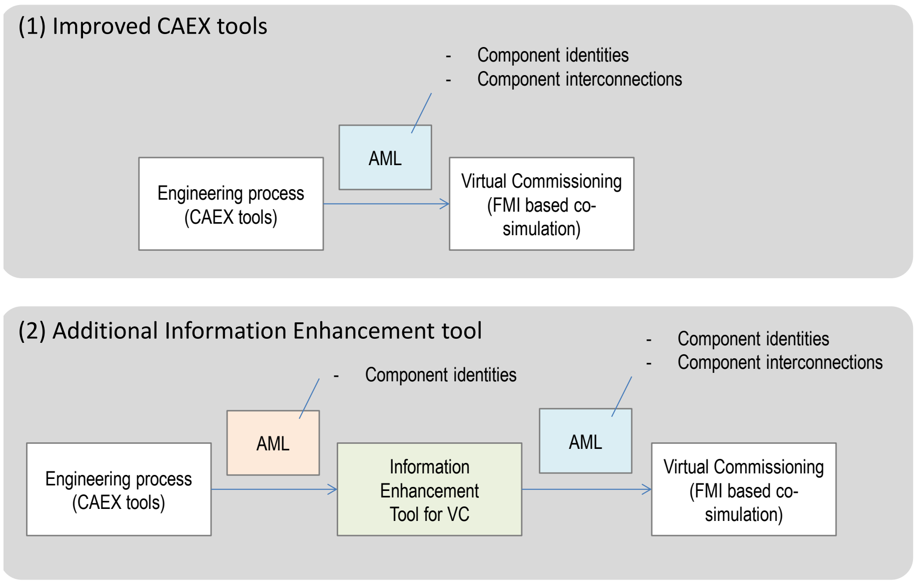
(1) Improved CAEX tools
- Component identities
- Component interconnections
AML
Engineering process (CAEX tools)
Virtual Commissioning (FMI based co-simulation)
(2) Additional Information Enhancement tool
- Component identities
AML
Engineering process (CAEX tools)
Information Enhancement Tool for VC
- Component identities
- Component interconnections
AML
Virtual Commissioning (FMI based co-simulation)

Figure 18: AutomationML inside the engineering process

- Goal:
  - The main goal is to re-use interconnection information of production components engineered in CAEX processes to set-up of Virtual Commissioning scenarios based on FMI
- Information to be formalized in AutomationML:
  - Components may be represented by Internal Elements (IEs) or System Unit Classes (SUCs). For further description we call it component-IE.
  - The component-IE contains a sub-IE, which refers an FMU. For further description we call it FMUConnector-IE, since it is an instance of RoleClass ="FMUConnector".
  - A component-IE may contain multiple FMUConnector-IEs for different goals of simulation. E.g. Hardware-in-the-Loop needs very performant simulation models, while other scenarios need high accuracy of the models, what possibly may cause lower performance.
  - An FMUConnector-IE will contain:
    - An IE (role=ExternalDataConnector) for referencing the FMU
    - Interfaces of input/output variables of the FMU in order to be connected inside of the AutomationML file

With this proposal we are able to reference from an AutomationML file to the path of the FMU. In a further step the expected engineering tool will unzip the .fmu file and find the defined IN – and OUT ports inside the modeldiscription.xml (FMU standard). This interface from the FMU can relate to other AutomationML parts e.g. as internal links (Figure 19).

