FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

An Ontology-Based Approach for Transformer Generation in a Multi-Disciplinary Engineering Environment

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Pircher
Matrikelnummer 0327031

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao. Univ.-Prof. Dr. Stefan Biffl
Mitwirkung: Univ.-Ass. Dr. Richard Mordinyi

Wien, 23.08.2014

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

An Ontology-Based Approach for Transformer Generation in a Multi-Disciplinary Engineering Environment

MASTER'S THESIS

submitted in partial fulfillment of the requirement for the degree of

Diplom-Ingenieur/in

in

Software Engineering & Internet Computing

by

Michael Pircher
Registration Number 0327031

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao. Univ.-Prof. Dr. Stefan Biffl
Assistance: Univ.-Ass. Dr. Richard Mordinyi

Vienna, 23.08.2014

(Signature oft he Author) (Signature oft he Advisor)

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

_____	_____
(Ort, Datum)	(Unterschrift Verfasser/in)

Danksagung

“It is good to have an end to journey toward; but it is the journey that matters, in the end.” by Ernest Hemingway

Nun ist es geschafft. Die fertige Diplomarbeit liegt vor mir. Es war eine aufregende, anstrengende, nervenzehrende und lehrreiche Zeit, die ich ohne die Unterstützung von liebenswerten Menschen nicht geschafft hätte.

Allen voran meine Frau Christine, die immer an mich geglaubt hat, mich trotz meiner Launen motiviert und niemals das Handtuch weder nach mir noch generell geworfen hat. Danke, ich liebe dich.

Ein Dankeschön an meinem Professor Stefan Biffl und Richard Mordinyi, die mir beide mit Rat und Tat zu Seite standen und mir die Zeit gegeben haben, die ich auf Grund meiner beruflichen Verpflichtungen benötigt habe.

Darüber hinaus möchte ich mich auch bei meiner Familie bedanken, meinen Eltern, meinen Schwiegereltern, Großeltern, Geschwistern, Schwägerin und Schwägern, Tanten und Onkels. Ohne eure Unterstützung und den Glauben an mich wäre das Studium nicht möglich gewesen.

Am Ende möchte ich mich noch bei meinen Freunden bedanken: Lenz, Tom und Ivana, Alex, Martin, Katharina und Max, Lena und Bobby, Tanja und Jürgen, Lukas, Michael, Markus, Markus, Andreas, Joachim, Julia, Christine und Heli, Kathi und alle dich ich vergessen habe. Ihr habt unzählbare Stunden mit mir über den durch die Diplomarbeit in Frage gestellten Sinn des Lebens diskutiert - mit dem Wissen, dass es keine Antwort gibt.

“All was well.“ by J.K.Rowling

Abstract

Modern industries such as energy, automotive, or chemical accommodate a multiplicity of different engineering areas. To satisfy the fast changing requirements of the market, the different engineering areas need to work together within multi-disciplinary engineering projects. This requires the exchange of information between different tools of each area to enable the collaboration with a maximum of a synergy effects. Each tool has a variety of data models which were originally defined with their domain-specific terms and notations. This results in a scenario with the need of semantic integration which deals with the integration of heterogeneous data.

A generic approach to face the challenge of semantic integration of engineering data is the *Engineering Knowledge Base* (EKB). The EKB stores explicit knowledge about engineering plans in different domains to support the access and the management of models and tools across different engineering areas. The main features are data integration based on mappings between local tools on a domain level, transformation between local tools and the domain level, and advanced applications built on these foundations. The creation of a transformation is complex because the engineers need to develop transformation logic as e.g. source code and need to understand the stored knowledge belonging to the tools.

This master thesis proposes the TransformerIDE – a solution to support the engineers in their task of creating quality-assured transformations based on the knowledge stored within the EKB. It offers an integrated development environment (IDE) with functions to create, modify, validate, manage and export transformations. The core of the TransformerIDE is the procedural *Transformation Language* (TL) based on the *Transformation Meta Language* (TML). This language is tailored to the requirements of the use case and can be easily extended. It consists of simple functions, e.g., *trim, concat*, with clearly defined input and output. By combining this functions complex transformations can be created. On top of this language the IDE consists of an Editor, Validator, Code Generator, and Transformer Manager. The editor lifts TL from a pure textual structure to a graphical structure with validation, execution, and export functions.

The evaluation of the TransformerIDE with the underlying *Transformation Meta Language* has shown that the automated creation and validation of the Transformer and the Transformations supports the user in reducing the effort and error rate of transformation design and operation processes.

Kurzfassung

Moderne Industriebereiche wie Energie, Automobil und Chemie vereinen eine Vielzahl von verschiedenen Ingenieurs Disziplinen. Um auf die sich ständig verändernden Anforderungen des Marktes zu reagieren, ist es notwendig, dass diese Disziplinen interagieren. Diese Zusammenarbeit setzt den Austausch von Information voraus. Jedoch verfügt jedes System über unterschiedlichste Daten-Modelle, bereichsspezifischen Bedeutungen und Bezeichnungen. Das resultiert in die Notwendigkeit einer semantischen Integration von heterogenen Daten.

Die Engineering Knowledge Base (EKB) repräsentiert einen generischen Lösungs-Ansatz für die semantische Integration. Sie speichert das explizite Wissen der verschiedenen Disziplinen, Systeme und Modelle. Die Hauptfunktion hierbei ist die Daten-Integration anhand der Verknüpfung zwischen den Modellen und dem gespeicherten Wissen. Außerdem fungiert die EKB als Grundlage für fortgeschrittene Anwendungen wie z.B. End-to-End Analysen oder Transformer Erstellung. Um einen solchen Transformer erstellen zu können, muss der Ingenieur die Logik des Transformers in Form von Quell-Codes und aufbauend auf dem gespeicherten Wissen der EKB erstellen.

Diese Arbeit stellt die TransformerIDE vor, welche die Tool unterstütze Erstellung von Transformer mit Hilfe der EKB ermöglicht. Die TransformerIDE verfügt über eine integrierte Entwicklungsumgebung, welche über Funktionen wie das Erstellen, Ändern, Validieren, Verwalten und Exportieren von Transformern verfügt. Der Kern der Arbeit ist die Transformation Meta Language (TML) und die darauf aufbauende Transformer Language (TL). Diese Transformer Sprache ist angepasst an die Anforderungen des Anwendungsfalls. Sie verfügt über einfache Funktionen wie z.B. „trim“, „concat“ und einer klaren Struktur mit vordefinierten Ein- und Ausgabewerten. Die Struktur der Sprache ermöglicht die wahllosen Kombinationen der zur Verfügung stehenden Funktionen, was das Erstellen von komplexen Transformationen ermöglicht.

Die Evaluierung der TransformerIDE hat gezeigt, dass das Tool unterstütze Erstellen von Transformern, aufbauend auf dem gespeicherten Wissen der EKB, den Aufwand und die Fehlerrate einer Transformer-Erstellung reduziert.

Contents

CHAPTER 1

Introduction

This master thesis is written in the context of multi-disciplinary engineering projects where systems strongly depend on distributed software solutions, which regulate and execute automated processes [1]. To develop a distributed system, different quality criteria such as predictability and testability must be met in order to guarantee that cost, risk and safety are kept within acceptable ranges. Beside the necessity to comply with the quality criteria, the industrial environment demands systems that are highly interoperable, flexible and responsive to the changing needs of the customers in terms of product variety, manufacturing agility, and low cost [1]. One of the leading fields of multi-disciplinary engineering is mechatronics, where the three engineering disciplines mechanics, electronics, and software are combined to achieve collaboratively an improvement within product performance and flexibility [2].

Typically, a distributed system and its components are developed from scratch. Rather, the different parts of such a system have their own individual history and evolution in which the requirements and goals have often changed multiple times. The underlying structure and data are based on domain- specific terms and notations. Therefore, the collaboration between the components is confronted with challenges such as “matching ontologies or schemas, detecting duplicate tuples, reconciling inconsistent data values, modeling complex relations between concepts in different sources, and reasoning with semantic mappings ”[3]. In order to deal with those challenges, integration areas with multiple engineering disciplines need to support new modelling formalisms to integrate

model transformations, new code generation techniques and the elaboration of formal analysis techniques to react on changes[2].

Halevy et al. [4] define semantic integration as the “solving of problems originating from the intent to share data across disparate and semantically heterogeneous data ”. This nature of semantic integration to share information indicates its importance for the discipline of ontologies. The following two statements emphasize this significance: According to Fensel [5], ontologies offer “ a shared and common understanding of a domain that can be communicated between people and application systems ”. In line with that, Ushold et al. [6] added that ontologies are used for interoperability, search and software specification.

This attempt to share information confronts the discipline with semantic heterogeneity. Jasper et al. [7] define different application scenarios for ontologies. One scenario which plays an important role for this master thesis is “Common Access to Information”. It describes a scenario where different systems need to exchange information among one another. Each of those systems disposes of different vocabularies and formats. To enable a bidirectional communication between these systems the scenario describes an interchange format as ontologies. The inter-operability and a more effective use and reuse of knowledge resources are supported.

All the different challenges caused by the purpose of sharing heterogenic data produced a variety of research areas. Noy et al. [8] consolidate them while taking into account the main task of the semantic integration:

- Mapping discovery: The matching of ontologies, also called ontology alignment, deals with finding similarities between two ontologies.
- Declarative formal representation of mappings: It refers to how the mapping between two ontologies is represented to enable reasoning.
- Reasoning with mappings: This research area involves the challenges of using mappings for further tasks such as data transformation or query answering.

A reasonable integration solution must argue with all of the listed research areas and their challenges. Different approaches, such as the common repository [9], were proposed in the last decades. Common repository is a metadata manager which stores data from engineering artifacts within a central repository. Engineering artifacts are shared objects such as information about software, documents, and information systems. They solve the problems of persistency and versioning of engineering data.

Based on the common repository, Moser et al. [1] propose the Engineering Knowledge Base (EKB), “ a framework for supporting engineering environment integration in multi-disciplinary engineering projects ” [1]. The EKB offers the following solution approaches for sharing and managing of models across disciplines:

- Data integration based on mappings between local and common engineering concepts.
- Transformations based on mappings between the engineering concepts.
- Advanced applications based on these foundations.

As a framework, the EKB offers architecture with an underlying supported process. The process consists of five steps which form the different parts of the architecture. The process starts with the extraction of tool data from the different tools as key value pairs. The second step stores the extracted data in the Engineering Data Base (EDB). This data base is like a common repository with features such as versioning and rollback of changes. The next step consists in describing the format, the meaning and the use of the tool and domain knowledge as ontologies. The domain knowledge is the collaborative view of the domain and is named as virtual model. Within step four, tool models and the virtual models are mapped based on model description from step three. This mapping consists in the assignment of attributes from one model to another. The last step describes the usage of the Engineering Knowledge Base.

Typical use cases are consistency check across tool boundaries and change impact analyses. Important for this use cases are transformations between the

tool models and the virtual model. Such transformations are created based on the mappings and the stored information. Kleppe et al. [10] define the model transformation as the ”automatic generation of a target model from a source model, according to a transformation definition”.

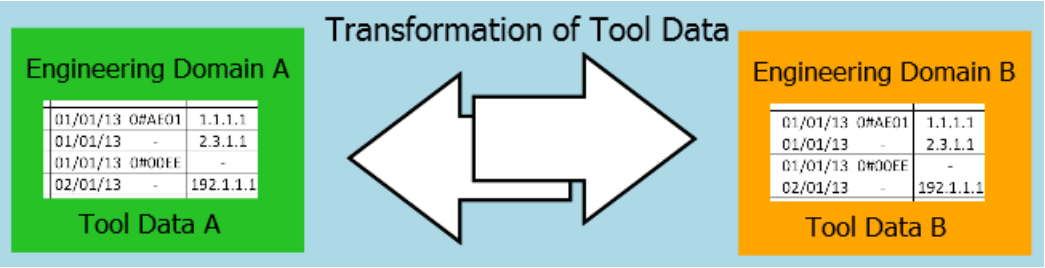

Figure 1-1 Motivation Scenario

The creation of a fully functional transformation is a complex and costly task which requires the involvement of different experts from the domain, tool or ontology area. The scientific community proposes different approaches for model transformation such as: direct model manipulation, intermediate representation and transformation language support (Sendal et al. [11]). The direct model manipulation uses a general-purpose language to develop a transformer. This means that a software developer or an engineer with development skills is able to create a Transformer. The disadvantage is that for the creation of a Transformer no tool support, beside the standard IDEs, is offered. The intermediate representation uses a standard format as intermediate model before it is transformed into the target model. Such approaches offer tool support with the drawback that cross model integrity cannot be guaranteed. The last approach is the transformation language support where a transformation language which is tailored to the problem area is used.

This master thesis proposes the TransformerIDE. A solution for the creation and management of transformers between data models of different engineering concepts. The two transformation approaches intermediate representation and transformation language support are used as basis. This mix enables the use of the benefits of both approaches such as the tailored Transformer language and the tool support as an IDE. To enrich the development process, the TransformerIDE uses the EKB as a knowledge base. Current solution

approaches such as the Atlas Transformation Language (ATL) focuses on the Transformer creation process itself and not what skills are necessary to develop such a Transformer. The TransformerIDE tries to satisfy also this requirement by reducing the Transformer creation process to a simple development task which requires no specific domain knowledge.

Within six sections, the TransformerIDE as a solution is approached step by step. The first section highlights the related work with topics such as semantic integration, ontology based semantic integration, the Engineering Knowledge base, Model Driven Architecture, Model Transformation and ontology modeling. They represent the theoretical basis for the TransformerIDE. Section three describes the research approach which consists of the description of the problem scenario, the use cases derived from the hydro power plants construction application area, the research method and the three research issues. Research issue RI-1 consists of a use case analysis and the elicitation of the **requirements** derived from the integration scenario of hydro power plant construction industry. The collected requirements build the basis for the design process of the TransformerIDE. Research issue RI-2 evaluates the **effectiveness and efficiency** of the TransformerIDE. The final research issue RI-3 evaluates the **learnability and understandability** of the created Transformations. After the research approach, section four highlights the scope, environment, stakeholder, requirements, architecture and the implementation of the TransformerIDE. Section five covers the evaluation of the TransformerIDE and the Transformer Meta language against the four quality criteria: effectiveness, efficiency, understandability and learnability. Based on the result of the evaluation, the three research issues are discussed. The last section summarizes the master thesis and the results of the evaluation and discussion and gives an insight into possible future works.

CHAPTER 2

Related work

In this chapter, a summary of the related work building the framework of the master thesis is given. In the first section, the research area semantic integration is described and the Ontology Based semantic integration, the most popular approach, is highlighted. The objective of the second section is to introduce the generic approach of semantic integration called Engineering Knowledge Base Framework. The framework builds the basis on which the proposed approach of this master thesis lies. The last sections Model-driven Architecture (MDA) and Model Transformation highlight the idea of MDA.

2.1 Semantic Integration and Heterogeneity

> "Semantic Integration is defined as the solving of problems originating from the intent to share data across disparate and semantically heterogeneous data"[12]

The growing need of integration from (enterprise) information systems within or across different areas demands a mix of data from different data sources. Typical sources are not only structured data sources like relational databases but also semi-structured data sources such as XML, HTML etc. Thus semantic integration is confronted with heterogeneous data. Hakimpour et al. [13] distinguish between two types of heterogeneity:

- Data heterogeneity:

"Data heterogeneity refers to differences among local definitions, such as attribute types, formats or precision."

- Semantic heterogeneity:

  “Semantic heterogeneity refers to differences or similarities in the meaning of local data.”

Contrary to data heterogeneity, which is a technical problem, semantic heterogeneity plays a central role for the research area semantic integration. Noy 2004[14] says that semantic heterogeneity " occurs on the large-scale, involving terminology, structure, and context of the involved sources, with respect to geographical, organizational and functional aspects related to information use". Halevy 2005 [4] describes it in a more broadly way. He says that if there is more than one way to structure data semantic, heterogeneity will appear. Therefore, we speak about data that was developed in a subject-specific and custom-designed way, from different user groups. The process schema matching consists in deciding if two elements from different database schemas match with the same real word concept[15].

According to Doan et al. (2005)[15], the 3 main challenges for this process are unknown semantics of the involved schemas, incomplete schema and data clues and the problem to decide if an element u of a schema U matches and element y of a schema Y. A lot of decisions need to be made during this process which, as a whole, cannot be made automatically because of the complexity of the decisions. To support this process, there exist rule-based and learning-based matching techniques. Rule-based techniques work on schema information like schema/element names, data types etc. They are inexpensive and quite fast because it is simple to create rules and they need no training but they ignore the data instances although they contain a lot of useful information. On the other hand, learning-based techniques take care of data information. The main concept is to match schemas on attribute specifications and statistics of data content[15].Semantic integration plays an important role not only for the database community but also for other disciplines like information-integration and ontologies [3].

Ontology Based Semantic Integration

The scientific discipline of Artificial Intelligence (AI) and the research area of Semantic Web have identified ontologies as a useful tool to formalize and

represent knowledge. According to Obitkio et al. [16], “ontologies play important role in knowledge sharing, reuse, and communication” due to the semantic interoperability which means “a possibility to understand shared data, information and knowledge”.

Gruber et al. [17] define an ontology as “a formal, explicit specification of a shared conceptualization ”A conceptualization represents an abstract model which focuses on a particular view of the real world. Szulman et al. [18] describe the usage of an ontology as the intent to assign a vocabulary of names to things from a domain of interest and to specify the meaning of the names as some logic. One of the main targets is with reference to Uschold et al.[6] the taxonomy of classes.

An ontology consists of classes, properties and their relations. A class represents a real world object e.g. a Signal, Tool etc. The properties which each class holds characterize attributes such as restrictions and features. Complex connections and the interaction of classes can be visualized by using relations. A combination of an ontology and a set of individuals incorporate a knowledge base.

The nature of an ontology is to represent and to share knowledge in such a way that a multiplicity of systems, stakeholders etc. can understand it and process it. Fensel [5] emphasizes that by saying that an ontology offers “a shared and common understanding of a domain that can be communicated between people and application systems”. This need implies the confrontation with semantic heterogeneity because of the different sources of knowledge from different environments. Therefore, it is hardly surprising that semantic integration is one of the key challenges of a feasible ontology.

Jasper et al. [7] define different application scenarios for ontologies such as Neutral Authoring, Ontology as Specification, Ontology-Based Search and Common Access to Information. Within this master thesis, the focus lies on Common Access to Information. It describes a scenario where different systems with unequal vocabularies and formats need to exchange information among one

another. The ontology represents an interchange format which enables a bidirectional communication between the communication partners. This scenario promotes a more effective use and reuse of knowledge get.

To formulate an ontology the World Wide Web Consortium (W3C) [19] has specified the Web Ontology Language (OWL). OWL is based primarily on the Resource Description Framework (RDF). Based on the operational area, OWL offers three sublanguages which are graded by their expressiveness. RDF builds the base. The next level is OWL Lite which adds a classification hierarchy with and simple constraints such as simple cardinality. OWL DL (description language) offers a maximum expressiveness while retaining computational completeness and decidability. The last level is OWL Full which offers a maximum expressiveness. The drawback is that no computational quarantines can be made. Beside OWL, different methodological framework are available to create reusable and usable ontologies. Such frameworks are DOGMA (Lastra et al.[20]) and MASON (Lemaignan et al. [21]).

Noy et al. [8] define 3 semantic integration tasks which are common also for other areas which are dealing with semantic heterogeneity and semantic integration:

- Mapping discovery: The mapping of ontologies, also called ontology alignment, deals with the finding of similarities between two ontologies.
- Declarative formal representations of mappings: It refers to how the mapping between two ontologies is represented to enable reasoning.
- Reasoning with mappings: This research area involves the challenges of using mappings for further tasks such as data transformation or query answering.

An example of semantic integration based on an ontology is the approach of Adams et al. [22] who use an ontology to represent heterogeneous information to answer complex queries from the aeronautical industry. Within the automotive

industry, Hefke et al. [23] use an ontology reference model to integrate data from heterogeneous disciplines.

2.2 Engineering Knowledge Base Framework

The Engineering Knowledge Base (EKB) is a generic approach of semantic integration defined as "an ontology-based data modeling approach which supports explicit modeling of existing knowledge in machine-understandable syntax."[12]. The motivation of such a generic approach comes from the ambition to improve the interoperability and flexibility of distributed software systems within modern industrial automation systems. Those software systems make sure that the operational sequences of the different areas of responsibility execute and behave correctly. The different belonging engineering/expert domains have a strong desire to share their data models and knowledge to improve their quality and safety. Typically, a strong and, of course, diverse evolution stands behind each software system which causes different structures, terminologies and processes of the same data. Therefore, it is quite complex and complicated to share information regarding the permanent changes that each system must undergo.

The EKB targets the following topics and challenges related to the described scenario of distributed software systems within modern industrial automation systems: semantic model integration between expert domains, semantic integration of expert knowledge, data model translation/transformation, managing of heterogeneous knowledge. Based on approaches from recent studies like the common repository [9] or the approach to provide engineering knowledge in machine-understandable syntax [24], the EKB identifies their weakness and optimizes them for the field of operation. The focus of the EKB is on the following issues:

- Providing links between data structures of engineering tools and systems.
- Support the exchange of information between engineering tools.
- Making systems engineering more efficient and flexible.

The Architecture of the Engineering Knowledge Base

Figure 2-1 illustrates, in terms of an application scenario in a simple and easy understandable way, the architecture of the Engineering Knowledge Base.

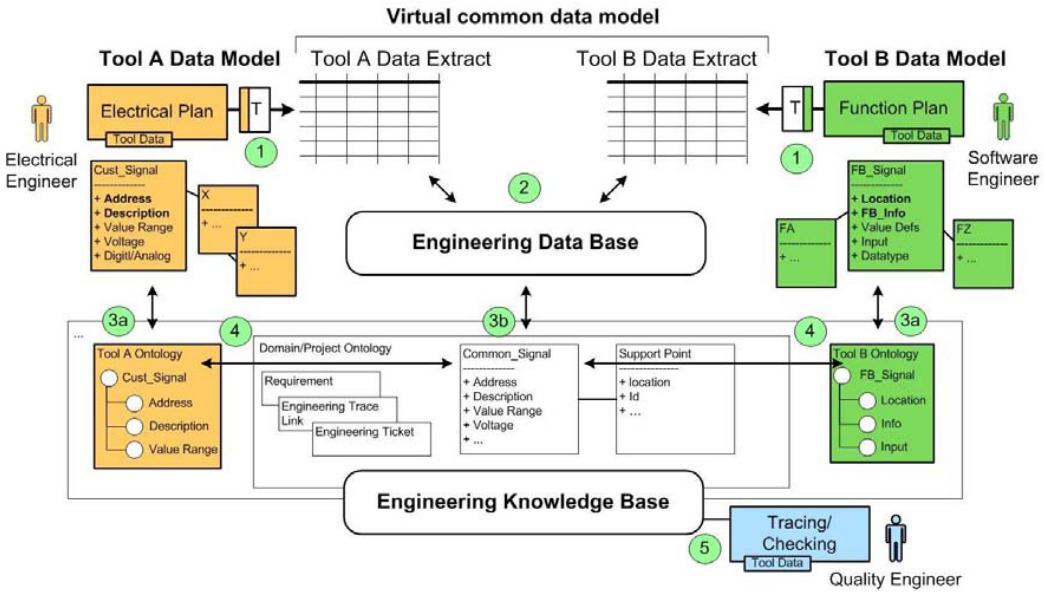

Figure 2-1 Application scenario of the Engineering Knowledge Base [1]

On the left and on the right side of the use case scenario, two engineering areas, the software engineering and electrical engineering areas, are displayed. Of course, there can be more then only two areas involved, such as mechanical engineering etc. Each area consists of one or more tools represented by tool-specific engineering knowledge, tool-specific data models and mappings between the tool data model and the virtual common data model.

