FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Dynamic Java-Based Model-To-Model Transformation Environment

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Felix Mayerhuber
Matrikelnummer 0825283

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao. Univ.-Prof. Dipl.Ing. Dr. Stefan Biffl
Mitwirkung: Dr. Thomas Moser
Dr. Estefania Serral Asensio

Wien, 08.07.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Felix Mayerhuber
Paul Hörbiger Straße 1, 3140 Pottenbrunn

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)

Abstract

In any environment where several heterogeneous tools have to work together (e.g. software integration frameworks), it is necessary to enable the communication between those tools. To deal with this challenge, the tools need 1) either share their models with each interested tool (higher coupling between the tools); or 2) use a translator component which transforms the models of one tool into models understandable for the other tools. This thesis deals with the second option since it separates the model format transformation from the independent tools.

In the context of this thesis, a model describes the internal data representation of a components' working domain object. For instance, an issue tracking component may describe an issue as a model. This model is represented through a class which contains information like the author or the involved components. This data can also be represented through different classes, e.g. an author may contain information like a name or project role.

Nowadays, the most common approach for implementing a translator component is to use one of the transformation solutions popular in the model-driven engineering world. These solutions are powerful, but complex and difficult to handle. Modelling experts are needed to set up such solutions in order to let them work as expected.

This thesis deals with the theoretical concept and the practical approach of the model transformation solution used in the Open Engineering Service Bus (OpenEngSB) project, a tool integration platform with the goal of archiving efficient interoperation between tools in multidisciplinary environments. The proposed approach provides an easy to use model transformation solution which is also flexible and runtime extensible.

The approach is based on Java. Transformation operations are Java methods and models are realised as Java classes' structures (specifically, in means of classes, their attributes and their relations to other classes). The advantage of this approach, is that it can be adapted by every Java developer to address project requirements and their changes (e.g. by adding new operations). In addition, no expertise in formal high-level abstraction modelling and meta modelling is needed; only knowledge about Java is required.

Based on the introduced transformation solution, an approach is introduced to enable a simple form of automatic model update propagation. With this approach, it is possible to define relations between model instances and enforce data integrity between them.

The prototype has been tested by its application in practical use-cases out of the OpenEngSB domain and the OpenEngSB customer area. In addition to these use-case tests, the introduced transformation definition language has been compared with other existing approaches and the prototypes performance has been tested.

Kurzfassung

In jeder Softwareumgebung, in welcher mehrere heterogene Tools zusammenarbeiten (z.b. Software Integration Frameworks), müssen die Tools miteinander kommunizieren können. Um dies zu gewährleisten, können die Tools entweder 1) ihre Modelle mit jedem interessierten Tool teilen (zusätzliche Kopplung zwischen den Tools) oder 2) eine Komponente benutzen, welche Modelle in Modelle anderer Tools konvertieren kann. Diese Arbeit beschäftigt sich mit der zweiten Möglichkeit, da diese die Modelltransformationslogik von den Tools trennt.

Im Kontext dieser Arbeit beschreibt ein Modell die interne Datenrepresentation eines Objektes aus dem Arbeitsbereich eines Tools. Ein Modell eines Issue Tracker Programms könnte z.B. ein Issue beschreiben, welches Informationen wie Autor und betroffene Komponenten enthält und als Klasse dargestellt wird. Jede dieser Informationen kann durch eine andere Klasse dargestellt werden (Ein Autor kann beispielsweise aus Namen und Projektrolle bestehen).

Der Einsatz einer Transformationslösung aus der Modellbasierten Entwicklung, ist heutzutage die gängiste Variante um eine Übersetzungskomponente zu realisieren. Diese Lösungen sind mächtig, aber komplex und schwer zu bedienen. Man braucht Spezialisten, um solche Lösungen effizient einsetzen zu können.

Diese Arbeit beschäftigt sich mit dem theoretischen Konzept und der praktischen Umsetzung der Modelltransformationslösung des Open Engineering Service Bus (OpenEngSB) Projekts, einer Toolintegrationsplattform mit dem Ziel der effizienten Zusammenarbeit zwischen Tools in multidisziplinaren Umgebungen. Die vorgestellte Lösung bietet einen einfach zu bedienenden und zu modifizierenden Ansatz einer Modelltransformationslösung.

Der Ansatz ist Java-basiert. Transformationsoperationen sind Java-Methoden, Modelle sind als Java-Klassenstrukturen realisiert (genauer: als Klassen, deren Attribute und die Relationen zu anderen Klassen). Der Vorteil dieses Ansatzes ist, dass er durch jeden Java Entwickler an (sich ändernde) Projektanforderungen (z.b. durch neue Transformationsoperationen) angepasst werden kann. Außerdem ist kein Expertenwissen in formaler High-Level Abstraktionsmodellierung und Meta-Modellierung nötig. Java-Wissen allein reicht aus.

Basierend auf dem Prototypen wird ein Ansatz vorgestellt, welcher eine einfach Form von automatisierter Modellaktualisierungsverbreiterung bietet. Mit diesem Ansatz ist es möglich, Relationen zwischen Modellinstanzen zu definieren und Datenintegrität zwischen ihnen zu gewährleisten.

Der Prototyp wurde mittels Anwendungsfällen aus der Domäne und Kundenumgebung des OpenEngSB getestet um seine Anwendbarkeit zu prüfen. Zusätzlich zu diesen Tests wurden sowohl die vorgestellte Transformationsdefinitionssprache mit anderen existierenden Ansätzen verglichen als auch Performance-Tests am Prototyp durchgeführt.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In most systems more than one component is involved in the daily processes (e.g. different tools, different project segments, individual software for each department, etc.). Typically these different components need to communicate with each other or at least with a sub-set of the other components.

A common problem in such systems, is that each component has its own data formats, object types and conventions. So it is quite common for each component to use its own internal data representations for every structured information they need to deal with.

In this thesis, these representations are referred as *models*. A model is realised by one or more Java classes, which have properties and relations among them. There exist one 'master' model class which have the other model classes as properties if they are needed. Such a model construct describes an internal data representation of a component. More information of how such a construct is realised, can be seen in section 6.6.

For example, a component which is working with an issue tracking system may have a model that describes an issue of the issue tracking system. The corresponding class contains information like the issue author, the date when the issue was created, which components are involved and other information the component is interested in. It may be that the issue author or the involved components are described in other classes themselves and the issue class contains properties referencing to these classes.

To enable the communication between the components there is the need to transform a model of one component into a model of another component. Specifically: if component A wants to communicate with component B, the data model used by component A needs to be converted into the data structure used in the data model managed by the component B (model that component B can understand). This converting process is called *model transformation* and is visualized in figure 1.1.

Currently there are two main possibilities to perform model transformations in company environments: manually or automated. Both approaches have their advantages and disadvantages.

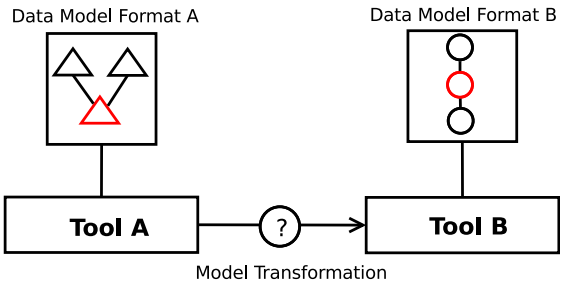

Figure 1.1: The general problem of model transformation

The **manual approach** consists of several steps which need to be performed by the user, therefore requires a high manual effort and is usually error prone. The user gets descriptions (e.g. in text documents) about how to transform model instances from one model into the other (copy an existent model, paste it into the local workspace and modifies it in that way that the other model is generated), or get scripts which can be performed to transform a model of a specific type into an other model type. The general process of this approach is shown in figure 1.2.

+ no installation of a model transformation infrastructure is needed.
+ no meta models needed, which saves resources.
- time expensive, since every transformation need to be done by hand.
- error prone, since it bases mainly on manual tasks (see book of Reason [Reason, 1990]).
- all workers need to be informed when a transformation changes.

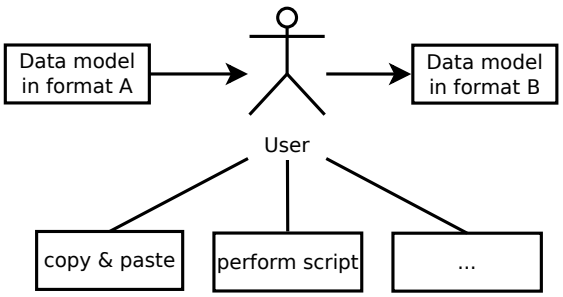

Figure 1.2: The manual model transformation approach process

There is a list of different **automated approaches** available today. In the scope of this thesis, this list is divided into two distinct parts: the *meta model based approaches* and the *static uninformed approaches*.

The **meta model based approaches** expect that there is a schema (also called meta model) for the models involved in the transformation process. In normal cases there exists a component which is aware of this schema and has access to all available transformation rules to be able to transform any model to a list of other models in the working environment. These approaches provide a high customizability to allow a wide range of use-cases which can be realized with them. The meta model based model transformations are the most used model transformation solutions. The general structure of such a solution with all involved models and meta models is shown in figure 1.3.

+ allow a high amount of use-cases because of their flexibility.
+ widely used (high usage in the model driven development world).
+ single point of failure, which lies in the transformation rules.
+ the workers do not need to be aware of the transformations.
- need for modelling experts to apply these solutions.
- a component that handles the transformation process is required.
- meta models are needed, which means higher costs at project start.

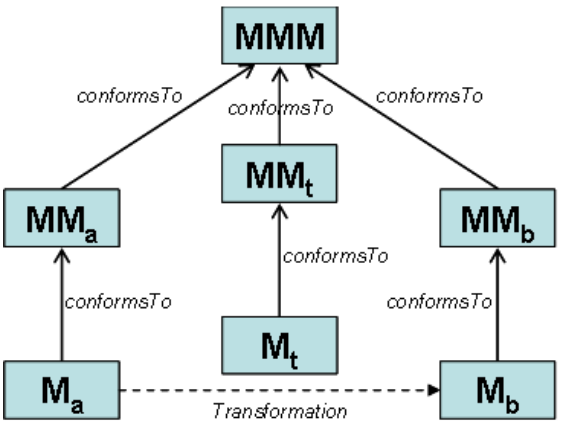

Figure 1.3: The meta model based model transformation approach process. [The Eclipse Foundation, 2012a]

The **static uninformed approaches** expect no schema from the models involved in the transformation process. They rely on information they get from the models at runtime and previously defined model transformation rules. Since they have no prior knowledge of the involved models before runtime, they are called uninformed. Because of the limited knowledge about the models at compile time, the possibilities in the areas of validation and testing are restricted.

Additionally, these approaches are very limited in their customizability since they do not (or only in an insufficient manner) allow to adapt the transformation possibilities to a projects need (e.g. by adding new transformation operations). This limitation makes these approaches static and makes them hard to use for projects with special transformation requirements. On the other hand, because of the limitations of these these solutions, they are highly optimized so their performance is very good. Figure 1.4 shows the main structure of a non dynamic model transformation process.

+ very fast because of optimizations on the transformation process.
+ often used in communication frameworks.
+ no explicit meta models are needed.
+ single point of failure, which lies in the transformation rules.
+ the workers do not need to be aware of the transformations.
- a component that handles the transformation process is required.
- very limited possibilities due to low customizability.
- limited possibilities in the area of validation and testing.

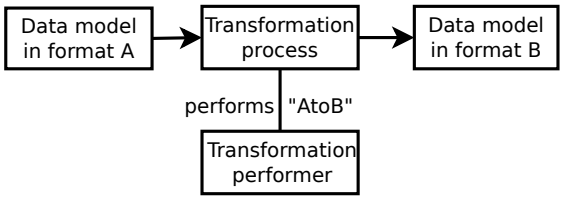

Figure 1.4: The static uninformed model transformation approach process.

The model transformation solution presented in this thesis takes a novel approach by introducing an automated transformation solution for Java projects that do not need the definition of meta models but provide a high customizability. The proposed solution provides the following advantages:

- **No meta models need to be defined.** The presented solution do not need to previously define meta models of the models that need to be transformed; therefore, transformation designers do not need to have previous knowledge about meta modelling.
- **Easy to understand for developers.** Models are defined by annotating Java classes which represent the models. The transformation definitions are defined with a concentrated view on simply transform one model to another, with as less meta data as possible.

- **Easy to extend/adapt.** The proposed solution is entirely Java based. This also includes the operations used in the proposed transformation language. It is easy for a developer to introduce, adapt and modify operations to address (changing) project needs.
- **Low user expertise required.** The users of the proposed transformation solution only need knowledge in either Java or XML since transformation rules can be expressed in both languages.
- **Easy existing project enhancement.** Under certain preconditions (see section 6.8), it is very easy to extend an existing project with the presented transformation solution. After the models and the transformation descriptions are defined, transformations can be performed.

1.2 Problem

Model transformation is a topic which need to be considered in many existing projects. In every situation where more than one existing software unit needs to communicate with other units (e.g. in every integration platform or integration solution) model transformation is a mandatory task. Because of this circumstance, there are already quite a lot of existing approaches to deal with this problem.

These approaches are typically based on existing high-level transformation solutions. These solutions are often very powerful but also complicated to use. Normally these solutions need meta models on which the transformations are defined. However, generating meta models for a set of models is not a trivial task and is normally done by experts with previous knowledge about meta modelling.

The aim of this thesis is the design and implementation of a transformation solution for the Open Engineering Service Bus[1] (*OpenEngSB*) project, a system integration framework with the main goal to support multidisciplinary engineering projects. This transformation solution shall be meta model less, easy to be used by/in other frameworks and work as a base for more sophisticated `OpenEngSB` features.

The three main challenges of the transformation solution research and development are listed here and explained shortly:

Transformation Path Search. Internally, the network of possible transformations are treated as a graph. In this graph, models are visualized as nodes and transformations are visualized as edges. If there is a *Transformation Request* given to the transformation managing component, this component needs to find a path from the source model to the target model. However, the search for a path is not trivial since there are a list of requirements which are not solved by standard graph search algorithms (e.g. the temporary deactivation of nodes or the mandatory presence of an edge subset in the path search result). If there is a possible path, the transformation can be performed (more details about the path search can be read in section 6.4). An illustrative figure of this challenge can be seen in figure 1.5.

[1] `http://openengsb.org/`

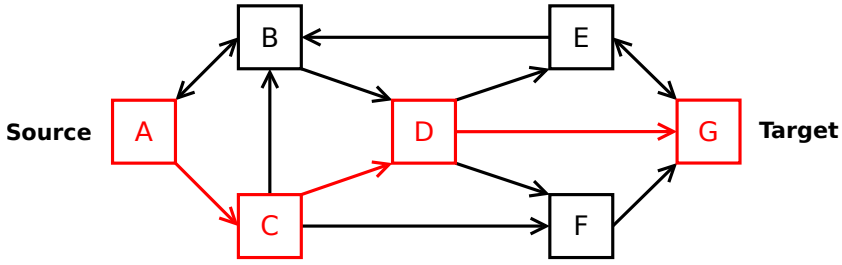

Figure 1.5: Transformation Path Search Challenge

Model to Model Transformation. After a path has been found, the actual transformations need to be performed. To be able to perform transformations, the transformation solution need descriptions of the transformations. These descriptions contain the information what values of the one model attributes are transformed in which way to values of the other model attributes. There is the need of a component which can understand these descriptions and execute the corresponding transformations. To enable a wide variety of possible use cases, this transformation descriptions need to support a collection of *Transformation Operations* and also allow the project user to define its own operations or adapt existing ones. Since models possibly consist of more than one Java class, there is the additional need to be able to transform such model constructs. This need is dealt with, through the possibility of nested field access (more details are given in section 6.3). An illustrative figure of this challenge can be seen in figure 1.6.

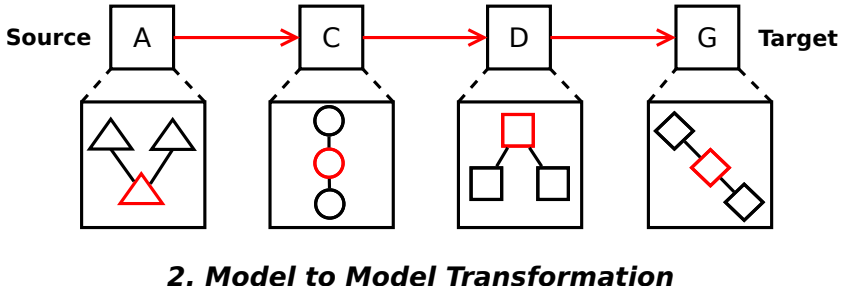

Figure 1.6: Model to Model Transformation Challenge

Automated Model Update Propagation. With automated model update propagation, the automatic update of models which are related to a model which has been modified is meant. The typical use case for such models are environments where models in different tools exists, which are semantically identical but are represented in different ways. In such environments it is normal that the changing of a model influences several other models

which are somewhat connected to the changed model. It is necessary to update also the influenced models to ensure data integrity.

To achieve this, it is necessary to specify explicitly dependencies between models. To enable this dependency definition, the *Engineering Object* concept has been introduced with this thesis. Engineering Objects are special models with which it is possible to create links between model instances. Based on these models, the automated model update propagation is performed in background by the OpenEngSB. As base for this sophisticated feature, the introduced transformation solution is used. An illustrative figure of this challenge can be seen in figure 1.7.

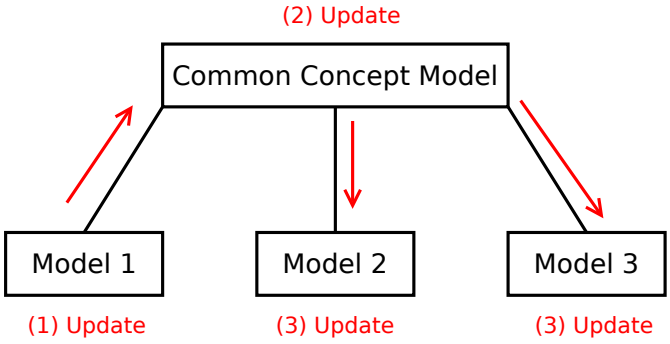

Figure 1.7: Automated Model Update Propagation Challenge

A specific example for an environment which needs the automated model update propagation is a company which develops electrical circuits. There are different engineering disciples involved in the creation of these circuits. Every discipline need a different representation of the same information. E.g. an electrical engineer is not interested in the variable name some transistor will have in the circuit programming part. This example environment is used as use-case for the model update propagation validation in this thesis (see chapter 4).

Another example use case of the Engineering Object concept can be found in the software modelling area. There exist a lot of modelling mechanism and tools, with which the same component can be described (e.g. Component View, Class Diagrams, Flow Diagrams, State Diagrams, ...). Some of them are tightly bound so that changes in the one diagram type force changes in the related diagram types.

Every environment which contains different syntactical representation of semantically equivalent models has to deal with update propagation between the models, since every change on the one hand could possibly change something on a different representation. If no update propagation is considered there is a high probability of producing inconsistencies.

With the Engineering Object concept the project designer is able to model such configurations and let the framework perform the update propagation between the models. In this way it can be ensured that changes of models are noticed by any worker which is working with

the changed models. This reduces (or even remove) the problem of data inconsistencies based on a lack of communication between the workers.

