Fakultäten für Informatik und Maschinenwesen

der Technischen Universität München

Interdisziplinäres Projekt

Model Transformations for Product-Service Systems

Bernhard Radke, Konstantin Govedarski

Fakultäten für Informatik und Maschinenwesen
der Technischen Universität München

Interdisziplinäres Projekt

Model Transformations for Product-Service Systems

Verfasser:	Bernhard Radke
	Konstantin Govedarski
Aufgabensteller:	Prof. Dr.-Ing. Udo Lindemann
Betreuer:	Christopher Münzberg
	Danierl Kammerl
	Konstantin Kernschmidt
	Thomas Wolfenstetter

Submission Date: 15.04.2014

Contents

Chapter 1

Introduction

In present-day economic circumstances, businesses are presented with numerous challenges. Some of those challenges originate form the businesses themselves, like for example the strategic aim of growing and expanding into new markets, or the forging of customer and market awareness. Other challenges originate from the business and regulatory environment of the business in question. These environments force a business to be ever more competitive and to optimize for sustainability. To achieve this, industries no longer put the product itself at the core. Rather, they incorporate the product into a number of services, which reach from the lowest-level technical detail to the highest-level customer interaction and provide for optimal product utilization. In this sense, what a company tries to sell nowadays is not just the product itself, but a service through which the customer consumes the product, while being isolated from technical detail and unnecessary responsibility. Such complete solutions are denoted as Product-Service Systems (PSS).

While the incorporation of Product-Service Systems in an industry requires an initial effort and an evolution of the companies' cultures, it can lead to a more concise and efficient utilization of resources to the benefit of both the companies themselves and their customers.

Importance of Integration

After their implementation and market introduction, Product-Service Systems can bring numerous benefits to a company. Nevertheless, their implementation brings a number of challenges of communication, integration and complexity. To explain this, let us consider the parties involved in the development and production cycles of a Product-Service System. Both development and production involve a number of disciplines, each acquainted with its own modelling languages and tools. Each of these domain-specific languages furthermore is concerned only with those aspects of the overall systems, which are relevant for the discipline at hand. As a result, each involved party has only a limited excerpt of the entire Product-Service System at its disposal. In the worst case, this leads to contradictions in the design and implementation of the system. In the moderate case,

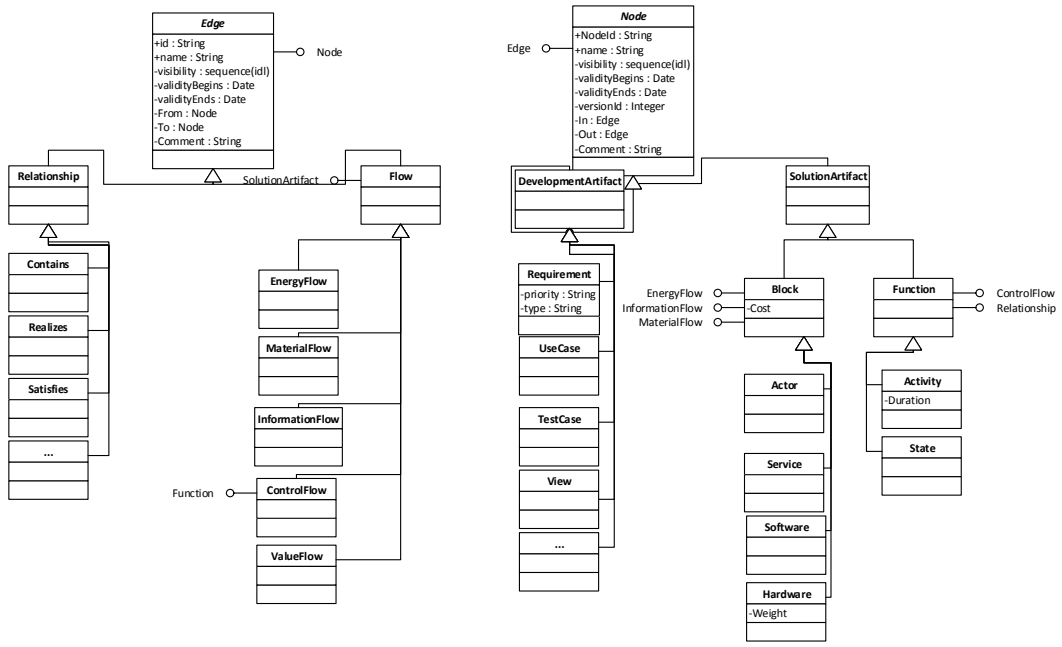

Figure 1.1: PSS-Integration Framework Common Structure

it only incurs significant synchronization and management costs to the company. It is thus of crucial importance for the success of a PSS venture to establish a mechanism though which the models developed by different disciplines can be transformed to each other, or even to incorporate them all in a single model, representing the Product-Service System in its entirety.

PSS Integration Framework

An approach to the integration of the disciplines involved in the development of a Product-Service System is researched jointly in the faculties of Informatics and Mechanical Engineering of the Technical University of Munich by C. Münzberg, D. Kammerl, K. Kernschmidt and T. Wolfenstetter. The approach is named Product-Service Systems Integration Framework (PSS-IF) and provides a methodology and semantics for bringing the business, computer science and mechanical engineering aspects of a Product-Service System together. In its core, the framework describes a common structure which is sufficiently expressive to incorporate the PSS-wide relevant features of each domain-specific aspect of the PSS. This structure is depicted in Figure 1.1.

TODO Diagram from Thomas.

The main goal of the PSS-IF is the usage of this structure for the integration of models of different disciplines. For this purpose, a key step is the ability to translate models from and to this common structure.

Scope of the Interdisciplinary Project

In the context of the PSS Integration Framework (PSS-IF), the goal of this interdisciplinary project is to design and implement a Proof of Concept software utility which:

- follows the PSS-IF methodology and semantics
- can transform a model described in one relevant domain-specific language into another relevant domain-specific language
- considers the following domain-specific languages relevant:
  - Event-driven Process Chain (EPC) Diagrams
  - Business Process Modelling Notation (BPMN) Diagrams
  - Flow-oriented Functional Modelling (UFM) Diagrams modelled using the BOOGGIE-Tool (Bringing Object-oriented Graph Grammars into Engineering)
  - Systems Modelling Language for Mechatronics (SysML4Mechatronics) Diagrams
- should, at the time of its delivery, support at least two of the relevant domain-specific languages

Structure of this Documentation

This documentation is structured as follows:

- **Chapter 2** discusses possible approaches to the conceptual structure of the Proof of Concept developed in this interdisciplinary project.
- **Chapter 3** describes the implementation of the Proof of Concept.
- **Chapter 4** provides the results obtained from the Proof of Concept.
- **Chapter 5** provides a few ideas about possible future developments on the basis of the provided Proof of Concept.
- **Appendix A** provides the distribution of tasks between the authors of the Proof of Concept.