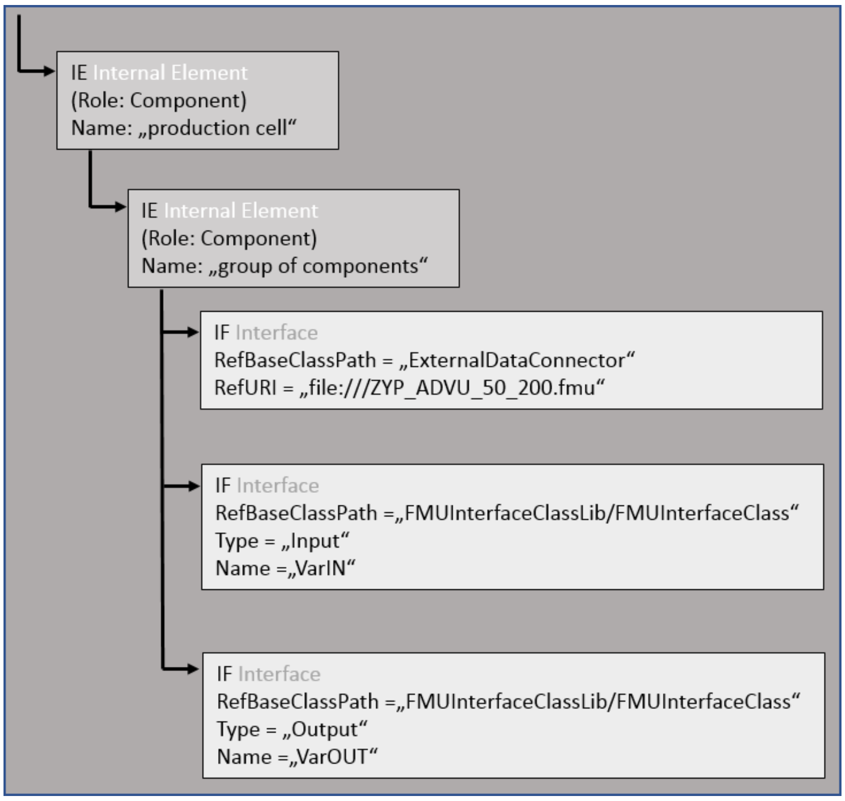
IE Internal Element
(Role: Component)
Name: „production cell“
IE Internal Element
(Role: Component)
Name: „group of components“
IF Interface
RefBaseClassPath = „ExternalDataConnector“
RefURI = „file:///ZYP_ADVU_50_200.fmu“
IF Interface
RefBaseClassPath =„FMUInterfaceClassLib/FMUInterfaceClass“
Type = „Input“
Name =„VarIN“
IF Interface
RefBaseClassPath =„FMUInterfaceClassLib/FMUInterfaceClass“
Type = „Output“
Name =„VarOUT“

Figure 19: FMU integration in the AutomationML structure

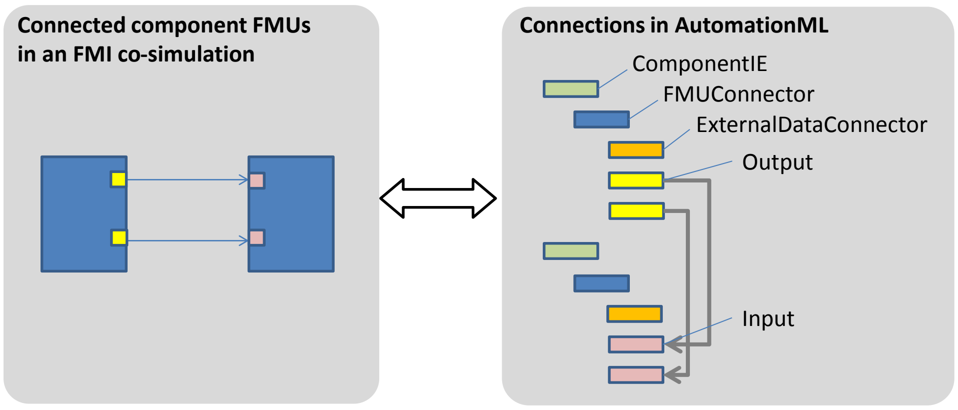
Connected component FMUs in an FMI co-simulation
Connections in AutomationML
ComponentIE
FMUConnector
ExternalDataConnector
Output
Input

Figure 20: Scheme to relate FMU´s

With this proposal, every vendor of FMUs is independent and free to use the interfaces permitted by the FMI standard. The AutomationML integration allows these interface descriptions to be read and linked to other AutomationML parts with internal links which is shown in Figure 20.