1.3 Contributions

The contributions provided by this thesis are:

- The design and implementation of a Model-To-Model transformation solution prototype with the previously mentioned properties (described in more detain in chapter 6).
- A Transformation Definition Language (*TDL*) for the transformation solution implementation named Dynamo Transformation Language (*DTL*).
- An implementation of an automated model update propagation solution realised through the `Engineering Object` concept (explained in section 6.7).

The implementation of the transformation solution is built for the `OpenEngSB` project, but the components could also be placed in other projects providing certain conditions, which are explained in detail in section 2.5.

Figure 1.8 shows a simple overview over the contributions which have been added in the practical part of this thesis. The figure is divided in three levels with a different detail grade.

- In **Level 1** the user wants to transform an instance of *ModelA* to an instance of *ModelC*. The transformation solution finds a path in the transformation graph ($A \rightarrow B \rightarrow C$), performs the transformations and returns the transformed instance.
- **Level 2** focuses on the transformation following the identified path. It shows that the instance of the *ModelA* needs to be transformed into an instance of the *ModelB* before this instance can then be transformed into an instance of *ModelC*. Between each transformation, a component of the *`Transformation Engine`* is drawn which performs the transformation, the *`Transformation Performer`*.
- **Level 3** shows a detailed view on how a model to model transformation is performed. Based on the transformation description language introduced in this thesis, the component perform the actual transformation steps. Every edge represents an operation on values of the source model, which results in a value for the target model.

In the `OpenEngSB` project the introduced transformation solution is part of the semantic core component, the Engineering Knowledge Base (*EKB*). The `EKB` is divided into a set of components which, as a whole, manage all operations related with the semantics of the `OpenEngSB` and is explained in more detail in section 2.2.3.

For transformation support, there were three new components added to the `EKB`.

- `Transformation Engine`: This component responsible for performing the transformations. It uses the *`EKB Graph DB`* component to find *Transformation Paths* and returns transformed models to the user. For more information, see section 6.2.3.

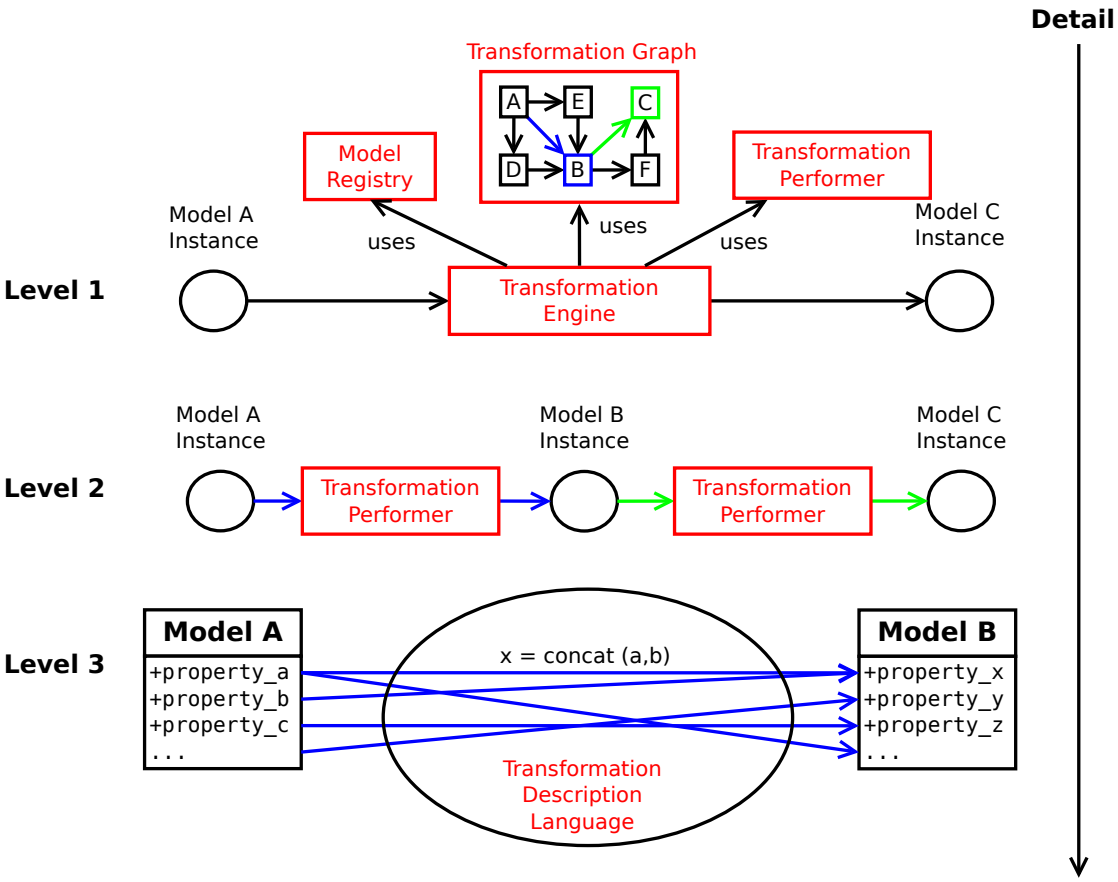

Figure 1.8: Contributions Overview

- *`Model Registry`*: This component is aware of all active models which are active in the running environment. It is able to load model classes and can retrieve meta data from models. For more information, see section 6.2.2.

- `EKB Graph DB`: This component is a graph based NoSQL database for internal use. A NoSQL graph database is used because they are designed for tasks like the finding paths in graphs. The `Transformation Engine` and the `Model Registry` give this component all information about active models and *Transformation Descriptions*. For more information, see section 6.2.1.

These three components and their functionality as a complete unit will be called *Transformation Environment* in this thesis. A more detailed explanation of these three components can be read in section 6.2.

The Transformation Environment need to be aware and up-to-date with all the models and descriptions of transformations which are present in the project, so that it can work properly. On the one hand, the `Model Registry` needs to have knowledge about models in the run-

ning environment. The Transformation Engine on the other hand needs definitions of transformations between models. Such definitions are called Transformation Descriptions.

As language for these Transformation Descriptions a new transformation language was defined, named DTL. One advantage is, that Transformation Descriptions can be expressed either in pure Java code or in XML documents. The complete list of reasons why DTL was invented and not an existing solution was chosen can be read in section 6.3.

A Transformation Description represents the list of mapping definitions between two models. It contains the source model, the target model, an identifier and a list of *Transformation Steps* which are needed to perform the transformation itself. Based on such Transformation Descriptions the Transformation Engine is able to perform the transformation from one model into another model. More information about the structure of Transformation Descriptions and their components is stated in section 6.3.

1.4 Thesis Structure

The remainder of this thesis is structured as follows:

- Chapter 2 shows the context where the explained transformation solution can be used. Also it illustrates the OpenEngSB framework which is used for the implementation of the presented solution.
- Chapter 3 summarizes related work and existing solutions which have influenced the work on the transformation solution of the OpenEngSB.
- Chapter 4 presents the use-cases which are used for the validation of the developed prototype.
- Chapter 5 explains the research issues for this thesis and introduces the use cases on which the prototype got tested.
- Chapter 6 gives an overview of the solution presented in this thesis. It shows detailed information about the components involved, the interaction possibility for the user and internal procedures.
- Chapter 7 presents the validation of the transformation language used and a comparison to other transformation languages as well as a validation for the transformation solution prototype based on the use-cases defined in chapter 4. Additionally, benchmarks on the presented prototype are presented in this chapter.
- Chapter 8 discusses the prototype implementation in the scope of features which could be constructed based on it.
- Chapter 9 concludes the thesis, summarizes the limitations of the current prototype implementation where no efficient solution has been found so far and gives an overview over future work which can be done based on the presented prototype.

CHAPTER 2

Context & Background

2.1 Overview

The goal of this chapter is to introduce the Open Engineering Service Bus (*OpenEngSB*) project where the introduced solution was developed and tested. The illustration of the OpenEngSB project in this chapter is limited to the aspects which are needed to understand the constructs of this thesis.

Since the OpenEngSB is an Enterprise Service Bus (*ESB*) similar project using Open Services Gateway initiative framework (*OSGi*) as integration technology, these two concepts are shortly explained in this chapter too. Finally, since the presented transformation solution is not limited to OpenEngSB projects, an explanation of the context in which the developed prototype can be used is given.

2.2 OpenEngSB project

The OpenEngSB project is an open source tool integration solution which tries to provide an easy to set-up and easy to adapt possibility to enable interoperability between different tools. In contrary to other existing approaches, the OpenEngSB tries to create an integration layer on top of the tools which are used by a company before, instead of forcing companies to use a new set of tools fitting for their integration solution.

Figure 2.1 gives an overview over the OpenEngSB structure relevant for this thesis. The green elements are parts of the OpenEngSB. The remaining shown components are tools that use the OpenEngSB to be able to cooperate among them. To do this, each tool is connected to the OpenEngSB using a connector, which is visualised as little rectangle containing the letter C.

This section only shows an overview over the parts of the OpenEngSB relevant for this thesis. However, a complete specification of the OpenEngSB is given by Andreas Pieber [Pieber, 2010] and Biffl and Schatten [Biffl and Schatten, 2009].

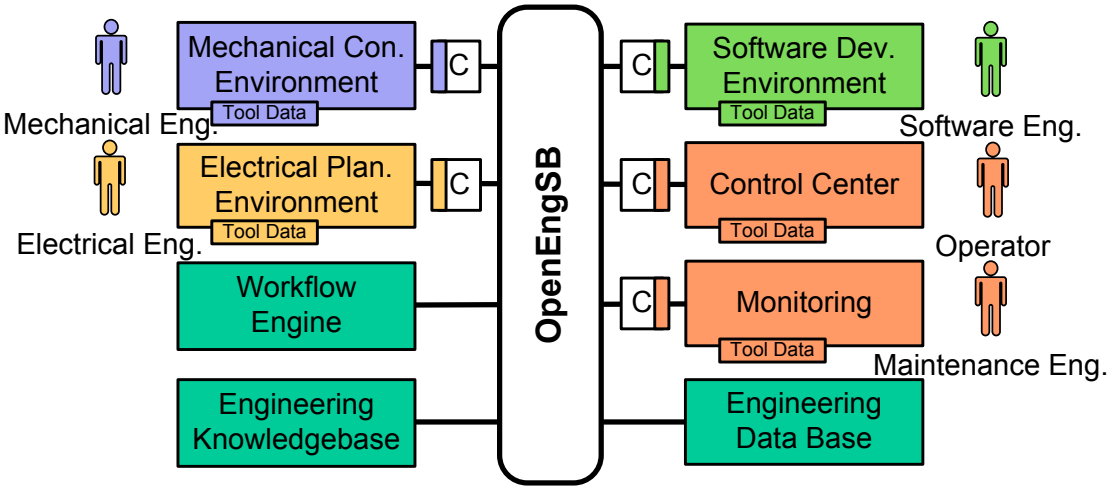

Figure 2.1: OpenEngSB Overview. Source: modified figure [Biffl and Schatten, 2009, p.7]

2.2.1 Domains & Connectors

As the `OpenEngSB` is a tool integration platform, tools need to be able to communicate with this platform. In this context, tools are user tools and not other integration technologies. The communication between the tools and the integration platform is structured in the `OpenEngSB` via the domain and connector principle (see [Pieber, 2010, page 81-90]).

Every tool communicates via one or more connectors with the `OpenEngSB`. A connector on the other hand is related to a domain, which defines in an abstract way the functionality every connector, which wants to use this domain, has to provide. It also defines the models which are used for the communication with the `OpenEngSB`.

A connector is a concrete implementation of the abstract domain to which it is connected. E.g. If there is a domain which defines how an issue tracking system should be addressed, there could be a connector for this domain written implementing a connection to a JIRA[1] instance.

2.2.2 Workflows

The `OpenEngSB` uses a workflow centred approach to allow the user to adapt working processes and enable the user reuse already existing working steps without reinventing the wheel with every project managed through the `OpenEngSB`.

To allow this, a workflow environment is needed, which is able to deal with rules and provides the possibility to trigger them. In the case of the `OpenEngSB` JBoss Drools [2] is used as such an environment.

Drools provide a Java similar syntax with which the processes of a project can be written without learning a complete new programming language. As already mentioned, the workflow

[1] `https://www.atlassian.com/software/jira/overview/`
[2] `https://www.jboss.org/drools/`

engine is a rule based solution where the rules are activated through events. These events are thrown in the OpenEngSB environment and can be processed in the engine.

Whenever an event is processed by the engine, 0 to n rules are triggered. A triggered rule now processes the event, can call active services in the OpenEngSB environment or even create new events which then could trigger other rules and so on and so forth.

2.2.3 Engineering Knowledge Base

The Engineering Knowledge Base (*EKB*) is the semantic core element of the OpenEngSB project. The EKB consists of several components which offer all semantic relevant functionality for the OpenEngSB. The concept of the EKB was introduced in the Work of Moser and Biffl [Moser and Biffl, 2010] and its details and concepts can be read in the dissertation of Dr. Thomas Moser [Moser, 2010].

At the time this thesis was written, the EKB was separated logically in two blocks: the transformation environment and the EKB persistence. The transformation environment is responsible for the transformation between models and is introduced with this thesis. Its details can be read in chapter 6.

The EKB persistence responsibility is the persistence of models. As persistence back end for the models the Engineering Database (*EDB*) (see section 2.2.4) is used. This responsibility is divided into two parts:

- **Persist Interface.** This service is responsible for the persisting of models. It takes a commit object as parameter which contains three lists of models: inserts, updates and deletes. This lists get checked and converted into an EDB usable format. When all this is done, it sends the models to the EDB for persisting.

- **Query Interface.** This service is responsible for the loading of models. Based on the function which is used to load the model(s), it queries the requested data from the EDB and converts them from the EDB format to models and return the result to the caller.

2.2.4 Engineering Database

The Engineering Database (*EDB*) is the model persistence solution of the OpenEngSB which is used by the EKB. The EDB is a versionized key/value store. This means it is possible to retrieve the saved models in any status they were ever persisted in the EDB.

The EDB was introduced in the work of Waltersdorfer et al [Waltersdorfer et al., 2010], realised in the bachelor thesis of Florian Waltersdorfer [Waltersdorfer, 2010] and was adapted and improved in the bachelor thesis of Felix Mayerhuber [Mayerhuber, 2011].

2.3 SOA & ESB

The Service Oriented Architecture (*SOA*) concept is a design approach in software engineering projects. The main point in this design approach is the separation of code functionalities in independent software pieces, which then be described as a service.

These services should be designed following the SOA service oriented principle. For example that they should only communicate through well defined interfaces or that their code should be internally independent from other services. More details about these service design principles can be found in the book of Thomas Erl [Erl, 2005].

For every service there is a specification needed, which tells the user or a possible service caller meta data about the service, e.g. which methods the service provides, which parameters the methods need or what the result of a method is.

With the SOA principle and this service specifications a loose coupled system can be constructed. Services are called in procedures only through interfaces which can be implemented by a lot of service providers. The SOA triangle shown in figure 2.2 illustrates the components which are involved in the service procedure.

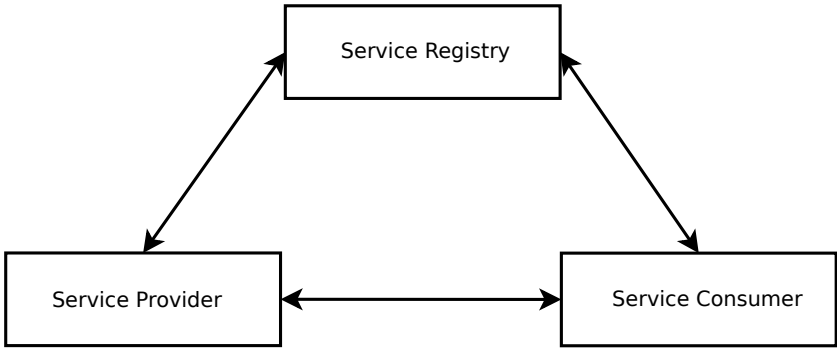

Figure 2.2: SOA triangle. Source: modified figure from [Papazoglou and Heuvel, 2007, p.392]

- **Service Registry.** The Service Registry holds a list of service specifications and the locations of the Service Providers.
- **Service Provider.** The Service Provider implements a service specification. The provider registers its implementation in the Service Registry so that consumers can use this service.
- **Service Consumer.** The Service Consumer searches for services in the Service Registry. If there is a fitting Service Provider (or a list of fitting Service Providers) the Service Registry returns the location of a Service Provider. If there is more than one Service Provider, the Service Consumer elects one of them to call the service.

The work of Papazoglou and Heuvel [Papazoglou and Heuvel, 2007] gives a good overview of the topic SOA. For a deeper insight of the SOA principles and their techniques, the book of Thomas Erl [Erl, 2005] can be read.

An Enterprise Service Bus (*ESB*) is an implementation possibility of the SOA concept. Mainly, the ESB adds an extra layer over a set of provided services, where the ESB provide features like monitoring, routing or transformation of messages between the services.

Like the bus on the motherboard of a computer, the main responsibility of an ESB is the enabling of communication between the components while the components only need to access one medium to send their messages.

An ESB is usually built on top of a Message Oriented Middleware (*MOM*) and all the features like routing or transformation of messages are built in the MOM or provided as extensions for the MOM (see work of Breest [Breest, 2006, p.6]).

For more information about the topic ESB the work of Chappell[Chappell, 2004] can be read.

2.4 OSGi

To achieve a high flexibility and reuse ability, the OpenEngSB uses Open Services Gateway initiative framework[3] (*OSGi*) as base technology for all components running in the OpenEngSB.

OSGi is a service platform specification which encourages a module based development style. Software solutions based on OSGi are separated in modules. Such a module is called bundle and represent the most basic element in OSGi. The specifications are available online at [The OSGi Alliance, 2012].

In an OSGi implementation such bundles can be added, started, stopped and removed at run-time. So there is a life-cycle defined for bundles. An illustration of the bundle life-cycle is shown in figure 2.3 and the explanation of the states can be seen in table 2.1.

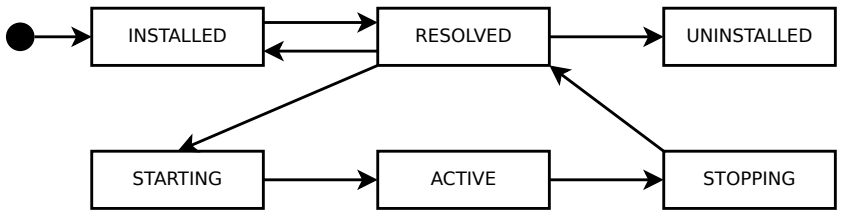

Figure 2.3: OSGi Bundle Life-cycle. Source: modified picture from [The OSGi Alliance, 2011, p.96]

Every bundle can export classes and provide services to other bundles. On the other hand, a bundle can import classes and use provided services of other bundles. In that way a dependency network is built, which can be resolved by OSGi implementations.

With OSGi the SOA communication principle can be easily realised. Referring to the OSGi Triangle explained in section 2.3 in the figure 2.2, all three entities are also present in an OSGi solution.

The OSGi environment takes the role as Service Registry. Bundles can provide services and if they do that, the OSGi environment is aware of this. A bundle which is providing a service, takes then the role of the Service Provider. A bundle which is using this service, uses the OSGi environment to search for the service and then use the service provided by the provider bundle. This bundle takes the role of the Service Consumer.