Chapter 2

Approach

This chapter presents and, to some extent, justifies the approach of the Proof of Concept (PoC) software utility developed in the scope of this interdisciplinary project. To achieve this, first an overview of the different levels of abstraction in a language are made in Section 2.1. Then, a comparison of different transformation methods on an abstract level is provided in Section 2.2. Thereafter, the chosen approach to the realization of model transformations is defined on an abstract level in Section 2.3.

2.1 Language Levels of Abstraction

TODO concrete , absract, semantics

2.2 Transformation Methods

Provided with the task to transform between different Models, there is an number of possible solutions. These can roughly be categorized into direct and indirect transformation methods.

2.2.1 Direct Transformation Methods

Direct transformation methods are such which define rules for the transcription from source to destination domain-specific language directly, i.e. such methods do not produce a (defined) intermediate result, but rather are always language-specific. Furthermore, it can be differentiated between syntax-dependent and syntax-independent technical solutions for this kind of transformations.

Syntax-Dependent Transformations

Syntax-dependent transformations would build upon a technology which is well-suited for processing the syntax of the source and destination domain-specific languages (DSLs). For example, considering DSLs which are both use the Extensible Markup Language

(XML) as their concrete syntax, an appropriate transformation technology might be EXtensible Stylesheet Language (XSLT).

Syntax-Independent Transformations

Syntax-independent direct transformation methods are a category of methods, which, while still transforming directly from a source DSL to a destination DSL, are not coupled to the concrete syntax of any particular DSL. Such transformation approaches can be realized in a high-level programming language, which relies on a number of serialization and de-serialization components for the transmission of own language-specific data-structures to the respective DSL concrete syntax serializations.

2.2.2 Indirect Transformation Methods

Indirect methods of transformation are methods which rely on a stable and well defined intermediate format. A particular transformation between two DSLs is performed by first transforming from the source DSL into the intermediate language and then transforming from the intermediate language to the target DSL. Once again, from a technological perspective, a differentiation between two categories of intermediate languages can be made: fixed and flexible intermediate languages.

Fixed Intermediate Language Transformations

Transformation methods with a fixed intermediate language can be realized in a high-level programming language. In this case the abstract syntax of the intermediate language is directly implemented as a data structure in the programming language. Thus, the intermediate language is described directly with the vocabulary of the programming language in use. Also, in this case, the transformations from and to DSLs can be implemented directly in the syntax of the particular programming language and are, therefore, most likely Turing-complete.

Flexible Intermediate Language Transformation

Transformation methods with a flexible intermediate language can also be technologically solved with a high-level programming language. As opposed to the last case, here the intermediate transformation language is not fixed, in the sense that it is not hard-coded. Note that the intermediate language is still likely to be fixed for the scope of a single transformation.

2.2.3 Discussion

In this section the different transformation methods presented above are compared with each other. Let us first consider syntax-dependent transformation methods such as XSLT. On the one hand, this kind of transformations can be advantageous, because of their closeness to the languages at hand. A direct transformation can always be defined

to enclose the maximal possible transmittable detail from one DSL to another. On the other hand, such approaches impose a limitation to the entirety of languages which can be supported, due to their binding to the concrete syntax of those languages.

Syntax-independent direct transformations resolve this issue by abstracting the transformation description from the concrete syntax of the particular language, but still have a number of significant disadvantages. The most important one of these is the fact that such transformation methods require an explicit implementation for each pair of source and target DSLs. As a result, with the introduction of each new language, a transformation procedure has to be defined for the combination of this language with every language already supported. Illustratively put, the result is a complete graph (each node being a DSL and each edge a language-to-language transformation) and the number of necessary implementations grows exponentially in the number of DSLs to support. In a dynamic field, where new languages may appear at any time, direct transformation approaches would thus incur significant costs on the longer run. Another disadvantage of these approaches is that with the growing number of transformation implementations, the code-base also grows proportionally. As a result, the code maintenance for such a utility also becomes more costly with time.

To address the issue of exponentially growing complexity, indirect transformation methods can be used. As already noted, the methods in this category use an intermediate format to and from which transformations are made for each DSL. As a result, the addition of a new DSL will require the additional implementation of just one transformation, as opposed to as many transformations as there are languages to support. The direct solution in this case would be to directly implement the intermediate language in the programming language of choice. Such a description of the intermediate language and its transformation would, on the one hand, have the advantage of being expressed in the concepts of the used programming language directly and thus being accessible to any person with knowledge of this programming language. On the other hand, a fixed intermediate language is likely to be more costly once the evolution of the transformation framework is taken into consideration. In particular, any conceptual change in the intended meaning of a transformation would directly impact the transformations to and from all DSL on the level of the programming language used. In essence, it would be necessary to potentially rewrite the code used for all transformations, which, considering a growing number of DSLs and a continuously evolving modelling methodology, would once again incur long-term costs. Also, while the code base in this case is significantly smaller than in the case of direct transformation approaches, it still is growing in proportion with the number of DSLs to support.

Finally, the most abstract approach is the usage of flexible intermediate language transformations. As opposed to the last approach, here the intermediate language is not bound by concepts of the underlying programming language, but rather is only expressed in those concepts. As a result, the intermediate language can be defined on a level of abstraction on which most, if not all, new requirements can be expressed in terms of instantiation instead of code generation. In particular, only requirements which introduce new concepts in the language would require a modification of the code

base. Requirements which merely imply a structural change can the implemented by configuration. The advantages of such an approach are numerous, most importantly that it would provide a viable, flexible and powerful tool with only limited costs for the introduction of new DSLs. The major disadvantage of such an approach is that it incurs a significant initial effort, as it requires the infrastructure for the desciption of the intermediate language to be implemented as well.

With this considerations made, the authors consider a flexible intermediate language approach to be the most appropriate one for the PoC software utility, as it best addresses the purposes of the PSS Integration Framework.

2.3 The PSS-IF Transformation Method

In the last section the choice of a flexible intermediate language as a transformation method was made. In such a transformation method, there are a few central concepts, around which further definitions revolve.

2.3.1 Levels of Meta

In the rest of this documentation, the terms Metamodel and Model are used continuously. For this reason, here these are defined somewhat more clearly from the perspective of levels of information and structure. The most basic of those levels is reality itself, which is, as such, incomprehensible in its fullness. This is why a model is defined as:

Model A simplified description of reality, which only contains certain aspects of it, relevant for the creator of the model.

A model, while depicting certain aspects of reality, does not necessarily conform to any particular structure. The structure of a model is provided by a metamodel, defined as:

Metamodel A description which defines how models are structured. A model is said to be an (ontological) instance of a metamodel, if the model structurally conforms to the metamodel.

