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,

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 Computer Science 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.

TODO PSS-IF Metamodel pics and a bit more text.

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

    - Conversion-oriented Functional Modelling (UFP) Diagrams

    - Bringing Object-oriented Graph Grammars into Engineering (BOGGGIE) Diagrams

    - 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** concludes this documentation and 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-independent transformations would build upon a technology which is well-suited for processing the syntax of the source and destination domain-specific languages (DSLs). For ex maple, 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 user programming language. 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 already existing language. 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 filed, 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 expression 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 only inherited if no local mappings substitute the ones inherited from an ancestor type. Also, Edge Types define a Multiplicity. 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. Note that more-concrete Connection Mappings on lower levels in the Edge Type hierarchy hide this generic association in accordance with the substitution rule stated above.

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 is intended for the purpose of the transformations.

Viewpoints and Operators

An operator is a generic bijective transformation function, which conforms to a certain signature and can be applied, in accordance with its signature, to elements of a given metamodel. The result of the operator is once again a metamodel, which internally contains the operator and thus operates on a model in accordance with the semantics of the operator. The modified metamodel is called a viewpoint. Note that in this sense the PSS-IF Canonic Metamodel is the unit viewpoint of the resulting algebraic structure. Through the chaining of operators different viewpoints can be constructed, whereby the ordering of the operators can often play a role.

On the lighter side: How about Viewpoint-Fixpoint-Arithmetics? How far are we then from graph grammars?

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

TODO list operators, with definitions

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 in XML by 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), then they can be achieved directly on the basis of 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

TODO

- technology: java. maven... expplain why

- guiding principles of development

- project structure

- core

- transform

- vsdx

- viz?

Chapter 4

Results

TODO

- general evaluation of the resulting framework
- for each supported language, what worked, what did not

Chapter 5

Conclusion

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*

- **Serialization and Deserialization**:

  - **GraphML**: *Bernhard Radke*

  - **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).