During the EKB preparation process illustrated in Figure 2-2, the implicit knowledge is determined. The process starts with identifying the overlapping common engineering concepts. This includes the comparison of the tool concepts from the different engineering areas and the identification of the correspondent identical (common) concepts. Step two and three of the process consists of the description of the common concepts and the tool-specific concepts as ontologies. During the last step, the mapping between the tool-specific concepts and the overlapping common concepts is performed. The mapping is the most complex

part of the process which T. Moser [12] confirms by saying “ the identification of complex mappings is directly related to the research area of Ontology Alignment”.

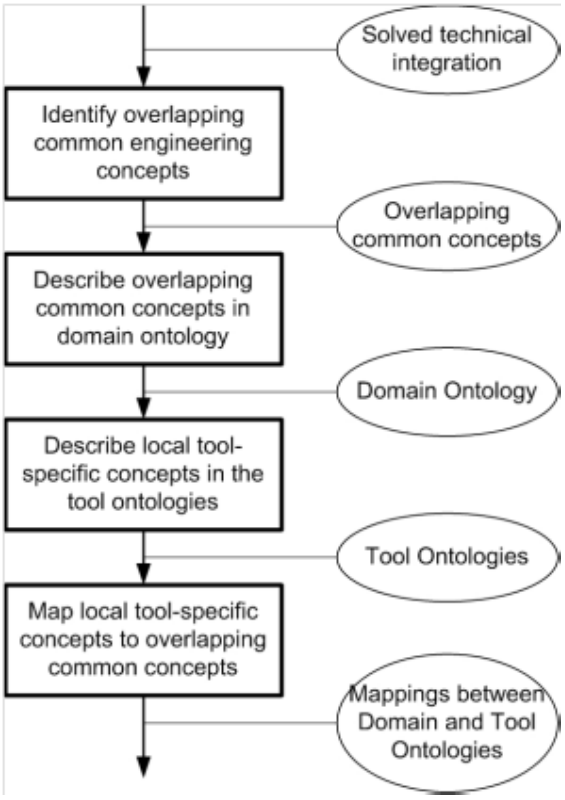

Figure 2-2 Preparation of the Engineering Environment [12]

Step one of the use case scenario is called "Extraction of Tool Data". This step handles the export of the tool data in the form of key value pairs so that the EKB framework can process the data from the different tool sources. As a rule, the tool data is not available in the correct formatting. Therefore, the "Extraction of Tool Data" step consists not only of the export of tool data but also of the parsing and the transformation of the tool data into the correct and readable EKB format. Step 2 consists of storing the extracted tool data. Such a storage component is called Engineering Data Base (EDB) and offers features like versioning or roll-back.

The next step is divided into two. Part 3a specifies the description of the tool knowledge. It refers to the format, the meaning and the use of the tool information. Part 3b comprises the domain knowledge from the production automation domain. It consists of information about the collaborative view of the

domain. The model of this domain is called virtual common data model. The mapping of the tool knowledge of part 3a to the domain knowledge of the part 3b is illustrated with step 4. Each data structure segment of the tool ontology is mapped to a corresponding domain attribute. This mapping works of course in both directions. When we take a look at all mappings, we can see the similarity between all domain tools. The chosen structure of the stored information is an ontology. Step 5 is a generic point describing the usage of the Engineering Knowledge Base.

The ontology-based mappings and domain descriptions offer the possibility to transform tool data model into the common data model and vice versa. The mapping defines which attributes are connected and the description of the attribute like data type, format etc. specifies how the result of the transformation should look like. A transformation can be created as an executable language like java or in a more generic way with an own model transformation language. Based on this transformation, more complex application can be realized like consistency check across tool boundaries, change impact analyses etc. This master thesis proposes a simple way to create Transformers based on the stored information within the Engineering Knowledge Base to enable such advanced applications.

The Engineering Knowledge Base Integration Process

The Semantically-enabled Integration Process introduced by Moser et al. [25] and showed in Figure 2-3 introduces an approach to “make expert knowledge explicit to facilitate tool support”[25].

Involved roles of the integration process [25] :

- Subject Matter Expert: The SME has the needed knowledge about the engineering tool to be integrated.
- Domain Expert: The DE is responsible for management of the problem domain.
- Network Administrator: The NA describes the architecture and capabilities of the underlying network infrastructure.
- Integration Expert: The IE selects suitable integration partners.

The approach consists of 6 process steps. The first step, Legacy System Description, is the task to describe the requirements and capabilities of the engineering tools by the SME (Engineers) as ontology. In the next step, Domain Knowledge Description, the DE describes the common knowledge of the problem domain and the NA the architecture and the capability of the underlying network infrastructure. As before, the outcome will be an ontology. Then, in the third step, Model Quality (QA) uses ontology-based reasoning techniques.

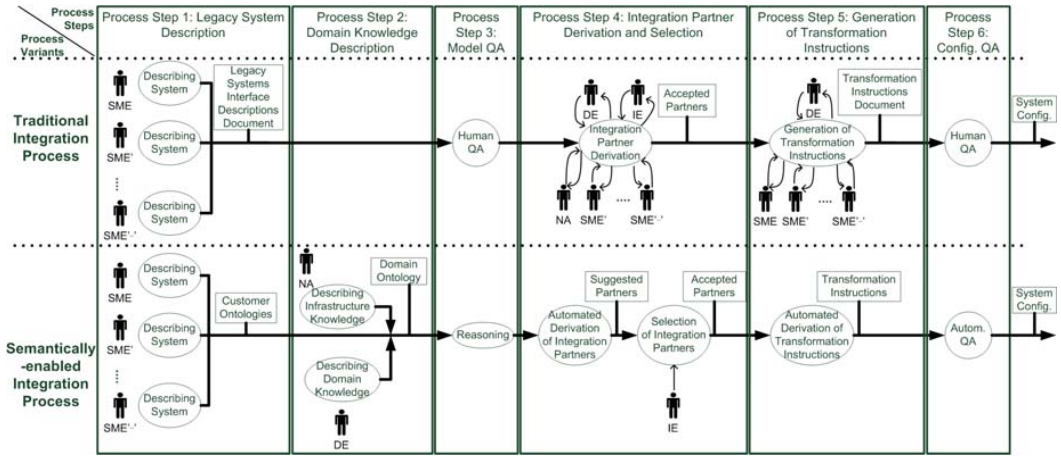

Figure 2-3 The traditional and the sematically-enabled integration proccess used by the EKB [25]

In the fourth step called Integration Partner (IP) Derivation and Selection, the previously determined knowledge is used to derive possible Integration Partners. The IE chooses suitable IPs which will be used in the further steps. Process step 5, Generation of Transformation Instructions, is composed of the derivation of the transformation instructions between the IPs. Step 5 is described by Moser et al. [25] as automated process; in reverse, the TransformerIDE offers a tool-supported solution to create the transformations. In the last, step Config. QA, the determined transformation inductions and the knowledge from process step 1 and 2 are compared.

2.3 Model-Driven Architecture

The Model-Driven Architecture (MDA) was introduced by the Object Management Group (OMG) as an approach of the software development methodology Model Driven Engineering (MDE). MDE, as Kent [26] says, has a wider scope then MDA. Favre et al.[27] confirm this characteristic by saying that MDE is a more general approach including but not limited to MDA.

The Model-Driven Architecture combines a set of standards which are based, according to Mellor [28], “on the idea that modeling is a better foundation for developing and maintaining systems”. Beside the standards defined within the bundle MDA contains also different technologies and techniques. The best-known are the Unified Modeling Language (UML) or the Meta Object Facility (MOF) but also others such as Mapping functions or Marking functions are part of the approach.

In general, the “Model-Driven Architecture (MDA) is a framework based on the Unified Modeling Language (UML) and other industry standards for visualizing, storing and exchanging software designs and models ”[10]. To put it in a nutshell, “MDA is a framework for software development ”[10]. The central role is taken by models used to model a software system. The software development process of MDA is based on the standard development process, with the difference that MDA deals with platform-independent models instead of a platform-close implantation.

MDA consists of different types of models and tools. Models are the central key of the MDA framework and of the MDE in general. Following the description of Mellor et al. [28], a model represents a real thing but is much cheaper to build. It describes a physical, abstract or hypothetical reality. The MDA uses different types of models for different levels of abstraction. PIM represents a platform-independent model (PIM) which reflects only the problem area without the need to take care of any platform specific details. PSM in contrast describes a system with the full knowledge of the final target platform[10].

The benefit of MDA is a direct result of the platform-independent development. Kleppe et al. [10] summarize them as a revised productivity and portability. Productivity because of the new abstraction layer introduced with the use of PIM. This abstraction enables a focusing on the business problems without the challenges of a specific target platform. Therewith the needs of the end-user comes more into the spotlight. At the end, a PIM must be transformed into PSM. This is of course a complex task. At this point, all the platform-specific details must be considered. The good news is that this task must be done only once but, nevertheless, the complexity is high and should not be underestimated. With a working transformation from PIM to PSM, the second benefit also, portability, is enabled. As far as different PSMs for different target platforms exist, the system can be developed for different platforms based on one source model.

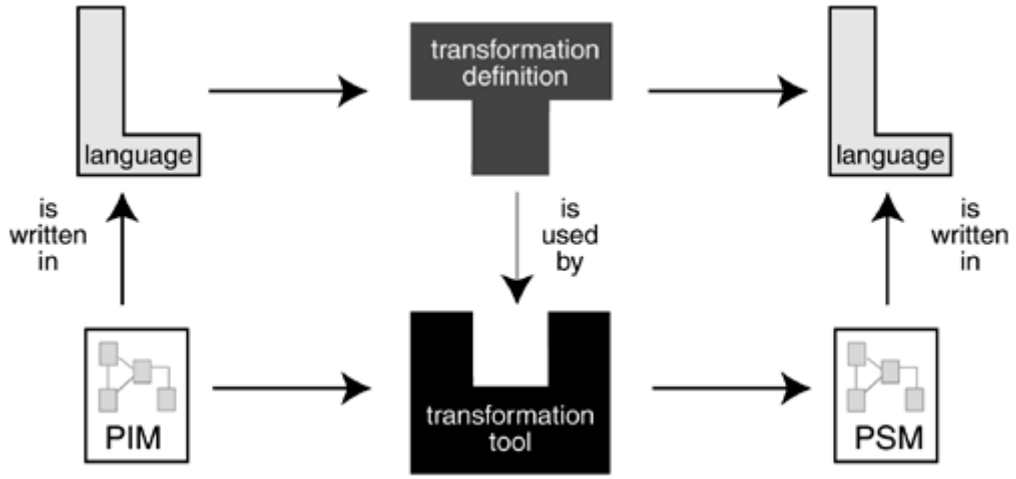

Figure 2-4 The MDA framework [10]

As visualized within Figure 2-4, both models are written in their own model language. The transformation tool understands both models and transforms with the help of transformation operations based on the transformation definitions, a type of dictionary, PIM into PSM. The sequence consists of 2 steps. The first step transforms the PIM model into the PSM model. Within the second step, the PSM model is finally transformed into a code e.g. Java. A working transformation is complex and complicated to achieve but, nevertheless, it is a core element of the MDA and is therefore highly critical.

A model language is used to develop a model such as PIM or PSM. This model language is defined by a metamodel which itself is written in a metalanguage. The mechanism to create a model language is called metamodeling.

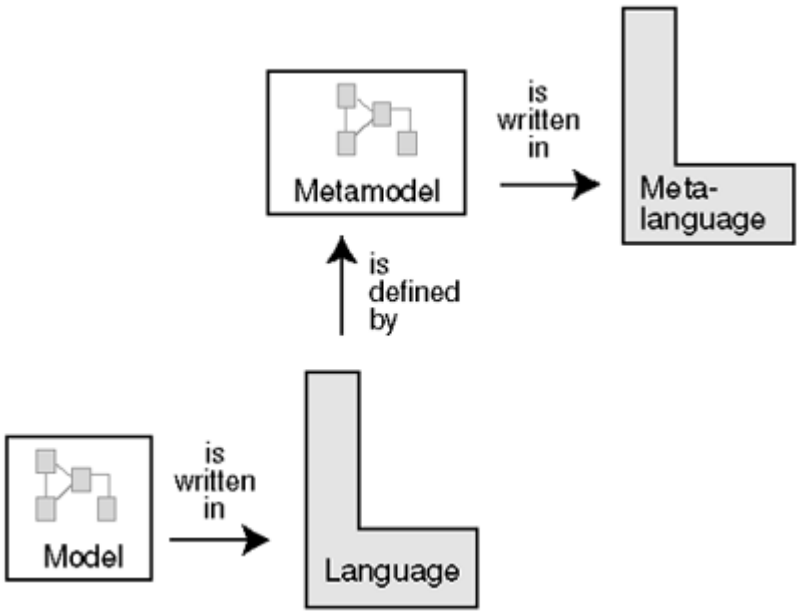

Figure 2-5 Metamodel and language relation [10]

According to Kleppe et al. [10], a model defines what elements a system can have and a language defines which elements can be used to define a model. For example UML, the Unified Modeling Language, is metamodel which can be used to define a model language. UML consists of a class, attributes and some other terms. To create a model house with windows, we need to create a class with the name “house” and the attribute “windows”. As UML is also a model, it is clear that there is another language with which it was defined. In the case of UML, the language is the Meta Object Facility (MOF). What is special about MOF, is that it is defined by itself.

2.4 Model Transformation

“A transformation is the automatic generation of a target model from a source model, according to a transformation definition ”[10]

The Model-Driven Architecture introduces the Model Transformation. Model transformation is important because of the need to transform models like PIM into

PSM. The benefits of the model-driven engineering are well known but hard to achieve. Within the area of model-driven engineering, model transformation represents the effort necessary to achieve a feasible MDA solution. Of course, the transformation must be defined once and then it can be used again and again but the initial investment can be high and should not be underestimated.

Sendal et al. [11] list three transformation approaches. The first describes the direct model manipulation. This means that a model is accessed directly via e.g. Java and transformed into the target model. Due to the use of a general-purpose language, developers do not need to learn a new language, which helps a lot to understand and develop a Transformation. The disadvantage is that the language offers no high level abstraction which means that a Transformation can be very complex. The next approach is the intermediate representation. It takes a model and transforms it into a standard form such as XML. From this moment onwards, an external tool can be used for further transformation. Therefore, a standard solution is needed to eliminate the requirements of the different model structures. As the Transformation is done outside of the tools, some information such as the cross-model-integrity constraints can be ignored and, therefore, result in an incorrect Transformer. The last approach is called transformation language support. It refers to languages which consist of constructs to express transformation. Those languages are tailored for the problem area and are therefore the most powerful approach.

A transformation language can be declarative, procedural or a combination of both. Also a graphical language to represent transformation might be a feasible solution. Sendal et al.[11] define the following four characteristics which should be considered for a good transformation language. The first characteristic is precondition. It means that, as a transformation works only under certain conditions, the transformation should be supported by a tool which enables direct feedback about the correctness of the created transformation during the developing. The next one is composition which means that already defined transformations should be reusable for other transformations. The third characteristic is form. It refers to the representation of the language as e.g. a visual editor. The last one is usability which means that the language should

support the user in his or her doing and not confront him or her with a lot of complex tasks.

Bézivin et al. visualize the basic idea of model transformation within Figure 2-6. Ma represents the source model which is transformed by Mt into Mb. Mt contains the transformation operations.

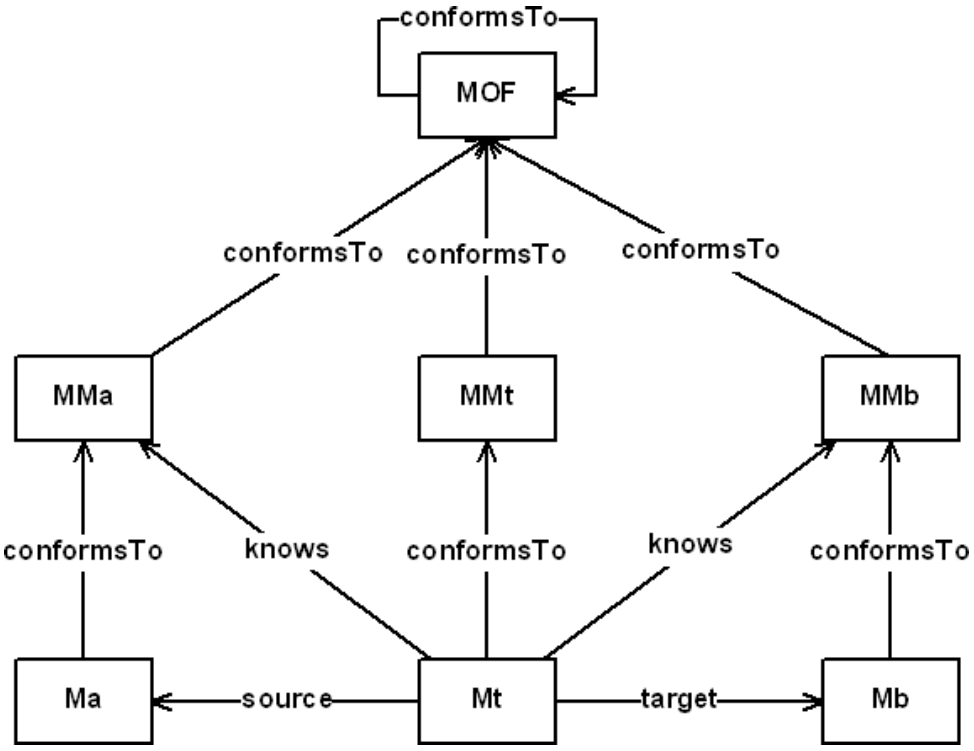

Figure 2-6 MDE Model Transformation [29]

The operations of the model Mt are created based on the transformation language MMt. This is a metamodel. The metamodel or model language of Ma and Mb is MMa and MMb. Mt is aware of all 3 metamodel languages. The metamodel languages too must be based on a model. The metametamodel of the Meta Object Facility (MOF) defined by OMG builds the base of all 3 metamodels.

2.5 Ontology Modeling

Djurić et al. [30] discusse with “Ontology Modeling and MDA” an approach to use the Model-Driven Architecture for ontology engineering. MDA is well accepted within the industry and therefore a multiplicity of tools such as simple

code generators or complex transformation tools are available [10]. The approach recommended by Djurić et al. respects this and uses it for its benefit.

Typically, an ontology is expressed with an ontology language such as Web Ontology Language (OWL). OWL offers different sublanguages with different levels of expressiveness [19]. Figure 2-7 illustrates the OWL stack and highlights the derivation of it. The Extensible Markup Language (XML) serves as syntax basis because of its “common, well defined and easy process able syntax”[30]. On top of XML, the Resource Description Framework (RDF) introduces elements to express semantics like relations. As RDF is not expressive enough for reasoning, OWL is introduced.

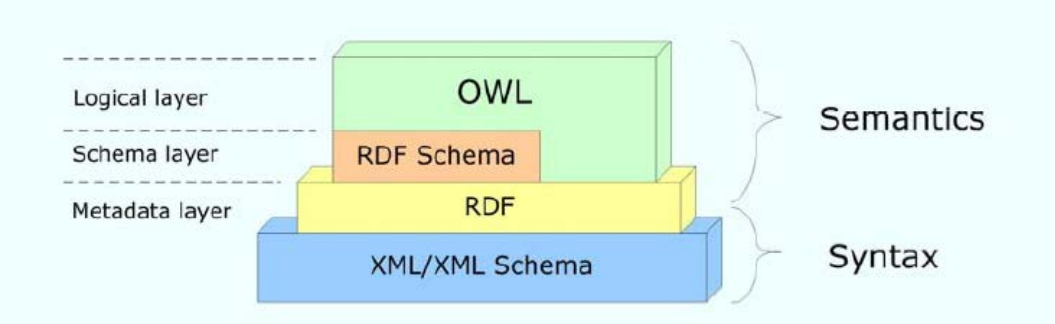

Figure 2-7 OWL language stack [30]

In contrast, the Model-Driven Architecture consists of a four-layer architecture M0-M3. M3 represents the metametamodel layer composed of the Meta Object Facility (MOF). M2 consists of UML as metalanguage defined by MOF and M1 represents models based on the metalanguage. The last layer M0 contains the instances of the models. Based on this architecture, Djurić et al. [30] introduce the Ontology modelling architecture. Figure 2-8 shows the architecture with the different layers of OWL and the MDA and it dependencies.

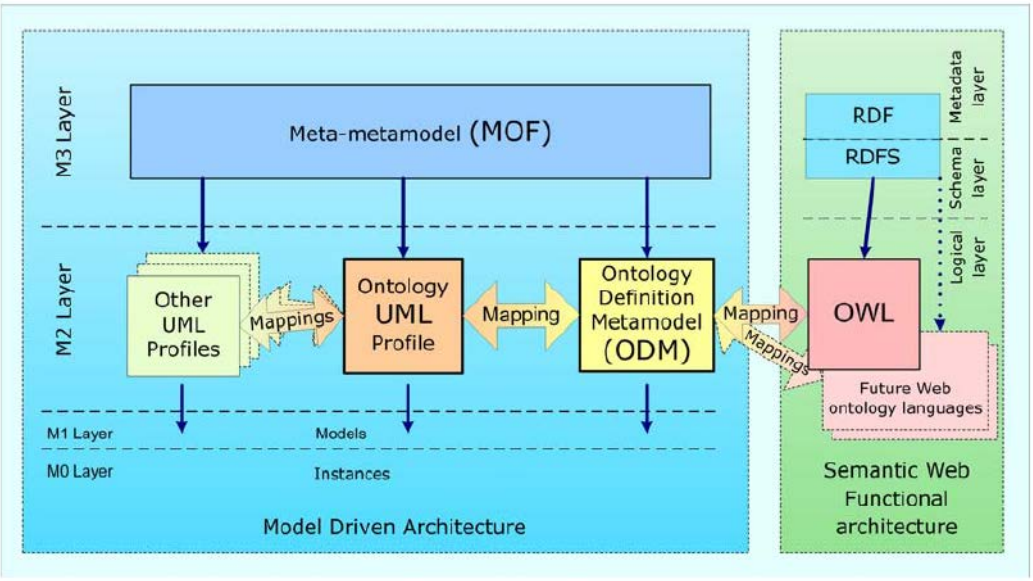

Figure 2-8 Ontology Modeling architecture [30]

The core element of the approach is the modelling language Ontology Definition Metamodel (ODM) based on the metametamodel MOF. ODM is a metamodel and, as defined by MDA, it is located within layer M2. The connection between ODM and OWL is defined by mappings. To execute those mappings, transformation instructions based on XSLT (Extensible Style sheet Language Transformations) are used. There is a specific Ontology UML Profile which enables the benefit of graphical modeling. In return, mappings and transformations based on XSLT are used to link ODM and the UML profile.