2.5 Container format ".amlx"

AutomationML offers the opportunity to create a container file to compress AutomationML data. In general, two types of AutomationML Containers [7] are allowed:

- Local linked container files
- Free linked container files

For the standalone container file which is an approach to work offline with the AutomationML files e.g. construction sites without guaranteed internet access is only allowed to link relative to local files. It´s not possible to refer to public addresses.

The free linked container files can refer to project links via URIs – but they need public access as a kind of online dependency.

Each container file consists of a **Root AMLfile** which serves the entry point for an AutomationML Container and a **Library AMLfile** which contains only RoleClassLibs, InterfaceClassLibs and SystemUnitClassLibs and is always definied as entry point (referenced from the root AMLfile). Within the structure of an .amlx container well known AutomationML data e.g. COLLADA, CAEX and PLCopenXML can be attached to the file (see Figure 21). An .amlx container can just contain library files and can include further .amlx containers.

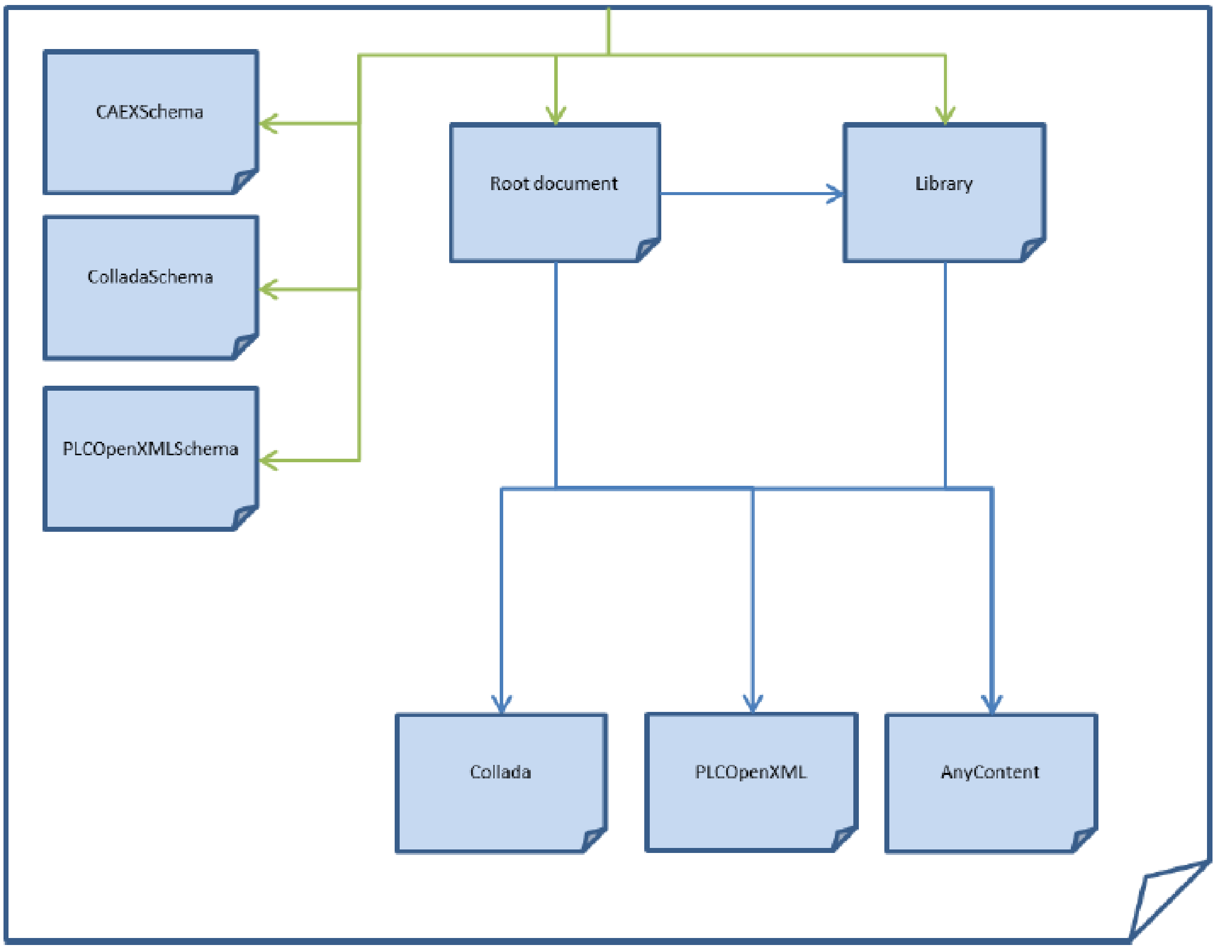

Figure 21: AutomationML container file structure [7]

With these Container files it is possible to simply transfer AutomationML data and integrate it into a higher-level data structure. As an example, the pneumatic and electric drive from the Component Demonstrator is shown as AutomationML container, see Figure 22 and Figure 23.

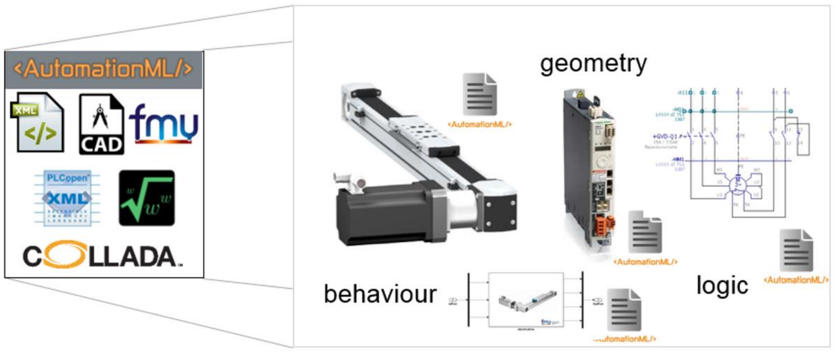

Figure 22: Container format for an electric drive

