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
  - Revenue-oriented Functional Modelling (UFP) Diagrams
  - Bringing Object-oriendted 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

TODO

- what are possible ways of providing
- advantages and disadvantages for them
- argue for metamodelling
- define pssif metamodel (abstract)
- define transformations (abstract)

here also: aim: maximize work done by the core, i.e. not just a graph, because then many things have to be done more than once, such as inheritance,type cheching, consistency verification etc.

Note: here: schema of Metamodel, i.e. node and edge types, attributes, inheritance (also w.r.t. features and some definition of the stuff), connection mappings, multiplicities... aka all the language rules

transformations: editable views

final statement: the pssif-canonic is only one instance of a metamodel

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).