With ODM, the fixed layer structure of OWL is persevered and the benefits of both worlds of, MDA and the ontology-based semantic integration, can be used. Ontologies get used for their expressiveness and MDA for their good tool support.

CHAPTER 3

Research Approach

This chapter introduces the research approach of the master thesis. The first section highlights the problem scenario of creating a transformer within an integration scenario based on the Engineering Knowledge Base. The use case is described by introducing the hydropower plants construction application area. The research issues are discussed in the next section. Each of the issues examines different aspects such as the requirements of the use case and the feasibility, the complexity, and the effectiveness of the proposed solution. The last section focuses on the methods to approach the related work, the prototyping and the research issues.

3.1 Use Case and Problem Scenario

This master thesis examines multi-disciplinary engineering projects from the electric power industry where different teams from various technical disciplines work together. The shared goal is to achieve an increased flexibility and reactivity to the needs of the market and the customers with a simultaneous decrease in costs. This general goal is confronted with the challenge of integration. Different tools and systems subject to the terms and notations of their own technical disciplines must work together and exchange data. A feasible solution of such an integration scenario is offered by the Engineering Knowledge Base introduced by Tomas Moser [12]. Based on ontology data modelling, EKB enables the explicit modelling of domain-specific knowledge to ensure data exchange between different domains. Each domain represents a discipline with a set of tools, terms and notations which can be assigned to a common concept. The application

scenario of EKB (as shown in Figure 2-1) consists of the following 3 steps: Extraction of data from each domain and tool; integration of the tool data into a common repository; and transformation of the tool data. Although the first two steps represent a plurality of exciting challenges, this master thesis focuses mainly on the third step: the transformation of tool data based on the explicit knowledge stored within the EKB.

The use case selected for this master thesis comes from the hydropower plants construction. To build a power plant, electrical, mechanical and software engineers must work together. This is complicated by the multiplicity of different stakeholders, tools, terms and notations. But, regardless of these differences, they mostly mean the same information; they just look at it respectively from different perspectives, different layers of abstraction etc. The synergy potential of a common communication exchange between those engineering areas would be huge. A missing shared orientation, independent development and no common language represent just a small number of the troubles of an environment with missing integration. To establish a communication between the tools of the engineering areas, the concept of signals can be used. A signal represents a mechanical interface, an electrical signal or a software I/O variable which transports information within complex automation systems. Signals are introduced by the application field signal engineering where the focus lies on the managing of information from different engineering disciplines.

To transform a signal from one discipline into another one, it is necessary to create a transformer. A transformer represents a sequence of instructions which alters the signal data in a way that it is converted into the needed form. The life cycle of a transformer (as shown in Figure 3-1) starts with the development phase, where simple instructions are combined into full qualified and complex transformers. Guiding principles here are tool models, mapping information combined into the implicit knowledge stored within the Engineering Knowledge Base. The occurrence of changes within the tool domain demand the update of the impacted transformer or, if not possible, the discharge. To prove the correctness of a transformer, real data provided from e.g. the Engineering Data

Base is converted from one format into another and reviewed respectively according to their corresponding model definitions.

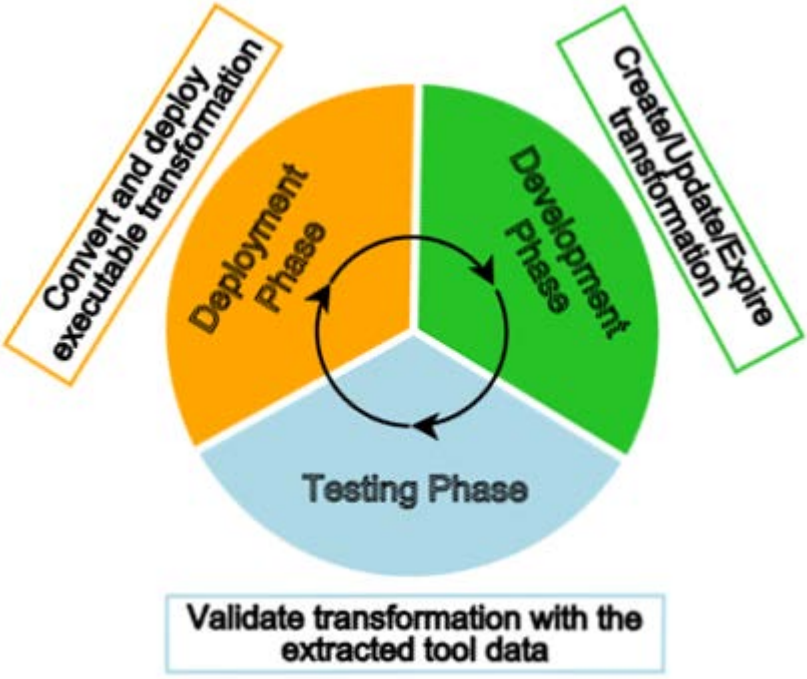

Figure 3-1 Life-cycle of a transformation

The validation of a transformer is part of the testing phase. If the transformer is a valid representative, the deployment phase can be initiated. First, the transformer is converted into executable code. This code can then be deployed to the executive system which deals with the integration of the domain tools at runtime.

This master thesis researches the problem scenario shown in Figure 3-2, as derived from the use case of signal integration and the application scenario of the Engineering Knowledge Base, Two different tool domains and a virtual common domain as intermediate layer need to be integrated. The first step consists of the extraction of tool data as signals from the two tool domains. The extracted signals are stored within the Engineering Data Base (EDB). In addition, the virtual common domain deposits its signals into the EDB. Next, the domain and integration experts store the domain specific knowledge in form of ontologies into the Engineering Knowledge Base. Each one of the ontologies represents a

domain with its specific signals. Part of the stored knowledge and highlighted as step three are the mappings between the ontologies as attribute mappings.

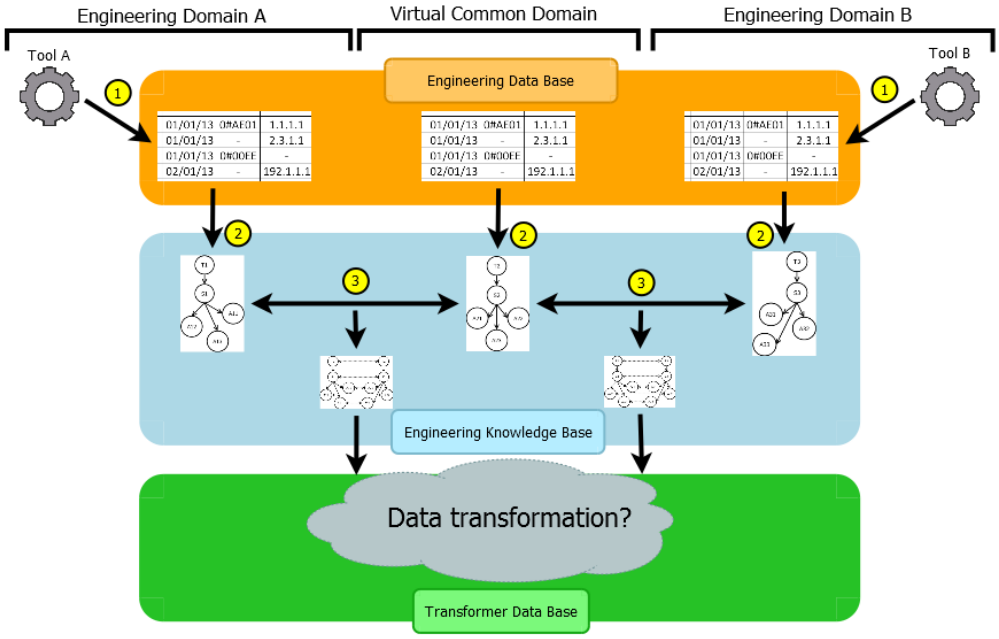

Figure 3-2 Problem scenario

After these three steps, representative knowledge about the domains, signals and their data is collected and stored in a computer-readable format. Nevertheless, the problem of integration is still not solved. What is missing are the Transformers themselves. The principal task of this master thesis is to offer a feasible solution for developing a Transformer based on the stored information within the EKB and the EDB.

3.2 Research Issues

With the research issues, defined by this section, the solution approach TransformerIDE is evaluated regarding different criteria such as effectiveness, efficiency and usability.

RI-1 Use case analysis, requirements elicitation and design process

This master thesis seeks a solution which enables an effective, efficient and usable way to create Transformers for an integration scenario with different tools from different engineering areas. The requirements are gathered and collected by analyzing the use cases, derived from hydropower plant construction area. These requirements represent the basis of the design process of the TransformerIDE. Thus, further analysis should be possible based on measures of an effective, efficient and usable solution.

RI-2 Efficiency and Effectiveness of the Transformer Creation Process

The TransformerIDE is the proposed solution for the problem scenario of this master thesis which consists of the challenge of creating a Transformer to enable communication between different tools from different engineering environments. The research issue investigates the ability of the TransformerIDE and of the underlying Transformation Meta Language to satisfy the identified requirements. Effectiveness and efficiency are analyzed for this purpose. Concerning the effectiveness, the relevant requirements are evaluated and checked to find out their fulfilment. Regarding the efficiency, the TransformerIDE is compared with another approach consisting of the use of a script language to create a Transformer to show the strengths and limitations of both approaches.

RI-3 Analysis of the usability of the proposed solution

This research issue examines the suitability of the proposed solution approach for the targeted audience. A strong focus here lies on learnability and usability. The most important question here is: Can a user with technical background but no knowledge about the integration scenario create a workable Transformer within an acceptable range of time and effort?

3.3 Research Method and Evaluation Concept

This section specifies step by step the approach of the master thesis to develop a feasible solution for the creation of Transformer within a multi-disciplinary engineering environment. The enclosing topics derived from scientific literature and the correspondent industrial application areas are analyzed by progressive investigation. The first step is the detailed familiarization with the baseline research areas “semantic integration” and “semantic heterogeneity”. This results in a basic understanding of the topic and the connected research fields. The next step is an excursion into the more specific fields of the related work such as ontology-based semantic integration approaches, the model-driven architecture and ontology modeling. Ontology-based semantic integration approaches is the main focus of this master thesis to address the problem of semantic integration. A basic idea and theory of model transformation is given with the model-driven architecture and, in particular, the MDE model transformation. Ontology modelling represents the junction between semantic integration approach and model-driven architecture. The literature research is based on the systematic literature review described by Brereton et al. [31]. This consists of three phases: planning, conducting and documenting the review. A structured approach with traceability and a clear focus on the result is achieved with the successive execution of the three phases.

Next comes an analysis of the Engineering Knowledge Base introduced by Thomas Moser [12]. It is a semantic integration approach which uses ontology-based data modelling to store explicit knowledge about tools from different domains. This knowledge allows the creation a Transformer which enables the overlapping communication between different domains and tools. The EKB acts as a capstone for the theoretical groundwork of this master thesis by linking all the related work.

The next step is the discussion of the use case with the objective to highlight the application area and the necessity of a feasible transformation solution. A result of this discussion are the requirements defined according to the IEEE STD

830-1998 [32] for a Software Requirements Specification (SRS). The standard defines the sections of a complete SRS such as requirements, product perspective, product functions, user characteristics, constraints, assumptions and dependencies. Subsequently, the problem scenario illustrates where within the use case the master thesis tries to find a solution. Three research issues are selected to narrow down the research problem. Each one of them has a different focus. The first research issue (RI-1 Use case analysis and requirements elicitation) analyses the use cases and the derived requirements. The second (RI-2 Efficiency and Effectiveness of the Transformer Creation Process) compares the approach with another more generic approach to determine the efficiency and effectiveness and the third (RI-3 Analysis of the usability of the proposed integration) questions the usability of the proposed solution.

A solution called TransformerIDE is designed to fulfill the requirements of the Software Requirements Specification. The design is discussed with the help of a Software Design Description (SDD). The SDD with its different sections such as system architecture, design patterns and the structure of the Transformation Meta Language gives a clear picture of the solution. The next step is the implementation of a prototype to enable any further evaluation. Floyd [33] characterizes different concepts of prototyping such as prototyping for exploration, experimentation and evolution. For this master thesis, a full functional prototype based on prototyping for experimentation is implemented.

Three quality criteria, defined within the ISO standard ISO/IEC 9126-1 [34] are used to evaluate the TransformerIDE. The first criterion is **effectiveness** which is defined as “the capability of the software product to enable users to achieve specific goals with accuracy and completeness in a specified context of use”. To prove that the proposed solution has the capability to achieve specific goals of the context, each requirement is analyzed regarding its possibility to be fulfilled with the TransformerIDE. The second criterion is **efficiency**. The ISO standard defines efficiency as follows: “The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions”. To show that the TransformerIDE supports the user to improve the performance and to reduce effort, four exercises are done on the one hand by

using the TransformerIDE and on the other hand by using a script language without any tool support. The two results are analyzed in the dimensions of the execution time, the complex situations and the resulting Transformer Code. The last criterion is **usability**. Usability has multiple sub-criteria; for this master thesis, learnability and understandability are analyzed. To evaluate those two criteria, a set of ten test users is requested to execute four exercises using the TransformerIDE. During the execution, the following three measured values are documented and used as the basis for the evaluation: successful execution, execution time and the count of occurring complex situations. The first value states if a user was capable to finish an exercise successfully. The second gives a clear overview of how long a user requires to accomplish the different exercises and the last one indicates which complex situation appeared during which step of an exercise.

CHAPTER 4

TransformerIDE Design Process

This chapter presents the scope, environment, stakeholders, requirements, architecture and the implementation of the TransformerIDE. The section Scope defines the proposed approach and illustrates the covered and not covered functions, the process of usage and the purpose. In the third section, the field of application in modern industries, the research areas such as signal engineering, semantic integration and model transformation are introduced. The next section contains the requirement specification based on the IEEE Std 830-1998 [32] standard, including the product perspective, product functions, user characteristics, constraints, assumptions and dependencies. The fourth section discusses design and architecture of the Transformation Meta Language and the TransformerIDE.

4.1 Scope

The TransformerIDE is a tool kit to manage and build a Transformer between data models of different engineering concepts. It offers an integration development environment. A Transformer comprises multiple Transformations which consist of instructions to translate a signal from an engineering concept A into a virtual common model and vice versa. To build well working

transformations, explicit engineering knowledge is needed. This information is provided by the different engineering domain experts within a knowledge base determined by the semantic integration approach EKB (Engineering Knowledge Base). The EKB stores mappings between local and common engineering concepts, data model descriptions and meta information like value range, value format etc. The data integration part of the EKB is not part of this master thesis but an interface to access the knowledge base is available.

By using the mapping information, the TransformerIDE provides automatically the base frame of a transformation between a data model of an engineering concept and the virtual common model. In order to create the complex intermediate transformation instructions, the TransformerIDE uses the Transformation Language which is based on the Transformation Meta Language (TML) which offers various transformation functions (e.g. trim, substring etc.). An overview about the supported process of the TransformerIDE is visualized in Figure 4-1. The inputs are the information provided by the EKB such as data models, use case data etc. OWL is used as the standard format of the provided information. A Subject Matter Expert (SME) defines the information and a Software Developer (SD) is responsible for the development of the Transformer logic. Beside the involved roles, the TransformerIDE has an interface to the EKB and the EngSB. The EngSB is the engineering bus which embeds the created Transformer for further use. Within this process, the IDE operates like a unit construction system where the functions are the bricks and the base frame the basement. This building process is the manual part of the transformation building. To support this creation step the TransformerIDE offers a verification of the transformation with regard to a correct structure and a correct operating.

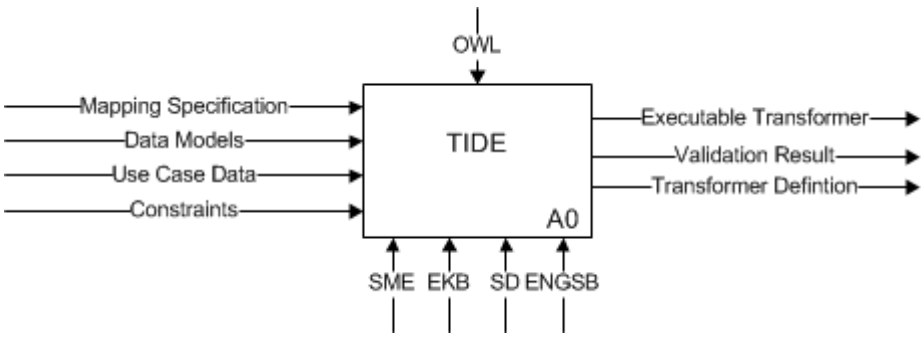

Figure 4-1 TransformerIDE Context

A correct transformation is given if the construct based on the TML consists of no missing or wrong elements. Typical errors are e.g. when a function needs an element that is still not set or when there is a boundary violation etc. By contrast, correct operation means that a transformation is validated on the basis of entered test data and available meta-information. Thereby, the transformation is executed so that the result of the transformation can be compared with the anticipated test data and verified on the meta-information like value format, value range etc. This verification checks enable the accurate understanding of created transformations and their behavior. To use transformations outside the development environment, the code generator interface allows the creation of code generators for different programming languages e.g. java. With this flexibility, there is no restriction to any target execution environment. If an engineering concept changes, an old transformation can be enhanced and updated without loss of any already created transformation instructions which can be reused.

4.2 Environment and Stakeholders

Modern industries like energy, automotive or chemical require complex automation systems. These systems are used and, of course, needed to optimize and increase the efficiency of the production processes. By so doing, complex machines, control systems and information technologies are used to achieve this target. Engineering disciplines like mechanical engineering, electrical engineering and software engineering are involved in such systems [12]. In such a multi-disciplinary engineering domain, each discipline has its own tools and models set for the same, or at least similar, information.

Such systems were mostly developed independently and enhanced over a relatively long period without compatibility with other areas. Moser et al. [35] introduce the application field "Signal engineering" where the managing of signals from different engineering disciplines is dealt with. The concept of signals is used in complex automation systems to link information across different engineering disciplines. This can be mechanical interfaces, electrical signals or software I/O variables. The 3 main challenges are a consistent signal handling, the integration of signals from heterogeneous data models/tool and the version management of signal changes across engineering disciplines [35]. Moser et al.[1] propose the

semantic integration approach Engineering Knowledge Base (EKB) for automation systems. The EKB "supports the efficient integration of information originating from different expert domains without a complete common data schema".

The architecture of the EKB consists of the following: extraction of tool data, storage of the data, description of tool knowledge and of domain knowledge and mapping of the tool knowledge to the domain knowledge. Each engineer must describe, with the help of ontology, the proprietary view on the information exchanged. This is information like the format, the meaning and the use of the exchanged information. The domain knowledge represents the collaborative view on the exchanged information. This knowledge is created in cooperation of all engineers and offers a standardized domain specific view.

The mapping between the tool knowledge and the domain knowledge enables the creation of transformations between a model of a tool and a common data model. The Transformer IDE supports the managing and creation of such transformations based on the stored knowledge. Jouault et al. [36] describes a common model transformation pattern (Figure 2-6), which is used within the Model-Driven Engineering (MDE). Just like the model transformation pattern, the TransformerIDE consists of a transformation program based on a metamodel language called Transformation Meta Language (TML). By using TML, the Transformation Language (TL) is defined to enable the transformation of models from different domains.

4.3 Requirements Specification

The following chapter presents the Software Requirements Specification (SRS) of the TransformerIDE based on the IEEE Std 830-1998 [32]: standard, product perspective, product functions, user characteristics and constraints. The first section, points to all terms, acronyms and abbreviations used within the SRS to understand it in the intended way. In the second section, the TransformerIDE is analyzed regarding their relation to other applications and solutions. In so doing, the requirements derived from these dependencies and the needed interfaces are introduced. The third section summarizes the functions performed

by the solution such as transformer management, transformer creation, transformer testing, transformer validation and code generator. Then, in the fourth section, the stakeholders and their goals are explained. The next section describes all constraints related to the problem field and having any impact on the decisions made for the proposed solution.

Product Perspective

The TransformerIDE is an IDE to manage and build transformers based on the Transformation Language (TL). In so doing, the TransformerIDE uses the Engineering Knowledge Base which "stores the engineering knowledge in ontologies and provides semantic mapping services to access design-time and run-time concepts and data" [12]. The data offered by the services of the EKB is used to create the frame of the transformer, validate the created transformations and to support the users to create the intermediate steps of the transformations. As the TransformerIDE manages also the created transformers, it makes sense to offer a service to get access to them in order to use them as described by Moser T. [12] for "more complex applications to perform advanced tasks like tracing of artifacts, consistency checking across tool boundaries, change impact analyses or notification of stakeholders in case of changes".

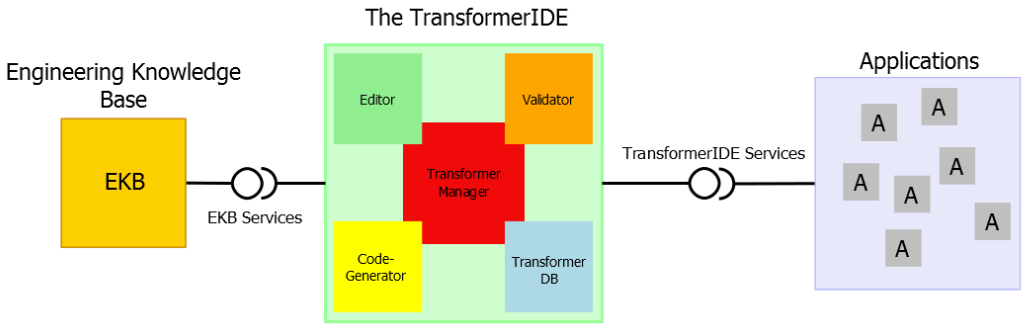

Figure 4-2 The interconnections of the TransformerIDE

Figure 4-2 visualizes the interconnections of the TransformerIDE with the EKB and the advanced applications. On the left side, we see the EKB with its offered services. In the middle, we have the TransformerIDE consisting of the 5 main parts: editor, validator, code generator, transformer manager and the

TransformerDB. On the one hand access is given to EKB services and on the other hand TransformerIDE services are provided.