Note that no constraint is given as to the number of meta-levels involved – one can easily define meta-metamodels etc.

2.3.2 Viewpoints and Views

In the last section we have defined the terms metamodel and model. In this section, the terms Viewpoint and View are added.

View A view is a subset of a model.

Viewpoint A viewpoint is a description which defines how views are created.

A viewpoint can thus be seen as a restricted or, more generally, transformed metamodel, which makes it possible to perceive model instances of this metamodel from a certain perspective (viewpoint). More precise definitions of this terms can be found in the ISO 42010 standard.

2.3.3 PSS-IF in a Nutshell

Consider a metamodel which describes all aspects of reality, which are of importance for the development of a Product-Service System. Such a metamodel is denoted as the PSS-IF Canonic Metamodel. For each domain-specific language, a viewpoint is defined on the basis of the PSS-IF Canonic Metamodel and captures only those parts of the canonic metamodel, which can be represented in the corresponding DSL. Furthermore, viewpoints are defined in such a way that they can be used for both reading from and writing to a model.

This makes it possible to achieve a transformation between two exemplary DSLs *A* and *B* through the following process:

1. Obtain the viewpoint for DSL *A*.
2. Create an empty model.
3. Add data to this model from an external source, whose abstract syntax is that of DSL *A*.
4. Obtain a viewpoint for DSL *B*.
5. Create a new external target, conforming to the abstract syntax of DSL *B*.
6. Extract data from the model through the viewpoint for DSL *B* and write it to the target.

Note that after step 3 the model actually contains data conforming to the PSS-IF Canonic Metamodel. This is because the data is added through a viewpoint which internally transforms it to conform to the canonic metamodel. This is why the data can, in step 6, be read directly with another viewpoint from the same model and is implicitly transformed to the abstract syntax of DSL *B*.

2.3.4 Parts of the PSS-IF Proof of Concept

To realize the PSS-IF transformation approach, a number of concepts need to be defined. While these concepts are briefly described here on an abstract level, they also closely resemble the implementation, presented in detail in the next chapter. The concepts required are provided in the following sections.

PSS-IF Metamodel

A PSS-IF Metamodel consists of a multitude element types and data types. Each element type can be either a node type or an edge type. Both node and edge types can have attributes, and each attribute is bound to a particular data type. Associations between node types are defined through connection mappings, which are always bound to a certain edge type, i.e. an edge type can have many connection mappings, allowing it to associate different pairs of node types. Furthermore, inheritance can be defined among the node and edge types. It holds that attributes are inherited, while connection mappings are not. Additionally, node types can be defined to have edge type semantics under certain circumstances, so that chaining of associations is possible. Note that element and data type names are unique in the scope of a PSS-IF Metamodel and attribute names are unique in the scope of an element type. Finally, a PSS-IF Metamodel always contains a top-level node type called 'Node' and a top-level edge type called 'Edge', both of which define a set of predefined attributes.

PSS-IF Model

A PSS-IF Model consists of nodes and edges, both of which have attributes. A model does not have any knowledge of its own structure with respect to any PSS-IF Metamodel, i.e. all structural information in the model is on the level of incoming and outgoing edges and (untyped) attribute values. Since all structural information is held in the metamodel, accessing the same model with different metamodels will yield different results, as it is intended for the purpose of the transformations.

Viewpoints and Transformations

A *transformation* is a function which, when applied to a metamodel, results in a transformed, slightly changed metamodel. This changed metamodel is denoted as a *viewpoint* and represents a step in the direction of defining the way in which the PSS-IF Metamodel is seen from the perspective of a certain domain-specific language. The metamodel on which a transformation is applied is called *base metamodel* of the resulting viewpoint. The viewpoint contains element types, i.e. node or edge Types, which, on demand, whenever they are used to operate on a model, apply the actual transformation of data. A new viewpoint can then be constructed by applying another transformation to an already existing viewpoint, which serves as it's base metamodel. Thus, through the chaining of transformations, different viewpoints can be constructed. This makes it possible to describe viewpoints for the different domain-specific languages the PSS-IF needs to support. It should be noted that, in general, the ordering of the application of transformations is of relevance.

So far, the following transformations have been identified as sufficient for the definition of the viewpoints of all DSLs relevant for this interdisciplinary project:

Rename The rename transformation is responsible for replacing the name of an existing element type with a new one in the scope of a viewpoint.

Alias The alias transformation allows the definition of multiple element types within a viewpoint, all of which are mapped to a single element type within the base metamodel of the viewpoint.

Artificialize The artificialize transformation is used for the creation of additional artificial entities in a model, when instantiating a node or an edge, respectively.

Hide Applying this transformation to a metamodel removes a specific element type together with its subtypes, or a certain connection mapping of an edge type, from the resulting viewpoint.

Deinstantify This transformation results in a viewpoint hiding all instances of a certain element type within a model. It is required to, in contrast to the hide transformation (see above), allow subtypes of the deinstantified type to be instantiated, while still hiding the instances of the deinstantified type itself.

Join This transformation is used to internally create an edge between nodes, which are neighbours of the nods externally designated as the two ends of the respective edge. The actual neighbours used for the connection are determined in accordance with specific paths defined in the transformation.

Generic Graph

A generic graph is required as an abstraction for the concrete syntax of all external representations. The proof of concept implementation relies on a generic graph in which nodes and edges have a string designating their type, yet no further structural information.

In essence, the generic graph is used as follows: The data from an external representation is first read into a graph, which is then processed generically with the viewpoint of the respective DSL. In the same manner, when exporting a view, the model is first converted to a generic graph in accordance with the viewpoint of the respective DSL, and the generic graph is then serialized in accordance with the specifics of the concrete syntax of the DSL.

This approach has the following advantages:

- Separation of concerns: By using the graph it is possible to separate the handling of concrete and abstract syntax of each language from each other. The concrete syntax is handled by a specific utility, which rather relates to the syntax than to the language. For example, more than one language might be serialized to XML using the same utility. The abstract syntax of the DSL is meanwhile handled independently in accordance with the DSL's viewpoint.

- Extension point for pre- and post-processing of data: It might be necessary, for certain DSLs, to perform certain pre- or post-processing modifications to the data, i.e. to normalize it. If such normalizations are out of the expressiveness of the operators of PSS-IF (i.e. are generic graph transformations), they can be applied directly onto the generic graph.

Mappers

Finally, mappers are utilities which encapsulate the whole transformation process to and from the PSS-IF Canonic Metamodel and a corresponding model. A mapper thus offers two functionalities, one for reading a model from an external representation and one for writing a model to an external representation. Each DSL has its own mapper and each mapper combines, in the appropriate order, the DSL's viewpoint creation, data transformation, any pre- and post-processing strategies, and the correct serialization utility.