[3] http://www.osgi.org

State	Description
INSTALLED	The bundle has been installed
RESOLVED	All required imports could be obtained. Ready for start or stop
UNINSTALLED	The bundle has been uninstalled and reached the final state
STARTING	The bundle got started, but haven't finished the start process now
ACTIVE	The bundle is ready to use
STOPPING	The bundle got stopped, but haven't finished the stop process now

Table 2.1: Life-cycle States Of OSGi Bundles. Source: modified table from [The OSGi Alliance, 2011, p.96-97]

Two examples for implementations of `OSGi` are Apache Felix[4] and Equinox[5] which is also the core of the Eclipse[6] IDE.

The `OpenEngSB` uses the Apache Karaf[7] project as its `OSGi` interaction platform. Karaf builds on top of an `OSGi` implementation and provides extended features, like an enhanced console for `OSGi` commands, a hot deployment folder, easy configuration possibilities through configuration files or a built-in security framework.

2.5 Usage context

It is important to note that the introduced transformation solution is a pure Java based implementation. This is because Java is a highly used programming language in the enterprise programming section and it is compatible with the `OSGi` concept (see section 2.4) which enables or at least simplifies the realisation of the constructs introduced in this thesis.

While the transformation solution presented in this thesis is mainly described and tested in the automated software engineering area, the solution is not restricted to such usage areas.

The introduced transformation solution can be theoretically used in any Java project which has the need of model transformations, but does not want to deal with the complexity of existing transformation solutions. These solutions are mainly based on meta models and need experts in the used technologies. The proposed approach is specially suitable when there is no need for complex scenarios which require additional features of existing transformation solutions.

The introduced transformation solution does not need to meta models. As a consequence of this and due to the fact that models are represented as simple Java classes, it is very easy to extend an existing project with the presented transformation solution. E.g. if a small project has grown over time, the involved component number has massively increased so that the communication between the components would be much easier with a model transformation solution.

An example for such a situation could be, that a company developed a personnel management system for other companies. The first years they do not need a transformation possibility

[4] `https://felix.apache.org/site/index.html`
[5] `http://www.eclipse.org/equinox/`
[6] `http://www.eclipse.org/`
[7] `https://karaf.apache.org/`

because their project was used at it is and they had full control over all used models. Soon there come the wish that the project should cooperate with existing solutions in the companies.

Now, if the company does not want to develop an own translator component for every other tool it should work with, the need for a transformation solution is born. This solution should be able to convert between the tool models and the models of the personnel management system. Under the correct circumstances it only takes a few hours to enhance this system with the proposed transformation solution (see section 6.8 for the effort needed to apply the solution to an existing project).

Another reason to use the introduced transformation solution could be the built-in automated model update propagation. To support this automated model update propagation, the *`Engineering Object`* concept was introduced. With `Engineering Objects` it is possible to construct model update relations. That means, whenever an `Engineering Object` or a model which is connected to an `Engineering Object` gets updated, the *Transformation Environment* searches automatically for models which need to be updated so that the models stay consistent. More information about this concept can be read in section 6.7.

CHAPTER 3

Related Work

3.1 Overview

Model transformations are a quite common problem in the integration framework world, where different components with different data representations are used, but the need for communication between them is given. As a result of this fact, there exist a collection of already existing solutions.

In general model transformations can be separated into two families:

Model to Text (*M2T*) A `M2T` transformation transforms a model instance into a textual form representing this model instance. Although this kind of transformations are not in the scope of this thesis, they are briefly explained for the completeness of the model transformation topic.

Model to Model (*M2M*) A `M2M` transformation transforms a model instance into another instance of a different model. From an abstract point of view, a `M2M` transformation is the procedure where one or more models are taken as input for the transformation execution which result consists of one or more different models.

The remaining of this chapter is structured as follows:

Section 3.2 gives a short introduction to the `M2T` topic for the completeness of the model transformation topic.

Section 3.3 introduces the `M2M` transformation topic.

Section 3.4 gives an overview over four important existing languages with which `M2M` transformations can be described.

Section 3.5 deals with the topic of infrastructures/frameworks which provide the possibility to perform transformations based on transformation definition languages.

Section 3.6 gives an overview over the model update propagation topic, since a solution for this problem is one of the challenges of this thesis.

Section 3.7 deals with the relation of models and ontologies in the scope of this thesis. Although in the presented prototype ontologies are not used, this section gives an idea about the topic since they should be used in the near future for the model definition.

3.2 Model to Text transformations

Whenever a transformation of a model of any form into text of any form is transformed, this process is a `M2T` transformation. However, the definitions of the model and the text are not clear and so quite a lot of different transformation use cases belong to the same family of transformations. Some examples are:

- transform an UML class definition into a piece of software code.
- transform a piece of software code or parts of it into documentation text.
- transform a programming language class into an equivalent XML document.

While there are a list of existing tools to perform this kind of transformations, there exist a specification which is the de facto standard for `M2T` transformations. This standard is defined by the *Object Management Group (OMG)* and is called *MOF Model to Text Transformation Language (Mof2Text or MOFM2T)*. The specification was released in January 2008 and is available online in [OMG, 2008]. This specification is analysed and compared to alternative `M2T` languages which were candidates for the specification in the work of Oldevik et al [Oldevik et al., 2005].

This standard defines a transformation language to transform `M2T` transformations. It reuses many concepts of the *Meta-Object Facility (MOF)* which will be explained in more detail in section 3.4. There is one famous open source implementation for this standard called *Acceleo*[The Eclipse Foundation, 2006], which mainly focuses on code generation from models.

3.3 Model to Model transformations

There are a lot of existing Model to Model (*M2M*) transformation solutions. Most of them have their roots in the model-driven engineering world and are designed for specific kinds of models and/or specific transformation requirements. Publications like the report of Czarnecki and Helsen [Czarnecki and Helsen, 2006] or of Mens and Van Gorp [Mens and Van Gorp, 2006] try to classify transformation solutions based on the different aspects that the solutions provide, mostly under the scope of model driven development.

The transformation solution presented in this thesis differs mainly in one point from most of the solutions: no meta models are needed to perform model transformations. Other than that, the presented solution has the following basic properties:

- **Java based.** All transformations are performed in Java. The transformation operations are written in Java and the solution transforms from Java objects to Java objects. However, the transformation descriptions are defined in an own format to decouple definition from execution. See section 6.3 for more details about the description definition format.
- **1:1 transformation cardinality.** At the point of time this thesis was written, the solution was limited to only 1:1 transformations. That means that every transformation has precisely one model as input and one model as output. Other cardinalities will be provided in future work, see section 9.3. However, until they are available natively, they can be realised by using work-flows.
- **unidirectional.** The transformations defined are only valid for one direction. That means if there is a transformation defined between model A and model B, there is no automatically added transformation between model B and model A.
- **dynamic transformation operations.** The user of the transformation solution can add, rename, modify and delete any transformation operation, since they are decoupled from the engine and written in Java.

3.4 M2M Transformation Languages

Since M2M transformations are a common problem in the software engineering discipline, there is already a list of existing solutions and approaches available solving this problem. For example in the master thesis of Huber[Huber, 2008], the author takes a look at existing solutions and compares them. Three of the existing solutions shown in [Huber, 2008] were also validated in this thesis for fitting the Open Engineering Service Bus (*OpenEngSB*) transformation solution needs.

This thesis calls Transformation Definition Language (*TDL*) to a language for defining transformations. Some of the already existing TDL solutions were also possible candidates for the transformation solution of the OpenEngSB. However, most of these solutions were either way too complex and/or unfitting for the OpenEngSB environment, or miss some features which are needed by the transformation solution.

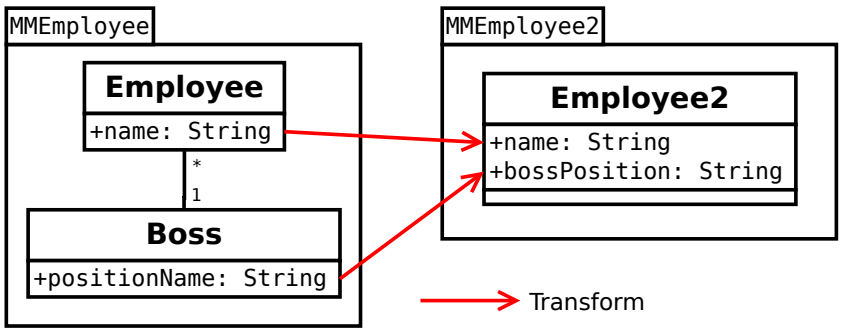

Figure 3.1: Overview TDL example

In this section a list of the most important TDL solutions including their drawbacks for the OpenEngSB are presented. Additionally examples for these TDLs will be given. All of them will be based on the simple scenario shown in figure 3.1. Based on the two meta models *MMEmployee* and *MMEmployee2*, transformation descriptions will be shown.

QVT (Query/View/Transformation)

QVT (see [OMG, 2011a] and Jouault and Kurtev [Jouault and Kurtev, 2006]) is a set of specifications for model transformation languages. The set and their exact definition was created by the Object Management Group[1] (*OMG*). As the name already reveals, these specifications cover not only the transformations of models, but also the viewing and the querying of models.

The specifications define two declarative language layers:

- **Relations.** This layer specify a declarative language which is used to define the relations between the models involved in the operation. This layers position is rather high-level and should help the user to express their needs easier than in the more low-level *Core* layer.
- **Core.** This layer operates on a lower level than *Relations*. Its main responsibilities are the pattern matching and the condition evaluation over sets of models. Theoretically, this layer is equally powerful than *Relations* and the goal of the specification is, that definitions on the *Relations* layer should be translated to *Core*, similar to the compiling procedure of programming languages where the source code (*Relations*) is transformed into byte code (*Core*). However, this is not entirely true, since it can be shown that there is no semantically equivalent transformation from *Relations* to *Core*(see the work of Stevens [Stevens, 2009] for more details).

However, these two specifications only operate at declarative level. But since there are use cases which need operational instructions too (e.g. situations where conditions or loops are needed), the OMG added two possibilities to add such instructions to *Core* and *Relations* level:

- **Operational Mappings Language.** This language defines a set of operational operations which can be used to define transformations in a more procedural style. It is even possible to define transformations only by these operations.
- **Black Box Implementations.** QVT allows the usage of custom transformation MOF operations which can be written in any language that is able to write such operations. With this custom transformation operations it is possible to outsource some more complicated operations which are difficult or even impossible to describe with the QVT built-in constructs.

```
transformation MMEmployeeToMMEmployee2
 (in source : MMEmployee, out target : MMEmployee2);

main(){
```

[1] http://www.omg.org

```
  source.objects()[Employee]->EmployeetoEmployee2();
}

mapping Employee::EmployeetoEmployee2() : Employee2 {
  name := self.name;
  bossPosition := self.boss.positionName;
}
```

Listing 3.1: Sample QVT transformation definition

Listing 3.1 shows an example for a transformation defined in operational style. This style has been chosen, since all other TDLs in this section use an operational design and in this way it is easier to compare the approaches. Each transformation has a starting method which is called *main*. In this example, all existing instances of *Employee* are given to the function *EmployeetoEmployee2* to transform them.

QVT specification implementations are mainly used with graphical user interfaces (e.g. as plug-in for the IDE), since the transformation descriptions are very complicate to write manually. Meta models, models, relations (also the conditions under which the relation is valid) and the transformations themselves need to be defined explicitly.

Transformations defined in QVT can only be applied to models conformed to MOF 2.0 (Meta-Object Facility 2.0) meta models. MOF is a standard for model-driven engineering defined by the OMG.

In general, QVT provides the following properties:

- **Model Driven Engineering (*MDE*) based.** QVT is able to work with MDE conformed models.
- **N:M cardinality.** The input and the output of the transformation can be one ore more model instances.
- **bidirectional.** All transformation descriptions are either valid in only one or in both directions.
- **hybrid paradigm.** QVT mainly operates in a declarative way, but there are extensions for using imperative instructions too.

The idea of the *Black Box Implementations* was also used in the developed transformation prototype. It is possible to define custom transformation operations for the presented transformation solution to accomplish a higher variety of use cases.

The major drawbacks of this language for the *Transformation Environment* are:

- Need to define models in accordance to MOF 2.0.
- Meta models required.
- Not easy to retrieve transformation meta data, because of the not well defined transformation description structure.

3.4.1 Atlas Transformation Language (ATL)

ATL[2] (see [The Eclipse Foundation, 2012b] and Jouault and Kurtev [Jouault and Kurtev, 2006]) is a model transformation language supported by the Eclipse[3] project. Its origins are located in the idea of developing a transformation language well suited for the challenges of MDE.

The language was created in order to create a counter-product during the standardization progress of QVT. While the origins of the both transformation languages are the same, both developed their own area of applications and differ in some significant points from each other.

Model definitions are separated in ATL into three different levels, which is according to MDE:

- **Models.** They represent the lowest level in ATL. They represent actual realisations of meta models.
- **Metamodels.** They represent the meta information about a model and model connections.
- **Metametamodels.** The meta model hierarchy is recursive. That means that also meta models need meta models. However, this level is the highest one and is normally a pre defined file which defines the structure of the meta models and stops the recursion.

Important point here is, that not only models need meta models, but also the ATL transformation descriptions, which then again needs to be conform to the meta meta model.

To describe meta models in ATL normally KM3 (Kernel MetaMetaModel) is used. KM3 is a domain specific language designed for the definition of meta models. It was designed by the developers of the ATL language and its full specification can be read in the specification written by Jouault et al [Jouault et al., 2006]. Listing 3.2 shows an example how the *MMEmployee* meta model could be expressed with KM3.

```
package MMEmployee {
  class Boss {
    attribute positionName : String;
    reference inferiors[*] container : Employee oppositeOf boss;
  }
  class Employee {
    attribute name : String;
    reference boss[0-1] : Boss oppositeOf inferiors;
  }
}
```

Listing 3.2: MMEmployee meta model written in KM3

A model instance is a realisation which is conform to its meta model. In the examples of ATL, normally XMI (XML Metadata interchange) is used to describe these instance realisations. XMI is a XML based data exchange standard which is specified by the OMG. Its specification is available online: [OMG, 2011b]. Listing 3.3 shows an instance that is conform to the meta model defined in 3.2.

[2] http://www.eclipse.org/atl/
[3] http://www.eclipse.org

```
<xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.0"
 xmi:version="2.0" xlmns="MMEmployee">
  <Boss positionName="BOSS">
    <Employee name="John Doe"/>
    <Employee name="Lucy Mill"/>
   </Boss>
</xmi:XMI>
```

Listing 3.3: Company Structure Instance in XMI

An ATL transformation needs a source meta model, a source model and a target meta model to perform a transformation. Listing 3.4 shows an example for an ATL transformation. This example uses the previously defined source meta model and model and expects a target meta model MMEmployee2 which saves a name and the position of the employees boss.

```
module MMEmployeeToMMEmployee2;
create OUT : MMEmployee2 from IN : MMEmployee;
rule EmployeeToEmployee {
  from
    s : MMEmployee!Employee
  to
    t : MMEmployee2!Employee2 (
      name <- s.name
      bossPosition <- s.boss.positionName
    )
}
```

Listing 3.4: Sample ATL transformation definition

In general, ATL provides the following properties:

- **MDE based.** ATL is able to work with MDE conform models.
- **N:M cardinality.** The input and the output of the transformation can be one ore more model instances.
- **unidirectional.** All transformation descriptions are only valid in one direction.
- **hybrid paradigm.** ATL uses both, imperative and declarative paradigms in the transformation definitions. However, the imperative style is the preferred way to define transformations.

The major drawbacks of this language for the Transformation Environment are:

- Need to define models in accordance to MOF 2.0.
- Meta models required.
- Not easy to retrieve transformation meta data, because of the not well defined transformation description structure.
- For custom functions, possibilities are limited to ATL language power.

3.4.2 Kermeta Transformation Language

The Kermeta Transformation Language (see Triskell Team [Triskell Team, 2007], Muller et al [Muller et al., 2005a] and Chauvel et al [Chauvel et al., 2007]) is specially designed to define models, and their behaviour and relations between them. Kermeta is compatible with the EMOF standard (subset of the MOF standard defined by the OMG).

Thus Kermeta is not really a transformation language, but rather a core for defining languages. However, it is well suited to define transformation languages with it like shown in the work of Muller et al [Muller et al., 2005b].

The biggest difference to most of the model driven approaches in this area is that this language is imperative and object-oriented. Developers with a certain amount of experience in the object oriented programming principle should be able to read the definitions created in this language.

```
package MMEmployee{
  class Employee
  {
    attribute name : String
    reference boss : Boss#inferiors
    operation foo(param : String) : String is do
      // put code here
    end
  }

  class Boss
  {
    attribute positionName : String
    attribute inferiors : Employee[0..*]
  }
}
```

Listing 3.5: Example Kermeta Class Definitions

Listing 3.5 shows how class definitions are structured in Kermeta. It can be seen that it is similar to other approaches like ATL for example. However, in Kermeta it is possible to define functions inside of classes and is also able to use more complicated structures like inheritance. Similar to typical programming languages, Kermeta can use typical programming structures, like conditions, loops and exceptions. Also it is possible to call methods of external Java code, which allows a wider range of possible transformation problems which can be solved with the help of Kermeta.

Like already mentioned, Kermeta is not directly a transformation language, but is used to define models and the relations between them. Thus the solution to use Kermeta as transformation language is to use relations to express transformations.

In this relations, there is custom code written which express the transformation instructions needed to perform the transformation. Thus Kermeta does not define a set of transformation instructions, but rather ask the developer to create an own relation for each transformation description. In these relations then the Kermeta imperative instructions is used to program the rules of the description.

```
operation transform(emp:MMEmployee::Employee) : MMEmployee2::Employee2 is do
  var emp2:MMEmployee2::Employee2 init MMEmployee2::Employee2.new
  emp2.name := emp.name
  emp2.bossPosition := emp.boss.positionName
  result := emp2
end
```

Listing 3.6: Sample Kermeta transformation operation

Listing 3.6 shows an example how such a transformation could look like. This operation transforms an *Employee* model instance into an *Employee2* model instance.

In general, Kermeta provides the following properties:

- **EMOF based.** Kermeta is able to work with EMOF conform models.
- **N:M cardinality.** The input and the output of the transformation can be one ore more model instances.
- **unidirectional.** All transformation descriptions are only valid in one direction.
- **imperative paradigm.** Kermeta uses the imperative paradigm when writing relations and models.

The major drawbacks of this language for the Transformation Environment are:

- Only a pseudo `TDL`. Kermeta is only used to define models and their relations. Transformations can be seen as a relation, thus transformations can be realised via a relation.
- The models need to be defined in Kermeta and that they need to be EMOF conform.
- Meta models required.
- It is very difficult to retrieve meta data about transformations with no defined *Transformation Operations*.

3.4.3 Smooks Transformation Language

Smooks[4] (see [Smooks developers, 2011]) is generally a communication framework which provides features like message delivering, message enrichment and also transformations. Smooks uses internally an own XML based format, from which a list of supported formats can be created. The same list of formats can be used as input source for the creation of the XML based format. One of these formats is Java.

So in order to be able to transform model transformations, Smooks transforms a model at first in its internal representation, which then can be transformed in a different model. Since these models can be Java objects, this approach does not need meta models, very similar to the presented transformation solution.