System Interfaces

The TransformerIDE is a web application. It is published on a compatible web server. As the application must store and access information a relational database is used. Any compatible web browser can be used to access the web application

User Interfaces

The User Interface of the TransformerIDE has the following characteristics to enable optimal handling:

- Common widgets: The UI of the TransformerIDE will consist of only common widgets like buttons, tables, check-boxes, drop-down-boxes, lists, links, context menus, tool tips, dialogs, text fields, search fields, labels, tabs and status icons.
- Page layout: The different masks of the tool will be laid out on different pages.
- Dialogs: Different types of dialogs will be used such as error, save as, create and information dialogs.
- Back button: From each page a back button will be available so that it is possible to return to the previous page.
- Icons: Different icons will be used to visualize actions, states and elements.
- Visualize errors: To visualize errors or problems dialogs, status icons or status labels are used.

Hardware Interfaces

The application consists of the following hardware requirements:

- Client side

  Since the TransformerIDE is a web application based on Java Server Pages and Wicket, the access must be with a browser which supports

HTML (Version 5.0+) and Java Script (Version 1.7+) like Google Chrome, Firefox etc. Therefore, each system with such a browser can be used.

- Server Side

  The web application is deployed on a web server with a Java Server Pages (JSP) compatible servlet container as e.g. Jetty or Apache Tomcat. The web server can run on any compatible system like Linux or Windows server. The TransformerDB needs a relational database where all TransformerIDE information is stored. Such database can be a relational database like MySQL or Oracle etc.

Software Interfaces

The TransformerIDE requires the following software products:

- Java Runtime Environment 6+

  This is the computing platform of the web application.

- Jetty Version 7+

  The web server Jetty is used to publish the web application.

- A browser which supports HTML (Version 5.0+) and Java Script(Version 1.7+)

  Any compatible web browser can be used to access the web application.

- MySQL Version 5.5+

  The TransformerDB uses a relational database to store the transformer information.

Interfaces to other applications:

- Engineering Knowledge Base
  The Engineering Knowledge Base stores the tool specific knowledge in form of ontologies. The service offered to access the information is used by the TransformerIDE.
- Applications

Communication Interfaces

- The protocol used for the communication between the server and the client and the server and the services of the Engineering Knowledge Base is HTTP.

Product Functions

The TransformerIDE offers a tool kit to manage and build transformations between data models of different engineering concepts. This includes the creation, modification, validation, storing, managing and exporting of Transformer and Transformations. To group the different functions the TransformerIDE consists of 5 modules called Editor, Validator, Transformer Manager, Code-Generator and TransformerDB (see Figure 4-3) and the Transformation Language which represents the basis of each Transformer and the different modules.

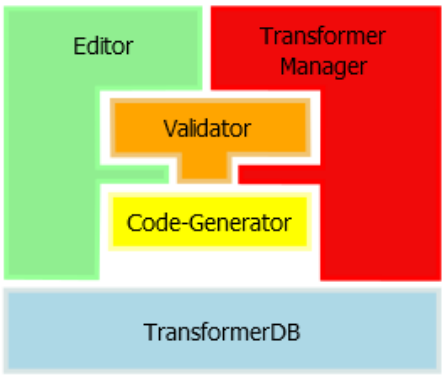

Figure 4-3 The TransformerIDE Modules

Transformation Meta Language

The Transformation Meta Language (TML) is the core of the IDE. the Transformation Language (TL) is defined based on the metastructure. TL offers functions such as trim, concat etc. to manipulate input variables to get as a result an output variable. TL is a tailored procedural language which can be extended by new functions to fulfil the changing requirements. Due to the use case, the language is limited to a simple structure of multiple source variables and a target variable and, in-between, a set of nested functions. The different Modules of the TransformerIDE are aligned to the structure of TL.

Editor

The Editor offers a visual surface to create, edit, validate, test and export a Transformation out of a mapping between domain model attributes and a tool model attributes. The Editor accesses the TransformerDB module to load the tool and domain model information and the Validator module to validate the created transformation.

Validator

The Validator validates a Transformation on the correctness respective the domain and the tool model definition by using the test data entered by the user. The Validator is used by the Editor and the Transformer Manager.

Code Generator

The Code Generator creates an executable transformer script on the basis of the stored Transformations within the TransformerDB for a mapping between a tool model and a domain model and vice versa. The Code Generator module is accessed by the Transformer Manager and the Validator.

Transformer Manager

New Transformers can be generated and available Transformer can be managed by using the Transformer Manager. The following functions are covered by the manager: Creating a Transformer based on the EKB mappings; changing, updating and copying of a Transformer; downloading a Transformer as executable script.

TransformerDB

The TransformerDB is the central database for the created Transformer and all associated information. This includes all Transformer detail information like the name, description, history information, test data and of course all Transformations with the transformation instructions created by the user.

User Characteristics

The TransformerIDE is considered to be used by users taking over a role based on the integration process described in the section The Engineering Knowledge Base Integration Process. That implies that the user has at least

some of the following knowledge: basic computer usage understanding; basic problem domain and integration problem understanding; basic knowledge about logical software functions like split, concat etc. Of course, also users who are not involved in the process but have the needed knowledge described previously, can use the software. This is possible because the TransformerIDE offers a straightforward view on the EKB information and a simple modular construction system to create the Transformations.

Constraints

The TransformerIDE will be a web application and therefore accessible for the user via web browser. The involved constraints regarding hardware and software are described in detail in the section Product Perspective. Additional constraints are the requirement of a user interface with an English localization and user identification to be able to identify any change made by a user.

4.4 Design/Architecture

> "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution".[37]

The architecture and design of the TransformerIDE and the Transformation Meta Language is presented in this section. First, the design patterns used are introduced. Next, the architecture of the TransformerIDE is addressed and then, the Transformation Meta Language structure is dealt with.

Design Pattern

Different well established design patterns have been used for the TransformerIDE. The focus has been on a clear architecture with well-defined components and tasks. Based on the book patterns of enterprise application architecture by Fowler [38], the following patterns have been used to design the TransformerIDE.

- Data Source Architectural Patterns
    - Data Access Object
    - Data Mapper

- Domain Logic Patterns
    - Service Layer Pattern
- Web Presentation Patterns
    - MVC Model View Controller
    - Template View Pattern

A common architecture for a web application is the client server three-tier architecture. It separates different logical aspects of an application and assigns them to different layers. Such typical layers are the Presentation-, Business- and Data Source-Layer. The TransformerIDE is based on this architecture. All design patterns used to design the TransformerIDE can be assigned to a layer. The Data Access Object and the Data Mapper are used for the Data Source Layer. The Data Access Object (DAO) acts as an intermediate between the application and the database. It offers an interface to the data of the database and hides all the complex database details. The full implementation of a DAO is expensive. A Data Mapper is used to decrease the effort; it handles the mapping between the domain model and the database table. The Service Layer Pattern encapsulated the business layer. In so doing, a well-defined interface is produced and the complex interactions are hidden. The common Model View Controller pattern is used for the Presentation Layer. This pattern divides the User Interface into a view, a controller and a model. A better handling of the view is provided by the Template View Pattern. Hence, not the static part but only the dynamic parts of an HTML page are touched.

Architecture of the TransformerIDE

A basic overview of the architecture of the TransformerIDE is given by Figure 4-4 The three-tier architecture of the TransformerIDE. It shows the three-tier architecture. The layer on top is the presentation layer consisting of the Transformer Editor and Transformer Manager Package which build the user interface. The middle tier, also called business layer, consist of the TransformerIDE Services and is an interface to the business logic of the application. The last layer called Data source Layer is composed of the Data Access Object Package.

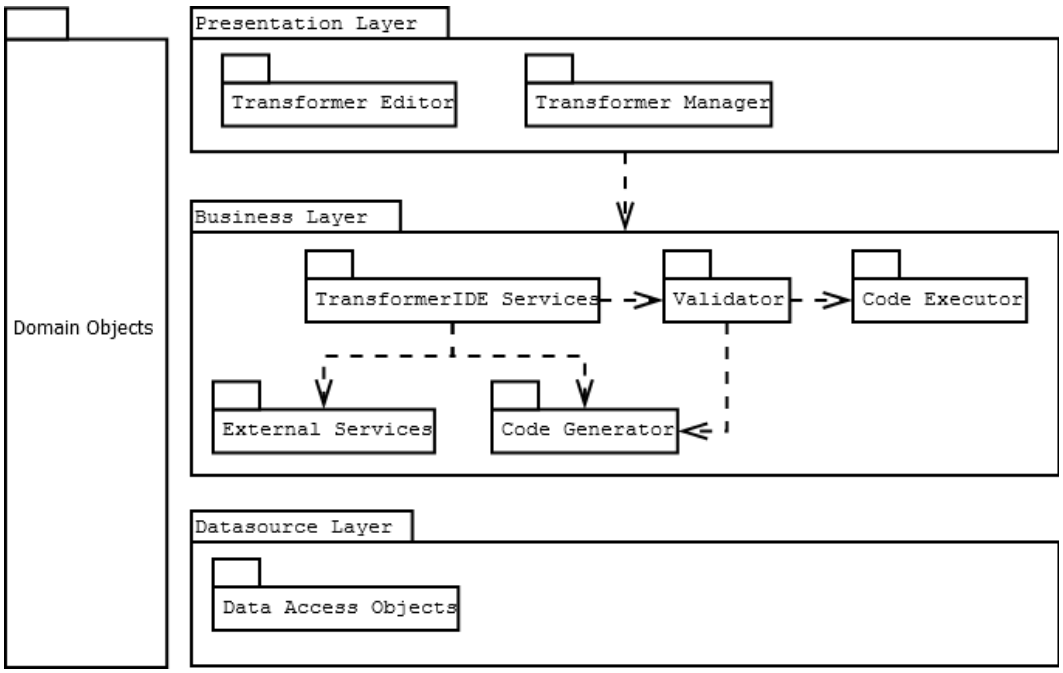

Figure 4-4 The three-tier architecture of the TransformerIDE

The Transformer Editor and the Transformer Manager are the visual representations of the IDE. Both components are implemented based on the Model View Controller and consist of a View, multiple models and a controller. The template view pattern of the web design patterns is used to implement the view. The controller of each component uses the Transformer Service which is the central point to access the Business logic. To get a better functional separation of the business logic functions such as the validation, generation and execution of the Transformers are divided into different packages such as the Validator Package, Code Generator Package and the Code Executor Package. Thus, it becomes evident that the TransformerIDE Service combines the different functions to offer a service with access to all needed functions. The Validator Package consists of the logic to validate a Transformer and the related transformation regarding the correctness. The Code Generator Package consists of the code to generate from the TML code an executable code. Finally, the Code Executor Package consists of the logic to execute the generated Code.

Transformation Meta Language

A transformation created with the TransformerIDE is expressed in a Transformation Language (TL) defined by the Transformation Meta Language

(TML). TML defines the procedural TL tailored for the requirements of the application area of this master thesis. As defined by Sendal et al. [11], a good transformation language must fulfil 4 characteristics. Those are tool support, defined conditions, visualization and a simple and focused language. The TransformerIDE represents the tool support allowing a supported usage of TL. The Editor module fulfils the requirements of a structured visualization. The remaining characteristics are explained in this section by introducing the design of the TML and the resulting TL.

The structure of a Transformer is a logical composition of attributes such as name, description, timestamp information and the mappings between the source and target variables of the signals which need to be transformed (Figure 5-6 Transformer Structure). The mappings represent the skeleton of the Transformer. Each individual mapping consists of a target variable and a set of source variables which must be completed with a working Transformation. The development of the transformation logic is the core challenge which must be handled by the Transformer developer.

The language TL represents the solution approach proposed by this master thesis to express and to develop the logic of a transformation. Following the basic structure of a mapping, each transformation formulated in TL has a root element which represents the target variable of the target signal. The root element can then be widened with child elements. A child element represents functions and necessarily all the source variables of the source signal. The shape of a transformation is a tree with one root and multiple nodes and leaves.

Figure 4-5 shows an example of a Transformation. The target variable with the name signalNumber is a combination of the source variables kks0, kks1 etc. with a comma delimiter. On the right side, we can see the tree-structure highlights root, nodes and leaves of the example transformation. To combine the source variables and the delimiter, the function concat requiring two input values to be valid, is used. This combination results in a nested structure which shows a traceable manipulation path.

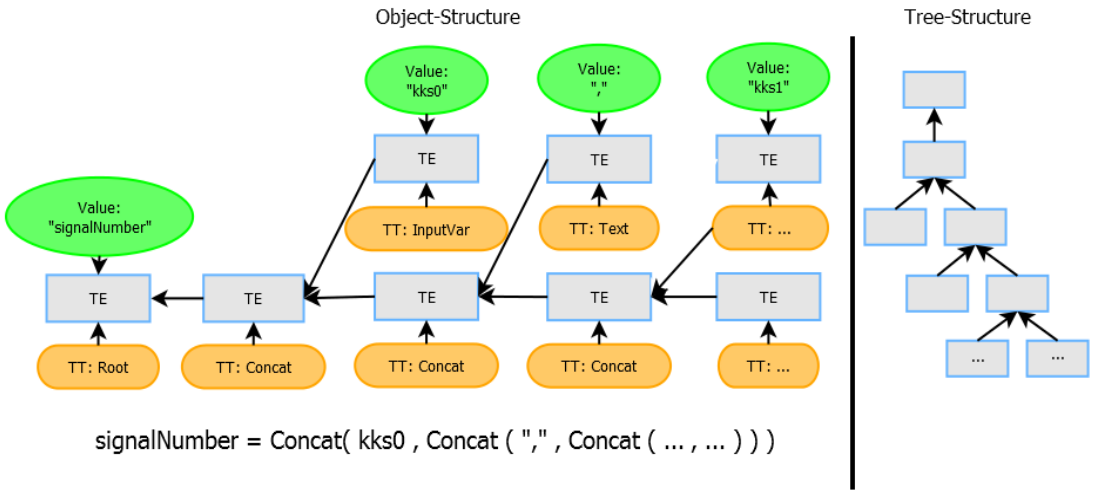

Figure 4-5 Transformation Structure

TML is a metastructure/language consisting of the components Transformation Element (TE) and Transformation Type (TT). A Transformation Element is a generic component having a parent element, children elements and an attribute id. The references to parent and child elements enable the nested structure of a tree. The type of TE is defined by the Transformation Type (TT). A TT has a name, a description, a count of needed children and the TT of the children. The possibility to define the type of a children shows that a TT represents a nested structure too. A TT definition can therefore be seen as a class definition of an e.g. materialized function (object) as part of a transformation. By now the defined structure based on TT and TE is generic and enables the creation of a nested structure without real meaning due to the absence of primitives. Such primitives are Element, RootElement, NumberField and TextField. The attribute element determines if a TT represents a primitive.

The creation language of a Transformation is created based on the definition of TransformationTypes. Two different categories of TT were created for the TransformerIDE prototype. The two categories are primitives and types composed of primitives.

Primitives:

- Root: The Root element indicates the first element of a transformation. It represents the target variable.
- Element: This is the simplest type since it represents a placeholder for any other type. This is used especially for the attributes of functions which expect another type as input.
- TextField: The TextField is an input field which enables the entering of a string value. A TextField can be used to enter a string value as a parameter of a function such as a delimiter etc.
- NumberField: A NumberField is identical to the TextField beside the restriction that only numbers are a valid value.

Composed types:

- Trim: Trim is a simple function which removes whitespaces at the beginning and the end of a string value. This function consists of a child element with the type Element.
- Concat: Concat combines two values into one element. It requires two children with the type Element.
- Substring: The function substring extracts the string between two indices. The function requires an Element as input and two numbers to indicate the indices.

The three composed types are only an example to show the functionality of the prototype. For the use within a real application environment additional types can be created to fulfil the requirements. The same applies to the primitives which can be extended.

Each language has natural restrictions which qualifies it. When all the conditions are fulfilled, then a Transformation based on TL is correct with regard to grammar. The following list shows all the conditions of TL defined by TML.

- The core element of the transformation is the TransformationElement.
- Each TransformationElement has a type.

- The starting element of a transformation has the type Root.
- Each parameter with the type Element must be set.
- Each leaf must consist of a closing type.
- The depth of the nested structure is not limited.
- The root element represents a target variable of the source model to target model mapping.
- Each source variable of the mapping must be part of the transformation.

The list of conditions is short and straightforward and it narrows the structure of a transformation to the nested tree structured highlighted within Figure 4-5.

Transformation Execution

Each Transformation created based on the Transformation Language must fulfil on the one hand the grammar requirements and on the other hand the restrictions of source and target values loaded from the Engineering Knowledge Base. To verify if a transformation is valid, it must be executed with test data. A transformation based on TL cannot be executed since as no compiler or interpreter exists.

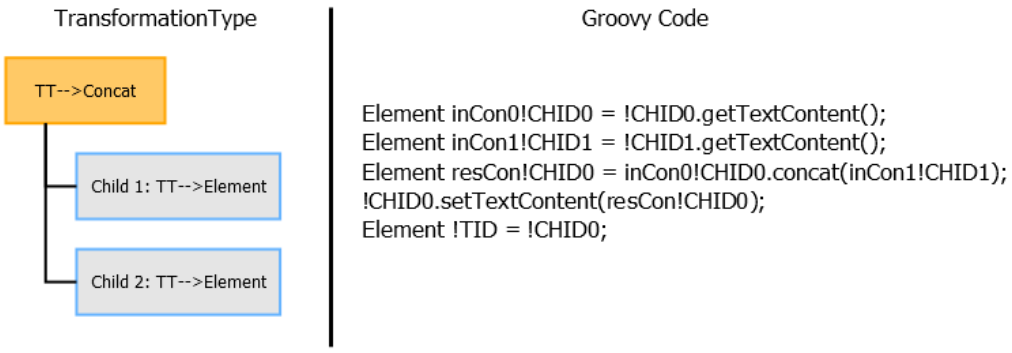

Figure 4-6 TransformationType Code Representation

The TransformerIDE offers anyway for this a framework to transform each transformation from the Transformation Language into an executable code such as Groovy, Java or any other types. The idea of the framework is simple: Each

TransformationType has a counterpart which is based on code which can be executed. Each counterpart must fulfil three conditions. The first demands that the interface equates exactly the output and input variables count of the TT. The second is that the type of the input and output values is string and the last requires that each function represents a closed system. When all 3 conditions are fulfilled, then each transformation can be transformed into executable code.

Figure 4-6 shows the TransformationType Concat and the counterpart based on the script language Groovy. Concat has two parameter and one output variable which can be found within the script code called inCon0!CHID0, inCon1!CHID1 and !TID. !CHID0, !CHID1 and !TID are placeholder of variable names replaced by the Generator. The Generator creates, based on the counterparts and a transformation, an executable code. With the placeholder, it can be guaranteed that each function is a closed system which does not affect the other functions.

CHAPTER 5

Evaluation

In this chapter, the TransformerIDE and the Transformation Meta Language are evaluated. The three research issues of the section 0 require the proof of four quality criteria: effectiveness, efficiency, understandability and learnability. Starting with the initiation of the environment, the basis of the following testing and execution framework with the use cases is defined. With the section 5.3, the effectiveness of the TransformerIDE is evaluated. The next section 5.4 compares the TransformerIDE with another approach to show the efficiency and the last section 5.5 evaluates the usability based on a user test.

5.1 Environment and Test Data Initiation

The need to integrate several systems to achieve a collective goal is a rampant challenge which has already led to a multiplicity of solution approaches for the most different application scenarios. The topic of this master thesis derives from the electronic power industry, more precisely from the hydropower plant construction. A project to build a hydropower plant demands the collaboration of diverse engineering disciplines such as electrical, mechanical and software engineering. Therefore, the systems of the various disciplines also need to interact to a certain extent with one another over the whole life cycle of the power plant. To enable a meaningful evaluation, the two systems EPLAN[1]

[1] http://www.eplan.de

and Toolbox 2 OPM[2], used typically for power plant building projects presented and explained. Extracted data from both tools and the implicit engineering knowledge deduced from the application scenario build the test data and therefore the basis of the evaluation.

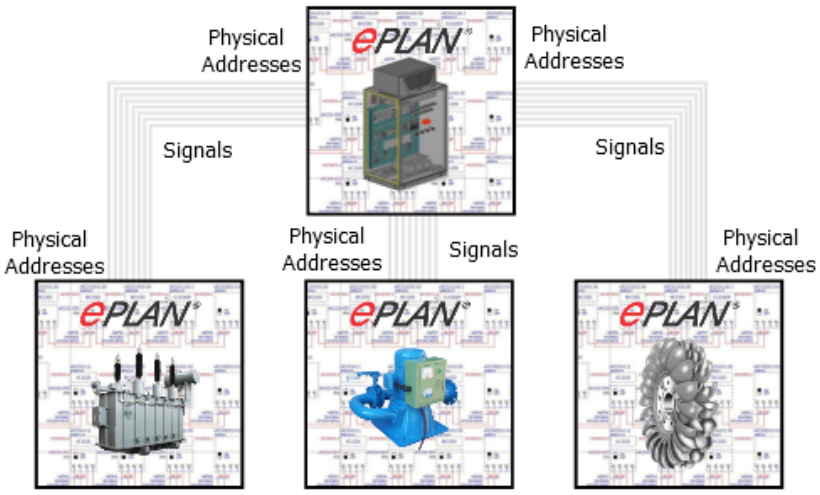

Figure 5-1 Symbolic EPLAN Diagram

EPLAN is an electronic planning tool which supports the electrical engineer during the process of design and implementation of electrical wirings based on diagrams for engines and facilities. The symbolic diagram shown in Figure 5-1 highlights sample components and their interaction. The focus of EPLAN is the wiring of the components itself and the interfaces defined with the physical addresses and the signal structures. EPLAN is used through all the project phases for the different levels of abstraction of the engine and facility with the beginning of the project. The second system ToolBox II provides the engineer with an integrated tool for the engineering of a facility. ToolBox II focuses on the planning and design of the facility and the interaction of its components. Each component is added as models with different parameters and attributes which enables an early simulation of the facility. Thus, engineers can concentrate on the design of the facility and the process information and not of each component individually. The models of the components are defined using OPM (Object-

[2] http://www.energy.siemens.com/hq/de/automatisierung/stromuebertragung-verteilung/stationsleittechnik/sicam-1703/sicam-toolbox-II.htm

Process Methodology). As shown in Figure 5-2, virtual addresses and, again, the signals are defined.