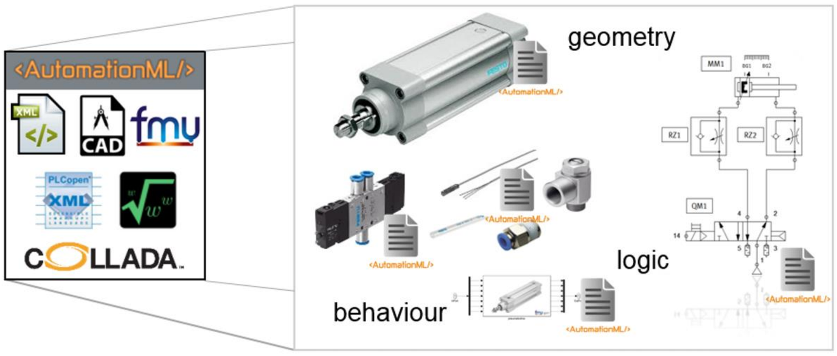

Figure 23: Container format for a pneumatic drive

For the use and handling with AutomationML Containers the depth of the AutomationML data for the ENTOC tool chain should be defined for each part when engineering results can be used by subsequent processes. If, for example the functional component is created by the component manufacturer then one or more AutomationML Containers can be handover to the plant manufacturer. The plant manufacturer again can build up the plant with existing standalone AutomationML Containers to one AutomationML file of the plant and zip this information for the OEM to provide an AutomationML Container of the plant.

3 Model Store package format

The common understanding of a package format is an archive containing files or additional metadata. For the ENTOC project it is necessary to find a solution how the component models and the specific dependencies can be managed through the whole engineering process.

To manage the individual aspects of components described in section 1.2 a holistic component description format has to be defined, which will be described in this section. These component descriptions will then be provided in the model store, which requires a further packaging format, incorporating the format described in this section

Looking at existing dependency management systems like Debian or Maven with package structures and management processes we suggest principles which may be adopted for the management of mechatronic component models to realize the engineering of complex production systems.