In the following, a simple transformation example is given, which uses Java objects as source and target models. Listing 3.7 defines the two source model classes, which will be transformed

[4] `http://www.smooks.org`

to a model of the model class given in Listing 3.8. Note here that there is no need to define the meta models for the models to define the transformations!

```
public class Boss {
  private String positionName;
  private List<Employee> inferiors;

  // getter and setter
}

public class Employee {
  private String name;
  private Boss boss;

  // getter and setter
}
```

Listing 3.7: Company Structure Source Java classes

```
public class Employee2 {
  private String name;
  private String bossPosition;

  // getter and setter
}
```

Listing 3.8: Company Structure Target Java classes

As base for the transformation, a configuration file of Smooks needs to be created. In this configuration file other mechanisms can be used, like message enrichment. For the example a simple transformation description is created with the name 'smooks-example.xml'. Listing 3.9 shows this description. The description defines:

- the value of the property 'name' from the source model will be set to the property 'name' of the target model.
- the value of the property 'positionName' which is present in the property 'boss' of the source model will be set to the property 'bossPosition' of the target model. Note here, that the property 'boss' is another Java class.

```
// put here smooks xml header
<bean beanId="emplNew" class="target.Employee2"
  createOnElement="source.Employee">
  <value property="name" data="name" />
  <value property="bossPosition" data="boss/positionName" />
</bean
```

Listing 3.9: Example Smooks Transformation Description

To perform a Smooks transformation description, it need to be started manually in a piece of code. Listing 3.10 shows an example how to do that. The example first loads the earlier created

description file. After the initialisation of the context and the definition of the source and result object, the transformation is started with the `filterSource` command.

The context is searched for a fitting transformation definition for the source object. After that, the result can be loaded from the result object. Note here, that the bean 'emplNew' must be explicitly called to get the transformed object based on the bean which was earlier defined. This is the case, since in this description there could a lot of transformations be defined in the description file.

```
private Employee2 performTransformation(Employee employee) {
  Smooks smooks = new Smooks("smooks-example.xml");
  ExecutionContext executionContext = smooks.createExecutionContext();
  JavaSource source = new JavaSource(employee);
  JavaResult result = new JavaResult();
  smooks.filterSource(executionContext, source, result);
  Employee2 emp2 = (Employee2) result.getBean("emplNew");
  smooks.close();
  return emp2;
}
```

Listing 3.10: Use Smooks in Java to perform a transformation

In general, Smooks provides the following properties:

- **XML based.** Internally every model is transformed first into an internal XML based format which is then transformed to another format.
- **1:N cardinality.** The input of the transformation can be one model instance and the output of the transformation can be one ore more model instances.
- **unidirectional.** All transformation description are only valid in one direction.
- **imperative.** Since Smooks is only a format changing tool, there are no declarative elements present.

The major drawbacks of this language for the Transformation Environment are:

- Only a pseudo `TDL`. The transformations are only a small part of the whole Smooks function set and the transformation descriptions are mainly mappings and not really transformation operations.
- It is very difficult to retrieve meta data about transformations with no defined Transformation Operations.
- Custom functions are not easily added. They need to be implemented as SAX event handlers.
- There is a huge amount of unneeded functionality in this language (since it is not only meant for transformations).
- Unneeded transformation Java object $\leftrightarrow$ internal format $\leftrightarrow$ Java object.

3.5 M2M Transformation Infrastructures/Frameworks

TDLs are needed to describe transformations. However, there are also infrastructures/frameworks that actually can perform the described transformations. A common approach to accomplish this is to use *Transformation Performers*. A transformation performer is capable to perform a transformation based on a description given in one TDL. A performer needs to be able to perform all operations which can be described by the TDL for which this performer is developed. At the performer creation level, there is usually the first validation of the usability and sanity of the TDL. If there are features which can be described, but it is technically impossible to perform them, there is something wrong in the TDL. An example for such a component is described by Ma et al [Zhibin et al., 2006]. This paper shows how such a component for OWL scripts can be designed and implemented.

The Transformation Environment of the OpenEngSB also uses a transformation performer. The *Transformation Engine* has its own performer which is able to perform transformations on models based on the Dynamo Transformation Language (*DTL*) transformation language. This performer is able to handle temporary variables and call operations dynamically from the OpenEngSB environment (more information about this topic is given in section 6.5).

However, transformation infrastructures/frameworks normally use one or a list of performers to support TDLs. Such solutions can be integrated into integration systems at different levels. So there are a list of solutions which not only deal with the transformations, but also perform other communication related features like message routing, message enrichment or security aspects.

The Transformation Environment presented in this thesis is an example for a transformation framework. It uses the TDL introduced in this thesis (see 6.3) and provides a transformation performer which is able to perform transformations based on *Transformation Descriptions*.

This section gives some examples for other transformation infrastructures/frameworks and explain the differences to the Transformation Environment.

3.5.1 ModelBus

ModelBus[5] is a tool integration framework with special focus on model-driven engineering. Its idea is the providing of services and encourage the users to add their own services too. Services in the resulting service collection can then be combined to create new services. In this way, the functionality grows over time.

As the OpenEngSB, ModelBus is designed by the Service Oriented Architecture (*SOA*) communication style. It is possible to add new tools to the ModelBus environment by writing connectors for commercial of the shelf tools which then expose the features of the tool for the ModelBus service pool.

ModelBus provides some built-in services like tracing or verification of models and services. However, the most important elements in this built-in concepts for the scope of this thesis is the *Model Storage* and *Transformation*.

Model Storage is the core component which is used for saving the model instances of the environment. SImilar to the Engineering Database (*EDB*) it is a versionised model storage

[5] http://www.modelbus.org

solution. The biggest difference here, is that the models need to be MDE conform, since the whole model interaction are defined in model engineering manner.

Transformation is the core component which performs the transformations. It uses ATL (see section 3.4), therefore the models need to be MDE conform.

The structure and features of ModelBus are described by Hein et al. [Hein et al., 2009]. The processing and orchestration of the services exported by tools is described in the work of Aldazabal et al. [Aldazabal et al., 2008]. However, the most recent information about ModelBus, a list of currently supported tool connectors and the documentation of the ModelBus functionality can be read online at [Modelbus Team, 2011].

3.5.2 Kermeta

Kermeta[6] is a very powerful environment for the meta model engineering process. For example, with Kermeta it is possible to define meta models, attach behaviour to them (e.g. by giving meta models states), add constraints to models (or only model properties) and also define relations between models. Based on these descriptions, Kermeta is able to create prototypes, run simulations and also perform model transformations.

Kermeta is mainly used as an Eclipse plugin. This extension provides the user a lot of GUI elements which eases to perform tasks such as: the definition of the descriptions, the writing of Kermeta rules (thanks to syntax highlighting) and the verification of models by the simulation functionality provided by the environment.

The Kermeta language creates definitions which are compatible with the standard defined in the EMOF specification. That means that definitions can be also used by non Kermeta environments. However, not all of the functionality will work in both situations, since Kermeta extends the EMOF specification. In the other direction, all definitions according to the EMOF specification can be used with Kermeta.

The Kermeta language was already introduced in section 3.4. Kermeta as a transformation environment has essentially the same major drawbacks. Another point here is that it is delivered as plug-in for the Eclipse IDE and there is no standalone Kermeta infrastructure available yet.

More information about this transformation solution and examples how it can be used are available online [Triskell Team, 2007].

3.5.3 Smooks

Smooks is a framework used to build the communication links in bigger enterprise solutions. It is specialized to construct the processing logic of enterprise solutions with a wide range of supported data formats. It does not only all the transformation work between and within data formats, but it is also capable of doing the message processing of a complete solution.

In the core concepts of Smooks, the transformation is only a small part of all its features. The core of Smooks is an event stream processor of structured data. To be able to process this

[6] http://www.kermeta.org

data in a way independent of the source data formats, Smooks introduces an internal XML-based format.

Since the internal engine of Smooks always have the same format to operate with, it is enough to implement all functionalities against this one format. In that way, Smooks can be seen as a bus where connectors can be used to connect to this bus. Each connector is responsible for one data format and so does the transformation between the internal model and the own data format.

As such a mighty communication solution, the features of these technologies are much more than the ones needed for a simple transformation environment. However, with the definition of the `TDL` of the Transformation Environment, one major interest was the easy to use and understand format of the transformation definition files. Smooks files are quite complicated due to the high amount of possibilities and features.

Since Smooks uses its own pseudo `TDL` (see section 3.4) and the transformation possibilities of Smooks as a transformation environment are exactly the same like the ones from its `TDL`, it has the same disadvantages like the `TDL` for the Transformation Environment.

More information about this transformation solution and examples how it can be used are available online [Smooks developers, 2011].

3.6 Model Update Propagation

Whenever there are related model instances in an environment, where the change of one model instance can influence a set of other model instances, it is not an easy task to ensure the consistency of these model instances manually. With the adding of a transformation solution it would be great if the solution is also able to deal with this problem.

However, there is no fully functioning solution for this problem. There are some approaches to enable the update propagation automatically, but there is no perfect approach found so far. The *`Engineering Object`* concept introduced in this thesis is another approach to deal with the automated model update propagation. In environments where similar models have a common concept model, this approach works quite well (see the work of Winkler et al. [Winkler et al., 2011] and section 6.7).

Xiong et al. [Xiong et al., 2007] show a concept and an implementation of an update propagation mechanism in an ATL based environment. Here the authors expect an uni-directional transformation description between models, so that they propagate changes to the models which have a transformation described where the changed model is the source model.

This concept was adapted for the transformation solution of the `OpenEngSB` to help finding a possibility to perform update propagation between the `Engineering Objects` and its referenced models (see section 6.7 for more information). In contrary to the solution presented in [Xiong et al., 2007], the `Engineering Object` concept need Transformation Descriptions defined in both directions (from the models to the `Engineering Object` and vice-versa) to work correctly. However, the presented solution has the drawback that the update propagation takes a high amount of time, since every model which is connected through any transformation description with one updated model gets also updated.

Also a very interesting paper is written by Sindhgatta et al. [Sindhgatta and Sengupta, 2009]. The authors deal with the problem of systems which use model-based developments and show a solution which analyses changes of rather static models and find out what the effect of these changes for other models means. It also introduces a graphical user interface, with which the users are able to see what the changes would be and which allows them to accept or decline the change. The drawback of this approach is, that the models are static in contrary to the dynamic environment of the `OpenEngSB`.

Parts of the concepts shown in this paper are also used in the `OpenEngSB` transformation solution. E.g. the Transformation Environment can calculate, based on Transformation Descriptions, which fields of the source model influence which fields in the target model.

3.7 Models/Ontologies

As soon as there is the discussion about models and the relation between them, ontologies are often mentioned. Ontology is a heavily overloaded term which has its origins in philosophy. From an informatics point of view, ontologies are the formal, structured description of entities which abstract elements of the real world.

As such, models (or classes in the ontology language) are one of the main components of ontologies. A model, from the ontological view, is the description of an element. These models can now be looked at under a specific domain. Every model can have multiple meanings based on the active domain. Based on the domain, there can be additional information added to the models, like attributes, rules for the model attributes or relations between models.

Ontologies are a very actual research field. Many papers are written how ontologies can be used, extended and reused for many use cases. Especially in the field of artificial intelligence, semantic web and other disciplines where the structure of data plays a key-role, ontologies are widely used.

With ontology engines you can save, maintain and query for information in a well structured way. Also transformations can be described and realized through ontologies (like shown in the work of Ma et al [Zhibin et al., 2006]).

In the work of Natalya Noy [Noy, 2004] a good overview over the topic ontologies is given. For example, it deals with the definition of ontologies or in which situations ontologies could help. In this paper, some open problems and challenges of the ontology topic are discussed.

Uschold et al [Uschold and Gruninger, 2004] deals with the problem how ontologies can be used in an enterprise integration solution. The paper presents a framework which enables two independent information system to work with each other through agents. The different models and the relations between them are defined by ontologies.

For the presented solution, ontologies are of a minor interest, since the model definitions are not ontology based at the point of time this thesis was written. However, the idea of defining relations between models was adapted for the transformation graph.

For the `OpenEngSB` they have an additional value. So far, they are not used by this framework, but it is planned that in near future the model descriptions are placed in ontology databases, to be able to use ontologies e.g. to reason over models or formally defining semantics for them.

CHAPTER 4

Use-case descriptions

4.1 Overview

In this chapter the both use-cases used to evaluate the implemented prototype are explained.

4.2 OpenEngSB Domains, transformation functionality

In every integration environment there are a lot of different tools present. These tools come with their own data formats and models, since the tools need to be runnable on their own without the integration environment.

Since these tools use their own models, but should also be able to communicate with different tools there is the need for translation of the models from one tool format into a different tool format. The easiest approach for dealing with this translation is to do the translations directly in the tool. However, this solution have some major drawbacks, especially the huge amount of translation code in the tools which grows with every added tool.

The most flexible and efficient way to deal with this translation problem is to have an own component which is independent of the tools and can be configured in an easier way than do updates in every tool. This is the approach followed in this thesis.

A highly common use case in this area is that there exists an integration solution that provides a collection of models (e.g. the Open Services for Lifecycle Collaboration (*OSLC*) integration solution, see OSLC Team[OSLC Team, 2011]). If other tools want to participate in the integration solution, they need to either use the provided models directly (which is normally never the case), or to translate their models to the provided models and the other way round.

If there is a model transformation solution used, then the translations of the internal models to the provided models and vice-versa can be defined in transformation descriptions which are performed by the transformation solution. In that way, if either the internal or the provided models are changed, only the descriptions need to be changed, but the code of the tool can remain untouched.

The presented use-case will use two domains with their models, which are provided by the Open Engineering Service Bus (*OpenEngSB*). For each of the domain models, there is an additional semantically equivalent but syntactically different (e.g. different property names and/or types) model defined. These additional models simulate the presence of models which are provided by another integration solution.

Figure 4.1 shows the structure of the models and their relations to each other. The question marks indicate that at this point a transformation solution need to be placed which performs the transformation between the models.

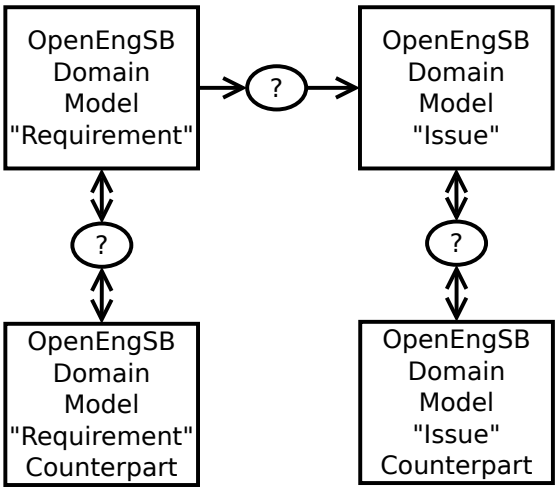

Figure 4.1: OpenEngSB Domains Use-case Overview

It needs to be possible to transform between the domain models and their syntactically different counter parts. Also a transformation will be defined between the both domain models, so that it is possible to transform directly between the two counter part models.

In this context, the following defined scenario was created:

To simplify the scenario description, the models shown in figure 4.1 got the following name schema:

- 'Issue' which resembles the model for the issue domain.
- 'ExternalIssue' which resembles the counterpart of the 'Issue' model.
- 'Requirement' which resembles the model for the requirement domain.
- 'ExternalRequirement' which resembles the counterpart of the 'ExternalRequirement' model.

The counterparts are models for some external tools which need to have a specific format to work with (e.g. `OSLC`). For this scenario it is assumed that the `OpenEngSB` can receive 'ExternalRequirement' models from an external tool(e.g. a planning tool) and can send 'ExternalIssue' models to an external tool(e.g. an external issue tracker).

1. An 'ExternalRequirement' model instance is received by the OpenEngSB from an external tool. This instance is analysed and it was found that this instance is also interesting for the developers of the project running in the OpenEngSB instance. The project uses its own requirement planning tool, which has its own model format. So it is necessary to transform the 'ExternalRequirement' model to the model of the internal planning tool, namely 'Requirement'. As soon as this transformation is performed, a semantically equivalent but syntactically different model instance is at hand.

2. The 'Requirement' model instance can now be reviewed, modified or simply saved depending on the logic defined in the OpenEngSB. For this scenario we assume that the created 'Requirement' instance is finished. However, since the planning of a new feature is only the first step in a software development process further actions need to be initiated.

   In this scenario, the next step is to create an 'Issue' model instance out of the 'Requirement' model instance. In this way, an issue for this requirement is created, which can be used in an internal issue tracker. For the transformation from a 'Requirement' into an 'Issue' model there are several possibilities, since both models are not semantically equivalent. For that reason, there are multiple transformation definitions defined between these two models. The decision which of these transformations are performed can be done in every way the developer wants to. In the scenario, a specific transformation will be picked, which already assigns a developer to the issue.

3. The 'Issue' model instance created in point 2 can now be published in the internal issue tracker. However, this issue should also be published in an external issue tracker. This can be possible e.g. if there are more than one issue tracking system is used in a project.

   To be able to send this 'Issue' model to an external tool, it first needs to be transformed into the model format needed by this external tool. So, similar to point 1, the instance is transformed into its semantically equivalent but syntactically different counterpart 'ExternalIssue' model.

4. With step 3, the first received 'ExternalRequirement' model instance is fully taken care of. After that a new 'ExternalRequirement' model has been received by the OpenEngSB, like already happened in the first step. The analyse of this instance results in the finding that it is not of interest for internal use. This may happen if the requirement and its issue should not be handled by the internal mechanisms but is only handled outside of the OpenEngSB scope.

   In this case there is the need to send this requirement to the external tool but without intermediate steps, since it do not need to be modified on its way to the external tool. With the models and the transformation descriptions shown so far, this would work directly, by just giving the transformation solution the request to transform between 'ExternalRequirement' and 'ExternalIssue'.

   However, in the mean time the 'Requirement' model may not available any more. This may happen if the bundle which contains the connection logic to the internal planning tool was shut down or if this bundle crashed because of a software error. Therefore, a

transformation between 'ExternalRequirement' and 'ExternalIssue' is not possible any more, since there is no connection between them.

To repair this and enable the direct transformation again, a new transformation description is added in this scenario. It will connect the 'ExternalRequirement' model with the 'Issue' model. In that way it is possible again to find a path in the transformation graph to directly transform from 'ExternalRequirement' to 'ExternalIssue'.

4.3 Andritz Hydro, model update propagation

In the Andritz Hydro environment different disciplines of technicians work together to create the different parts that are needed for building power plants. Specifically, three different domains are involved in the development of hydro power plants:

- software engineering domain.
- electrical engineering domain.
- hardware configuration domain (i.e. a supplier domain responsible for providing the base elements for the parts).

Each one of these domains work with a different tool and its corresponding data model. However, for being able to build the hydro power plant, these heterogeneous models must be integrated to be able to exchange their data.

These tools of the different professions store their data models in local data sources, which produce and/or consume data with heterogeneous data structures. Specifically, the data model of the software engineering domain contains information about programmable logic controller (PLC) variables (e.g. name, type - boolean or float, etc.), the data models of the electrical domain comprises information about monitoring sensors (e.g. id, type - digital or analog, etc.); and hardware configuration domain contains meta information about the parts (e.g. price, date of creation, etc.).