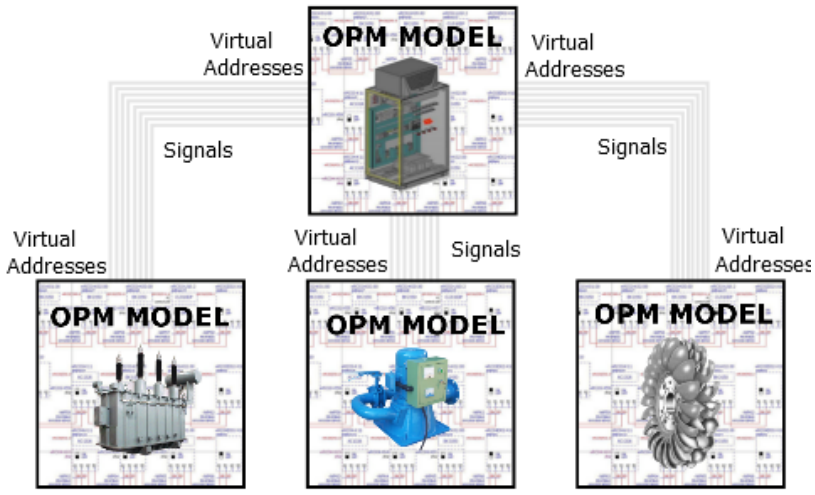

Figure 5-2 Symbolic OPM Diagram

Both tools provide an export of process data as signals. A signal transports information within a complex system and it represents a mechanical interface, an electrical signal or a software I/O variable. A signal consists of a set of attributes which are important for another tool and represents therefore the need of the integration. As intermediate model, the Virtual Common Data Model (VCDM) is used. It represents and structures the integrated data. Figure 5-3 highlights the structure of the extracted signals of the tools EPLAN and OPM and their mapping to the VCDM model. This shows that not all attributes are used or are available.

EPLAN	VCDM	OPM
signalNumber	kks0	GRP
Separator		
signalNumber	kks1	SYS
Separator		
signalNumber	kks2	EQP
Separator		
signalNumber	kks3	SIG
Funktionstext	longText	Longtext
	measureRangeStart	Y_0
	measureRangeEnd	Y_100
	measureUnit	UNIT
	textSwitchOn	Zustand_Ein
	region	CASDU1
plcAddress	componentNumber	Komp
dataType		
Symbolic Address		
	cpuNumber	BSE
	rackId	SSE
	dp	DP
	position	DP
	channelName	DP
	textSwitchOff	Zustand_Aus
	cat	LK_CAT

Figure 5-3 EPLAN, VCDM and OPM models

The information highlighted in Figure 5-3 is stored in form of an ontology in the Engineering Knowledge Base from which the TransformerIDE will load the needed information. Of course the structure information and the mappings are not enough for the creation of a workable and accurate transformer. To complete the information, each attribute is characterized with additional information which is often only implicit available. Figure 5-4 describes 3 attributes of each model with different complexity. Each attribute comes with a data type (String, Integer etc.), a default value and some restrictions such as not null, max value or a regex definition. The characterization is not limited to the values mentioned below, they can be dynamically extended with further properties like min value etc.

Attribute	Tool	Properties	
BSE	OPM	Mapping to VCDM	CPU Number
		Properties	Max value = 255 Min value = 0 Default value = 0 Data type = Integer
		Example	1,12,20 etc.
measureUnit	VCDM	Mapping to OPM/EPLAN	OPM = Unit EPLAN = na
		Properties	Not null = yes Default value = „“ Data type = String
		Example	kV
signalNumber	EPLAN	Mapping to VCDM	kks0-3
		Properties	Regex = ([A-Za-z0-9]+)\.([A-Za-z0-9]+)\.([A-Za-z0-9]+)\.([A-Za-z0-9]+) Not null = yes Data type =String
		Example	AEA00.CE001.EP000.XM00

Figure 5-4 Characterized Attributes

The evaluation of the TransformerIDE is carried out based on these tools and the related attributes.

5.2 Use Cases

The TransformerIDE is an approach to satisfy the requirements of the use case. But what are exactly those requirements derived from the hydropower plant construction application area. To illustrate them in a readable way, the different requirements are described in the form of user stories. The basic structure of a user story consists of a named role, a goal or desire and a benefit. The role used for the following user stories is John the engineer who wants a tool to develop transformers so that he can transform different signals from different engineering environments.

UCI To develop a Transformer, Ron the engineer would like ...

UCI-1 ... an Integrated Development Environment.

UCI-2 ... to use the knowledge stored in the EKB.

UCI-3 ... that the Transformer Language consists of functions to manipulate the source attributes.

UCII To receive a correct and complete Transformer, Ron the engineer would like …

UCII-1 ... to test the developed Transformer with real and test data.

UCII-2 ... that the IDE can verify with the EKB knowledge if a Transformer is correct or not.

UCIII To develop the Transformer in an efficient way, Ron the engineer would like …

UCIII-1 ... that the IDE builds as much parts of the Transformer automatically based on the knowledge from the EKB such as the frame based on the mappings between the source and target attributes.

UCIII-2 ... reuses as much parts as possible from a Transformer which must be updated due to a version change.

UCIII-3 ... that the IDE shows where a found error lies.

UCIII-4 ... that the IDE shows which source attributes are not used by the current Transformer.

UCIV To develop an executable and portable Transformer, Ron the engineer would like …

UCIV-1 ... to export the Transformer into executable code.

UCIV-2 ... that the Transformer can be exported into different programming languages such as java etc.

UCV To develop an understandable Transformer, Ron the engineer would like …

UCV-1 ... that the IDE visualize a Transformer in a structured and graphical way.

UCV-2 ... that a Transformer is created based on simple attribute manipulating functions.

UCVI To develop a maintainable Transformer, Ron the engineer would like …

UCVI-1 ... that the Transformer can be changed and updated.

UCVI-2 ... that the target and source model Version can be changed.

As a result of the uses stories, the requirements can be categorized in two types of requirements: the functional and non-functional requirements. The

following section tries to find an answer to the question if the TransformerIDE can satisfy the needs of John the engineer. Therefore the requirements are analyzed from different perspectives.

5.3 Requirement Fulfilment

The TransformerIDE is a web application developed according the Software Requirements Specification (SRS) specified in section 4.3 Requirements and implemented following the Software Design Description (SDD) of section 4.4 Design/Architecture. It consists of 5 main components: Editor; Validator; Transformer Manager; Code Generator; TransformerDB. The Editor is the central point in developing new transformations. It offers functions to create, edit, validate, test and export a transformation. With the Validator, a transformation can be verified according its correctness based on test data. If a transformation is validated, the Code Generator enables the creation of executable code (e.g. Java) which can then be used outside the tool. The Transformer Manager enables the management (e.g. updating, copying, etc.) of the transformer and their related transformations. The created and managed transformer is stored in the TransformerDB.

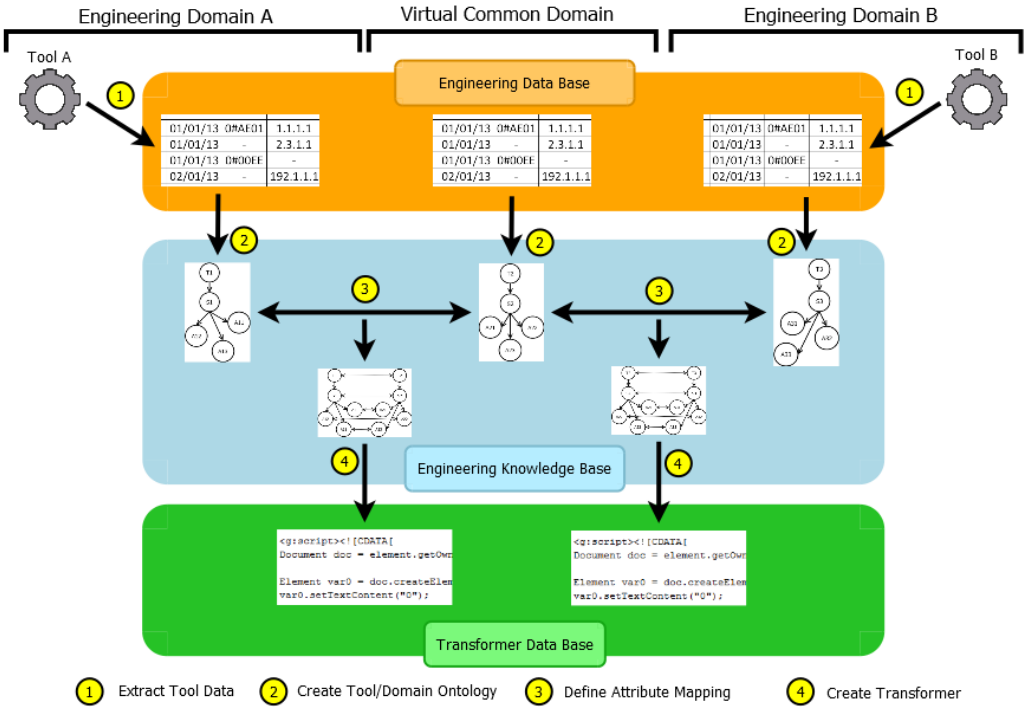

Figure 5-5 Use Case Scenario

The Use Case Scenario of Figure 5-5 consists of 4 steps. The first three comprises mainly the extraction of tool data from the tools such as EPLAN and OPM and the description of the data in form of ontologies and the mapping of the attributes between the tool signals and the Virtual Common Domain Model. These three steps are mainly the preliminary work which is necessary to enable the fourth step which is called Create Transformer. It consists of the following three sections Create, Update and Execute a Transformation.

Create a Transformation

The creation of a Transformation starts with the loading of the TransformerIDE. The start procedure loads the Transformer Manager which represents the entry point of the tool. It offers functions like creation of a Transformer, managing of already created Transformers and a general overview of the available Transformers and tool mappings. Before we design a Transformation, it is necessary to create a Transformer. A Transformer represents a container of mapping information and logic in form of Transformations to enable a transformation from a source to a target model (tool) e.g. EPLAN to VCDM. As highlighted in Figure 5-6, a Transformer consists of a name, a description, a status and mappings. With the status flag, a Transformer can be set as inactive if it is not needed anymore. The mapping area in green color represents the information loaded from the EKB. It consists of a source model and a target model with its corresponding attributes, the associated mappings and the implicit engineering knowledge. The frame of the Transformer is loaded and created automatically.

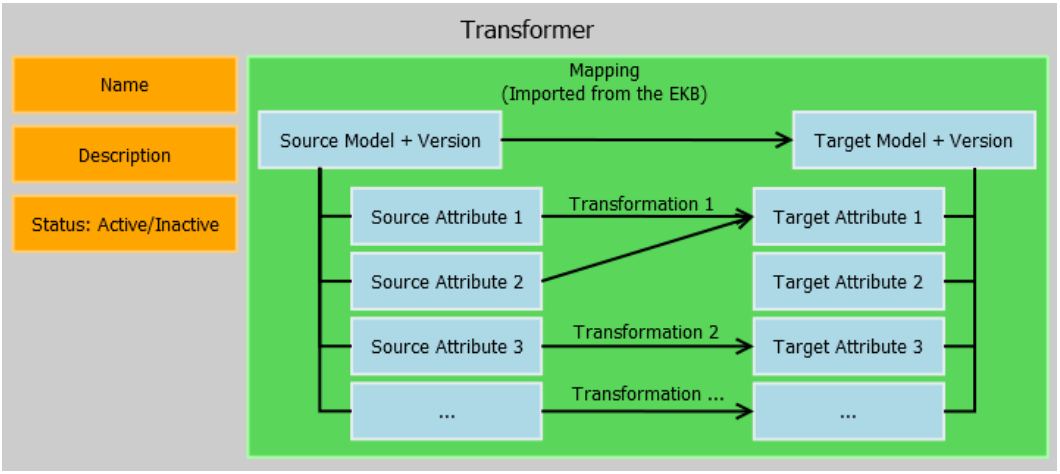

Figure 5-6 Transformer Structure

The challenging part is the creation of the Transformations which consist of the logic to convert the data from multiple source attributes to a target attribute. One Target attribute can have multiple source attributes but a source attribute can have only one target attribute.

In the Transformer Manager (Figure 5-7), we select the mapping *EPLAN V1.0 → VCDM V1.0* from the mappings table (label 1) and click on the button *Create new Transformer* (label 2). The new Transformer shows up in the Transformer table (label 3) with the name *New Transformer*. On the right side in the Transformer-Info area an overview with some detailed information about the Transformer is given.

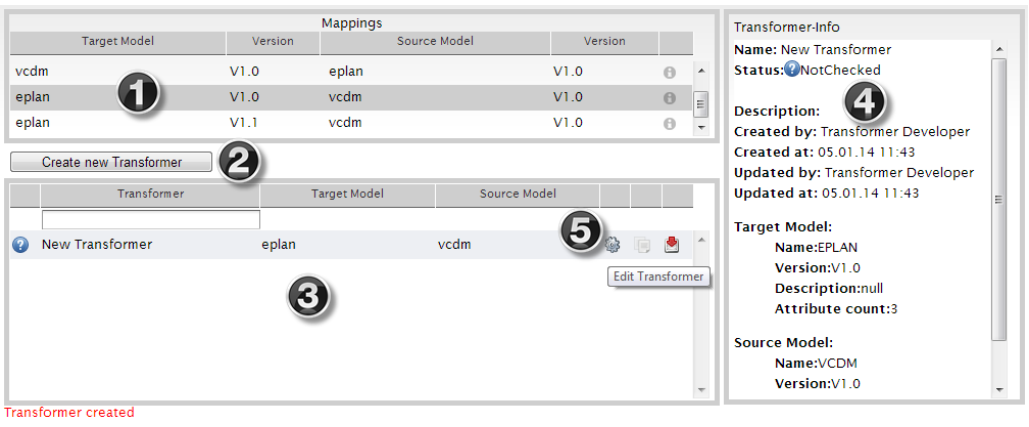

Figure 5-7 Transformer Manager

The status says NotChecked, which means that the Transformer has not been validated. The validation will be done automatically the first time the Transformer is open. Next, we will open the Transformer by clicking on the *Edit Transformer* icon (label 5). Beside the Edit function, a Transformer can also be copied or downloaded.

With the opening of the Transformer the Detail Transformer View (Figure 5-8), is loaded. It represents the main view of a Transformer with all the related information and functions. In the top area (label1), the Transformer name can be edited, a description can be added and the status can be set to inactive. The middle area (label 2) consists of information such as the status, target and source model and a drop-down box where the versions of the mapping can be selected. With this function a switch to a newer mapping can be done easily based on the information stored in the EKB. All changes are applied to the whole Transformer which includes also the attribute mappings. One should be aware that a change of the mapping version will lead to a Transformer which might not work anymore because of the changed information. The Validator will highlight the problem areas.

The status of the current Transformer is Incorrect. This means that the Transformer could not be validated because one of the Transformations is not working properly. When we take a look at the table of the attribute mappings (label 4), we can see that every mapping, further called Transformation, has the status Incorrect. As we have not created till now any of them, this is a normal status. With the function *Generate report* (label 3), a detailed printout of the Transformer, including general and detailed information about the transformations and, if applicable, the error messages estimated by the Validator, are created.

Next thing, we will finally create our first Transformation. To do so, we need to select first one of the available source attribute sets showed by the drop-down box. There are more than one set if a target attribute can be created based on different sets of source attributes. For a Transformer, only one set per time can be active. This requires that the developer selects the preferred one. Since for

this Transformer only one set is available, we take the current selection and make the next step by clicking on the button *Edit Transformation* (label 5) beside the Transformation *signalNumber → kks0, kks1, kks2, kks3*.

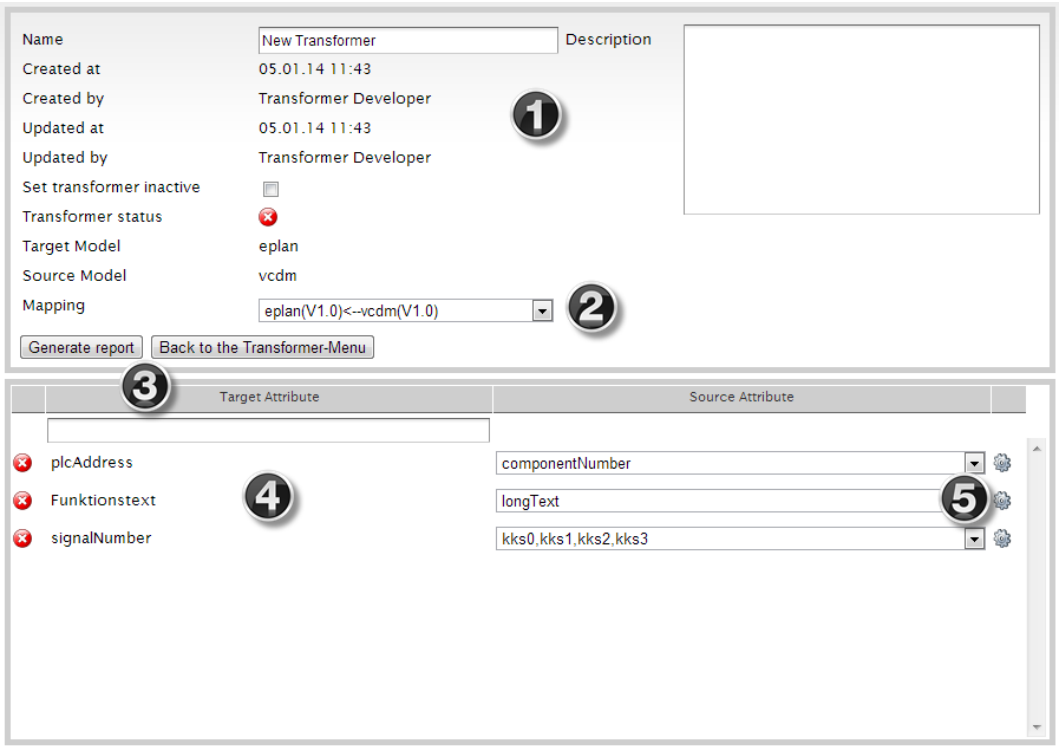

Figure 5-8 Detail Transformer View

The Editor (Figure 5-9) consists of three areas: editor area; testdata area; validator area. In the editor area, a Transformation is created by combining the attributes with the available and, if needed, extensible functions. The testdata area offers the possibility to add test data which can then be used by the Validator to validate the developed Transformation. The validation area shows a detailed description of the problems found by the Validator in a structured way.

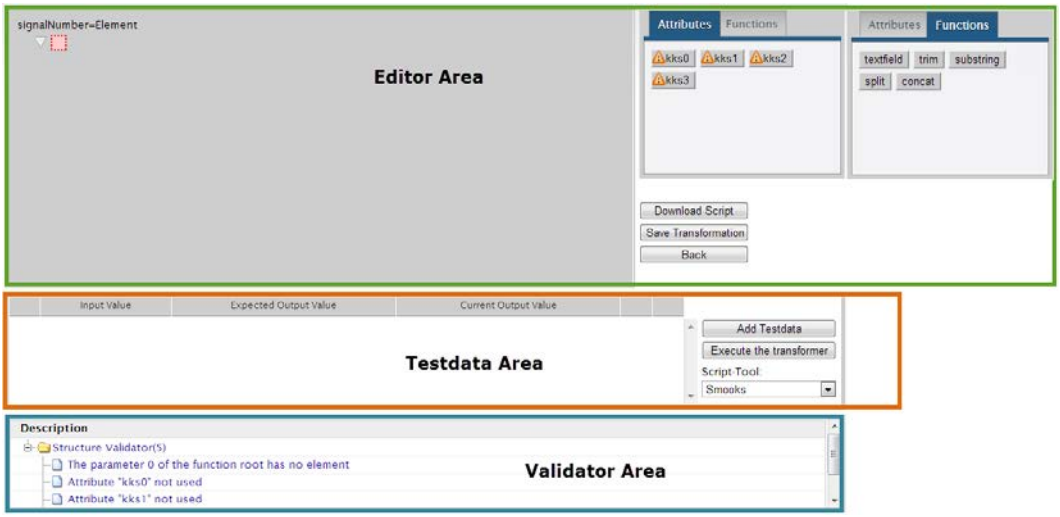

Figure 5-9 Transformation Editor

As the Transformation is not created, the following error messages shows up:

- The parameter 0 of the function root has no element
- Attribute "kks0" not used
- Attribute "..." not used

The first error message says that the Transformation has a parameter which has no input set. By clicking on the error message, a red flag indicates in the editor area to which parameter the error message is related. The other messages indicate that no source attribute is used. This can be also seen by the exclamation mark near the attributes on the left side. Based on this information loaded from the EKB, we know that the target attribute is a combination of the four attributes kks0-3. Of course, this information is not enough to create a correct Transformation but let us start with a first draft where we combine each of the four attributes into one string.

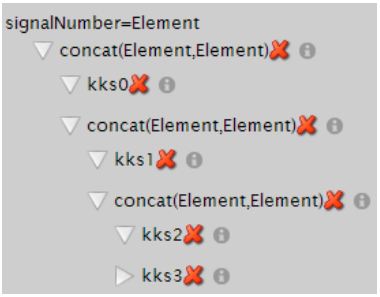

Figure 5-10 First Transformation

First, we need to click in the editor area into the red dotted box below the signalNumber=Element. Element indicates that the function signalNumber=Element needs one parameter. Now we need to select the function Concat to combine the fields. By clicking on the button with the label Concat, the function is added. We can see that Concat has two parameters. As we have 4 attributes, the first parameter will be the attribute kks0 and the second again the function Concat. Following this approach, we create the nested structure showed in Figure 5-10. When we take a look at the Validator output, we can see that the error messages have vanished and only the message No Testdata found shows up. From a validation point of view, a Transformation can be only correct if it is tested with real data.

To enter testdata, we click on the button *Add Testdata* in the testdata area. This will open the following dialog showed in Figure 5-11. The dialog consists of text fields to enter the test data for the target attribute and the source attribute independent of the number source attributes. The info icon beside each text field gives, by pointing the mouse on it, the restrictions of the related attribute. In this example, the attributes kks0-3 have no restrictions and are therefore a simple string. The target attribute however has a StringRestriction in the form of a regex expression.

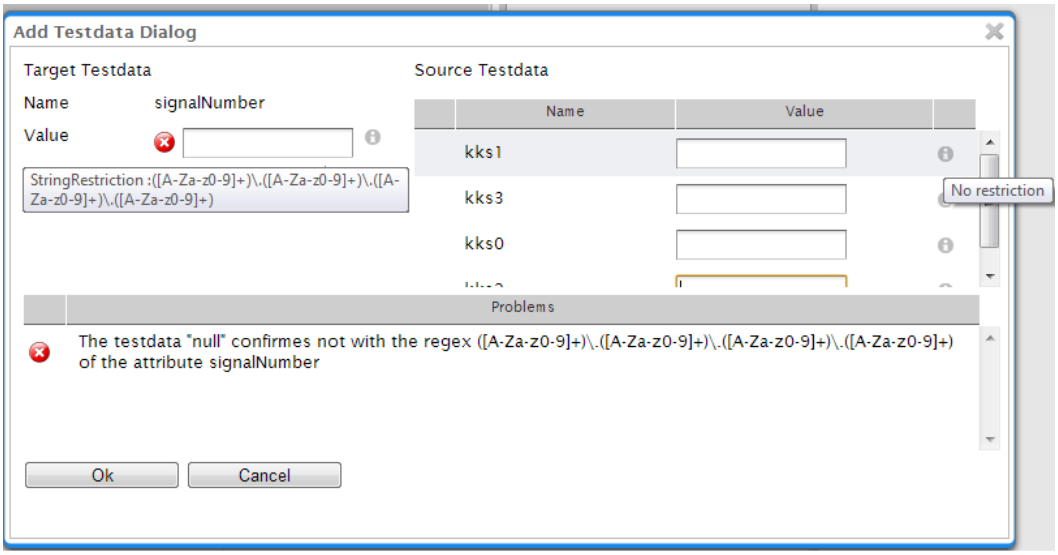

Figure 5-11 Testdata Dialog

Beside the text fields, the dialog consists also of a validation area like the one available for the transformation. This Validator verifies if the entered data corresponds to the rules and restrictions of the appropriate attribute. Currently, one message displays: “The testdata “null” confirms ...”. This problem indicates that the current entry of the target attribute is not valid due to the fact that no entry has been done till now. When we enter the following values, we can see that no problem displays and we can save it.