3.1 Debian package format

Debian is a modern Linux distribution, which is available for long time (since 1993). Currently it includes more than 51000 software packages [8]. Most of them depend on each other. Debian has been used as base for other Linux derivatives (e.g. Ubuntu, Knoppix, Linux Mint), which means that the majority of Debian software packages are used and are slightly modified and supplemented by further software packages. Thus those associations maintaining Debian derivatives reduce the effort for package management and could be a base for the ENTOC package format.

3.1.1 Debian package structure

Debian packages contain all files necessary to provide a single functionality for the user of the operating system. There are different types of packages:

- Binary packages, containing executables, configuration and documentation files. They are distributed within a Debian-specific archive format (file suffix .deb).
- Source packages, consisting of a .dsc file describing the source package (including names of source files), a .orig.tar.gz file, containing the unmodified original source code and a .diff.gz file, containing Debian-specific modifications against the original source code.

Users of mechatronic component models will probably be not in charge of compiling source code of the models. Thus we will consider the approach of handling Debian binary packages as a similar process to the distribution of mechatronic component models and analyze related package structures and processes more in detail.

The Debian binary package is based on a specific archive format, which can be understood by an archive tool available for almost all Linux systems (the tool is called "ar"). A package can contain member files up to 9 GB. There are three member files defined for the debian package, which must appear in the following order (see [9]):

- debian-binary: a text file of multiple lines (currently there is only one line containing the deb version "2.0")
- control.tar: contains the control information for the package in a compressed format (gzip or xz). Within this compressed container there is a mandatory file "control", which is explained later.
- data.tar: contains the file system with all executables, configuration and documentation files either uncompressed or compressed (gzip, xz, bzip2, lzma).

Further members of the .deb-file are ignored by current Debian package management tools, but are not prohibited.

Debian package filenames are structured as follows: "<ProgramName>_<VersionNumber>-<DebianRevisionNumber>_<DebianArchitecture>.deb". The file name elements have the following meaning (see [10]):

- <ProgramName>: indicates the main executable or functionality provided
- <VersionNumber>: the version number of the program
- <DebianRevisionNumber>: the version of the Debian operating system
- <DebianArchitecture>: the hardware and binary format indicator e.g. x86, amd64, arm, …

The Debian package member control.tar contains descriptive information about the package, which is used by package management client systems to install program systems without conflicts and with resolution of dependencies. Figure 24 shows an example of a control file. It is taken from [10]:

```
Package: hello
     Version: 2.9-2+deb8u1
     Architecture: amd64
     Maintainer: Santiago Vila <sanvila@debian.org>
     Installed-Size: 145
     Depends: libc6 (>= 2.14)
     Conflicts: hello-traditional
     Breaks: hello-debhelper (<< 2.9)
     Replaces: hello-debhelper (<< 2.9), hello-traditional
     Section: devel
     Priority: optional
     Homepage: http://www.gnu.org/software/hello/
     Description: example package based on GNU hello

The GNU hello program produces a familiar, friendly greeting.  It
allows non-programmers to use a classic computer science tool which
would otherwise be unavailable to them.
      .
Seriously, though: this is an example of how to do a Debian package.
It is the Debian version of the GNU Project's `hello world' program
(which is itself an example for the GNU Project).
```

Figure 24: Example of a control file

All possible fields of such control files are described in [11]. We will explain some selected fields in the following:

- Package: contains the name of the package as <ProgramName> in the name of the .deb package file name.
- Version: contains the <VersionNumber>-<DebianRevisionNumber>+<DebianRevisionNumber> as described above.
- Architecture: contains the <DebianArchitecture> as described above.
- Relationship fields (Depends, Pre-Depends, Recommends, Suggests, Breaks, Conflicts, Provides, Replaces, Enhances)

  - Those fields contain relations to other packages by specifying name and version information of those packages and taking conjunctions and disjunctions of those packages into account
  - Example: "Depends: libc6 (>= 2.2.1), exim | mail-transport-agent" specifies, that this package is dependent on package libc in version 2.2.1 or above and from either the package exim or mail-transport-agent.