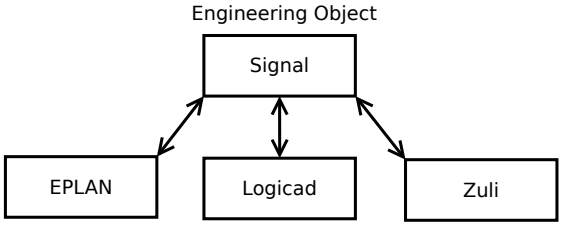

Figure 4.2: Andritz Hydro Use-case Overview

From these three domains, a common concept can be extracted which contains the information of a part in a generic way. Figure 4.2 gives an overview over the models and their connections to each other. EPLAN is the tool for the electrical engineering domain; Logicad is

the tool for the information engineering domain and Zuli is the tool of the hardware configuration domain. All these tools define a model which has the same name like the tool. The signal is the common concept of these three models which contains all information about a part.

All mentioned tools work independently from each other, but the models are not really independent. E.g. a software engineer can not simply add another variable without a proper electrical circuit on a board. In addition, when a data model of one domain is changed, this often has an impact on the models of the other domains, which makes the propagation of changes necessary.

Thus, there is the need of the defining model update propagation connections between models, so that the changes of one model in one domain, also changes all influenced values of all related models of other tools.

In this context, the following defined scenario was created:

To simplify the scenario description, the models shown in figure 4.2 got the following name schema:

- 'Eplan' which resembles the model for the electrical engineering domain.
- 'Logicad' which resembles the model for the software engineering domain.
- 'Zuli' which resembles the model for the hardware configuration domain.
- 'Signal' which resembles the common concept for the other three models.

These four models are not exactly the same like the ones used in the Andritz Hydro environment since they are not public domain, but they are conceptually equivalent. In this way they are valid candidates for the use-case validation implementation. For simplicity reasons, the identifier schema of the model instances have been changed in that way that related models have the same suffix number but a different prefix (e.g. signal-1, eplan-1, ...).

1. The Andritz Hydro company gets a new order. For simplicity reasons and because quantity does not help for the understanding of the concept, this order is conceptionally only one 'Signal'. The first step is, that a hardware configuration engineer searches for a fitting piece of hardware and saves the corresponding 'Zuli' model. In background, a new signal is created, which references to the 'Zuli' model and to the corresponding 'Eplan' and 'Logicad' models, even though they do not exist yet.

2. After that, an electrical engineer starts planning based on the hardware specified by the hardware configuration engineer. During the planning the electrical engineer see that this hardware is not fully functional for the order. In the models there is a boolean value which defines for every domain, if the domain is satisfied. So the electrical engineer sets this value to false and adds a comment why this hardware is not fitting. After saving the corresponding 'Eplan' model, also the 'Zuli' model updates, so the hardware configuration engineer can see that, and act accordingly.

3. The hardware configuration engineer found a new hardware which fits the needs of the order and saves the 'Zuli' model. The 'Eplan' model gets updated accordingly and the electrical engineer can continue working on his part of work. After he has finished, he saves his configurations and his domain is satisfied.

4. After that, the software engineer starts his work by creating his 'Logicad' model and saves it. With the next load of his model, all values which have been set so far by the electrical engineer and the hardware configuration engineer are set in his model. The software engineer does his changes and is not quite satisfied with the work of the electrical engineer. He sets his domain to be not fulfilled and writes to the electrical engineer that some changes are needed.

5. The models get updated as long as there is a configuration found where every domain engineer has set his domain to be satisfied. After that, the construction of the so planned part can start. Without the automated model update propagation there could easily some mistakes regarding synchronization and consistency be dropped, which could cause serious problems during the construction or after the delivery of the product.

CHAPTER 5

Research issues

5.1 Overview

In this chapter the research issues and their goals for this thesis are explained in detail. First all research issues are named and illustrated. Afterwards the research method used in this thesis is described.

5.2 Research Issues

In many software engineering projects and tool integration solutions there is the requirement of model to model transformations. Whenever data exchange should be enabled between tools, a solution need to be found how the translation between the different models can be done.

The common approaches are either to perform the translations manually in the tools or to use one of the existing model transformation approaches available. Both have their disadvantages. E.g. the manual approach grows rather complex if there are a lot of possible exchanging tools. On the other hand, the existing model transformations can help reduce the amount of code since the translations are exported to configuration files, but have the drawback that all solutions are rather complicated and/or bloated. Normally there is an expert needed, which is able to use the specific transformation solution.

This thesis deals with the transformation solution developed for the Open Engineering Service Bus (*OpenEngSB*) project. The goal is to introduce a prototype which is easier to use than the existing transformation approaches (e.g. by removing the need of meta models), and also to overcome the problems of the manual model translating process.

For the planning and developing of the `OpenEngSB` project transformation solution prototype, three research issues were defined:

- Representation and handling of models and transformations in the OpenEngSB project.
- Comparison of existing TDLs with the developed TDL in OpenEngSB context.

- OpenEngSB run-time mechanisms to enable an efficient transformation process.

In the following sections, the three research issues are explained in more detail.

5.2.1 Representation and handling of models and transformations in the OpenEngSB project

This research issue deals with the question how models and the definitions of transformations should be structured and how they interact with each other. Also an important point to consider here is the architecture of the components involved in the transformation process.

While almost all existing transformation solutions/approaches require models conformed to Model Driven Engineering (*MDE*), one important point for the models in the `OpenEngSB` project is that they should be represented as normal Java classes.

In the prototype implementation, the decision was made that the models stay normal Java classes. This approach allows existing projects to be easy extended to support model transformations. The models can stay as they were before except that they need to be annotated as models.

For the transformation definitions used by the prototype there were a list of requirements defined:

`OpenEngSB` models compatible Models are just normal Java classes in the `OpenEngSB`. Since there is no more information about a model (like it is the case with meta models in other approaches) all transformation descriptions need to access the properties of the models. Additionally all components involved in the transformation process perform their actions on `OpenEngSB` models.

dynamic transformation operation list In Transformation Definition Languages (*TDLs*) there exist normally a collection of operations which can be used to describe the transformations. However, it may happen that this collection does not satisfy all needs of the user or that the user want to rename some operations (e.g. if he is used to a name from a different `TDL`).

To satisfy this requirement, the solution was found that *Transformation Operations* are loaded from the *Transformation Environment*. Every bundle can provide its own operations by exporting a service with a specific interface. This interface defines all methods needed for a Transformation Operation, including documentation how to use this operation. With this approach it is possible to add/modify/delete Transformation Operations at run time.

transformation definitions can be placed in configuration files It is possible to write the descriptions of the transformations in pure Java. Since this is approach is not very dynamical and also forces the transformation describer to write Java code, it would be better to allow the describer write the descriptions in configuration files. This also allows the changing of the transformation descriptions without touching the source code.

To satisfy this requirement, the restriction was given to provide a transformation description format which is not written in Java. So a XML like language was defined, which

is able to express the transformation descriptions. Also a file watcher was added, which scans the configuration folder of the OpenEngSB for transformation description files. In that way, the definition is separated from the execution of the transformations.

Based on these requirements, a new TDL was introduced (named Dynamo Transformation Language (*DTL*)). More details about the introduced TDL can be read in section 6.3.

Additionally to the new TDL, the following components were added to the OpenEngSB project to enable the transformation process:

Transformation Engine This component is aware of all active *Transformation Descriptions* and is responsible for the actual execution of transformations. Also it is the main connection point for the user to the Transformation Environment.

Model Registry This component is responsible for the location and loading of the models in the running OpenEngSB instance.

EKB Graph DB This component is a graph based Not only SQL (*NoSQL*) database. The nodes of the graph are the models and the edges are the transformation descriptions of the running OpenEngSB instance. It gets the information for the nodes and edges from the Transformation Engine and the Model Registry. It uses the graph to find *Transformation Paths* so that the Transformation Engine is able to transform one model into another one.

More details about the components and their responsibilities can be read in section 6.2.

5.2.2 Comparison of existing TDLs with the developed TDL in OpenEngSB context

During the development of the OpenEngSB transformation prototype, a new TDL was created with the name DTL. Before this TDL was created, there was the question if an already existing TDL can be used for the prototype.

To answer this question, this research issue was defined. The main problem about this issue was to find a way to check if a solution is sufficient for the OpenEngSB transformation prototype. To accomplish this, first the requirements of the TDL were defined.

All TDLs described in section 3.4 plus the TDL defined in this thesis have then been checked if they fulfil these requirement. Since a normal binary decision (yes/no) was not enough, also a third possibility was introduced. This possibility refers to special circumstances like for example that the requirement can be fulfilled with some work around or additional components.

It is shown that based on the requirements every analysed TDL had some major drawbacks, which stood against the requirements. This leaded to the decision to define an own TDL which fits to the given requirements. In section 7.2 a detailed explanation of the requirements is given including the results of the comparison.

5.2.3 OpenEngSB run-time mechanisms to enable an efficient transformation process

This research issue deals with the problem of how it is possible to perform transformations in the `OpenEngSB` in an efficient way:

model weaving Like already mentioned, the decision was made that models in the `OpenEngSB` are normal Java classes. However, there is a list of functionality every model has to provide so that the Transformation Environment can work with it. The implementation of this functionality is a repetitive and quite static programming task which would look quite the same in every model class.

To avoid this programming task, a different solution was found. Instead of letting the programmer write all these functions, a byte code manipulation of these classes implements them. The Java classes representing models just have to be annotated as models and then they get enhanced with the model functionality at run-time. In that way, all complexity only needed for the model related tasks are hidden from the user and the models just behave like normal Java classes. For more information about this topic, see section 6.6.

transformation paths When requesting the Transformation Environment to transform a model instance from its source model to a target model, this is an easy task if a Transformation Description is defined between them. However, if there is not a direct Transformation Description defined between them, it is necessary to check if there is a possible transformation path between those two models in the transformation graph. If a path can be found, the transformation can be performed. See section 6.4 to read how the path search works.

model providing It was already pointed out that source and target objects for transformations are Java classes or more precisely Java classes with a specific annotation set. This circumstance brings benefits for the user, since the user can work with models like normal Java objects and use them directly without any intermediate steps.

However, this feature brings also a problem with it: The Transformation Environment needs to be able to provide models to the `Transformation Engine`, since this component needs to instantiate model instances. In normal Java environments, this is not a big deal, since all classes of the project can be loaded without any problem. In Open Services Gateway initiative framework (*OSGi*) environments it is not that easy. There is the need to load model classes from any bundle providing models in a dynamic way.

To accomplish this, an additional library was used, which is designed to let `OSGi` bundles provide classes which can then be loaded from any bundle. This enables a dynamic and robust class loading mechanism (see section 6.6 for more information).

model update propagation In many environments, there are models which influence other models. If model instances are changed, this may cause a needed change in other model instances. To add a way to support such environments with an automated model update propagation, the *`Engineering Object`* concept was defined in this thesis practical

part. With this concept, it is possible to link a list of model instances together and, based on Transformation Descriptions, model update propagation can be achieved. For more details about this concept, see section 6.7.

5.3 Research Approach

To deal with the research issues previously defined, an adapted research methodology based on *Constructive Research* (see work of Kasanen et al [Kasanen et al., 1993]) is used. The complete process is divided into a list of sequential steps. In this section these steps are named and described:

Define Requirements The first step was to gather the requirements for the `OpenEngSB` transformation solution. To derive them, discussions were held with the `OpenEngSB` developers, involved scientists and involved `OpenEngSB`-using companies. The outcome of this phase was a common list of requirements.

Literature Research The research for the state-of-the-art in the model to model transformation scene started. The main sources were literature and tool descriptions out of the areas:

- Model transformation approaches.
- Model transformations in Enterprise Application Integration solutions.
- Model transformations in Service Orientated Architecture environments.
- Ontologies used in Software Engineering and System Engineering.

In this step, also the analysis of the most important existing `TDLs` were done in order to check them for compatibility with the defined requirements.

Use-cases Definition In order to be able to check if the later developed prototype is working as expected and can be used in a practical environment, use-cases were defined. These use-cases are used in the prototype validation and also in the orientation during the development of the prototype concept and implementation. The use-case definitions for this thesis are described in chapter 4.

Prototype Concept A design for the prototype was developed. The concept contained the definitions of the components needed to be developed for the prototype and defined how they interact with each other. In addition to that, a new `TDL` was created, which fulfils the needs of all requirements in an easy and understandable way.

Prototype Implementation The designed prototype was implemented. In order to archive a high acceptability of the prototype, the implementation was divided in a list of development phases. With every completion of a phase, the result of this part was checked and discussed with the `OpenEngSB` developers and involved scientists.

Prototype Validation The finished prototype was validated against the use-cases defined in a previous step. To accomplish this, the previously defined use-cases were implemented and the prototype was integrated in the use-case environment. Then the use-cases were performed and the results were checked for validity. With this validation it could be concluded that the prototype is working as expected. Based on this prototype functionality, more sophisticated functionalities can be realised. In chapter 8 are some examples of such features are listed including benefits of realising these with the introduced prototype.

CHAPTER 6

Model Transformations in OpenEngSB project

6.1 Overview

This chapters goal is the explanation of the components, core concepts and mechanism descriptions which are involved in the *Transformation Environment*. It also explains the structure of the developed transformation definition language and how this language can be used to transform model instances. The chapters structure is:

Section 6.2 illustrates an overview over the prototype solution and explains the proposed architecture (involved components).

Section 6.3 explains the TDL proposed for achieving model to model transformations in the Open Engineering Service Bus (*OpenEngSB*) in detail.

Section 6.4 shows how *Transformation Paths* are searched and found in the transformation graph.

Section 6.5 illustrates the basic internal procedure of a transformation process.

Section 6.6 gives all needed information to annotate classes as models and explains details of the procedures in background.

Section 6.7 explains the model update propagation and how it is realised with the help of the `Engineering Object` concept.

Section 6.8 gives an overview how an existing project can be extended with the introduced prototype.

6.2 Transformation Environment

This section describes all components which are involved in the transformation procedure. All components as an overall concept will be called Transformation Environment.

Figure 6.1 gives an overview over the structure of the Transformation Environment. The *Model Registry*, the *Transformation Engine* and the *EKB Graph DB* form the core of the transformation solution. The other parts represent the context where the Transformation Environment is running. All components are implemented as Open Services Gateway initiative framework (*OSGi*) bundles and are optimized and tested with within the OpenEngSB framework.

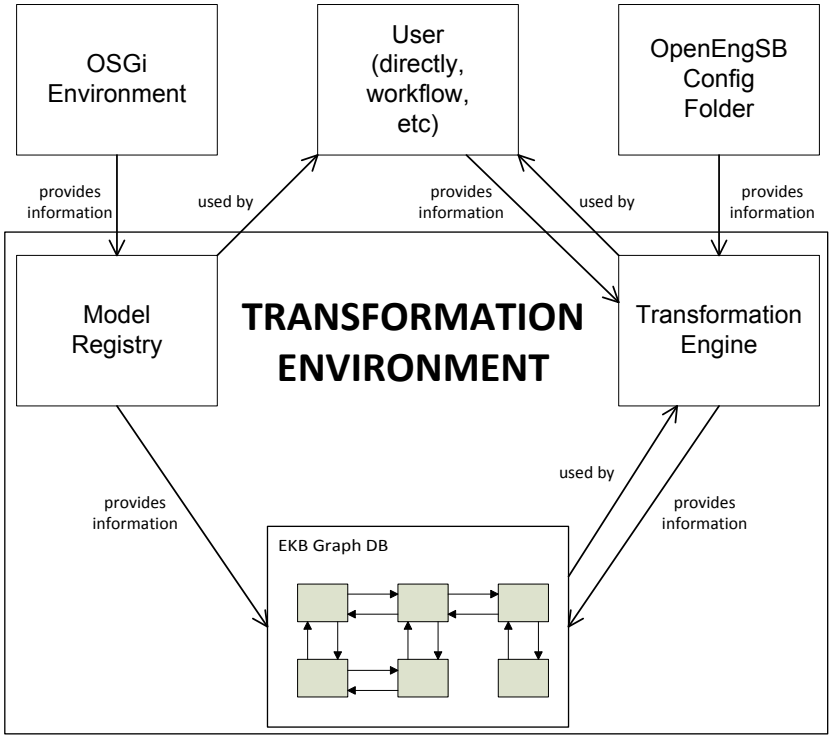

Figure 6.1: Transformation Environment Overview

The Model Registry retrieves the information about active models directly through the running OSGi environment. This component is implemented as a listener for bundle state changes. If a bundle status changes to the state stopped or started, the Model Registry checks for models contained in this bundle. If so, the Model Registry notifies the EKB Graph DB component about the status change of the models in the previously scanned bundle. In addition to this information based function, the Model Registry can also called directly by users to retrieve information about models or to load model classes.

The Transformation Engine component is the most noticeable component of the Transformation Environment. It contains all methods to start the actual transformation pro-

cess between models. To perform a transformation, this component uses the EKB Graph DB component to find a *transformation path*. A Transformation Path represents a list of *Transformation Descriptions* which are needed to perform a transformation from the source model to the target model of a *Transformation Request*. More information about Transformation Paths can be read in section 6.4.

The Transformation Engine uses Transformation Descriptions to fulfil its function. It retrieves this Transformation Descriptions either directly from users if they add a Transformation Description in its Java representation to the Transformation Engine, or through Transformation Description files placed in the configuration folder of the OpenEngSB. Whenever Transformation Descriptions are added or removed (again by one of the previously mentioned possibilities) the Transformation Engine informs the EKB Graph DB component about this changes. This information is important so that the EKB Graph DB component can find Transformation Paths. More information about the Transformation Descriptions and their representations will be explained in section 6.3.

6.2.1 EKB Graph Database

The EKB Graph DB is aware of all models and all Transformation Descriptions between models of the running project. This information is retrieved through the Model Registry component and Transformation Engine component. Based on this knowledge, the EKB Graph DB component is able to give the Transformation Engine component all necessary information to perform a transformation from one model into a different model, even if there is the need to perform multiple transformations in a row (a so called Transformation Path, see section 6.4 for more details) to obtain the target model.

As persistence back end, the EKB Graph DB component uses a graph based NoSQL database. The decision on such a special kind of database was taken because such databases are highly specialised for tasks like finding paths in graphs, which is needed for the searching of Transformation Paths. Since this procedure is the main job of the EKB Graph DB component, it is important that this task is as fast as possible.

The information about the models of the environment are given by the Model Registry component. The Model Registry component informs the EKB Graph DB component whenever a model is added or removed inside the OSGi environment. Based on this information the Model Registry performs one of the following actions:

Model added If a model is added, either a new node in the graph database is created for this model or if the model was deactivated before, the model gets activated again.

Model removed If a model is removed, the corresponding node gets deactivated. It is not deleted, because bundles can be started and removed as the user wants. And if they were removed, also all transformations which are having this node as start or end point would get removed, which makes problems when the nodes get reactivated at a later point of time.

The information about the active Transformation Descriptions is passed to this component by the `Transformation Engine` component. Based on this information, the `Model Registry` performs one of the following actions:

Transformation Description added If a Transformation Description is added to the environment, the `EKB Graph DB` component adds an edge from the node of the source model of the description to the node of the target model of the description. This edge contains all information so that the corresponding transformation can be executed.

Transformation Description removed If a Transformation Description is deleted, the corresponding edge get removed from the graph.

Every node and edge can be enhanced with generic custom properties, so that it is easily possible to add meta-information about nodes and edges directly to the according information unit, which means that there is no need for an additional component that maintains this meta-information. These meta-information is needed in the Transformation Path search procedure.