- signalNumber = 01.MEA12.AA101.S01
- kks0 = 01
- kks1 = MEA12
- kks2 = AA101
- kks3 = S01

After the dialog has closed, the created transformation is executed with the entered testdata. The testdata table has 3 columns. Input Value and Expected Output Value contain the entered testdata. The column Current Output Value contains the result estimated by the execution of the Transformation with the testdata as input. For our example, the result is correct. This is indicated by a red cross icon near the row of the testdata and the error message “Result matches

not the testdata". When we compare the testdata and the calculated result, we miss the separator "." between the source attributes.

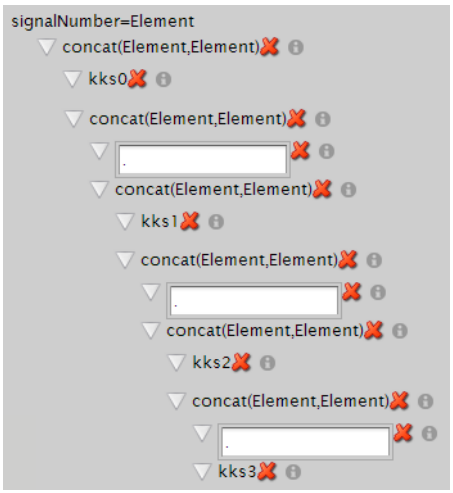

Figure 5-12 Correct Transformation

Following the new information gathered from the test result, we need to change the Transformation into the following structure shown in Figure 5-12. As expected. the Validator comes to the right result and prints out no error. Therefore, the created Transformation is the correct one and it is now ready to be used. For a working Transformer nevertheless, also the other Transformations must be created.

In this section, we showed how a transformer developer can create a Transformer and the related Transformations by using the TransformerIDE. Following steps were highlighted: create a Transformer; configure a Transformer; create a Transformation; enter testdata; validate a Transformation. The functions offered by the TransformerIDE enables the creation of a Transformation in a supporting way. This is possible due to the fact that the Transformation creation process is imbedded in the corset of the information loaded from the EKB. The challenges starts were the implicit engineering knowledge is not enough anymore. At this moment, the developer must be creative to develop a workable structure. Implicit knowledge and real data are used as reference point. When a Transformation is created, the Validator can use the real data to validate the correctness. This requires, of course, that enough testdata is available and that

the testdata is correct. This leads to the conclusion that a Transformation/Transformer can only be as good and correct as the information stored in the EKB is complete. The TransformerIDE fulfils the requirements to create a Transformation/Transformer because it bundles and offers all the needed functions and information to create a workable Transformation.

Update a Transformation

In the last section, we have shown how to create a Transformation with the TransformerIDE. In this section we will show how the TransformerIDE supports the developer when a tool of the Transformer must be updated due to e.g. a new version etc. The following scenarios will be considered:

- New mapping added
- Remove of a mapping
- Change of a mapping
    - Attribute removed
    - Attribute added
    - Restriction changed

The scenarios are typical changes which can occur over time and where the developer must react.

The attributes of a tool model needs sometimes to be changed due to changing versions of the tools. Such a change is entered as a model with a higher version and a mapping in the EKB. As soon as this is done, the TransformerIDE also has access to the new model and the developer can start to create the Transformation. The first possible change is the extension of the model with a new attribute. The model EPLAN V1.0 from the section before consists of three Transformations. EPLAN is enhanced with the attribute dataType. As VCDM has also an attribute called dataType a new mapping is created which maps dataType to dataType. To apply these changes to the created Transformer with the version 1.0, the TransformerIDE offers two possibilities. The first consists of the complete new creation of the Transformation following the process described in the section before. This is, of course, complicated and makes only double work

as we have already created a complete Transformer. To reuse the old Transformer and Transformations, the TransformerIDE offers the possibility to change the mapping version of a Transformer. This can be done in the Detail Transformer View (Figure 5-8) by using the drop-down box called Mapping. There, you need to select the right version. As soon as the mapping has changed also the mapping entries of the Transformer are updated.

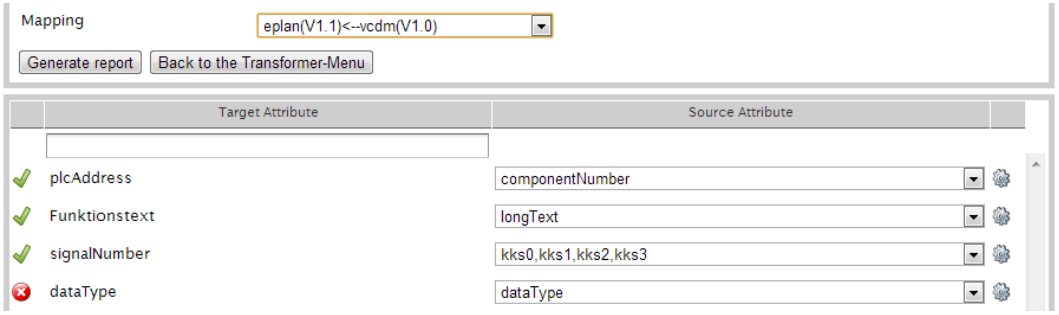

Figure 5-13 Add a new attribute

As showed in Figure 5-13, new entries in the table for dataType can be found. At this moment, the Transformer is not working as the mapping dataType → dataType is not configured.

In this section, we saw that the TransformerIDE offers the possibility to reuse Transformer if a version has changed. The worst case is, if the tool changes completely for this, a complete new Transformer must be created and the functions offered by the IDE bring then no benefit. Nevertheless and based on the fact that a change does rarely change the complete tool model, the old structure can be reused partially.

Execute a Transformation

The final phase of a Transformer is its execution. To do so, the Transformer must be exported from its current structure based on the Transformation Meta Language into an executable code. The Transformation Meta Language is an intermediate structure which is focused on the easy development of new Transformers and their visualization. Nevertheless, as it cannot be executed, a created Transformer must be transformed into any executable code. The architecture of the TransformerIDE supports the use of multiple types of

executable code. In the current environment of the TransformerIDE, each TransformationType has a representative based on Groovy code. This means that a Transformer can be executed as a Groovy script.

To export a Transformer, it is necessary to open the Transformer Manager, which is, as mentioned before, the main entry point of the TransformerIDE. The manager lists all available transformers. Figure 5-14 shows an extract of the Transformer Manager. For this example, the Transformer with the name Workable Transformer is selected.

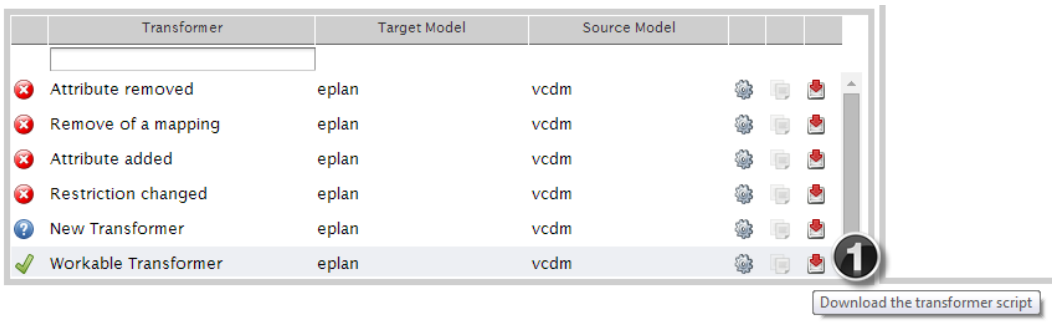

Figure 5-14 Export a Transformer

Number 1 highlights the export button a transformer. By clicking on it, a dialog shown in Figure 5-15 opens and gives the possibility to select a script tool which is Groovy in our case.

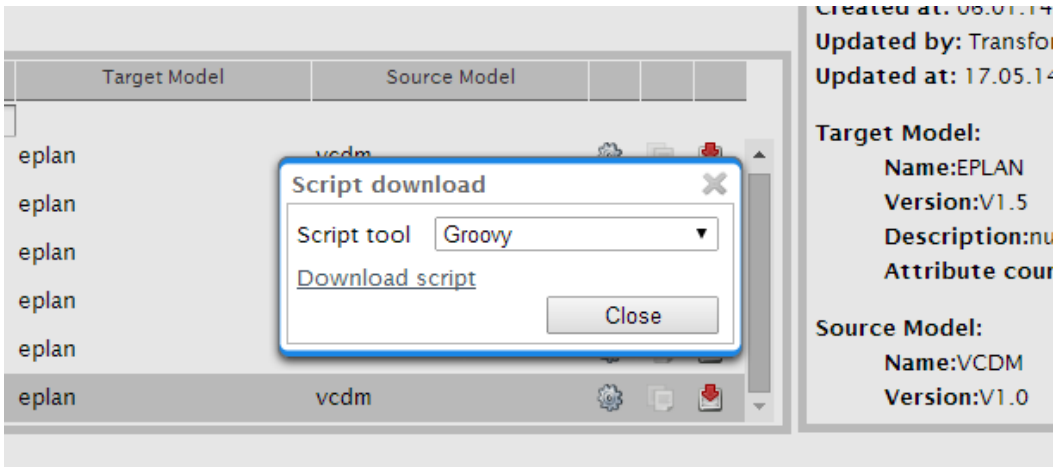

Figure 5-15 Download Script

The next step is to click on the function Download script which will create, based on the selected Transformer, the executable source code and put it into a zip file. The export consists of the following files.

- Funktionstext.xml
- plcAddress.xml
- signalNumber.xml
- Workable Transformer.txt

The first 3 files which have the ending xml contain the source code of the Groovy script. The last file is a description of the Transformer and the files which are part of the export.

```
<?xml version="1.0"?>
<smooks-resource-list xmlns="http://www.milyn.org/xsd/smooks-1.1.xsd"
xmlns:jb="http://www.milyn.org/xsd/smooks/javabean-1.1.xsd"
xmlns:g="http://www.milyn.org/xsd/smooks/groovy-1.1.xsd">
<g:groovy executeOnElement="csv-record" executeBefore="true">
<g:script><![CDATA[
Document doc = element.getOwnerDocument();

Element var0 = doc.createElement("var0");
var0.setTextContent(DomUtils.getElement(element,"kks0",1).getTextContent());

Element var1 = doc.createElement("var1");
var1.setTextContent(".");

Element var2 = doc.createElement("var2");
var2.setTextContent(DomUtils.getElement(element,"kks1",1).getTextContent());
```

Figure 5-16 Groovy Code

Figure 5-16 shows the first lines of the signalNumber Transformation. The script itself expects as input a csv-record with the different attributes of a signal. Based on the Transformation structure, it transforms the input attributes into the defined output attribute.

In this section, the export function of a workable transformer with all its requirements has been highlighted. The current environment has a clearly defined interface to the transformation, which is not fully developed but, as this was not the focus of the master thesis, this may be the subject of future research.

Fulfillment Grade

Effectiveness: “The capability of the software product to enable users to achieve specific goals with accuracy and completeness in a specified context of use”[34]

As highlighted by Figure 3-1, a Transformer undergoes three phases during which it is created (updated), tested and executed. The use cases highlighted in the section 5.2 define those 3 phases with all the needed functions and features. If the use cases are fulfilled and all the requested features and functions implemented, this is a question answered by this section. Non-functional requirements required by the research issues RI-2 and RI3 are not part of this section.

Chapter 5.3 shows how the different functions and features of the TransformerIDE can be used to create, update and execute a Transformer. Based on this information, Table 1 summarizes which use cases are fulfilled by the functions and features of the TransformerIDE. A use case can therefore be fulfilled, not fulfilled and partially fulfilled.

Use Case	Y	N	P
UC-I: Support the development of a Transformer			
UCI-1: Integrated Development Environment	X		
UCI-2: Interface to the EKB	X		
UCI-3: Offers a Transformer Language to create Transformer	X		
UCII: Create a correct and complete Transformer			
UCII-1: Test the Transformer with real and test data	X		
UCII-2: Verification of the correctness based on data from the EKB.	X		
UCIII: Create efficiently a Transformer			
UCIII-1: Automatic creation of the Transformer frame based on the EKB knowledge	X		
UCIII-2: Reuse of already created Transformer	X		
UCIII-3: Visualization and highlighting of produced errors	X		
UCIII-4: Visualization and highlighting of unused source attributes	X		
UCIV: Create executable and portable Transformer			

UCIV-1: Export of created transformer into executable code	X		
UCIV-2: Export of created transformer into different types of executable code	X		
UCV: Create understandable Transformer			
UCV-1: Graphical visualization of a Transformer	X		
UCV-2: Transformer is created based on simple manipulation functions	X		
UCVI: Create maintainable Transformer			
UCVI-1: Transformer can be changed and updated	X		
UCVI-2: Change of the target and source model version	X		

Table 1 Use Case fulfilment table

Table 1 shows 15 use cases. Those use cases demand e.g. an IDE, a graphical visualization of a Transformer, the execution of a Transformer, an interface to the EKB etc. Those are partially basic functions and functions that are nice to have. The basic functions are essential to manage a transformer through the whole life cycle and the nice to have functions, if not available, do not harm or hinder the creation of a transformer. A basic function is e.g. described by the use case UCIV-1 which states that a transformer must be exported into executable code. A nice to have function is e.g. the use case UCV-1 which requests that a Transformer is graphical visualized.

The result of the fulfilment evaluation is 15 out of 15. This means that all use cases and the derived there from functions are implemented and usable to create, update and execute a transformer.

5.4 Script Language vs. TransformerIDE

> Efficiency: “The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions”[34]

A Transformer represents a bundle of logic to transform data from on format into another. One approach to develop such a Transformer is the TransformerIDE with all its functions and the Transformation Meta Language as basis. The IDE prescribes the development process and the requirements, and the Meta Language the possibilities to create a Transformation. Other approaches such as

the development of a Transformer from scratch based on a computer language do not dictate such a tight corset of rules. Both approaches have their advantages and disadvantages. To highlight them, in this section, these two approaches are compared. To do so in a feasible way, four exercises are defined, with regard to typical tasks and situations, derived from the Use Cases. By executing the exercises, the two processes are presented and analyzed while taking their respective complex situations into consideration. As a final criterion, the Transformer also, resulting from the exercises, is evaluated in respect of its structure and readability.

Comparison Basis

To compare the TransformerIDE with the development approach of a Transformer from scratch, a common basis must be established before meaningful statements can be made. Both approaches have in common that they try to achieve the same result, which is a set of workable transformations. These transformations tied together represent a Transformer. Taking this as a starting point, the requirements for the comparison basis can be rolled up from behind. The first requirement is the need that the resulting Transformations have the same computer language. For this reason, the framework Smooks [39] with the integrated object-oriented script language Groovy is used. The TransformerIDE supports the export of the Transformer into this format and the script language approach is developed based on this framework. The next requirement is the knowledge needed to be able to create a Transformer with the TransformerIDE or based on the script language. E.g. the script language software development knowledge is needed, which, in contrast to the TransformerIDE, is not needed in this extent. For possible comparison, the needed knowledge is not a criteria and it is presupposed. The last requirement is a common set of test data, tool model definitions and the tool model mapping information.

The use cases listed in the section 5.2 describe typical scenarios and requirements of the process to create a Transformer. Derived from this use cases, four exercises are defined; these exercises cover fours main tasks of the transformation life cycle (see Figure 3-1): creating and updating, testing and the deployment of a Transformer.

- Exercise 1 – Create a Transformer and one correct Transformation

  The goal of this exercise is it to create a Transformer based on the mapping and tool model information and one of the Transformations. The test data and the tool model information prove if the created Transformation is working correctly or not.

- Exercise 2 – Finalize the Transformer

  After exercise 1, the Transformer frame and one of the Transformations are finished. The next step is the finalization of the Transformer, which means that all remaining Transformations need to be developed and tested.

- Exercise 3 – Deploy the Transformer

  The working Transformer can now be deployed as executable code.

- Exercise 4 – Change the version of a Transformer

  Due to a change of target tool model the created Transformer must be updated. The target of this exercise is it to create a Transformer which works with the changed tool model.

The four exercises and the requirements for the comparison are now defined. The next step is it to state what should be evaluated. The main focus lies on the Transformer creation process, the differences between the two approaches and the complex situations in each process. In general, a complex situation is a situation where the tester is confronted with a challenging moment. This can be e.g. a step where an error can be made easily, or where the tool or the process does not clearly indicate what must be done next. Additionally to the process analysis, the created Transformer itself is evaluated in respect of the complexity and size of the Transformations.

Script Language

The first approach evaluated is the script language approach. Following the test setup defined in the chapter Comparison Basis Smooks, with the script language, Groovy is used to execute the four exercises. Smooks is a framework which enables the transformation of data from one format into another. The typical input data format is e.g. CSV. Smooks integrates Groovy which enables

the implementation of transformation logic during the import process. Before the execution of the exercises can start a basic Transformer environment must be set up. This environment consists of a main application class, a definition of the import file and a basic frame for each Transformation. The main application class has the task to import the test data, execute the transformations and visualize the result. The import file of the test data will be a CSV file which contains for each test data set an own row. Finally, the frame for a Transformation must be defined so that the Transformations created during the exercises have a specified structure with clear interfaces. A Transformation expects a XML string which can be processed. With the setup of the Transformer environment, everything is in place for the four exercises.

Exercise 1 consists of the creation of a Transformer and one correct Transformation. The first step is it to understand the structure of the two tool models (the model mappings, the test data) and all other information provided. This information is exported from EKB as it represents the main knowledge pool for the tool information. As soon as a common understating of the test environment is gathered, the import file as a CSV file can be created. Here, it is important to make sure that the correct separator file is used. Next, the csv reader which represents the import logic must be adjusted to fit the csv file, which means mainly that the correct column name is assigned to the right column. A first test run to see if the information is imported correctly can be done.

With a working import routine, the mapping in the Transformation frame can be set up. The Transformation frame defines that the input and output format of a Transformation must be XML. The task of the mapping setup is to determine which source attribute is transformed into which target attribute. In general, a mapping consists of one target attribute and one or multiple source attributes. Here, it is important to make sure that the attributes are corresponding to the mapping definitions. Next thing, the logic of the Transformation is implemented. To process the import data, each Transformation uses the Document Object Model (DOM) interface. DOM enables the simple manipulation of the XML structure and the values. The first target attribute is selected is the attribute plcAddress. According the model definition and the mapping information

plcAddress is a simple String value which has no value restriction. The source attribute is componentNumber, which also has no value restriction. Following this information, the logic to be developed is a simple value assignment. Now, the line to assign the value from one attribute to another is implemented. The next step is to verify if the Transformation works correctly. To do so, the transformation is executed with the test data. The result is then compared with test data set. This is a complete manual task and it requires that the tester proceeds accurately. If the testing result is not correct, the Transformation must be adjusted until it gives finally the correct result.

As a result of exercise 1, a complete frame of the Transformer with all its mappings is provided. This means that, in order to accomplish exercise 2, the missing transformation logic of the remaining target attributes funktionstext and signalNumber must be developed. Before the development can start again, the definitions of the model attributes with their mappings must be understood and be clear. As soon as this is done, the first of the missing Transformations with the target attribute funktionstext can be developed. Funktionstext is again, like plcAddress, a String with no value restriction. The source attribute is longText and therefore a simple value assignment can be set up. The second attribute signalNumber has four source attributes called kks0-4 and it consists of a format restriction defined as a regex ([A-Za-z0-9]+)\.([A-Za-z0-9]+)\.([A-Za-z0-9]+)\.([A-Za-z0-9]+). The restriction indicates that the value must consist of 4 parts composed of characters and numbers. Each part is separated by the separator dot. Since the mapping has four source attributes and the target attribute consists of four parts, it is obvious that those four attributes represents the four parts. The only question is the order of the attributes. This information can be gathered by checking the test data. Once equipped with this knowledge, the program logic can be developed by stitching together the four parts with dot as separator in the right order. The final step is the testing of the new Transformations with the test data.

Exercise 3 requests the deployment of the Transformer as executable code. Since the Transformer was developed with a script language, the code is in the

current structure and format already executable. Therefore, no additional action is needed and the Transformer can be deployed in this form.

Till now, a new Transformer was developed and deployed. In the last exercise, the target model EPLAN is changed from V1.0 to V1.1. This means that a new version of the Transformer must be created as well. Again, the first step is the study of the model information. Additionally, the model definition of V1.0 and V1.1 is compared to verify the changes occurred in the model. The change includes a new target attribute type. This means that most parts of the Transformer can be reused. The adjustment consists of the update of the csv reader, the implementation of the mapping and the transformation logic and the final testing.

All four exercises have been successfully completed with the script language approach. The process consists of the setup and update of the input file, the input reader, the Transformer frame, the implementation of the logic and the testing of the Transformations. During the execution, three complex situations were identified. The first one is the requirement of understanding of the model and mapping information so well that the different Transformer parts such as the import reader or the transformation logic can be built. The second situation is the creation of the transformation logic. This is development and represents the creative part of the whole process. It requires that the test data and the model information be understood in the right way and interpreted so that a correct Transformation can be developed. Additionally, a good understanding of the scripting language is required also to use the functions offered appropriately. The third complex situation is the testing itself as it represents a manual task i.e. a task with high risks of errors.

TransformerIDE

The TransformerIDE, as proposed by this master thesis, represents the second approach evaluated and compared. The script language represents a development from scratch as it is done completely without any tool support. The TransformerIDE, on the other hand, offers a set of composed functions which automate and support the developer through the whole life cycle of a Transformer. Once again, for the execution of the exercises, the same

information, such as the test data and the model information, is available. In contrast to the first approach, the TransformerIDE accesses the model and mapping information directly via the interface to the EKB.