At least relationships as "depends", "conflicts" and "replaces" are expected too in case of engineering of complex production systems by use of mechatronic component models.

3.1.2 Debian binary package management

There is a variety of tools to manage packages of a Debian system and the Debian package repository. They provide following functionalities [10]:

- Manipulate and manage packages or parts of packages,
- Administer local overrides of files in a package,
- Aid developers in the construction of package archives, and
- Aid users in the installation of packages, which reside on a remote FTP site.

Similar tools are expected to support the management of mechatronic component models within engineering workflows.

3.1.3 Conclusions for the AutomationML package format

There are a variety of consequences, which should be taken into account, when designing package formats and tool chains for the management of mechatronic component models:

- A package should contain the following information:
  - a package format version indicator (see debian-binary member of a Debian package)
  - component name (see <ProgramName> of the Debian package file name)
  - version of component model (see <VersionNumber> of the Debian package file name)
  - AML version (see <DebianRevisionNumber> of the Debian package file name)
  - Information about relationships to other packages (see Depends, Conflicts, Replaces, … elements of the control.tar member of a Debian package)
  - a container for the payload data of the package.
- There is a need to specify data formats:
  - for the definition of a control.tar equivalent
  - for the data container (including compression algorithm)
- There is a need for tools
  - to create and modify packages for the mechatronic component model developers
  - to provide functionalities of a package server (Debian uses a kind of FTP/SFTP here)
  - to provide functionalities for users of packages, which have access to control.tar equivalents and provide the functionalities of computing all dependencies and possible conflicts of package installations; search of packages should be easy in those tools and it shall also consider commercial aspects
- There is a need for a package development strategy
  - e.g. the concept of packaging single functionalities of a program system into separate packages allows the user a better control over his package installations (e.g. for a mechatronic component considered as a model system, there should be separate packages for behavior model, documentation, 3D-information, …)

- within the Debian package system, there exist meta-packages, containing only dependency information about all separate packages of an according program system and in case of mechatronic component models, the concept could be used to create meta-packages with dependency-references to all sub-models (behavior, 3D, … models).

3.2 Maven

Maven [12] is a tool to manage building software and is Java based. It describes how the software should be build and describes the necessary dependencies to build and manage software projects.

Core of any Maven project is the Project Object Model represented by the XML file `pom.xml`. The Project Object Model is used for describing a Maven Package as well as a Maven development project. For Maven Packages, this file contains metadata, packaging information and dependency information. Obligatory metadata for a Maven Package consists of a <groupId> in reversed-DNS notation, an <artifactId> used for identifying the package within a group, a <version> and information about the packaging itself.

The combination of <groupId>, <artifactId> and <version> serves as a unique identifier for each Maven Package. Additional metadata for a Maven Package can be added as key-value pairs in the <properties> section of the Project Object Model. A lot of pre-defined metadata can be added to the Project Object Model such as license information, owning organization or URL to the project page. The Project Object Model can contain information about other packages this package relies on, also known as dependency information. Dependencies in the Project Object Model are referenced via the unique identifier consisting of <groupId>, <artifactId> and <version>, where the version can be specified as an exact version, version ranges or a list of exact versions and/or version ranges.

The Maven client can download all dependencies including the dependencies of the dependencies, also known as transitive dependencies, when performing any Maven goal that requires them. Maven goals specify actions on a Maven project, which can be extended by Maven plugins. Such goals can include building, running or packaging a Maven project. Maven is used as a de-facto standard in JAVA software development, but can be used in other contexts as well. Available package formats include a zip-format, which can contain any file-based payload data for a project.