Chapter 3

Implementation

In the previous chapter the conceptual approach to the realisation of the PSS-IF Proof-of-Concept was described. In particular, the latter sections of Chapter 2 describe the principles behind the chosen transformation method. In this context, this chapter focuses on the implementation of the framework. The chapter is structured as follows: Section 3.1 describes the technology stack used for the implementation. In the consequent Section 3.2 the adopted software development guiding principles are presented. Thereafter, Section 3.3 provides an overview of the project structure. Section 3.4 follows with a description of each component of the PSS-IF PoC and, finally, Section 3.5 describes the import and export processes and the collaboration of all components of the framework.

3.1 Technology Stack

This section defines the technologies used for the implementation of the PSS-IF Proof-of-Concept. Each of the following subsections addresses a particular aspect of the technological stack. Note that all the software and tools used for the realization of the PoC are widely accepted industry standards.

3.1.1 Programming Language

To enable an easier and more rapid development of the prototype, a high-level programming language can better be utilized. Due to previous experience and know-how, the authors have chosen the Java Programming Language. Furthermore, the code is compliant with Java version 7, as distributed by Oracle.

3.1.2 Source Code Management

For improved parallelisation, better code maintenance and easier documentation, a distributed version control system can be used. While there are numerous alternatives, the authors have chosen GIT as a modern and powerful solution.

3.1.3 Build Process and Dependency Management

For the automated build process, as well as for the management of dependencies to external libraries, Apache Maven has been used.

3.1.4 Test Framework

For the execution of automated tests, JUnit, an industry standard framework within the Java universe, has been used.

3.1.5 Used Libraries

Next to the libraries provided by Oracle's Standard Java Runtime Environment, the following additional libraries have been used:

apache-compress An API for the manipulation of different kinds of compressed files. In the scope of the Proof-of-Concept, the API is used in the Visio VSDX processing component, for the zipping and unzipping of VSDX files.

guava Google Guava is a set of common libraries, mainly developed by Google. The package includes useful APIs for the manipulation of collections, the usage of function and predicates, and others.

emf The Eclipse Modelling Framework (EMF) library is used for the serialization and de-serialization of eCore and XML Metadata Interchange (XMI) files, enabling the processing of SysML4Mechatronics data.

junit JUnit is an industry standard unit-testing framework.

3.2 Guiding Principles

During the design and implementation of the PSS-IF PoC the authors have followed the principles and best-practices for software development. Some of the guiding principles were the following:

Standardization: Usage of widely used and accepted industry-standard tools, technologies and formats, to render the produced solution more accessible to new developers and compliant to other pieces of software.

Patterns: To maximize code quality and understandability, common architecture and design patterns have been utilized.

Object-Orientation: The code is developed in accordance with the paradigms of the Java programming language – it is mostly object-oriented and imperative.

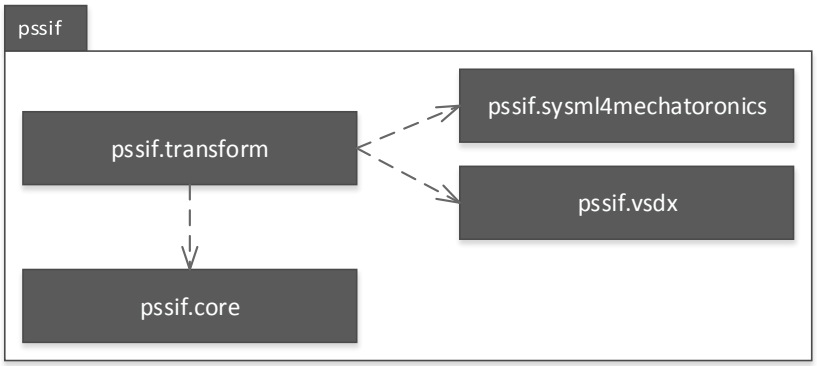

Figure 3.1: Project structure

Separation of Concerns: The implementation follows the separation of concerns paradigm.

3.3 Project Structure

The PSS-IF PoC is developed as an Apache Maven project, further divided into a root project, called "**pssif**" and a number of sub-modules. The root project is used for the provision of common development and build configuration, as for example the specification of common dependencies with a fixed version over all modules. This avoids redundancy and improves the manageability of the developed code. Each of the modules represents a component of the PSS-IF PoC architecture on a coarse level of abstraction. Note that while it is possible to establish a project with fine-grained Maven modularity, division in fine-grained modules would make it more difficult to capture the project structure. This is why the authors have tried to find a balance between capturing coarse-level architectural concepts through Maven modularization, while fine-grained modularization of components is achieved through the packaging mechanism of the Java programming language.

This section covers the coarse-grained separation realized through Maven modularization, while Section 3.4 is concerned with the fine-grained architectural modularization of the project. Currently, the PSS-IF PoC consists of four modules, as depicted in Figure 3.1.

Core The "**core**" Maven module contains the fundamental Application Programming Interface (API) of the framework. This API defines the concepts through which the PSS-IF Domain-Specific Language (DSL) is described, such as Metamodel, Model, NodeTypes, Nodes etc. Furthermore, the core module provides an implementation layer for the concepts of the PSS-IF DSL, as well as a number of common utilities, like for example

a generator for the canonic PSS-IF Metamodel, as depicted in Chapter 1.

Transform The "**transformation**" Maven module provides the Application Programming Interface (API) used for the definition and execution of transformations, as well as for input and output (I/O) operations. Next to the APIs, this module also contains their implementation, as well as a number of commonly used helping utilities, concerned with transformation or the serialization to or de-serialization from external formats. Finally, this module contains implementations for the supported source and target languages.

VSDX The "**vsdx**" module is a dedicated module which provides an API and an implementation for the processing of Microsoft Visio 2013 VSDX documents. The module defines an abstraction layer describing the structure of a Visio document in an object-oriented fashion, and is used for the serialization and de-serialization of VSDX files.

SysML4Mechatronics The "**sysml4mechatronics**" module contains a number of APIs and implementations for them, used for the serialization and de-serialization of SFB768 SysML4Mechatronics files.

3.4 Components

TODO verbatimize class names

After, in the previous section, the coarse-level division of the PSS-IF PoC parts through Maven modules was presented, this section focuses on a more detailed overview of the fine-level architectural components of the tool. Each of the following subsections describes the function and structure of a specific component of the transformation framework PoC.

3.4.1 Core

The core is the central component of the framework and is responsible for the realization of key concepts, used for the definition and processing of transformations. The core encloses two major concepts – those of a `Metamodel` and a `Model`, which were already presented in Chapter 2.