In the current implementation of the `EKB Graph DB` component, OrientDB[1], a powerful NoSQL graph database, is used as transformation path storage. OrientDB is able to handle different modes, even a document based mode. The modes differ from built-in functionalities and performance. The more features the mode provides, the less performance the database serves. The `EKB Graph DB` component uses the mode with the best performance, since the solution should be as fast as possible and additionally there is no need for the more sophisticated features which are provided at higher modes.

Even though the `EKB Graph DB` component is exported as a universally accessible service in the `OSGi` environment, there should be no direct interaction between the user and this component. Manually adding models or Transformation Descriptions can make problems and may introduce malfunctions in the processes which are using the Engineering Knowledge Base (*EKB*) component.

6.2.2 Model Registry

The main function of the `Model Registry` component is the search, adding and removal of models from the Transformation Environment. To accomplish this task, the `Model Registry` is implemented as a bundle tracker[2].

A bundle tracker is aware of all running bundles in the `OSGi` environment and can save a custom object for every bundle the tracker is aware of. In the case of the `Model Registry` component, this custom object is the list of models the corresponding bundle contains.

Whenever a new bundle is started in the `OSGi` environment with a running bundle tracker, the method `addingBundle` of the tracker is called. In the `Model Registry` component, this method searches the bundle for model objects. If there are any, they get registered in the `EKB Graph DB` component. On the other hand, if a bundle gets stopped, the method

[1] `http://www.orientdb.org`

[2] `http://www.osgi.org/javadoc/r4v42/org/osgi/util/tracker/BundleTracker.html`

removedBundle of the tracker is called, which tells the EKB Graph DB component that the models of the stopped bundle, if there were any, got inactive.

Since the search of model classes in a bundle is quite a expensive operation (all classes the bundle contains need to be checked, loaded and examined if a specific annotation is present), the OpenEngSB team introduced a bundle header with the name ProvideModels. If a bundle is added, the Model Registry component checks first if the bundle contains this introduced header. If the header is present, the bundle is analysed for models. If the header is not present, the bundle is skipped. This procedure reduces a lot of unneeded overhead for analysing classes which do not have any models.

Besides of the model location function, the Model Registry component also offers methods for getting meta-information about models, or to load a model in a specific version from the OSGi environment. The methods this service currently provides are:

- registerModel. This method is used to register a model class in the Transformation Environment.
- unregisterModel. This method is used to cancel a registration of a model class in the Transformation Environment.
- loadModel. This method loads a model from the OSGi environment. It throws an Exception if there is no model registered.
- getAnnotatedFields. This method checks a model class for fields annotated with the given annotation.

6.2.3 Transformation Engine

The Transformation Engine component is the starting point for user interaction with the Transformation Environment. It is responsible for the maintenance of all Transformation Descriptions and performs Transformation Requests by the user. The maintenance functionality of Transformation Descriptions contains the normal maintenance functionalities like saving, updating and deleting of Transformation Descriptions. A Transformation Description can be added to the Transformation Environment through two possibilities:

1. Through the Transformation Engine itself. To accomplish this, a Transformation Description is needed to be defined as a Java object, which is then passed to the Transformation Engine. The Transformation Engine will add the Transformation Description to the EKB Graph DB. How to create such an object will be explained in section 6.3.
2. Get the Transformation Descriptions from XML files in the OpenEngSB configuration folder automatically. Every Transformation Description can be expressed either as Java object or as a XML document. The Transformation Engine is aware of XML documents containing Transformation Descriptions located in the configuration folder of the OpenEngSB and scans this folder autonomously.

So whenever new XML documents containing Transformation Descriptions are added, existing documents get updated or deleted, the `Transformation Engine` can add, update or delete the corresponding Transformation Descriptions from the Transformation Environment without direct user interaction with the `Transformation Engine`. In that way it is possible to completely separate the development of Transformation Descriptions from written Java code.

Besides of the maintenance functionality, the `Transformation Engine` is able to perform Transformation Requests. To accomplish this, the `Transformation Engine` uses the `EKB Graph DB` to find a Transformation Path between the source and the target model. If there is such a path, the `Transformation Engine` performs the transformations on this path (with the help of the *`Transformation Performer`*) and returns the source model transformed into a model of the target type. If there is no such path, the method throws an exception.

The execution of the transformation is done by the `Transformation Performer` component. This component is a sub component of the `Transformation Engine`. This element is able to operate a Transformation Description on a source model instance and returns a target model instance. It takes care of the correct searching and invocation of operations and deals with temporary fields, which are needed to perform complex transformation sequences.

This component is the main communication point between the user and the Transformation Environment, it exports a service with which its functions are accessible for the user. The methods this service currently provides are:

- `saveDescription(s)`. This methods add or update Transformation Descriptions in the Transformation Environment.

- `deleteDescription`. This method is used to remove a Transformation Description from the Transformation Environment.

- `deleteDescriptionsByFile`. This method removes all Transformation Descriptions previously added by a XML document with the given name. This functionality is needed whenever a XML document containing Transformation Descriptions is removed from the config folder of the `OpenEngSB`.

- `getDescriptionsByFile`. This method returns all corresponding Transformation Descriptions Java objects that were added by a XML document containing Transformation Descriptions.

- `performTransformation`. There are a list of implementations for this method based on different parameters which are passed to the method. They are all supposed to perform transformations of a source model to a target model. They differ from each other by giving or not a list of transformation identifiers to the method, or giving an already instantiated target model which shall be used as base for a merge operation. More details about these functions are given in section 6.5.

- `isTransformationPossible`. These methods return a boolean value which tells the user if there is a possibility to transform a source model to a target model i.e., if there is a Transformation Path from the source model to the target model.

6.3 Transformation Definition Language

To enable the transformation from one model into another, a transformation description needs to be present. This description tells the transformation performing instance what exactly should be done in the transformation process. In general, a description language with which the transformation steps are described is called Transformation Definition Language (*TDL*). In this section Dynamo Transformation Language (*DTL*), the `TDL` of the transformation solution presented in this thesis is introduced and explained.

Additionally to the explanations presented in this section, appendix A show some example definitions of models and appendix B shows some example Transformation Description files.

6.3.1 TDL Design

The description of a transformation is separated into two parts:

- meta-data about the transformation containing:
  - The source model description, which defines the model type and the version of the source model.
  - The target model description, which defines the model type and the version of the target model.
  - An optional file name which defines from which file the description has been loaded by the autonomous loading mechanism. This property is empty if the description was created as Java object.
  - An optional identifier, which can be used to define exactly which specific transformation should be performed in a specific situation. If no identifier is set, the `Transformation Engine` uses a generated value for it. More information about the usage and the possibilities enabled through the Transformation Description identifier will be given in section 6.4.
- a list of *Transformation Steps*. A Transformation Step is the most basic mechanism in the Transformation Environment and describes the mapping of one (or more) field(s) of the source model to one field of the target model based on a specific operation. A Transformation Step consists of four elements:
  - A list of source properties. This list is allowed to contain properties of the source model and temporary properties.
  - A target property, which have to be a property of the target model or a temporary property.

– A *Transformation Operation* name. This name defines which operation will be used in this Transformation Step.
– A map of Transformation Operation parameters. The parameters differ from operation to operation.

Since it is possible that properties are models themselves, the possibility of accessing the values of these nested model properties was added. E.g., a model has a property with the name 'nested'. This property is a model itself, which has two properties with the names 'value1' and 'value2'. Then it is possible to access them in a Transformation Step by writing the string 'nested.value1' or 'nested.value2' in the property name.

To accomplish a wider variety of use-cases, the possibility of using temporary properties was added to the TDL. They get acquired the first time they are used in the Transformation Description and get deleted after the transformation is finished. To use a temporary field and to distinguish them from the actual fields of the models, the prefix '#' is used. E.g. '#propertyA'.

The most important advantage of temporary fields is the workaround to support cascaded expressions. E.g. the result of operation 1 must be used as input data for operation 2.

These Transformation Steps are not only used to transform one model into another, but also give the Transformation Environment meta-information about the transformation, like which properties of the source model affect which properties of the target model.

Java Object Structure

In the previous section the general structure of Transformation Descriptions was illustrated. This section proposes the Java object structure of a Transformation Description. Figure 6.2 shows a class diagram overview over the classes with their attributes omitting the methods of these classes.

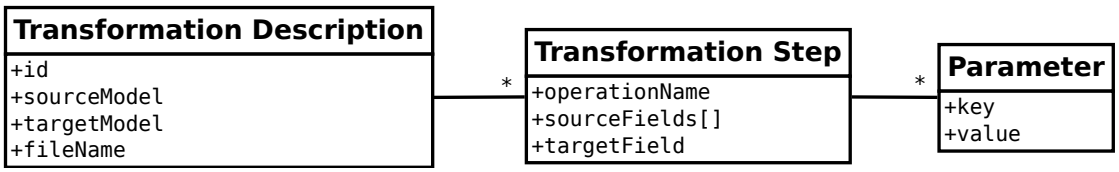

Figure 6.2: Java Object Structure

The main class of a Transformation Description is the **TransformationDescription** class. The properties of this class are the same as the elements which were mentioned in the general structure description: a source model description, a target model description, an optional identifier and a list of Transformation Steps. In addition to the mentioned fields also a field is present which saves a file name if the description was loaded from a file (see next section for more details).

This class also provides a collection of methods which add Transformation Steps to the Transformation Description. One of these methods is a very general method where the operation

name, the parameters etcetera can be manually specified. The other methods are used for an easy insertion of steps using standard (built-in) operations to the Transformation Description.

The Transformation Step Java object is called **TransformationStep**. This object consists of four elements:

- A set of source field names where every source field is either a property of the source model or a temporary property.
- A target field name which is either a property of the target model or a temporary property.
- An identifier string which defines the operation this transformation step uses.
- A map of operation parameters. The keys of the standard transformation operations parameters are defined in the **TransformationConstants** class where all parameter key names are defined as constants.

The operation identifier string is used during the transformation process to load the corresponding Transformation Operation from the `OSGi` environment. So it is possible to define in this property values which correspond to no transformation operation. In such a case, the error is noticed during the transformation process, an exception is thrown and the user gets informed.

XML Structure

To enable the projects to separate their transformation descriptions and also to make it easier for non developers to define a Transformation Description, it is possible to write Transformation Descriptions in a XML formatted file. Thus, Transformation Descriptions can be written by persons with no Java programming knowledge which only need to know about XML and the XML structure of the introduced `TDL`.

Each Transformation Description file need to have the file name suffix '.transformation'. The root element of each of these Transformation Description files is shown in listing 6.1. The examples given in the following are based on a fictional Transformation Description. However, complete example Transformation Description files are given in the Appendix B.

```
<transformations>
  // put your transformations here
</transformations>
```

Listing 6.1: Transformation XML file root element

In this root element there can be added a list of Transformation Descriptions. Listing 6.2 shows how a Transformation Description root element is defined.

```
<transformation
  source="org.openengsb.presentation.ModelA;1.0.0"
  target="org.openengsb.presentation.ModelB;1.0.0"
  id="transformModelAToModelB_1">
    // put your transformation operations here
</transformation>
```

Listing 6.2: Transformation root element

The source and the target attribute define the source and the target model. These attributes are separated into two parts, split by a semicolon. The first part of the model description defines the class name of the model and the second part is the version of the model. The version is the same like the OSGi version of the bundle providing the model.

The id attribute is optional and gives the Transformation Description an unique name which can be used for the Transformation Path calculations (see section 6.4). If there is no id attribute defined, the Transformation Engine will set this value to a generated value. However, it is recommended to set the id, since there can easily occur not desired results if there are a lot of unnamed Transformation Descriptions in the Transformation Environment, especially if there are several Transformation Descriptions between two models.

Inside this root element a list of Transformation Steps can be added. A sample transformation step is shown in listing 6.3.

```
<concat>
  <source-fields>
    <source-field>date</source-field>
    <source-field>#time</source-field>
  </source-fields>
  <target-field>timestamp</target-field>
  <params>
    <param key="concatString" value=" "/>
  </params>
</concat>
```

Listing 6.3: Transformation operation example

The operations root element defines the operation name. Note that this name need to be the same string with which the corresponding operation is provided in the OSGi environment. In the example of listing 6.3 the Transformation Operation is the 'concat' operation, which was introduced in the section about the standard Transformation Operations.

The source fields define the list of fields which are used as input for the operation. Note here, that the fields are either fields of the source model or temporary fields. If an operation need only one source field, then the parent element 'source-fields' is not necessary and can be skipped. In the example of listing 6.3 the first source field is the field with the name 'date' of the source model. The second source field is a temporary field with the name '#time'.

The target field defines where to write the result of the operation. This can either be a field of the target model, or a temporary field. In the example of listing 6.3 the target field is the field with the name 'timestamp' of the target model.

The 'param' elements with their parent element 'params' define the parameters for the Transformation Operation. Every operation has its own parameters defined where many of them are optional. The parameters of the standard Transformation Operations were already introduced. In the example of listing 6.3 the parameter which is defined tells the 'concat' operation that it should put a blank between the source field values.

6.3.2 Transformation Operation Design

To ensure a maximum flexibility of DTL and an easy adaptation to the needs of a project the transformation operations used in DTL descriptions are not hard coded. Every supported opera-

tion is exported to the OSGi environment with a specific interface called Transformation-Operation[3].

This interface defines all methods to retrieve meta data of and to perform a transformation operation. The methods are:

- The getOperationName method returns the name of the transformation as a string. This value is needed for the description of the operation, but do not need to be the name of the operation with which it is accessed through DTL (even though it is recommended to use the same name for both usages to avoid complications).

- The getOperationDescription method returns the description of the operation as a string. This string should contain what the operation does and how it should be used.

- The getOperationInputCount method returns an integer value representing the amount of input values this operation can take. If the operation can take 1 ore more input values, -1 should be returned and if the operation can take as many input values as it likes, -2 should be returned as value.

- The getOperationParameterDescriptions method returns a map which contains information about the parameters this operation can handle. The key is the name of the parameter and the value is the description of the parameter. The description should contain all needed information including if the parameter is optional or if there are existing standard values.

- Finally the performOperation method gets a list of input values, a map of operation parameters and returns an object. This object is the result of the operation. Additionally to the operation performing, this method should add checks for the validity of the input and the parameters. If an error occurs during the operation (e.g. incorrect input value size or missing parameters), a TransformationOperationException should be thrown.

With this interface it is very easy to add custom operations which then can perform more sophisticated things like loading values from a database, perform complex mappings or retrieving data from other services.

Figure 6.3 shows the basic interaction between the Transformation Performer, the Transformation Operations and the Transformation Operation providers. A set of bundles provide the Transformation Operations to the OSGi environment. The process to execute an Transformation Operation is as follows:

1. Search for the Transformation Operation service in the OSGi environment

2. Execute the found service with the parameters defined in the corresponding Transformation Step.

[3] https://github.com/openengsb/openengsb-ekb-api/blob/master/src/main/java/org/openengsb/core/ekb/api/transformation/TransformationOperation.java

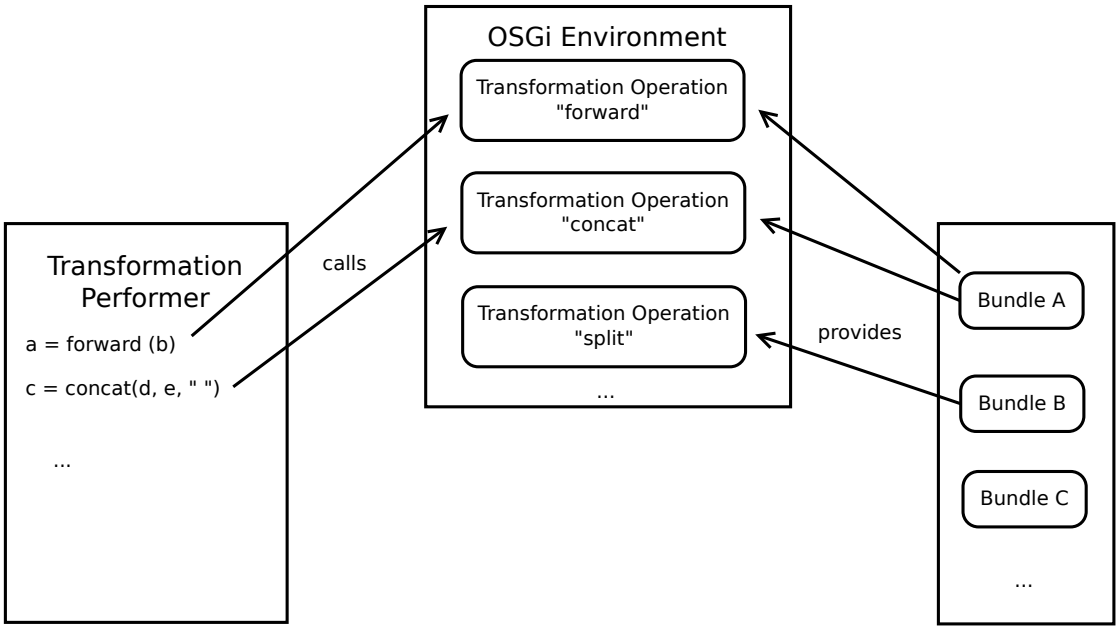

Figure 6.3: Transformation Operation Interaction

If there is no Transformation Operation with the given name, the Transformation Step is ignored.

To provide a custom function to the `DTL` performer, this operation needs to be exported to the `OSGi` environment as shown in listing 6.4. In this example the Apache Aries Blueprint[4] project is used to export the operation. Blueprint is a tool which allows the configuration of a bundles behaviour with the `OSGi` environment in a separate file. The shown example exports a Transformation Operation with the name 'forward'. An explanation of this operation and how it can be used is shown later in this section.

```
<service id="forwardOperation" interface="org.openengsb.core.
  ekb.api.transformation.TransformationOperation">
  <service-properties>
    <entry key="transformation.operation" value="forward" />
  </service-properties>
  <bean class="org.openengsb.core.ekb.transformation.wonderland.
    internal.operation.ForwardOperation" >
    <argument value="forward" />
  </bean>
</service>
```

Listing 6.4: Export transformation operation

[4] `https://aries.apache.org/modules/blueprint.html`

6.3.3 Transformation Operation List

In the following a complete list of all operations which are delivered with the OpenEngSB is shown. Although this operation set is already enough to fulfil a lot of use-cases, it is always possible to extend it with custom operations. An up to date list of the pre-defined operations can be seen in the OpenEngSB manual[5]. Note in the explanations that the source fields and the target fields can be either fields of the models or temporary fields.

For each of the operations explained in the following, an example is added which instantiates a Transformation Step with the corresponding operation. For the explanation here, the XML structure is chosen, since this format can be written in a more compact form. In the first operation explanation, additionally an example how to instantiate this operation in Java is added to compare the both formats.

Additionally, all properties defined in for the source fields are properties of the source model and the property defined tor the target field is a property of the target model.

Forward Operation

The **forward** operation is the simplest operation. It just forwards a field value from a source field to a target field.

```
<forward>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
</forward>
```

Listing 6.5: Example forward operation

Listing 6.5 shows an example how to use the **forward** operation.

```
TransformationDescription desc =
  new TransformationDescription(sourceModel, targetModel);
desc.forwardField("fieldA", "fieldB");
```

Listing 6.6: Example forward operation in Java

Listing 6.6 shows the equivalent Transformation Step instantiation to the example given in listing 6.5. It excepts that the variables 'sourceModel' and 'targetModel' are model descriptions.