The start screen of the TransformerIDE is the Transformer Manager. It represents the entry point for exercise 1. Following the description of the exercise, a Transformer between the tool models VCDM V1.0 and EPLAN V1.0 should be created. As the TransformerIDE accesses the EKB, the Mapping Table contains the mapping information between the two tool models. By selecting the preferred mapping and the use of the Create Transformer button, a new Transformer is created based on the tool model and mapping information. Here, it is important to select the correct mapping. Verification may be done by checking the information provided by the EKB beforehand. Next, the new created Transformer can be opened. The new Transformer has a default name which can be changed in the Transformer View. The current status of the Transformer is Incorrect as only the Frame of the Transformer and the Transformation was created automatically but the logic is missing. All target attributes with their source attribute are listed and represent one Transformation. Exercise 1 demand the creation of the Transformation for the target attribute plcAddress. The Transformer Editor for this Transformation is opened by clicking on the Edit button. The Editor view consists of an editor area where the Transformation can be developed and a list of source attributes and functions which can be used for the development. Before the development can start, the available test data must be added. The Test Data dialog requires for each target and source attribute a valid value which is automatically checked based on the information provided by the EKB. The attributes plcAddress and componentNumber have no restriction as they are simple String values. Following this information, we have already enough information to understand that the mapping is a simple value mapping so that no real logic must be implemented. By clicking the attribute componentNumber, it is assigned directly to the target attribute. The Transformation is automatically validated against the test data and the status set to valid by closing the mapping. This having been done, the first exercise is accomplished.

The next exercise is exercise 2. It consists of the finalization of the remaining transformations. Again, the starting screen is the Transformation Manager which means the right Transformation must be selected in order to update the Transformations. The Transformation with the target attribute functionsText is a simple value assignment and can therefore be created based on the same steps as during exercise 1. Of a somewhat different nature is the target attribute mapping signalNumber. This mapping consists of multiple source attributes and a format restriction. From a process perspective, the Transformation must be selected and the test data added again. As mentioned before, the attribute signalNumber has a restriction which is defined as a regex. The regex restriction restricts the format of the attribute to a set of four strings joint by a dot. To understand in which order the source attributes must be assigned the test data must be verified itself as it reflects the order. With this information and with the function concat the source attributes can be merged to a single string. The automatic validation of the TransformerIDE gives immediate feedback, if – based on the test data – the Transformer is correct or incorrect. Now, the Transformer is complete and, by going back to the Transformer View, the status should be valid.

The Transformer is valid and, therefore, it can be deployed as required by exercise 3. To deploy a Transformer, the TransformerIDE offers an export function. This function allows the transformation of Transformation Meta Language code into an executable computer language. The script language Groovy is used for this example. The export is a bundle of 3 Transformation files and a description of the Transformer.

Exercise 4 indicates that the EPLAN Model has changed from V1.0 to V1.1. This change does impacts the created Transformer also, which means that the next task requires the creation of a new Transformer based on the new model information. The TransformerIDE offers the possibility to reuse the Transformer already created by copying it. As soon as this is done, the copied Transformer can be opened. The current Transformer has still the old Model version and this can be changed by selecting the right mapping in the Mapping drop-down box. With the change of the mapping version, the complete Transformer structure is updated. All Transformations still valid are reutilized and all new ones are

created. According the Transformer View, the change consists of a new target attribute with the name type; all other Transformations are not impacted. To complete the Transformation, the new mapping must be developed by following the process highlighted in exercise 1.

With the TransformerIDE, all four exercises could be successfully accomplished. The process, given by the TransformerIDE, consists mainly of the development of the Transformations; all other steps, such as the setup of the Transformer frame and the testing, are done automatically. During the execution of the exercises, only the development of the Transformation was identified as a complex situation as it requires the understanding of the model structure, the attribute restrictions and the test data and creativity.

Summary

Both approaches, the TransformerIDE and the script language approach, allow the creation of a Transformer based on the available model and mapping information. The script language approach does not need the use of any tool support whereas the TransformerIDE offers a set of functions to support the development process. This main difference is also reflected in the two development processes. A developer using the script language approach must make each step manually, such as the setup of the import reader, the mappings, the testing etc. In the TransformerIDE, those steps are automatic, which means that the only real task under the responsibility of the developer is the development of the transformation logic itself. Both approaches have also in common that the most complex situation is the development of the Transformation itself. The reason is that it requires a good understanding of the model and mapping information, test data and knowledge about the use the functions of the approach. It requires also some creativity as not always all necessary information is available. As both approaches require understanding of the available information, the TransformerIDE pre-process the information and offers the information to the developer during the step where it is needed in a readable way. Both approaches offer also the possibility to reuse already developed Transformers if a version change occurs. By using the script language approach, the complete Transformer can be reused. The challenging situation is

finding all the changes and making sure that they are considered. The TransformerIDE, on the other hand, identifies automatically the changes and reuses only the Transformations where no changes to the source attribute occurred. The downside of this approach is that Transformations which could be reused partially must be developed again. The last aspect which should be considered is the created source code which, as required by exercise 3, is deployed. For the Transformations consisting of a simple value assignment, the differences between the two generated source codes are very small. This changes when the logic of the Transformation is more complex, such as e.g. the combination of multiple input values. Figure 5-17 shows an extract of the Transformation of the target attribute signalNumber with its source attributes kks0-3.

```
Document doc = element.getOwnerDocument();

Element var0 = doc.createElement("var0");
var0.setTextContent(DomUtils.getElement(element,"kks0",1).getTextContent());

Element var1 = doc.createElement("var1");
var1.setTextContent(".");

Element var2 = doc.createElement("var2");
var2.setTextContent(DomUtils.getElement(element,"kks1",1).getTextContent());
```

Figure 5-17 A part of the signalNumber Transformation

The script contains multiple intermediate variables which transport the values of the source attribute to the right function. If, instead, the script language is used, such intermediate variables are not needed. As a result from these circumstances, the difference between the two approaches is that, by using the script language as an approach, the Transformation can be optimized according the requirements and, by using the TransformerIDE, the source code can only be as good as the export function of the used computer logic implemented.

5.5 The Usability of the TransformerIDE

The proposed solution of this master thesis consists of the TransformerIDE and the underlying Transformation Meta Language. Whereas the target of the research issue RI-2 was to discuss the effectiveness and efficiency of this approach, it is the target of this section to show that it fulfils the quality criteria

learnability and understandability. The section is split into two parts. The first is Test Method which explains the approach necessary to prove usability. The second part, called Test Result, highlights the result of the tests.

Test Method

> Understandability: “The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use” [34]

> Learnability: “The capability of the software product to enable the user to learn its application”

Learnability and understandability are the two quality criteria defined in the ISO/IEC 9126-1 standard [34] which is analysed and evaluated to show that the TransformerIDE represents a usable solution for the application scenario. To prove this, ten test users are asked to execute four prepared exercises. The exercises demands from the test user to create, update, and deploy a Transformer. Understandability and learnability require different measured values. Understandability, following the definition, calls for a proof that the targeted user can use the application to achieve the intended task in a suitable way. To prove this, during the observation, it is noted if an exercise was executed successfully and how many complex situation have occurred. Learnability proves if a user can improve the use of the application in a short period of time. Ensuring that the exercises consist of similar tasks which, based on the noted execution time and count of complex situations, can be compared.

Test Audience

The TransformerIDE represents a solution which tries to simplify the process of creating a Transformer and to support the user. One of the targets was to enable non-experts, not familiar with the application domain, also to use the application to create a Transformer. Therefore, the test audiences are users which have good computer knowledge in general and commands of basic software development knowledge. Ten test users took part in the test. They were selected from a pool of persons in my environment: students, software engineers and hobby software developers. As the test was anonym, no names were documented.

Test Setting and Exercises

To show that the TransformerIDE represents a usable solution, the test setting consists of different exercises which must be executed by test users. The tests are done in a controlled environment, which means that, for this purpose, only the test user and the tester were in the test rooms. A Windows 7 machine, with English set as default, is available for the exercise. The machine consists of a keyboard and a mouse and is not connected to the internet to prevent that the user tries to search online for a solution. As the TransformerIDE is a web application, the web browser Chrome is used for the test. When an exercise starts, the web browser is open with the TransformerIDE as start page. Besides, each user receives a description of the exercises, a pencil, a writing block and the test data. The tools of the tester are a stopwatch, an exercise protocol and a pencil.

Before a test user can start with the execution of the test, each one of them receives a short introduction of the tool, the background and target of the application. At the end the process, of the test is explained. If the user has any questions, these also are answered, of course. The whole presentation does not take longer than 15 minutes.

The test consists of four exercises. Each exercise asks the user to use one or many functions of application and, in so doing, test all the user requirements. An exercise consists also of different parts which help the more granular time keeping and distinguishing between the steps to accomplish an exercise. The first exercise called “Create a Transformer and one correct Transformation” demand that a specific Transformer, presented in the exercise description, be created. The second exercise “Finalize the already started Transformer” invites the test user to finalize the remaining Transformations of the created Transformer. The result of the two exercises is a workable Transformer. The next exercise is the export of the created transformer. The final exercise consists of the scenario that the one tool model of the created Transformer has changed from version 1.0 to version 1.1. The test user must reuse the created Transformer and adapt it according to the changes in the new version. Exercise one, two and

four consists of similar tasks, which enables the comparison and the identification of any improvement.

For each exercise, an ideal path is defined before the tests are executed. This path is not communicated to the user but it is used as a reference to show if the path was followed or if the user has taken another path. The used test data is derived from the application scenario and is similar to the test data used in section 5.1 Environment and Test Data Initiation.

Measurement

In order to make reliable statements about the usability of the TransformerIDE, the execution of the exercises by the test users is observed. During the observation, a set of defined measured values is collected. Each one of these values serves the quality criteria understandability and learnability.

- Successful execution

  The first value indicates if a test user was capable to accomplish the requested exercises; which means if the result corresponds to the defined test settings. This value does not indicate the approach taken by the test user to achieve the result.

- Execution time

  The time to accomplish a part of an exercise is the execution time. This value, determined for each test user, represents the basis for the analysis of learnability based on the difference between the duration times of similar exercise parts.

- Complex situation count

  During the processing of the exercises a user can face complex situation. The amount of complex situations and the reduction of their occurrence for similar tasks enable the evaluation regarding learnability and understandability.

Those three measured values represent the basis for the evaluation. They are noted as an execution protocol created with a stopwatch by the tester. A complex situation is a situation where the tester is confronted with a challenging moment.

As this is a very general definition, a complex situation is identified based on the following list. A complex situation is the following:

1. Deadlock I: The user knows from a task perspective what must be done next but finds no appropriate function or actions.
2. Deadlock II: The user has no idea about the next action that she/he must perform.
3. Deadlock III: The user produces an error and she/he cannot solve it because she/he does not know how.
4. Choices: The user has identified for this situation multiple solutions but she/he does not know which one the right one. The user stops at this point or she/he tries the different solutions and hopes that at least one of them leads to the right solution.
5. Guessing: The user has no idea what might be right action for the next step but guessing seems to be a feasible solution.
6. Return: The result of a used function does not match with the user's expectations and she/he goes back to the initial point.
7. Dead end: The result of a used function does not correspond to the user's expectations and no possibility of return to the initial point is available.
8. Missing: The user does not execute a step which is part of the ideal path.
9. Error: The user makes an error.

The amount and the type of the complex situations occurred is an indication about the usability of the TransformerIDE. Complex situations with the number one, two, three and seven should not occur as they result in a dead end, which means the exercise cannot be accomplished successfully. The amount of all other complex situations, on the other hand, gives a clear picture which steps are complicated and represents an obstacle. Due to the time pressure, it might be sometimes the case that a complex situation could not be identified one hundred percent. Therefore, in order to prevent that any complex situation is overlooked,

the tester makes short notes about the situation and assigns the correct complex situation after the test.

Test Result

As described in the section Test Method, for this master thesis, ten test users executed a set of four prepared exercises. A tester observed the following measured values successful execution, execution time and complex situation count which represent the basis for this section.

The first test result was that all the test users where capable to execute the exercise and to achieve the predefined criteria. To achieve the goal, each user took in general the right path with only small discrepancy to the predefined ideal path such as a different sequence or a missing step. Figure 5-18 visualizes the appearance of the complex situations per exercise.

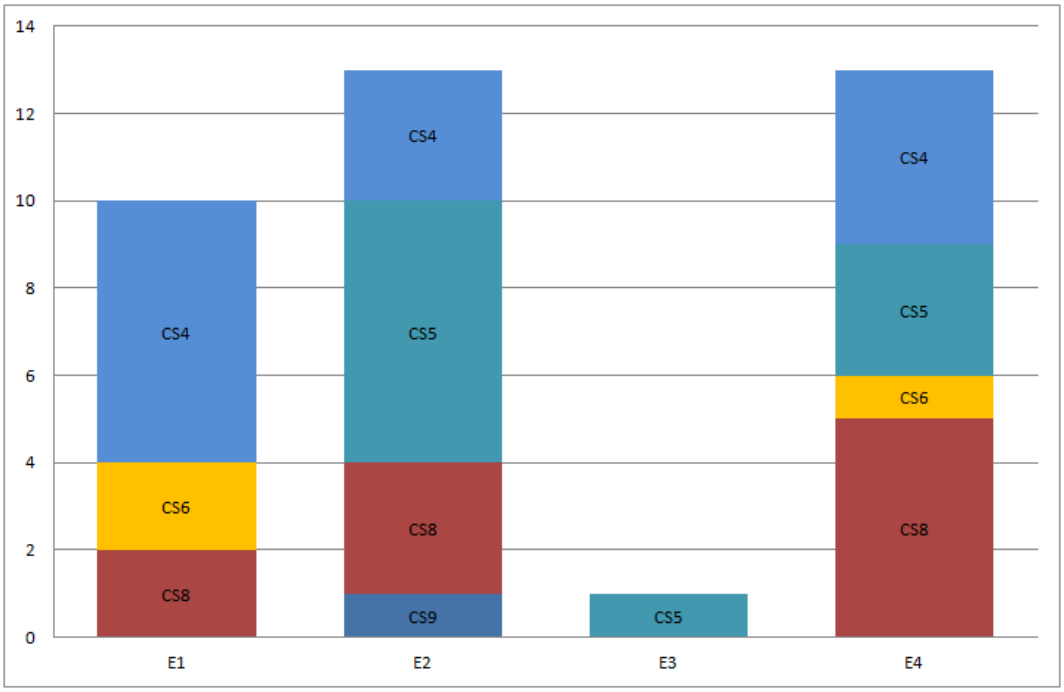

Figure 5-18 Observation of process issues during the evaluation

Not all complex situations of the list defined in the section Test Method appeared. In the following part, all occurrences of complex situations are explained and highlighted.

4. Choices: This complex situation occurred in exercise one, two and four. In general, the test user identified multiple options when it was necessary to select the correct mapping, or when they needed to decide if they should create first the transformation or add the test data and finally when they needed to adapt the Transformer to the changed mapping when they had the option to copy the old transformer or to create a new one.

5. Guessing: This situation appeared in exercise two, three and four. The test users were uncertain when they needed to use the functions concat and text field in the test editor and the function export and copy in the Transformer Manager. They solved the problem always by guessing which might be the right one. If they selected the wrong function, they tried the other one.

6. Return: In the Editor, it is necessary to select the red square before a function of a source attribute can be added. This was not obvious for all of the users and resulted in some confusion as they expected by clicking on an attribute or a function that these are added. The complex situation appeared in exercise one and four.

8. Missing: In different situations, some of the users missed a step, but it had no effect on the overall result. Such situations were that a user forgot to update the Transformer name, to add a second set of test data or, when she/he did not check the Validator output, to verify if a transformer is working properly. All these situations had no impact on the achieved goal to create a Transformer.

9. Error: In exercise two, a test user entered the wrong test data which had the impact that the transformer could not be created correctly. As soon as the user was aware of his mistake, he corrected the test data and the exercise could be accomplished successfully.

The complex situations one, two, three and seven did not appear as no test user came into the situation where he/she was forced to abort the test. Figure 5-19 shows the parts 1.3 of exercise one, 2.3 of exercise two and 4.5 of exercise four.

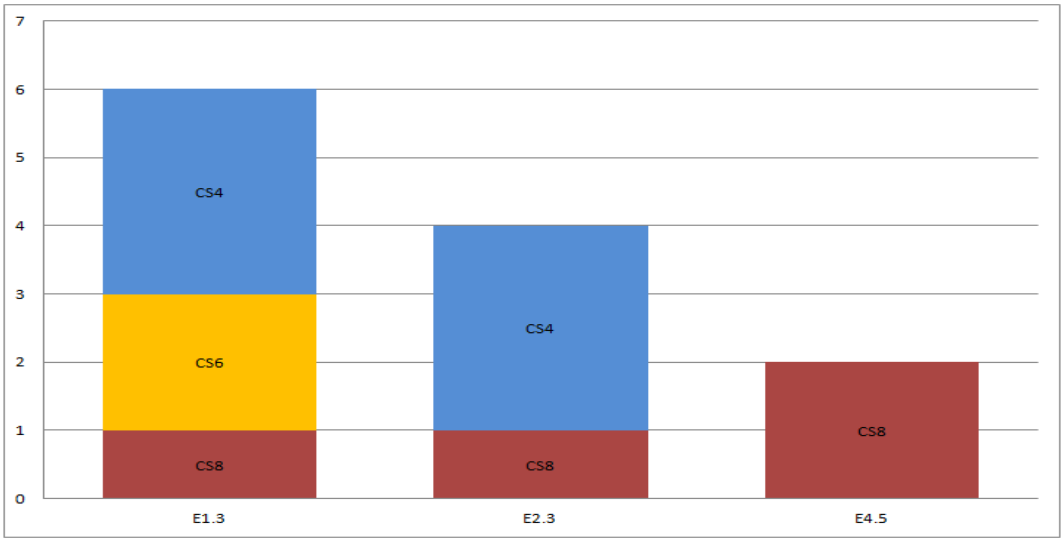

Figure 5-19 Complex situation appearance of similar tasks

These three parts have in common that they are similar as they require that a transformer be created based on an identical mapping of a source and a target attribute. In the first exercise, the test users had six complex situations which at the end, in exercise four, are reduced to two.

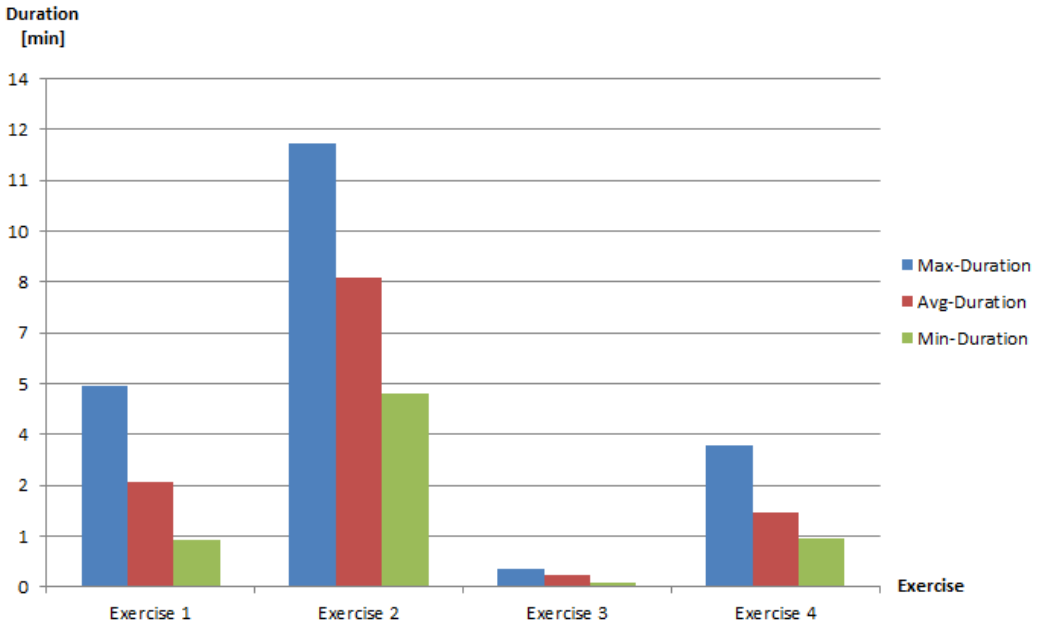

Figure 5-20 Min-Avg-Max execution duration

Next, the execution time of each exercise is evaluated. As a general reference value, it was defined that the complete duration of all four exercises should not

take longer than 30 minutes. If it takes longer, then the test can be rated as failed. Figure 5-20 visualizes the minimum; the average and the maximum duration which the test users needed to execute an exercise. The test users took between one and five minutes to execute exercise one, between five and thirteen minutes to execute exercise two, between eight and thirty seconds to execute exercise three and between one and two minutes to execute exercise four. This results in an overall average duration of twelve minutes. It took the slowest one twenty-one minutes and the fastest one 5 minutes.

Based on this duration data, also the execution time of the similar tasks can be shown, which is done in Figure 5-21 with the duration of the three runs and the reference value. The reference value starts with the average duration of the first run and is reduced after each run by twenty percent. If the average value is below the reference value then the execution time is in the expected range and can hold the quality value. If not, then the quality value is not reached.

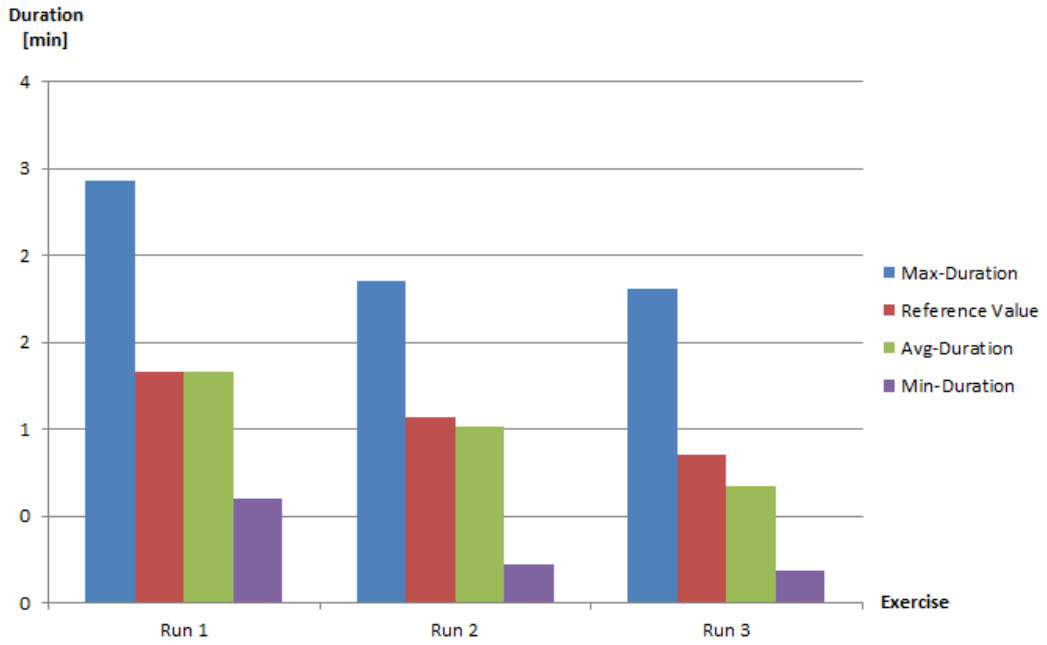

Figure 5-21 Accelerated execution time of similar tasks

CHAPTER 6

Discussion

In the section Research Approach, three research issues are defined. Those are the subject of discussion of this section. Research issue RI-1 is addressed by discussing the importance and complexity of the use cases and the resulting requirements for the proposed solution. Research issue RI-2 requires a detailed discussion about the effectiveness and efficiency of the TransformerIDE and the Transformation Meta Language. The third research issue RI-3 focuses on usability by discussing the two quality criteria of learnability and understandability. The structure of this section follows the three chapters where each one is related to a research issue. The results of the section Evaluation are used as discussion foundation.