Metamodel

The `Metamodel` (see Figure 3.2) is a concept which enables the users of the framework to define the structure of their data. In this sense, a `Metamodel` also defines, on the abstract syntax level, the language in which the elements of a PSS-IF `Model` are described. A parallel in the domain of the Extensible Markup Language (XML) are the XML Schema Definitions (XSD). In essence, a PSS-IF `Metamodel` is the schema definition in accordance with which a particular `Model` is created and processed.

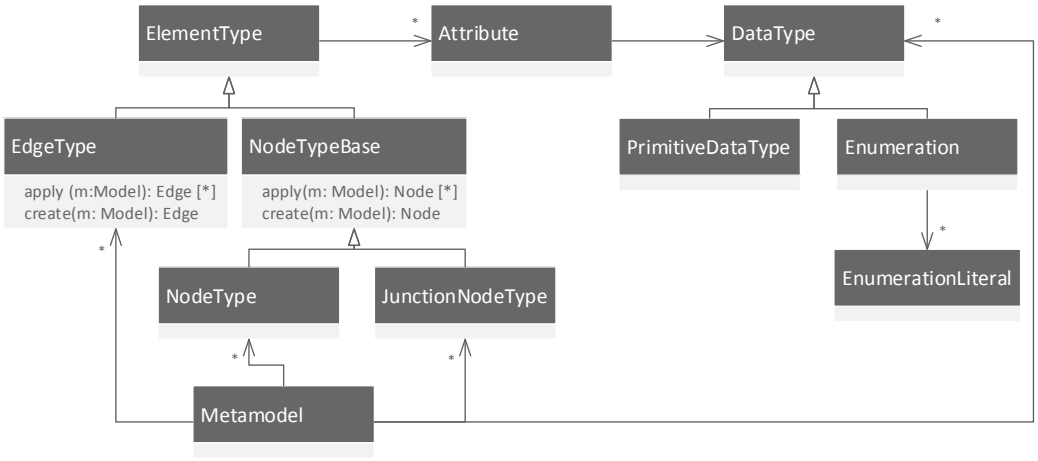

Figure 3.2: Metamodel API

A PSS-IF `Metamodel` captures a number of concepts. In particular, it consists of `NodeTypes` and `EdgeTypes`, i.e. elements which enable the user to define what kinds of nodes and edges their model may contain, what kinds of features they may have, and how they may relate to each other. Furthermore, in the PSS-IF PoC, inheritance relations can be defined between both `NodeType` and `EdgeType`.

Node Types `NodeTypes` are used for the description of the different kinds of nodes in a model and have a number of features. All `NodeTypes` are named and the name must be unique in the scope of all `NodeTypes` and `EdgeTypess` within a `Metamodel`. Furthermore, `NodeTypes` can have a number of attributes, and are connected to other `NodeTypes` over `EdgeTypes`, which can be both incoming and outgoing.

There are two categories of `NodeType` – conventional ones and `JunctionNodeTypes`. The latter actually describe nodes with edge semantics, i.e. they are used for the representation of hyper-edges.

Edge Types `EdgeTypes` are used for the description of possible edges in a user's model. `EdgeTypes` also have a name, which is unique in the scope of a `Metamodel`. For it to be possible to associate the same `EdgeType` with different pairs of `NodeType` instances, the concept of a `ConnectionMapping` is introduced. A `ConnectionMapping` is an association assigned to a particular `EdgeType`, which includes an incoming and an outgoing `NodeType`.

To illustrate the usage of the `ConnectionMapping`, consider the following example: Assuming two `NodeType` instances, denoted "State" and "Function" and an `EdgeType` "Control Flow", which has to connect both node types in both directions. In the PSS-IF PoC this is achieved by instantiating a single `EdgeType` "Control Flow" and assigning

two ConnectionMapping instances to it: one from "State" to "Function" and one from "Function" to "State".

Attributes For both NodeType and EdgeType, Attributes can be defined. Attributes are divided into AttribugteGroups, which can be used to separate different kinds of attributes conveniently, for example in a user interface. Furthermore, Attributes are identified by their names, which have to be unique in the scope of the owning node or edge type. Also, each attribute has a DataType. The PSS-IF PoC defines a number of primitive data types, like String, Integer, Date and Boolean, and also provides the user with the ability to define custom enumeration data types. Finally, Attribute instances can optionally have a Unit associated with them, which can be particularly useful for numeric attributes. Finally, attributes are divided into categories. Currently, the following categories are defined: Monetary, Weight, Density, Time, Geometry, MetaData and Material.

Inheritance As already noted above, for both NodeType and EdgeType, inheritance relations can be defined. When a NodeType inherits from another NodeType, it holds that attribute groups and attributes are inherited from the parent.

Built-In Metamodel Elements Every PSS-IF Metamodel has a number of predefined elements. These are the root node and edge types. The root NodeType has the name "**Node**" and provides the following predefined Attributes:

- **id**: An identifier of data type String for the node instance of the node type, categorized as Metadata.
- **name**: A name of data type String, categorized as Metadata.
- **validity start**: A Date, designating the begin of the validity period of the given node, categorized as Time.
- **validity end**: A Date, designating the end of the validity period of the given node, categorized as Time.
- **version**: The version of the node, of data type String, categorized as Metadata.
- **comment**: A comment of the node, of data type String, categorized as Metadata.

The root EdgeType "'**Edge**" has all built-in Attributes defined for the root NodeType "Node", and also an additional attribute **directed** of data type Boolean and categorized as Metadata.

The root node and edge types are also the roots of the inheritance hierarchies for NodeType and EdgeType within a Metamodel. In this sense, any NodeType or EdgeType instance automatically inherits from the root the respective root type. Thus, it is guaranteed that the set of attributes provided above is automatically defined for

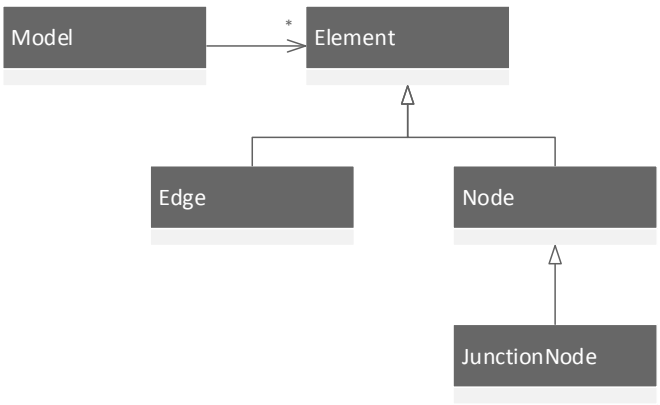

Figure 3.3: Model API

all instances of both `NodeType` and `EdgeType`. If a node or edge type inherits from a non-root node or edge type, the attributes are inherited transitively, together with all attributes of all ancestors throughout the generalization closure.