Substring Operation

The **substring** operation takes a part of the string from the source field and set it into the target field. The operation has the following parameters:

from This parameter defines the character index of the string at which the substring should start. This parameter is optional and its default value is 0, which stands for the start of the source string.

[5] http://openengsb.org/index/documentation.html

to This parameter defines until which position of the source string the new substring should be taken. This parameter is optional and its default value is the length of the input string, which stands for the last character of the source string.

```
<substring>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
  <params>
     <param key="from" value="0" />
     <param key="to" value="4" />
  </params>
</substring>
```

Listing 6.7: Example substring operation

Listing 6.7 shows an example how to use the **substring** operation. Based on this example and with a string value 'Hello World!' set in the field 'fieldA', the value 'Hello' would be written in the property 'fieldB'.

Concat Operation

The **concat** operation combines the string values of the source fields into one string and writes it in the target field. The order of the fields is the same as in the definition. The operation has the following parameters:

concatString This parameter defines which string should be placed between the separate input values to form the resulting string. This parameter is optional and its default value is the empty string.

```
<concat>
  <source-fields>
    <source-field>field1A</source-field>
    <source-field>field2A</source-field>
  </source-fields>
  <target-field>fieldB</target-field>
  <params>
    <param key="concatString" value="-" />
  </params>
</concat>
```

Listing 6.8: Example concat operation

Listing 6.8 shows an example how to use the **concat** operation. Based on this example setting and considering 'test1' as value of 'field1A' and 'test2 as value of 'field2A', the string 'test1-test2' is written to the field 'fieldB'.

Map Operation

The **map** operation takes the string value of the source field. This value is then used to search in the map given through the parameters for an entry where the value is equals to the map key.

If a matching key is found, the value to this key is returned. If a value is set in the source field for which there is no mapping defined in the parameters, a `TransformationOperationException` is thrown.

```
<map>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
  <params>
        <param key="animal" value="cat" />
        <param key="cheese" value="cheddar" />
  </params>
</map>
```

Listing 6.9: Example map operation

Listing 6.9 shows an example how to use the **map** operation. Based on this example and with a string value 'dog' set in the field 'fieldA', the value 'cat' would be written in the property 'fieldB'.

Split Operation

The **split** operation splits a string into parts and writes one of these parts (based on a given index) into the target field. The operation has the following parameters:

splitString This parameter defines which string should be used for the split operation. If this parameter is not set, the empty string will be taken instead which means that there is no splitting performed.

resultIndex The split operation splits a string into a set of strings. This parameter defines which of the created strings is written to the target field. This value has to be a number. If it is not a number, 0 will be taken instead. If the index does not fit to the result of the split operation (e.g. 5 if the split only creates 3 resulting strings) a `TransformationOperationException` is thrown.

```
<split>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
  <params>
    <param key="splitString" value="-" />
    <param key="resultIndex" value="0" />
  </params>
</split>
```

Listing 6.10: Example split operation

Listing 6.10 shows an example how to use the **split** operation. Based on this example and with a string value 'test1-test2-test3' set in the field 'fieldA', the value 'test1' would be written in the property 'fieldB'. If the **resultIndex** parameter would be set to 2, the value 'test3' would be written in the property 'fieldB'.

Trim Operation

The **trim** operation removes the blanks at the beginning and end of the source fields string value and writes the result into the target field.

```
<trim>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
</trim>
```

Listing 6.11: Example trim operation

Listing 6.11 shows an example how to use the **trim** operation. Based on this example and with a string value ' test ' in the 'fieldA' property, the string 'test' will be written to the 'fieldB' property.

Length Operation

The **length** operation calculates the length of the value saved in the source field and writes the result as string into the target field. The operation has the following parameters:

function This parameter defines which function of the object in the source field should be used to calculate the length of this object. This parameter is optional and its standard value is 'length'. If the set function name is not supported by the object in the source field, a `TransformationOperationException` is thrown.

The reason why this parameter is needed, is that there can be any object type used with this operation. For example, the source field has as type a Java object defined by the user and the method used to retrieve the length of this object is called 'foo'.

```
<length>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
  <params>
    <param key="function" value="size" />
  </params>
</length>
```

Listing 6.12: Example length operation

Listing 6.12 shows an example how to use the **length** operation. Based on this example and with a list set in the property 'fieldA' containing 4 elements, the value '4' would be written in the property 'fieldB'.

Replace Operation

The **replace** operation replaces a certain part of the string in the source field with another string and copies the result to the target field. The operation has the following parameters:

oldString This parameter defines which part of the source string must be replaced. If this parameter is not set, a `TransformationOperationException` is thrown.

newString The parameter defines which string must be used as a replacement for the old string. If this parameter is not set, a `TransformationOperationException` is thrown.

```
<replace>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
  <params>
    <param key="oldString" value="old" />
    <param key="newString" value="new" />
  </params>
</replace>
```

Listing 6.13: Example replace operation

Listing 6.13 shows an example how to use the **replace** operation. Based on this example and with a string value 'old string' set in the property 'fieldA', the value 'new string' would be written in the property 'fieldB'.

toLower Operation

The **toLower** operation converts a string from the source field into its lower case equivalent and writes it to the target field.

```
<toLower>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
</toLower>
```

Listing 6.14: Example to lower operation

Listing 6.14 shows an example how to use the **toLower** operation. Based on this example and with a string value 'TEST' set in the property 'fieldA', the value 'test' would be written in the property 'fieldB'.

toUpper Operation

The **toUpper** operation converts a string from the source field into its upper case equivalent and writes it to the target field.

```
<toUpper>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
</toUpper>
```

Listing 6.15: Example to upper operation

Listing 6.15 shows an example how to use the **toUpper** operation. Based on this example and with a string value 'test' set in the property 'fieldA', the value 'TEST' would be written in the property 'fieldB'.

Value Operation

The **value** operation is used to set a constant value into a target field. The operation has the following parameters:

value This parameter defines the string which must be set in the target field. If this parameter is not set, a `TransformationOperationException` is thrown.

```
<value>
  <target-field>fieldB</target-field>
  <params>
    <param key="value" value="Hello World" />
  </params>
</value>
```

Listing 6.16: Example value operation

Listing 6.16 shows an example how to use the **value** operation. Based on this example the value 'Hello World' would be written in the property 'fieldB'.

Pad Operation

The **pad** operation takes the string of the source field and fills it with characters until the string has reached a specified length. The result is written into the target field. The operation has the following parameters:

length This parameter defines length the resulting string should have after the filling with the characters. If the string is longer than the given value here, the operation does nothing. If this parameter is not set, the default value 0 will be taken instead, which result in no filling at all.

char This parameter defines which character should be used for filling the string. If the given value is longer than one character, the first character will be taken. If this parameter is not set, a `TransformationOperationException` is thrown.

direction This parameter defines from where the character must be placed/filled. There are two modes for this parameter, namely 'Start' and 'End'. 'Start' means that the padding is performed prefix style, 'End' means postfix style. This parameter is optional. If this parameter is not set or the set value is neither 'Start' nor 'End' the standard value 'Start' is taken instead.

```
<pad>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
  <params>
    <param key="length" value="4" />
    <param key="char" value="0" />
    <param key="direction" value="Start" />
  </params>
```

```
</pad>
```

Listing 6.17: Example pad operation

Listing 6.17 shows an example how to use the **pad** operation. Based on this example and with a string value '1' set in the property 'fieldA', the value '0001' would be written in the property 'fieldB'.

Instantiate Operation

The **instantiate** operation is used to support other field types than strings. With the **instantiate** operation it is possible to create an object instance, where the target object type and the method for instantiating the target object type can be specified.

This method can take 0 to infinite source fields, where the values of these source fields are used for the instantiation process. If no source field is set, the given method/the constructor with no parameter is called. The operation has the following parameters:

targetType This parameter defines either the fully qualified Java class name or a model description in the form of 'modelName:modelVersion' of the class that should be initiated. If this parameter is not set or the class with the given name can not be found a `TransformationOperationException` is thrown.

targetTypeInit This parameter defines which function should be used to initiate the object of the desired type. This parameter is optional and if it is not set, the constructor which has objects as parameter which fit to the values in the source fields is used. If this parameter is set, the function with the given name is used to create the object. Again, the values of the source fields are taken as parameter for this method. The function can either be an instance method or a static method.

This parameter is needed, because the **targetType** can be any type. It is not known before which method should be used to initiate the object. E.g. simple constructor versus factory methods:

- `Integer.parseInt('938');`
- `Uri.fromString('sampleUri');`
- `MyObject.createInstance();`

```
<instantiate>
  <source-fields>
    <source-field>fieldA</source-field>
  </source-fields>
  <target-field>fieldB</target-field>
  <params>
    <param key="targetType" value="java.lang.Integer" />
    <param key="targetTypeInit" value="parseInt" />
  </params>
</instantiate>
```

Listing 6.18: Example instantiation operation

Listing 6.18 shows an example how to use the **instantiation** operation. Based on this example and with a string value '1' set in the property 'fieldA', an integer instance with the value 1 would be written to the the property 'fieldB'.

removeLeading Operation

The **removeLeading** operation takes the string from the source field and removes all elements which match a regular expression beginning from the string start until a specified maximum length or until the regular expression does not match any more. The operation has the following parameters:

regexString This parameter defines the regular expression string which is used to identify the elements which should be removed from the start of the string in the source field. If this parameter is not set a `TransformationOperationException` is thrown.

length This parameter defines how many leading elements which fit the given regular expression string should be removed at maximum. This parameter is optional and its standard value is 0. 0 means in this operation that there is no maximum. All elements at the beginning of the input string which fit to the regular expression are removed until there occurs a character which does not fit to the regular expression.

```
<removeleading>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
  <params>
    <param key="regexString" value="[?#]+" />
    <param key="length" value="0" />
  </params>
</removeleading>
```

Listing 6.19: Example remove leading operation

Listing 6.19 shows an example how to use the **removeLeading** operation. Based on this example and with a string value '??##id' set in the property 'fieldA', the value 'id' would be written to the the property 'fieldB'.

Reverse Operation

The **reverse** operation takes the string from the source field, reverses it and writes the result into the target field.

```
<reverse>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
</reverse>
```

Listing 6.20: Example reverse operation

Listing 6.20 shows an example how to use the **reverse** operation. Based on this example and with a string value 'test' set in the property 'fieldA', the value 'tset' would be written to the the property 'fieldB'.

Split Regex Operation

The **split regex** operation works exactly like the split operation, but takes a regular expression as base for the splitting and not a simple string. The operation has the following parameters:

regexString This parameter defines the regular expression string which is used to split the source string. If this parameter is not set, the empty string will be taken which means that there is no splitting at all.

resultIndex This parameter defines which of the results of the split regex operation should be written to the target field. This value has to be a number. If it is not a number, 0 will be taken instead. If the index does not fit to the result of the split operation (e.g. 5 if the split only creates 3 resulting strings) a `TransformationOperationException` is thrown.

```
<splitRegex>
  <source-field>fieldA</source-field>
  <target-field>fieldB</target-field>
  <params>
    <param key="regexString" value="[^#]+" />
    <param key="resultIndex" value="0" />
  </params>
</splitRegex>
```

Listing 6.21: Example split regex operation

Listing 6.21 shows an example how to use the **split regex** operation. Based on this example and with a string value 'value1#value2#value3' set in the property 'fieldA', the value 'value1' would be written to the the property 'fieldB'.

6.4 Transformation Paths

Like already mentioned in section 6.2, the `EKB Graph DB` component saves model definitions as nodes and Transformation Descriptions as directed edges in a graph based NoSQL database. In this way, a graph is built which models the transformation possibilities of the Transformation Environment.

If the `Transformation Engine` gets a Transformation Request, it first needs to check if there is a transformation possible. To accomplish this, it uses the `EKB Graph DB` component to check if a transformation is possible and how. In the most trivial case, the source model and the target model have a direct Transformation Description defined between them. Then only one Transformation Description needs to be performed to reach the target model.

But if the transformation between the source and the target model needs to pass an amount of models to perform the transformation, a path in this graph need to be found. A found path represents a list of Transformation Descriptions which need to be performed one after another to fulfil a Transformation Request. This list of transformations is called *Transformation Path*.

Since the graph is an ordinary graph with nodes and directed edges, the searching of a path in this graph can be seen as an instance of the reachability problem. Every problem, where a path

from one node in a graph to another node in the same graph needs to be found is a reachability problem. Thus, in the context of this thesis the reachability problem is the challenge of finding a Transformation Path between the source and the target model.

There are quite a lot solution approaches for this problem, like for example the famous Dijkstra's algorithm, defined by Dijkstra in [Dijkstra, 1959]. The following requirements are needed for the searched algorithm:

- The graph is a directed graph. That means every edge can only be used in the direction it is defined.
- There can be infinite transformations defined between two models. In that way we can give the user the possibility to transform the models on more than one way if needed (e.g. use a different transformation for models with a specific property set).
- Models can be set inactive. Inactive models (= nodes) in the graph cannot be used for Transformation Paths.
- Transformations can have an identifier. If the user initiates a transformation, he can pass a list of identifiers with the invocation. The search algorithm then needs to find a Transformation Path, where for every identifier given in the passed list there is a Transformation Description present in the Transformation Path with the identifier.

Based on these conditions, a modified version of a depth-first-search defined by Tarjan [Tarjan, 1972] is used to search a fitting Transformation Path. This search algorithm has the following attributes:

- It ignores inactive nodes.
- It prefers edges which are in the given identifier list. If any of this identifiers cannot be fulfilled, the search is interrupted and the user get informed, that there is no Transformation Path with the given parameters.
- Even if there is more than one transformation between two models, the same path isn't checked twice.
- A loop-checker is included to prevent endless Transformation Paths.

In the following, two examples are shown which illustrate the functionality of the path search algorithm. Both examples are based on the same base transformation graph. In both, the source model is 'A' and the target model is 'B'.

Figure 6.4 shows the base graph extended with the information of inactive models coloured red. Every inactive model, according to the algorithm, can not be used for a Transformation Path. As a result of this, there is a limited amount of valid Transformation Paths, namely:

- $A \rightarrow C \rightarrow D \rightarrow G$.
- $A \rightarrow C \rightarrow D \rightarrow E \rightarrow G$.

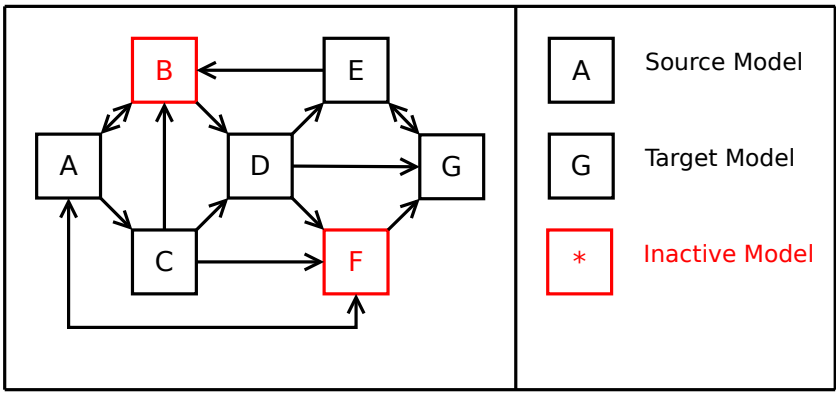

Figure 6.4: Transformation graph with inactive models

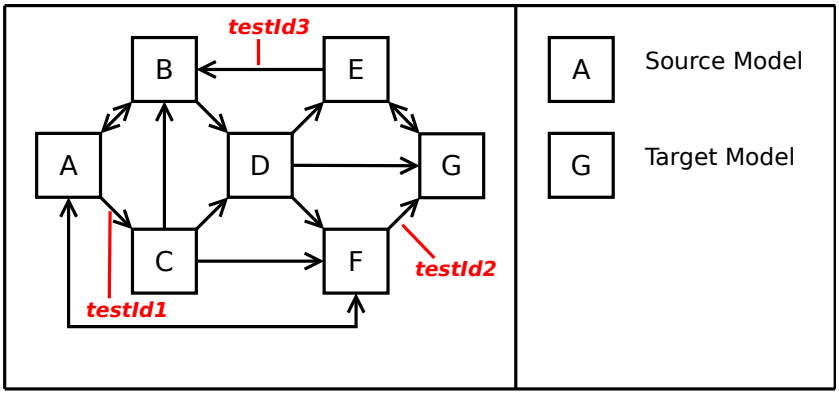

Figure 6.5: Transformation graph with transformation identifiers

Figure 6.5 shows the base graph extended with the information of identifiers for three transformation definitions. If transformation identifiers are passed to the search algorithm, the valid Transformation Paths are limited to paths which contain transformation definitions with the fitting identifiers for every passed identifier. Note here, that there can be infinite transformation descriptions between two models, which can all have an identifier set. With the help of these identifiers it is possible to explicitly require the Transformation Environment to use a specific Transformation Description when transform between two models.

In the following there are four scenarios with their valid Transformation Paths described:

Scenario 1 shows the influence if there is no identifier passed to the search algorithm. This results to a list of valid Transformation Paths which contain every non cyclic path in the directed graph from node 'A' to 'G'.

Scenario 2 shows the influence if there is the id 'testId1' passed to the search algorithm. This results to a reduced list of valid Transformation Paths, namely:

- $A \rightarrow C \rightarrow F \rightarrow G$.

- $A \rightarrow C \rightarrow D \rightarrow G$.
- $A \rightarrow C \rightarrow D \rightarrow F \rightarrow G$.
- $A \rightarrow C \rightarrow D \rightarrow E \rightarrow G$.
- $A \rightarrow C \rightarrow B \rightarrow D \rightarrow G$.
- $A \rightarrow C \rightarrow B \rightarrow D \rightarrow F \rightarrow G$.

Scenario 3 shows the influence if there are the ids 'testId1' and 'testId2' passed to the search algorithm. This results to a reduced list of valid Transformation Paths, namely:

- $A \rightarrow C \rightarrow F \rightarrow G$.
- $A \rightarrow C \rightarrow D \rightarrow F \rightarrow G$.
- $A \rightarrow C \rightarrow B \rightarrow D \rightarrow F \rightarrow G$.

Scenario 4 shows the influence if there is the id 'testId3' or a list of ids where 'testId3' passed to the search algorithm. This results in no valid Transformation Path, since there is no possible path from 'A' to 'G' without a cycle which contains the transformation description with the id 'testId3'.

The selection which neighbour should be taken for the next step is an uninformed approach. The only informed decision is that it handles edges earlier which have the fitting identifier. This search algorithm can be further improved by replacing the old uninformed approach with an informed approach where, for example, the neighbour with the highest edge-out count or the edge with the fewest Transformation Operations is chosen.

6.5 Transformation Execution Process

This section purpose is to give a more detailed view on the algorithm used to actually perform transformations. This algorithm is performed for every transformation request and is illustrated in figure 6.6:

1. A transformation request is given to the `Transformation Engine`. A transformation request contains all information needed to perform a transformation, like the start and the desired end model.

2. The `Transformation Engine` ask the `EKB Graph DB` for a Transformation Path which fits the needs of the given transformation request.

    a) If there is no fitting Transformation Path, an exception is thrown which tells the requester that there is no transformation possible. This can have multiple reasons, e.g. the start or target model are inactive, a wrong Transformation Description identifier was given in the parameters or there is simply no Transformation Path.