3.3 ENTOC specific Model Store file

An additional file is required to use the generated behavior models with the AutomationML description within the ENTOC tool chain in combination with the dependency management of the ENTOC Model Store. The content of the file includes basic information about the behavior model e.g. version and compatibility information to enable the functionality of the dependency management. It is also possible to refer and use the attributes and structures of eClass [13] specifications or other common discipline-specific definitions. This allows to search for components with common used basic attributes. Figure 25 shows an example of an additional file which will be used for the prototype Model Store of the ENTOC Project.

```
//ENTOC additional model store file
{  "descriptorVersion":"1.0.0",                            // informs about the version
   "name": "theNameOfThePackage",                          // name of the model package
   "version": "v1.0.7",                                    // version of the model package
   "deprecated": false,                                    // declares whether the package is deprecated or not
   "previousVersion" : "v1.0.5",                           // [not in first implementation]
   "description": "Text with a description of the Package.", // a short description of the model package
   "typeIdentification":{                                  // modelPackageType itself is dependent on its identifier
      "eClass":{                                           // link to eClass structure possible
         "modelPackageType": "rollerConveyor",             // type of the model package (search filter)
         "modelPackageTypeId": "13-47-42-11",
         "attributes": {
            "Speed": "3",                                  // attributes are stored (for the search filter)
            "SpeedUnit": "m/s"
         }
      },
      "customIdentifier":{                                 // link to individual customized structure also possible
         "modelPackageType": "rollTransportUnit",          // type of the model package (for the search filter)
         "modelPackageTypeId": "",
         "attributes": {
            "Acceleration": "3",                           // attributes are stored (for the search filter)
            "UnitOfSpeed": "m/s"
         }
      }
   },
   "behaviorModels":{                                      // the behavior model included in the model package
      "behavior.fmu":{
         "replaces":"vendorXYZ/testPackage - v1.0.2"       // [not in first implementation] breaks, conflicts, etc. ...
      }
   },
   "3dModels":[                                            // the 3d models included in the model package
      "test.dae"
   ],
   "others":[                                              // other files included in the model package
      "values.xslx",
      "manual.pdf"
   ],
   "dependencies":[                                        // paths of all depending model packages are stored
      "ENTOCvendorABC/testPackage - v1.0.4"                // path to the vendor and the model package
   ],
   "author": "ENTOC member",                               // author of the model package
   "createdOn": "2018-03-30T11:01:26.9755646+02:00",       // timestamp when the model package was uploaded
}
```

Figure 25: first version of the ENTOC Model Store file

3.4 Engineering Model Store package format

The entry point to get the Engineering Model from the Model Store is that the plant builder knows which kind of product or related specifications will be used inside the plant. This information normally is defined in the specification sheet of the planned production plant. The component vendors have the opportunity to generate a functional component including topology, geometry, behavior and logic in AutomationML and can pack that to the AutomationML container format. To use the functional components inside the Model Store an additional Model Store file has to be generated for each functional component to enable the dependency management which will be used inside the store. This additional file allows further functions inside the Model Store like versioning, actualization, searching and lot more helpful functionalities. At least it is easily realizable to detect which special characteristic of each functional component is stored.

To increase the range of components inside the Engineering Model Store component vendors could offer their product portfolio of functional components or they generate a functional component triggered from a request coming from the Engineering Model Store – which can enable new business models.

The Engineering Model Store package format consists of the AutomationML file of the functional component and the additional Model Store file, which contains versioning information and specification attributes, see Figure 26.