Model

The second key component of the PSS-IF PoC core is the `Model` (see Figure 3.3), which can be seen as a simple graph, consisting of `Node` and `Edge` instances. A `Node` is an ontological instance of a `NodeType` and an `Edge` is an ontological instance of the `EdgeType` of a PSS-IF `Metamodel`. Note that the elements of the `Model` are not type-aware themselves and can thus be accessed with different `Metamodel` instances.

In the PSS-IF PoC this strategy is used extensively, and a `Model` does not provide any information or allow any operations itself. All access points to a `Model` have to be navigated through a `Metamodel`, i.e. a `Metamodel` *operates* on a `Model`. This makes it possible to switch between metamodels as the user sees fit, but also to ensure data integrity at all times, by enforcing all modifications to be applied through the `Metamodel`, which plays the role of the schema for the data contained in the `Model`.

3.4.2 Transformations

Besides the core, the next most important component of the PSS-IF PoC is the transformation API. As already noted in Chapter 2, the authors have adopted the ISO 42010 approach, which defines different stakeholders as having different views on their data. Since the utility defines structures on the meta-level, the PoC describes the views of different stakeholders (source and destination languages) through corresponding Metamodels, called Viewpoints. These are mutable Metamodels, which can be *transformed*. For this purpose, Viewpoints accept a `Transformation`, which is a function mapping a Metamodel to a Metamodel.

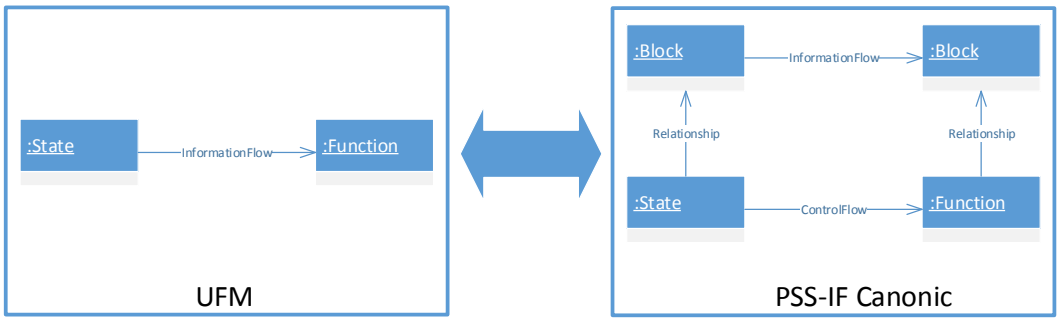

Figure 3.4: Example Transformation

Thus, the application of a `Transformation` results in a `Viewpoint`, i.e. a Metamodel, which modifies the behaviour of the input Metamodel in accordance with the particular kind of transformation used. As a result, transformations can be applied consequently, each one operating on the result of the previous. This process is illustrated in Figure 3.4. The same transformation is defined in Java as follows:

```
Metamodel viewpoint = PSSIFCanonicMetamodelCreator.create();

// create the artificial blocks
viewpoint = new CreateArtificialNodeTransformation
  (state, relationship, block).apply(viewpoint);
viewpoint = new CreateArtificialNodeTransformation
  (function, relationship, block).apply(viewpoint);

// join the informationflow to be created
// between the artificial blocks
viewpoint = new JoinConnectionMappingTransformation
  (informationflow, informationflow.getMapping(state, function),
  new JoinPath(relationship.getMapping(state, block)), block,
  new JoinPath(relationship.getMapping(function, block), block))
    .apply(viewpoint);

// create the artifial controlflow
viewpoint = new CreateArtificialEdgeTransformation
  (informationflow.getMapping(state, function),
  controlflow.getMapping(state, function)).apply(viewpoint);
```

Since the behaviour of the `Viewpoint` is independent of the particular kind of transformation used, the definition of a `Viewpoint` is reduced to the definition of all necessary transformations and their combination in the appropriate order. The current set of

available transformation covers all the ones described in Chapter 2.

3.4.3 Generic Graph

A simple component used for intermediate steps in the transformation process is the generic graph component of the PSS-IF PoC. This component is a simple undirected graph consisting of nodes and edges, which can have string-named attributes with string values, as well as the name of their assumed type. A UML Class Diagram describing the generic graph is depicted in Figure 3.5. In this sense, the graph is an untyped and unstructured equivalent of a PSS-IF Model.

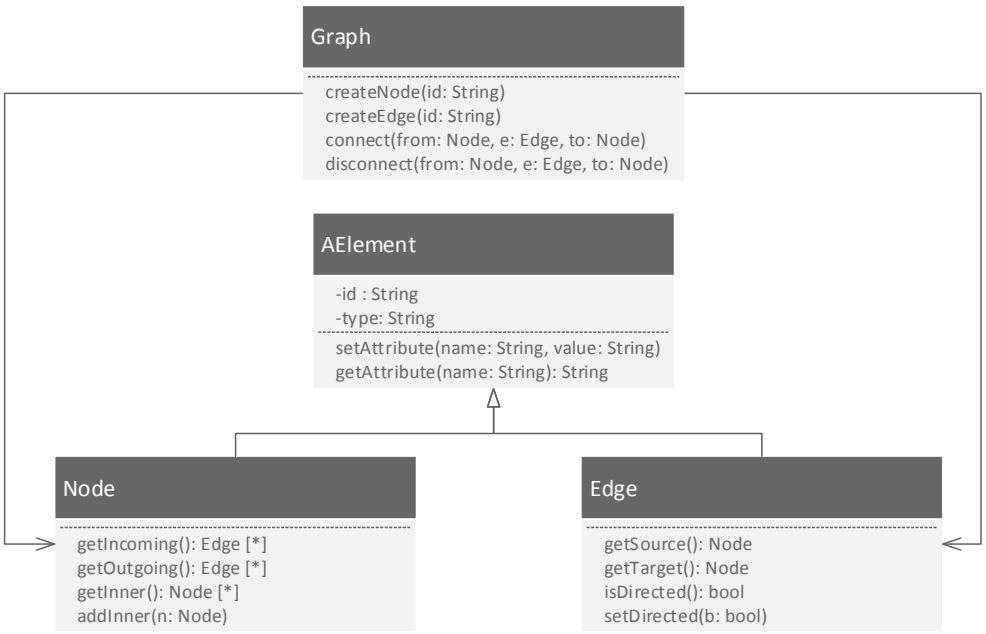

Figure 3.5: UML Class Diagram of the Generic Graph

In the transformation process, the generic graph is used as an intermediate format, separating the concrete syntax and the abstract syntax of each supported language. The concrete syntax is defined by the file provided by the user and is handled by an `IoMapper` (see Section 3.4.4), while the abstract syntax is defined by the `Viewpoint` and is processed by a `ModelMapper` (see Section 3.4.5). The details of the process are provided in Section 3.5.