6.1 Use Case Analysis and Requirements Elicitation

The TransformerIDE with the underlying Transformation Meta Language represents the proposed solution of this master thesis. It was developed based on a set of requirements. To determine these requirements, research issue RI-1 demands the analysis of the application scenario and the related use cases. The analysis resulted into a list of uses cases with related requirements which are discussed in this section. In general, the whole application scenario is located in multi-disciplinary engineering projects where a set of different teams from various technical disciplines must work together. Each engineering discipline represents an own domain as they have their own tools, notations, terms and stakeholders.

In order to collaborate, to achieve a common target, they must understand one another.

The application scenario of this master thesis is located in the hydropower plant construction. To build a plant, electrical, mechanical and software engineers must work together and therefore also the interoperability of the used tools is required. For the test scenario, the two systems EPLAN and Toolbox 2 OPM are analyzed. Both tools use signals and, therefore, the need to share those signals is the target. A signal represents a mechanical interface, an electrical signal or a software I/O variable. This integration scenario is addressed with the Engineering Knowledge Base which is introduced by section 2.2. The EKB addresses the following challenges: semantic model integration between expert domains; semantic integration of expert knowledge; data model translation/transformation; managing of heterogeneous knowledge.

The steps defined by the EKB approach consists of the domain knowledge gathering, the model definition and, finally, of the mapping definition between the virtual common model and the tool domain model. Based on the knowledge stored in the EKB, a Transformer can be created to enable intercommunication between the domains. This is where the use cases of the TransformerIDE are important. The effort and the knowledge needed to create the EKB is not part of this master thesis and will not be discussed any further.

In section 5.2, a set of use cases are identified and described as user stories. The design and the development of the TransformerIDE are highlighted in the chapter 4 TransformerIDE Design Process. The first step was the definition of the scope based on the user stories. Thus, a first limitation of the functions, the environment and the stakeholder was achieved. Next, a clear picture of the prototype was developed with the Software Requirements Specification (SRS) based on the IEEE Std 830-1998 [32]. A SRS demands a description of the product, the functions and the constraints. This was a logical conclusion following the gathered requirements. With the specification in the SRS, the architecture of the TransformerIDE was designed. An important aspect was the use of common

design patterns such as MVC pattern. The last step was the design of the Transformation Meta Language which represents the basis of each Transformer.

All of these requirements defined by the user stories and the SRS are derived from the creation process of a Transformer. The process consists of the following steps: reading and understanding of tool model knowledge, creation of the Transformer frame, implementation of the Transformation logic and the afterwards testing of the Transformer with test data. The first set of requirements demands the automated creation support of a Transformer based on the domain knowledge. This means that the proposed solution must access and understand the knowledge in the EKB. To enable the development of complex Transformation logics, it is necessary to offer any kind of computer language with powerful functions. The creation of the Transformation logic itself is a creative task and cannot be done automatically but it can be supported by offering an automated validation of a created Transformer based on the stored knowledge and test data.

All these requirements together aim the same objective, which is to enable the creation of a Transformer without any deep understanding of the domain and their knowledge. By achieving this objective, the condition that only an expert can create a transformer as only she/he has the needed knowledge and understanding is eliminated and the whole Transformer creation process is liberalized and opened to non-experts also. Consequentially, the effort and the creation and maintenance costs of a transformer are reduced.

6.2 Efficiency and Effectiveness of the Transformer Creation Process

In this section, the research issue RI-2 is discussed. It consists of two parts. The first part analyzes the effectiveness and second part challenges the efficiency of the solution approach.

Effectiveness

The solution approach consists of the TransformerIDE and the underlying Transformation Meta Language. To prove the effectiveness in section 5.3, the

fulfilment of each requirement is evaluated. In so doing, each function of the TransformerIDE and the Transformation Meta Language is described and linked to the requirement derived from the user stories of section 5.2. The result of the evaluation was unequivocal. Fifteen out of the fifteen user stories are covered and, therefore, the requirements also fulfilled.

The first set of requirements demands the automated creation support of a Transformer based on the domain knowledge. The TransformerIDE addresses this need with an interface to the EKB to read the stored knowledge. Besides, the TransformerIDE can interpret this knowledge and use it to create automatically the frame of a Transformer. If a model changes due to an update e.g., the TransformerIDE is capable to understand the changes and to adapt the Transformer frame to the new circumstances and enable therefore the reuse of a Transformer. To create complex transformation logic, the Transformation Meta Language is used. This language is tailored to the scope of application. It consists of functions e.g. trim, concat etc. with a clear defined input, output and nested structure and it can be extended if needed. TML consists also of a visual representative who helps the non-senior developer to understand and to develop a Transformation. Hence, the second set of requirements which requires the creation of complex transformation logic are also fulfilled. The last group of requirements demands the supported creation of the Transformation logic. The development of such logic is a creative task and is not automated by the TransformerIDE but it is supported with a Validator which verifies, based on test data and the knowledge stored in the EKB, the correctness of a Transformation.

This positive result is from a theoretical nature as it shows that the prototype implementation of the TransformerIDE and the Transformer Meta Language can be used to create a Transformer. This test does not show if it is efficient or usable. This is the topic of the next sections Efficiency and 6.3 Analysis of usability of the proposed solution.

Efficiency

With the awareness of the requirements of the application scenario and the grade of effectiveness conformity in mind, the next step can be made. This is the

evaluation of the efficiency. Section 5.4 deals with this task by comparing the TransformerIDE with the creation approach of a Transformer based on a script language.

To use a script language to develop a Transformer means to perform every step of the Transformer creation process manually. This means that a user must have enough knowledge and skills to design a software development framework where a Transformer can be set up and tested. As soon as the framework is created and, therefore, the high initial effort handled, a developer can create any complex transformation. The condition is that the user understands the knowledge stored in the EKB and be able use it to build a Transformer frame and the Transformations. In order to deploy the Transformer, it can be used directly provided that the environment of use can utilize the selected computer language of the framework. This is of course is a limitation which, in an environment where a Transformer must run on multiple platforms, can result in additional effort. If a model definition changes, the user must compare the two model versions and adjust, according the changes, manually the Transformer.

By contrast, using the TransformerIDE with the underlying Transformation Meta Language means that all the manual steps such as the setup of the framework, the creation of a Transformer frame, the verification of the correctness of a transformer and the supported adjustment of a Transformer to any model version change is done automatically which means the only manual task under the responsibility of the developer is the development of the transformation logic itself. Thus, the error sensitivity and the time-consuming property of the manual tasks are eliminated. Another beneficial aspect of the TransformerIDE is that it offers the possibility to export a Transformer into different computer languages. This results in an elimination of the time consuming task of conversion of a Transformer into another computer language.

Although the TransformerIDE offers a better efficiency due to the fact that all tasks, beside the Transformation development itself, are automated, it is important to be aware that such a development environment has some limitations. The first limitation is the Transformation Meta Language itself. This

language comes with a small number of functions which may not be enough to create complex Transformations. By using a script language, this cannot occur as the full spectrum of the language can be used. If such a situation occurs, TML must be extended, resulting in additional effort. A consequence of this extension is the need to adapt also the Code Generators which are in charge of the export of a Transformer into an executable computer language. If the number of the Code Generators is high, the effort to add a new function will also increase.

The Transformer Meta Language is a language tailored for the purpose of use and corresponds to the architectural approach Transformation Language Support called in chapter 2.4 Model Transformation. The observation confirms the advantage mentioned by Sendal et al. [11] by saying that functions can be adapted to the purpose of use and, therefore, reflect the direct need better than a general purpose language as only required functions are offered. With an increasing number of functions and requirements, a more general approach might be necessary but, as long as the application scenario is manageable, the tailored approach offers a better support. Sendal et al. [11] write also that a transformation language has preconditions that reflect the definition of the models which can be transformed. For TML, such a limitation is the structure of a signal and all other information provided by the EKB. Those are very important and must be communicated to the developer to eliminate any kind of misunderstanding.

The current interface to the EKB can handle and understand a predefined set of information so that it can be reused by the Validator and all other automated processes. As soon as this changes, the TransformerIDE also must be adapted, resulting, depending on the extent of the change, in a lot of work. The final limitation is the signal structure. The current TransformerIDE prototype supports signals which have a set of attributes not nested such as entities in an entity relationship model. As soon as this precondition is not valid, the current solution approach must be adapted to an extent requiring probably a redesigning of the complete solution approach.

6.3 Analysis of Usability of the Proposed Solution

The TransformerIDE offers a development environment which consists of a Transformer Manager, an Editor, a Validator, Code Generators and a TransformerDB. The central topic of the last section was the effectiveness and the efficiency of all this components. In this section, the usability and, more specific the understandability and learnability of the TransformerIDE, is discussed. The result of the usability evaluation is described in section 5.5. With an experiment, where ten test users executed four exercises which demand the use of each function of the Transformer, the following measurable values were collected: successful execution, execution time and complex situation count. The two quality metrics understandability and learnability were proved by measurable values.

The result of the evaluation was that all of the test users were capable to accomplish the demanded exercises. Some of them were faster and some slower, but looking at an overall duration none of them exceeded the threshold of 30 minutes. All of the test users had some software development background but no one had anything to do with the application scenario addressed by this master thesis before. As a result, this indicates that the tool can be used without any understanding of the application scenario. During the different test phases, multiple complex situations occurred and, as shows Figure 5-19, the complex situation count was reduced for similar tasks. The test users had the most problems with understanding the use of functions, such as export, copy, or Transformation functions, such as concat and text field. But they solved the problem always by guessing and trying. Sometimes, some of the users missed some steps, such as the adding of a second set of test data or saving. The reason might be TransformerIDE but also the test setting itself. Concluding from the missing appearance of complex situations which results in a dead end, one can say that the TransformerIDE offers an understandable solution.

To state a meaningful conclusion about the learnability of the TransformerIDE, the attitude of the measurable values over time must be analyzed. The four exercises contained three similar tasks and enabled a comparison of them. Let’s first look at the complex situation count. Following Figure 5-19 in the first task six

complex situations occurred. The main reason was that the users were not sure which functions must be used to achieve the desired goal and, therefore, they guessed the next and were sometime surprised by the wrong result. Looking at the second task, the number is reduced to four. At this time, no situations occurred when a user was surprised about the result of any used function. The third time, each test user knew what is necessary to create a Transformation and therefore no uncertainty about the used function could be found, only two of the users forgot to add the second set of test data. Beside the improvement of the understanding of the IDE, also the needed execution time is analyzed. Figure 5-21 shows the minimal, average and maximum time it took the users to execute the three tasks. Going from the left to the right, it is clear that the duration is reduced of approximately forty-five percent. Based on these two results, it can be said that the TransformerIDE represents a solution which fulfils the learnability quality criteria some a clear improvement over time can be noticed.

CHAPTER 7

Conclusion & Future Work

This chapter summarizes the master thesis and the results of the evaluation and discussion of the 5 and 6. It gives an overview on the achieved knowledge in using the TransformerIDE as a solution approach. At the end, perspectives for future work are given.

7.1 Conclusion

The *TransformerIDE* and the *Transformation Meta Language* represent the solution approach developed and evaluated by this master thesis. The thesis uses an experiment to find a feasible solution for an integration scenario where the Engineering Knowledge Base is used. While the EKB offers a solution to store explicit knowledge in a reusable way and make it accessible for further use, the TransformerIDE uses this information and supports engineers in creating a Transformer and Transformation in an effective, efficient and usable way.

The environments of this master thesis are multi-disciplinary engineering projects with focus on the hydropower plants construction application area. In such an environment, a set of different engineering disciplines must work together to achieve a common goal. The challenge of such an integration scenario is the heterogeneity of the information which must be shared. Before a solution, such as the TransformerIDE, can be developed, it was necessary to understand the related work, the status of research and to find possible synergies. The outcome of this research was the chapter-related work. Three main research areas were identified: Semantic integration, Model-Driven

Architecture and the Ontology Modelling. The first one semantic integration of course deals with the integration of heterogeneous information. Here, the EKB provides a solution as it pre-processes the tool model information and provides them to further use. The EKB does not offer a solution for the development of a Transformer but it proposes an automated or semi-automated process based on the stored information. And this is where the TransformerIDE fits in as it provides a semi-automated approach to create a Transformer.

In general, a Transformer is some logic which transforms a signal as a model from one definition into another. This simplification brings us to the Model-Driven Architecture. The MDA is a framework to visualize, storing and exchanging software models and it provides architecture to transform a model based on a model definition language. Taking this approach into account and by making further research in the area of ontology modelling, the need for an own transformation language for the TransformerIDE is obvious.

Leaving the related work, the next step is the requirement analysis to understand the requirements of the hydropower plant construction application scenario. The determined high level requirements were the need to handle signals of the tools used in the application scenario such as EPLAN or OPM, the supported creation of a Transformer based on the knowledge stored in the EKB and possibility to deploy them as different executable computer languages. To satisfy these needs, the TransformerIDE with a transformation language called Transformation Meta Language was designed and implemented as a prototype. With this approach, a clear decision to develop an adapted solution and not to use any generic transformation approach was made.

Three research issues demanded the suitability of the proposed solution by challenging the effectiveness, efficiency, learnability and understandability. To prove the effectiveness, the thesis checked if the TransformerIDE provides functions which satisfy the requirements identified. It could be proven that all function needed are covered. This result is positive but it gives no clear information about the quality of the provided function. To do so, the efficiency was proven by comparing the TransformerIDE with the use of a script language.

Both solutions were capable to accomplish the defined tasks but the effort to accomplish them was differently. While by using the script language, a developer must develop and to do each step manually, the TransformerIDE does everything automatically, except the Transformation development itself. This is a high benefit and reduces the whole effort. Despite its beneficial effects, the TransformerIDE has some limitations which are the limited set of functions of TML, the maintenance effort and the strong dependencies to the model information structure of the EKB. The last quality criteria evaluated were usability and learnability. The target was to show that also non experts can use the TransformerIDE without any deep knowledge about the application scenario. Ten users were asked to test the application and different measured values were observed. The result was that no test user took longer than 30 minutes, which was the threshold, to execute the exercises. Some of the user had situation where they were not sure what must be done next but, in general, they found their way without any bigger issue or show stopper. By analyzing the tasks of the exercises over time a learning effect have been observed due to the reduced duration and problems could. Therefore, also the understandability and the learnability were proven.

As a final conclusion it can be stated, that the TransformerIDE as a prototype offers a solution which can bring benefit into the application scenario as it makes it possible that also non experts can create a Transformer without the involvement of experts. The development of a Transformation itself is the creative part and requires that the software developer is creative in interpreting and using the available information. The TransformerIDE supports the creation process by visualizing the information and validation of the correctness but it does not take care of it. Nevertheless, the effort to maintain and extend the TransformerIDE is only acceptable if the application scenario is big enough.

7.2 Future Work

The questions of this section are “What is not covered by this thesis” and “What can be done next based on the achieved results?” The following list of topics tries to find an answer to these questions and should give an outlook to future work.

- Complex signals

  The current TransformerIDE is capable to handle signals which are flat. This means that a signal model has a name and a set of atomic attributes. Looking at other environments it becomes clear that information can be much more complex. Therefore, one future plan can be to extend the TransformerIDE to enable the handling of nested signals.

- Complex functions

  For the application scenario analyzed for this master thesis only simple string and number manipulation functions were necessary. As this might not be enough for all situations, the implementation of complex functions such as when structures or loops might be required.

- Reuse of changed Transformations

  With a change of a model version a created Transformer can be reused and updated. The Transformations which are not impacted by the change is reused and all new and changed Transformations must be developed new. If a Transformation was complex and only small changes to the source and target attribute occurred, it might be still usable to reuse the old Transformation. In order to do so, the TransformerIDE must have a more granular detection of differences to enable the reuse of Transformation parts.

- Reuse of Transformations

  Some Transformations are complex and a high benefit would be to reuse them for other Transformations. The possibility to define customized Transformations and to provide them as a function to other Transformations might be useful.

Bibliography

[1] T. Moser, S. Biffl, W. D. Sunindyo, and D. Winkler, “Integrating production automation expert knowledge across engineering stakeholder domains,” in *Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on*, 2010, pp. 352–359.

[2] W. Schafer and H. Wehrheim, “The challenges of building advanced mechatronic systems,” in *2007 Future of Software Engineering*, 2007, pp. 72–84.

[3] N. F. Noy, A. H. Doan, and A. Y. Halevy, “Semantic integration,” *AI Mag.*, vol. 26, no. 1, p. 7, 2005.

[4] A. Halevy, “Why your data won’t mix,” *Queue*, vol. 3, no. 8, pp. 50–58, 2005.

[5] D. Fensel, “Ontologies: A silver bullet for Knowledge Management and Electronic-Commerce (2000),” *Berlin: Spring-Verlag*.

[6] M. Uschold and M. Gruninger, “Ontologies and semantics for seamless connectivity,” *ACM SIGMod Rec.*, vol. 33, no. 4, pp. 58–64, 2004.

[7] R. Jasper, M. Uschold, and others, “A framework for understanding and classifying ontology applications,” in *Proceedings 12th Int. Workshop on Knowledge Acquisition, Modelling, and Management KAW*, 1999, vol. 99, pp. 16–21.

[8] N. F. Noy, “Semantic integration: a survey of ontology-based approaches,” *SIGMOD Rec.*, vol. 33, no. 4, pp. 65–70, 2004.

[9] P. A. Bernstein and U. Dayal, “An overview of repository technology,” in *Proceedings of the International Conference on Very Large Data Bases*, 1994, p. 705.

[10] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA explained, the model driven architecture: Practice and promise*. Addison-Wesley Professional, 2003.

[11] S. Sendall and W. Kozaczynski, “Model transformation: The heart and soul of model-driven software development,” *Software, IEEE*, vol. 20, no. 5, pp. 42–45, 2003.

[12] T. Moser, “Semantic Integration of Engineering Environments Using an Engineering Knowledge Base,” *Vienna Univ. Technol. Vienna, Austria*, 2009.

[13] F. Hakimpour and A. Geppert, “Resolving semantic heterogeneity in schema integration,” in *Proceedings of the international conference on Formal Ontology in Information Systems-Volume 2001*, 2001, pp. 297–308.

[14] S. Bergamaschi, S. Castano, and M. Vincini, “Semantic integration of semistructured and structured data sources,” *ACM Sigmod Rec.*, vol. 28, no. 1, pp. 54–59, 1999.

[15] A. H. Doan and A. Y. Halevy, “Semantic integration research in the database community: A brief survey,” *AI Mag.*, vol. 26, no. 1, p. 83, 2005.

[16] M. Obitko and V. Marík, “Ontologies for multi-agent systems in manufacturing domain,” in *Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on*, 2002, pp. 597–602.

[17] T. R. Gruber and others, “A translation approach to portable ontology specifications,” *Knowl. Acquis.*, vol. 5, no. 2, pp. 199–220, 1993.

[18] P. Szulman, M. Hefke, A. Trifu, M. Soto, D. Assmann, J. Doerr, and M. Eisenbarth, “Using ontology-based reference models in digital production engineering integration,” in *Proceedings of the 16th IFAC World Congress, Prague, Czech Republic*, 2005.

[19] D. L. McGuinness, F. Van Harmelen, and others, “OWL web ontology language overview,” *W3C Recomm.*, vol. 10, no. 2004–03, p. 10, 2004.

[20] J. L. M. Lastra and I. M. Delamer, “Ontologies for production automation,” in *Advances in Web Semantics I*, Springer, 2009, pp. 276–289.

[21] S. Lemaignan, A. Siadat, J.-Y. Dantan, and A. Semenenko, “MASON: A proposal for an ontology of manufacturing domain,” in *Distributed Intelligent Systems: Collective Intelligence and Its Applications, 2006. DIS 2006. IEEE Workshop on*, 2006, pp. 195–200.

[22] T. Adams, J. Dullea, P. Clark, S. Sripada, and T. Barrett, “Semantic integration of heterogeneous information sources using a knowledge-based system,” in *Proceedings of 5th Int Conf on CS and Informatics (CS&I’2000)*, 2000.

[23] P. Szulman, M. Hefke, A. Trifu, M. Soto, D. Assmann, J. Doerr, and M. Eisenbarth, “Using ontology-based reference models in digital production engineering integration,” in *Proceedings of the 16th IFAC World Congress, Prague, Czech Republic*, 2005.

[24] D. R. Kuokka, J. G. McGuire, J. C. Weber, J. M. Tenenbaum, T. R. Gruber, and G. R. Olsen, “SHADE: Technology for knowledge-based collaborative engineering,” in *of the AAAI Workshop on AI in Collaborative Design*, 1993.

[25] T. Moser, R. Mordinyi, A. Mikula, and S. Biffl, “Efficient system integration using semantic requirements and capability models,” in *Proc. 11th Int. Conf. on Enterprise Information Systems*, 2009.

[26] S. Kent, “Model driven engineering,” in *Integrated Formal Methods*, 2002, pp. 286–298.

[27] J. Bezivin, M. Barbero, and F. Jouault, “On the applicability scope of model driven engineering,” in *Model-Based Methodologies for Pervasive and Embedded Software, 2007. MOMPES’07. Fourth International Workshop on*, 2007, pp. 3–7.

[28] S. J. Mellor, *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional, 2004.

[29] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow, “Model transformations? transformation models!,” in *Model Driven Engineering Languages and Systems*, Springer, 2006, pp. 440–453.

[30] D. Djurić, D. Gašević, and V. Devedžić, “Ontology modeling and MDA,” *J. Object Technol.*, vol. 4, no. 1, pp. 109–128, 2005.

[31] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, “Lessons from applying the systematic literature review process within the software engineering domain,” *J. Syst. Softw.*, vol. 80, no. 4, pp. 571–583, 2007.

[32] “IEEE Recommended Practice for Software Requirements Specifications,” *IEEE Std 830-1998*, p. i, 1998.

[33] C. Floyd, “A systematic look at prototyping,” in *Approaches to prototyping*, Springer, 1984, pp. 1–18.

[34] International Organization For Standardization Iso, “ISO/IEC 9126-1,” *Software Process: Improvement and Practice*, vol. 2. pp. 1–25, 2001.

[35] W. D. Sunindyo, T. Moser, D. Winkler, and S. Biffl, “A process model discovery approach for enabling model interoperability in signal engineering,” in *Proceedings of the First International Workshop on Model-Driven Interoperability,* 2010, pp. 15–21.

[36] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, “ATL: A model transformation tool,” *Sci. Comput. Program.*, vol. 72, no. 1, pp. 31–39, 2008.

[37] I. A. W. Group, “IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems,” IEEE, 2000.

[38] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Professional, 2003.

[39] “Smooks Data Integration.” [Online]. Available: http://www.smooks.org.

List of Figures