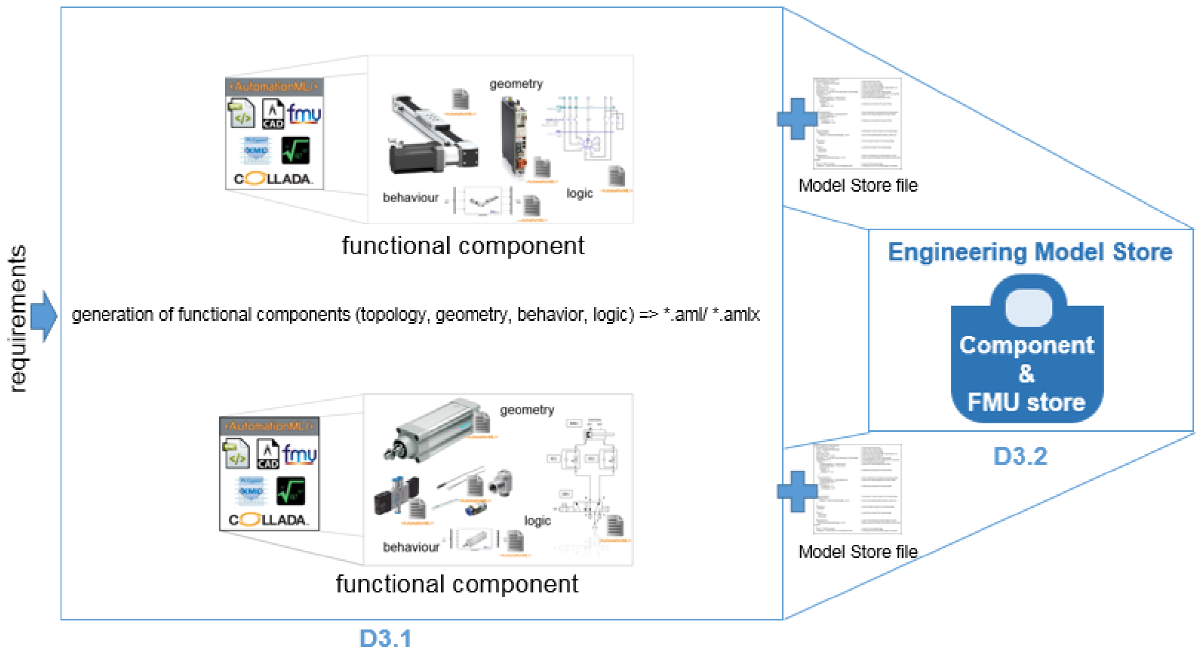

Figure 26: Scheme of the Model Store package format

4 Summary

The content of Deliverable 3.1 focuses on the various ways to create a component package format based on AutomationML, where the functional component contains characteristics of behaviour (PLCopenXML/MathML/FMU), logic (PLCopenXML) and geometry data (COLLADA).

It is possible to generate a component package with different kinds of e.g. behaviour models. The package format and the interface description itself are independent and can contain one or more variants of behaviour models for the different ways of engineering. From a technical point of view, the FMI standard offers the most promising potential for using maximum accuracy and an independent simulation architecture. Ongoing standardization activities regarding the integration of the FMI concept within AutomationML will be forced from WP3.

One further focus of D3.1 is the discussion about the usage and content of additional files to enable an Engineering Model Store within the ENTOC toolchain. The concept of the Engineering Model Store itself will be detailed described in Deliverable 3.2.

For the ENTOC tool chain an engineering package format inside the Model Store will be offered. It is a AutomationML component file with an associated Model Store file. This additional Model Store file enables functionalities of the dependency management system which will be implemented in the Model Store prototype.

5 References

[1] AutomationML – www.automationml.org

[2] COLLADA – www.collada.org

[3] PLCopenXML – www.plcopen.org

[4] MathML – www.w3.org/Math/

[5] FMI – www.fmi-standard.org

[6] MODELLISAR – www.itea3.org/project/modelisar.html

[7] BPR_008: AutomationML Container – https://www.automationml.org/o.red.c/dateien.html

[8] The Debian Project – www.debian.org

[9] deb(5) manual page of the Debian operating system (run "man deb" on such a system)

[10] The Debian GNU/Linux FAQ, Chapter 7 - Basics of the Debian package management system – www.debian.org/doc/manuals/debian-faq/ch-pkg_basics.en.html

[11] Debian Policy Mailing List – www.debian.org/doc/debian-policy/policy.pdf

[12] Maven – www.maven-apache.org

[13] eClass – Classification and product description: www.eclass.de

All links visited on 2018-03-13