3.4.4 I/O Mappers

The I/O Mapper component is responsible for the serialization and de-serialization of a generic graph to and from a stream. In this sense, the component has the task to

abstract over the concrete syntax used for the representation of the user's data in any particular language. The API of the component is the `IoMapper` interface, which has the following signature:

- `Graph read(InputStream in);`
- `void write(Graph graph, OutputStream out);`

Note that the same `IoMapper` may be used for more than one language. For example, the `VsdxIoMapper` could be used for both EPC and BPMN.

3.4.5 Model Mappers

The Model Mapper component is responsible for the translation between generic graphs and PSS-IF Models under the provision of a corresponding PSS-IF `Viewpoint` (Metamodel). In this sense, it handles the translation between the abstract syntax of the external representation and the abstract syntax of the language's viewpoint in PSS-IF. The component is realized through the `ModelMapper` interface, which has the following signature:

- `Model read(Metamodel metamodel, Graph graph);`
- `Graph write(Metamodel metamodel, Model model);`

In the simplest case, a particular model mapper would simply use the provided viewpoint to directly transfer information between a Model and a graph, processing all nodes, edges and attributes. In more complex cases, pre- or post-processing of the graph may be necessary, to "normalize" it into a structure compatible with the viewpoint defined for the particular language (and, hence, with the provided Model). This may, for example, be the case, if the external representation requires the nodes of a graph to be ordered in accordance with a certain rule, or if there are implicit existential dependencies between nodes, implied by the nature of the `Viewpoint` or a specific `Transformation`.

In general, it is assumed that one `ModelMapper` implementation is necessary for each format to support. Theoretically, if the same language is to be imported from or exported to more than one format, and the generic graph is sufficient to express the abstract syntax of both external formats, the same `ModelMapper` and `Viewpoint` can be used with different `IoMapper` implementations, to obtain different serializations of the same data.

3.4.6 Microsoft Visio VSDX I/O

For the serialization and de-serialization of Microsoft Visio 2013 VSDX files needed to support EPC and BPMN, a special component was developed in a dedicated Maven module. The component defines an API for the manipulation of VSDX files, operating with the following abstractions:

- `VsdxDocument` An object-oriented representation of an entire VSDX document.
- `VsdxPage` A page within a VSDX document.
- `VsdxMaster` A master shape defined within a VSDX document.
- `VsdxShape` A shape, contained either in a page, or in another shape.
- `VsdxConnector` A connector shape, i.e. a shape which connects two other shapes.

The inter-relation of the different concepts of the VSDX API is depicted in Figure 3.6.

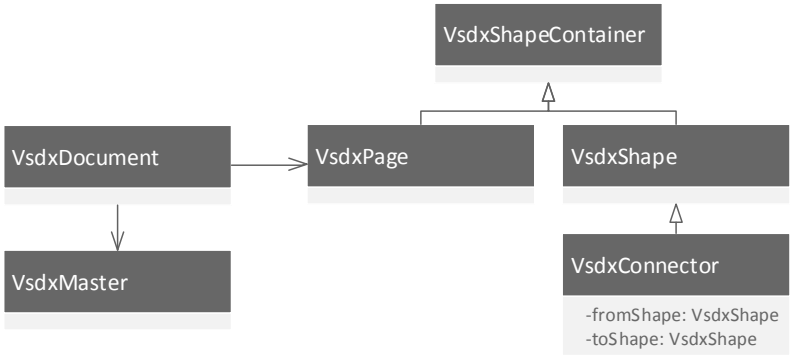

Figure 3.6: API of the VSDX Component

It should be noted that the current VSDX component does not provide any mechanisms for layouting, colouring, or other finer operations on a Visio file. This is because the component is aimed explicitly at the extraction and generation of data. The layout of Visio files exported from PSS-IF can be fixed through the usage of algorithms which are built-in in Microsoft Visio. If special decoration of shapes is required, the user's can provide their own template files with correspondingly modified master shapes. These template files can then be used in combination with the VSDX component to produce appropriately formatted Visio files. Finally, the API can be extended in future works, to support finer layouting and formatting operations.

3.5 Mapping Process

After, in the previous section, the different components of the PSS-IF PoC were presented, this section focuses on how the components hang together and describes the mapping (import and export) process in detail. Here, the term mapping is used, as it covers the processes in both directions and the PSS-IF PoC implementation handles both cases in the same fashion.

API The mapping processes are triggered over the corresponding API, represented by the `Mapper` interface, which has the following signature:

- `Model read(Metamodel metamodel, InputStream inputStream);`
- `void write(Metamodel metamodel, Model model, OutputStream outputStream);`

Implementation While each specific language, for each file format, requires its own `Mapper` implementation, the internal procedure for all languages and file formats is the same and is described by the `AbstractMapper` class, which is the super-class of all `Mapper` implementations. The `AbstractMapper` defines the methods of the `Mapper` API as follows:

```
public abstract class AbstractMapper implements Mapper {

 @Override
  public final Model read(Metamodel mm, InputStream in) {
    Graph graph = getIoMapper().read(in);
    Metamodel view = getView(mm);
    ModelMapper modelMapper = getModelMapper();
    return modelMapper.read(view, graph);
  }

  @Override
  public final void write(Metamodel mm, Model m, OutputStream out) {
    Metamodel view = getView(mm);
    ModelMapper modelMapper = getModelMapper();
    Graph graph = modelMapper.write(view, m);
    getIoMapper().write(graph, out);
  }

  protected abstract Metamodel getView(Metamodel metamodel);

  protected abstract ModelMapper getModelMapper();

  protected abstract IoMapper getIoMapper();
 }
```

As the code snippet demonstrates, a PSS-IF `Model` instance is obtained from an external representation by transcoding the external representation into a generic graph, obtaining the `Viewpoint` of the current language and finally mapping the generic graph to a model in accordance with the viewpoint.

Symmetrically, a file is generated from a PSS-IF `Model` by first obtaining the `Viewpoint` for the current language and using it to translate the model into a generic graph, and then serializing the graph to a stream through the corresponding `IoMapper`.

Process Recall the 6-step process described in Section 2.3.3, which describes the transformation process on the conceptual level. In the PSS-IF PoC, the transformation process is implemented through the consequent invocation of two mappers. The first one is the mapper which de-serializes the source data and transforms it into a Model conforming to the PSS-IF Canonic Metamodel. The second mapper is the one for the target format and is used to write the obtained model into an external representation. For an exemplary source language A and target language B, the Java code describing the transformation process is the following:

```
public void transform(Mapper aMapper, Mapper bMapper,
  InputStream inputStream, OutputStream outputStream) {

  Metamodel metamodel = PSSIFCanonicMetamodelCreator.create();
  Model model = aMapper.read(metamodel, inputStream);
  bMapper.write(metamodel, model, outputStream);
}
```

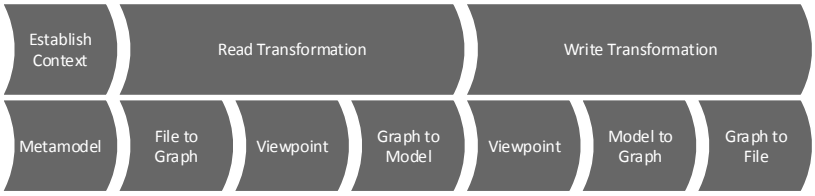

Figure 3.7: The Transformation Process of the PSS-IF PoC

In Figure 3.7, the transformation process is depicted in two levels of abstraction. The first level addresses the separation into two distinct read and write steps, while the second level demonstrates how both the read and write processes are accomplished by the respective language mappers.

Chapter 4

Results

After, in the last chapter, the implementation of the PSS-IF Proof-of-Concept was described, this chapter continues to present the results of the interdisciplinary project. In Section 4.1 the key features of the PSS-IF PoC, considered as a framework, are addressed. Section 4.2 proceeds with achieved results for each of the four domain-specific languages referenced in Chapter 1. Finally, in Section 4.3 a brief overview of the results is given.

4.1 PSS-IF Proof-of-Concept

Throughout the implementation, the authors follow the principles described in Chapter 2. Furthermore, the usage of industry-standard libraries and established software-development practices provide for a good long-term manageability of the resulting software. Finally, through the definition of small, simple, yet powerful APIs, the resulting tool can conveniently be used as a framework to build upon in other projects. The authors are convinced that the PSS-IF PoC fulfils the following quality features:

Generic The architecture of the implementation defines the transformation process on multiple levels of hierarchy, so that in most cases addition of new features can have a limited impact. Furthermore, divergence from the generic process on each level is possible through the usage of different implementations of the API on that level.

Extensible The clear separation of tasks between components and their collaboration only through well-defined APIs allows new features to be added to each component without intermediate effect on other parts of the software. Also, behind each API, implementations can be changed, or new ones can be added, to better suit the needs of the tool's users.

Flexible Changes in the PSS-IF Canonic Metamodel, or in one of the supported languages can easily be incorporated through adjustment of the metamodel definition and the corresponding viewpoints.

Expressive Through the adopted meta-modelling approach, the expressiveness of the resulting tool is comparable with that of the Meta-Object Facility (MOF), while at the same time being tailored to the set of modelling structures sufficient for the specific field of application.

Accessible Through the comparatively simple and well-defined APIs, the PSS-IF PoC can easily be used, once the concepts behind it have been clarified.

4.2 Supported Languages

Through the chosen implementation approach, the objective of transforming models between languages is reduced to the transformation from and to the language defined by the PSS-IF Canonic Metamodel with minimal information loss. The following sections provide the results for the four languages relevant for this work.

4.2.1 Flow-oriented Functional Modelling (UFM)

Models in the Flow-oriented Functional Modelling (UFM) language can be translated to PSS-IF canonic. The transferred information is restricted to States and Functions and the Flows between them, as well as the functionary attribute, which is used to forge artificial blocks, or dummy blocks, if no value for this attribute is provided. The original Flow between the States and Functions is then transferred to the artificial blocks, and a ControlFlow is created between the States and Functions. The creation of artificial blocks required the development of the artificialize transformation to create the artificial respectively dummy blocks out of the functionary attribute and the join transformation to transfer the original flow the additionally created blocks. Furthermore another artificialize transformation is required to create the additional ControlFlow between the States and Functions.

4.2.2 Event-Driven Process Chain (EPC)

Due to their structural similarity, EPC models can be translated into PSS-IF without much difficulty. In the case of this language, the key challenge was the development of an own object-oriented API for Microsoft Visio 2013, so that the models can be extracted from and written to VSDX files.

4.2.3 Business Process Modelling Notation (BPMN)

The objective to translate from and to BPMN models described in Microsoft Visio 2013 could not be completed. This is because the BPMN extension of Visio does not use the Visio graph strucuture for the encoding of data, but rather stores the BPMN-specific information into formulae of concrete and abstract nodes. Since the interdisciplinary project has a limited time horizon, the reverse-engineering of this kind of encoding was not possible.

4.2.4 Systems Modelling Language for Mechatronics (SysML4Mechatronics)

work in progress...
trouble: original xml format not clearly interpretable
trouble: bad planning on the side of the authors, complexity recognized at a late time
outcome: ecore + xmi solution, work in progress

4.3 Summary

In summary, the interdisciplinary project has resulted in the development of a powerful yet flexible approach to the task of transforming between models used by different disciplines in their collaboration in the scope of a Product-Service System. Also, for three of the four languages in the original objective, an implementation could be provided.

Chapter 5

Future Work

TODO

- brief summary of what is achieved
- future work: speculate

Appendix A

Distribution of Tasks

This appendix describes the distribution of tasks between the developers of this Proof of Concept (PoC) implementation of the PSS Integration Framework.

Research

The research necessary for the development of the PSS-IF PoC was done by both Bernhard Radke and Konstantin Govedarski.

Conceptual Development

The conceptual development of the PSS-IF PoC was done by both Bernhard Radke and Konstantin Govedarski. Both authors are in comparable knowledge of the concepts and technology behind each component of the PoC system.

Implementation

The implementation of the PSS-IF PoC was made by both Bernhard Radke and Konstantin Govedarski. The approximate distribution of specific tasks is provided below.

- **Utilities**: *Bernhard Radke* and *Konstantin Govedarski*
- **Core**:
  - **Metamodel**:
    - **Node and Edge Types**: *Bernhard Radke* and *Konstantin Govedarski*
    - **Connection Mappings**: *Bernhard Radke*
    - **Attributes**: *Konstantin Govedarski*
    - **Inheritance**: *Bernhard Radke*
  - **Model**: *Bernhard Radke* and *Konstantin Govedarski*

  - **Operational API**: *Bernhard Radke*
  - **Canonic Metamodel**: *Konstantin Govedarski*
  - **Canonic Persistence**: *Bernhard Radke*

- **Transformations**:
  - **Views**: *Bernhard Radke*
  - **Operators**: *Bernhard Radke*
  - **API**: *Konstantin Govedarski*
  - **Generic Graph**: *Konstantin Govedarski*

- **Languages and Serialization**:
  - **GraphML**: *Bernhard Radke*
  - **EPC (Visio)**: *Konstantin Govedarski*
  - **BPMN (Visio)**: *Konstantin Govedarski*
  - **SysML4Mechatronics**: *Konstantin Govedarski*

Documentation

The documentation of the PSS-IF PoC was created by both Bernhard Radke and Konstantin Govedarski. The documentation includes:

- This document.
- The documentation of the source code (JavaDocs).