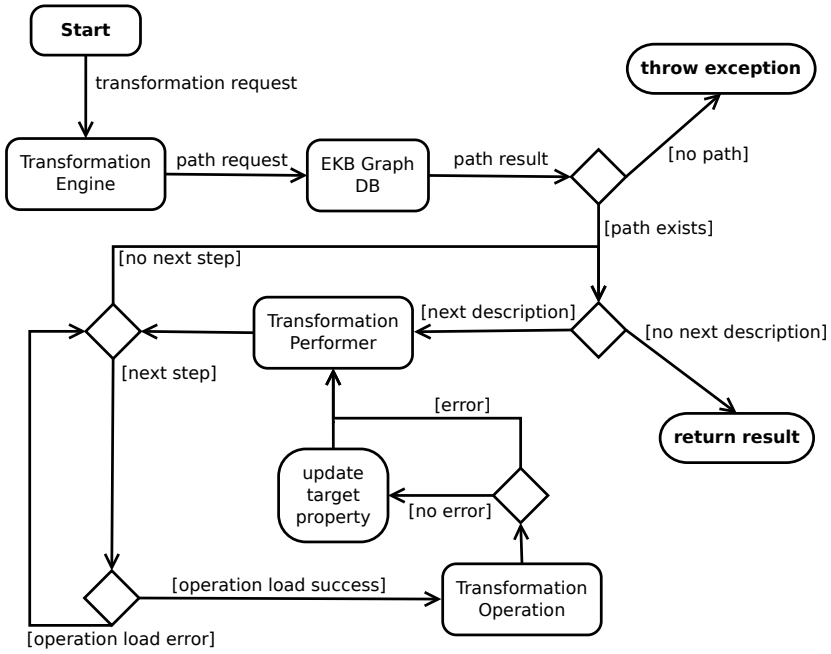

Figure 6.6: Overview figure of the transformation request processing

b) If there is a fitting Transformation Path, a transformation is possible and the path is returned to the `Transformation Engine`.

3. Based on the retrieved Transformation Path, the transformation is performed. The path is a list of Transformation Descriptions, where one after another need to be applied. The first on the source model instance, the second on the result of the first application etcetera.

   a) If there is no Transformation Description left, the transformation has been finished and the result is returned.

   b) If there is a Transformation Description left, it is sent to the `Transformation Performer` (point 4).

   c) If an error occurs in this transformation chain (e.g. a model is no longer available), an exception is returned to the calling instance.

4. The `Transformation Performer` gets a Transformation Description and a source model instance on which the Transformation Description should be applied. A Transformation Description contains a list of Transformation Steps.

   a) If there is no more Transformation Step, the resulting model instance is returned to point 3.

   b) If there are more Transformation Steps, the `Transformation Performer` takes the first and performs it. To perform the Transformation Step, the corresponding Transformation Operation needs to be loaded by the `Transformation`

`Performer`. If the requested Transformation Operation does not exist, the Transformation Step is skipped. If the Transformation Operation exists, it is called with the parameters of the Transformation Step. The result of the operation is written to the property specified in the Transformation Step (either a property of the target model or a temporary property).

5. The Transformation Operation tries to perform its action on the source model instance.

   a) If there was no error, the result is returned to point 4.

   b) If an error occurred in the Transformation Operation, a warning is printed to inform the user about this problem and the corresponding Transformation Step is skipped.

6.6 Model Handling

In the presented transformation solution, models are Java class structures, where each class is are annotated with a specific Java annotation (in fact, to enable all features two annotations). However, there are some run-time mechanisms needed so that these objects can be treated as models, e.g. the enrichment with all model related functions.

Model Format

Before the details of the models background knowledge are presented shows this section how a model looks like. It was already said, that models are realised as Java class structures. In such a structure there is one 'master' model class. This model class has properties which describe the connections to the remaining classes of the model. Since only a 'master' model class has the access to all other objects (through the properties), this class is used for the model transformation executions.

```
public class Notification {
  private Recipient recipient;
  private Text text;

  // put your fields and getter/setter pairs here
}

public class Recipient {
  private String name;
  private String address;

  // put your fields and getter/setter pairs here
}

public class Text {
  private String locale;
  private String message;

  // put your fields and getter/setter pairs here
```

```
}
```

Listing 6.22: An example multi-model class structure

Listing 6.22 shows an example for a model class structure where `Master` represents the 'master' model class. This class can act as a parent for the model since it can access every other model class structure member. The classes `Sub1` and `Sub2` could also have other objects as properties and so on and so forth.

Model Interface

The `OpenEngSB` project defines the interface for a model with the name `OpenEngSBModel`[6]. This interface contains all methods which are expected of model instances by components which are working with them. With this interface it is possible to handle all models in an abstract way without knowledge about the real implementation of the model. The methods this interface defines are the following:

- `toOpenEngSBModelEntries`. Returns a collection of generated and previously added `OpenEngSBModelEntry`[7] objects. These objects contain a triple of information, containing field name, field type and field value for every field of the corresponding model instance. This function is mainly used when models get persisted in the Engineering Database (*EDB*).
- `addOpenEngSBModelEntry`. Adds a `OpenEngSBModelEntry` object. With this function it is possible to add custom information to a model with a specific key. This may be interesting if there are a family of tools which want to share data with each other, not directly related with the model. The list of added entries is called *model tail*.
- `removeOpenEngSBModelEntry`. Removes a `OpenEngSBModelEntry` previously added to the model instance.
- `getOpenEngSBModelTail`. Returns the list of all previously added `OpenEngSB-ModelEntry` objects.
- `setOpenEngSBModelTail`. Sets the model tail explicitly.
- `retrieveInternalModelId`. Returns the internal model identifier of the corresponding model instance. This identifier is defined through a field with a special annotation.
- `retrieveInternalModelTimestamp`. Returns the time stamp when the corresponding model has last been persisted in the `EDB`. If the model has never been persisted, this method returns null.

[6] `https://github.com/openengsb/openengsb-api/blob/master/src/main/java/org/openengsb/core/api/model/OpenEngSBModel.java`

[7] `https://github.com/openengsb/openengsb-api/blob/master/src/main/java/org/openengsb/core/api/model/OpenEngSBModelEntry.java`

- `retrieveInternalModelVersion`. Returns the version of the model defined in the `EDB` (e.g. if a model was saved three times, the last version of this model has the value 3).
- `retrieveModelName`. Returns the fully qualified name of the object implementing this interface.
- `retrieveModelVersion`. Returns the version of the model as a string representation of the bundle version from the bundle containing the model.

Every model in the Transformation Environment need to implement this interface so that the semantics components can work with them correctly.

Run-time Model Weaving

Since implementing the methods for the `OpenEngSBModel` is a simple but time consuming, repetitive and annoying task for developers, the `OpenEngSB` team decided to add these functions and their implementation via byte code manipulation to the model objects. This byte code manipulation process in which a model is enhanced with the model functions is called *model weaving*.

There are two supported ways in the `OpenEngSB` to perform this weaving. While both possibilities work for every solution which uses `OSGi`, only the first way works for projects without `OSGi`:

- Use a Java Agent[8] which does the model weaving. The corresponding Java Agent jar file is created during the installation of the `OpenEngSB` weaving service. A Java agent can be defined as parameter at the start or at the compilation of a Java application and is able to manipulate the byte code of Java classes when they are first loaded by the Java Virtual Machine. Using this construct, the model functionality can be weaved into the model classes.
- With the help of a WeavingHook[9] which is started in an `OSGi` environment. Whenever a class is loaded by the `OSGi` environment, this class is passed to all active WeavingHook instances in the `OSGi` environment. Similar to a Java Agent, a WeavingHook allows the byte code manipulation of already compiled classes. This is the solution preferred by the `OpenEngSB` team.

Both implementations use the same byte code manipulation process in background, which is using JBoss Javassist[10] as byte code manipulation library. Javassist is a powerful tool which makes it possible to generate a class object from a byte array and allows modification of this class with normal Java syntax, which then will be translated back to machine readable instructions.

[8] `http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html`

[9] `http://www.osgi.org/javadoc/r4v43/core/org/osgi/framework/hooks/weaving/WeavingHook.html`

[10] `https://www.jboss.org/javassist`

To register an object for the model weaving process, a special annotation need to be set for the class of the object, which will be introduced in the following.

Model Annotations

There exist a set of annotations that are important to know when working with models of the presented transformation solution. In the following, every annotation in the model context is listed and explained.

@Model

This annotation is a class annotation which mark a class as a model object. Listing 6.23 shows an example how this annotation is set.

```
@Model
public class ModelObject {
  // put your fields and getter/setter pairs here
}
```

Listing 6.23: Model annotation example

If a class is annotated with this annotation, it is activated for the model weaving process, it gets recognized by the `Model Registry` component, can be persisted in the `EDB` through the `EKB` and also transformations are possible. However, to make a model fully accessible by all Transformation Environment components, it also need to be loadable through the loading mechanism which is explained later in this section.

@OpenEngSBModelId

This annotation is used to indicate the field that acts as the internal identifier of the model object. Listing 6.24 shows how to define the identifier of a model with the annotation.

```
@Model
public class ModelObject {
  @OpenEngSBModelId
  private String id;
  // put your fields and getter/setter pairs here
}
```

Listing 6.24: OpenEngSBModelId annotation example

The identifier of the model is important whenever a model should be loaded from the `EDB` by its identifier or if the `Engineering Object` concept should be used, since the references defined there are based on references to other model instances.

It is important to note here that if no identifier is defined for a model, a random Universally Unique Identifier (*UUID*) is taken instead.

@IgnoredModelField

When the model interface was explained, there was a method described which returns all fields of a model as a list of key/value/type containing objects. However, it is not always required or

wanted that all fields of a model are converted/returned in that way. E.g. if the fields are only helper values, or statically set values which should not be persisted in the EDB.

To enable a way to avoid the converting/returning of specific fields, this annotation can be used to mark fields to be skipped in the converting procedure. Listing 6.25 shows how a field can be marked for being ignored.

```
@Model
public class ModelObject {
  @IgnoredModelField
  private String temporary;
  // put your fields and getter/setter pairs here
}
```

Listing 6.25: IgnoredModelField annotation example

@OpenEngSBForeignKey

One of the challenges of this thesis is the enabling of automated model update propagation. To accomplish this, the Engineering Object concept was introduced, which is explained in detail in section 6.7. With this construct it is possible to define links between models, which then are used for the update propagation.

The links are defined through this annotation, like shown in listing 6.26. This annotation defines which model type the link is referring to (combination of the model class and the model version) and needs to annotate a String property. This property needs to contain the identifier of a model identifier of the given type.

```
@Model
public class ModelObject {
  @OpenEngSBForeignKey(
      modelType=org.openengsb.presentation.OtherModelObject.class,
      version="1.0.0")
  private String keyToOtherModel;
  // put your fields and getter/setter pairs here
}
```

Listing 6.26: OpenEngSBForeignKey annotation example

Model Loading Mechanism

Like already mentioned in this thesis, the introduced transformation solution is able to deal with models in different versions. However, to enable such a possibility there need to be a mechanism that allows the loading of models in every available version. This causes problems, since the loading of the same model class in different versions through a Java class loader is not possible.

Another problem is the providing of the model classes for the Transformation Environment components. In OSGi it is often unknown what classes a bundle will need to load during run time. It can be defined that a bundle is dynamically loading packages from other bundles at run time, but this causes other problems like that these bundles cannot be refreshed properly.

The solution found for this problem, is the using of a library which is able to load classes provided by bundles. Additionally, the classes are not added to the class loader of the caller, which means the class loader of the calling instance is not involved in the class loading process.

This library is the labs-delegation[11] project that allows the dynamic class loading in `OSGi` environments.

Classes that should be loadable by this library can either be defined via the bundle manifest header or be annotated with an annotation. In both cases there can be a list of contexts defined with which specific 'families' of classes are built. The service which loads classes through this project can be set to search for all exported classes, only classes of a specific context and/or classes of a bundle with a specific version.

If classes are annotated as models, but don't get provided with the labs-delegation project with the context 'models', they get the whole model functionality, but can not be used in the transformation solution, since the model classes are loaded via the labs-delegation project.

```
...
Provided-Classes-models: org.openengsb.presentation.ModelObject,
  org.openengsb.presentation.*
...
```

Listing 6.27: Provide manifest header example

Listing 6.27 shows how models can be provided via the bundle manifest header. The 'Provided-Classes' key defines which classes should be available through the labs-delegation project. Optionally, a context can be set as suffix. In this case, these objects are exported with the context 'models', which is the context used for models in the transformation solution. Like shown in this example, it is possible either to export classes directly or use wildcards to export a whole package with all its classes.

```
Manifest Header:
....
Delegation-Annotations: true
....

Java class:
@Provide(context = { "foo", "bar" }
public class ProvidedClass {
  ...
}
```

Listing 6.28: Provide annotation example

Listing 6.28 shows how models can be provided via a Java class annotation. The first thing to do is to put the value 'Delegation-Annotations: true' in the bundle manifest header. If this flag is set, the classes of this bundle get scanned for annotated classes. These classes will then be provided when the bundle is started. Like in the manifest header way, it is also here possible to define contexts. In the shown example, the `ProvidedClass` class is accessible in the context 'foo' and 'bar'.

[11] `https://github.com/openengsb/labs-delegation`

6.7 Model Update Propagation

As soon as a project is able to transform models, more sophisticated algorithms and processes can be built with the based on this feature. One example for such a process is the update propagation between models. Update propagation in the context of this thesis means, that a change of a model instance is propagated to all model instances which are influenced through the changed model. Then again these updated models need to update models which are connected to them and so on.

It is a typical situation in tool integration solutions, that the different tools use their own models even though there are other tools which work with similar or related models. However, it is quite common that these loosely connected models influence each other. That means whenever a model is changed, all related models need to be updated too or otherwise data inconsistencies are created.

The project and its structure introduced in section 4.3 gives an example for such a situation. Whenever a model instance is changed, this change needs to be propagated to the *Common Concept* and from there it needs to be propagated to the other models which are connected to the *Common Concept*. Figure 6.7 shows the sequence of updates which need to be performed.

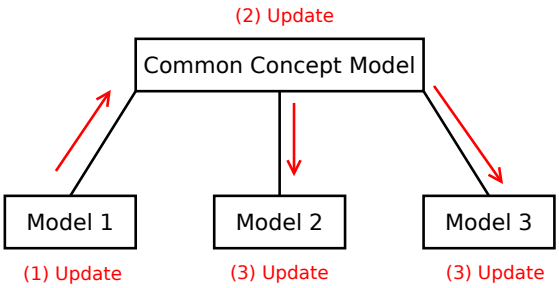

Figure 6.7: Model Update Propagation Sequence

In this thesis, the `Engineering Object` concept is used [Winkler et al., 2011]. With `Engineering Objects` it is possible to define constructs like the one described in this section. Internally such objects are just like any other model, but with the difference that it is possible to define references to other model instances. Most of their attributes are defined by the models they are referring to, but it is also possible to add attributes to `Engineering Objects` which are in no way related to the referenced models.

To define references to other models, attributes need to be annotated with a special annotation which define the referenced model type and model version. The identifier of the corresponding model need to be set as value of the annotated field. More details about this annotation is given in section 6.6. As soon as a model have at least one attribute which is marked with this annotation, it is treated as an `Engineering Object`.

The `Engineering Object` concept and the according automated model update propagation logic is placed as addition to the `EKB` model commit procedure. During the practical work for this thesis, the component responsible for the commit procedure received an additional

sub component which takes care of the automated model update propagation. This component analyses all incoming commits, check if updates need to be propagated and enhance the commit with the additionally required updates.

The additionally added updates can then again cause other updates and so on and so forth. To accomplish this, the enhancement is implemented as a recursive algorithm. The cases in which an enhancement need to be done are the following:

`Engineering Object` insert. Whenever an `Engineering Object` is inserted, the enhancer first loads all referenced models. If a reference string is not set or the referenced model instance is not existing, these model instances are not involved in the enhancement process. The reason why a referenced model instance may not exist, can be if the `Engineering Objects` are created at an earlier point in time than the referenced model instances. After all referenced model instances are loaded, they are transformed to the `Engineering Object` model and the results are merged accordingly.

`Engineering Object` update. Whenever an `Engineering Object` is updated, the enhancer needs to load all its referenced model instances and propagate the change to them. It need also to be checked if the `Engineering Object` is itself referenced by another `Engineering Object`. If this is the case, then this `Engineering Object` need also to be updated, which cause also updates of its referenced model instances etcetera. Like already said, the algorithm is implemented recursively so that an update is really propagated to all referenced model instance.

model insert. Whenever a model is inserted, the enhancer check for already existing instances of `Engineering Objects` which references this model. In that case, these instances were created before inserted model instance and so the enhancer needs to load the model instance now, perform the transformation to the corresponding `Engineering Object` and to merge the model with it.

model update. Whenever a model is updated, the enhancer check for already existing instances of `Engineering Objects` which references this model. In that case, these instances need to be updated too. This `Engineering Objects` reference other models and can be referenced by other `Engineering Objects`, which then need to be updated too etcetera.

The `Engineering Object` concept is a good example for a more sophisticated feature built on top of an existing model transformation solution. In this case, the model update propagation can even be performed without any extra work for the developer. The preconditions that this feature can be directly used without extra coding work are:

- Use the presented transformation solution.
- Use the `OpenEngSB`, the `EKB` and the `EDB`.
- There are Transformation Descriptions defined in both directions between the involved models and the `Engineering Object`.

6.8 Apply the solution to a project

The following section will show in which projects the introduced transformation solution can be used and which modifications need to be done in order to start performing model transformations in an existing project.

Major pre-condition is that the project is a Java project, since the transformation process is only performing with Java objects.It could be also possible to work around this issue by setting up an `OpenEngSB` and connect the project via a bridge to the `OpenEngSB`, but this may be more complicate and bloated than using an alternative transformation solution.

	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6
`OpenEngSB` application						X
OSGi-based					X	X
want to use EKB persistence + EDB			X	X	X	X
want to use EKB persistence	X		X	X	X	X
want to use EDB		X	X	X	X	X
only transformation solution wanted	X	X	X	X	X	X

Table 6.1: Steps needed for applying the transformation solution

Table 6.1 shows which modifications need to be done step by step to adapt a project so that the introduced transformation solution can be applied to it. Based on the pre-conditions on the most left column, every step that need to be done is marked. The first pre-condition that fits gives the needed steps.

That means for example, if the project already uses an `OpenEngSB`, then only step 6 have to be performed, but if the transformation solution should be used, but no of the listed preconditions are fulfilled, then all 6 steps have to be performed.

Step 1 - Implement EDB The `EDB` is the model store solution of the `OpenEngSB` and is explained in section 2.2.4. It is a versionized persistence solution, which means for instance, that the history of model changes can be seen, models from every point in time can be loaded and also the status of all models for a specific point in time can be queried.

To write an own `EDB` implementation, the `EngineeringDatabaseService`[12] interface needs to be implemented. This interface contains a lot of different functions for saving, deleting and loading of objects. How this interface is implemented and which technology is used as persistence back end is free of choice.

Step 2 - Implement EKB persistence The `EKB` persistence is the second big part of the `EKB` so far. The `EKB` is the semantic core component of the `OpenEngSB` and is explained in section 2.2.3. The `EKB` persistence consists mainly of two services and uses an `EDB` implementation as persistence back end.

[12] `https://github.com/openengsb/openengsb-edb-api/blob/master/src/main/java/org/openengsb/core/edb/api/EngineeringDatabaseService.java`

To write an own EKB persistence implementation, the interfaces PersistInterface[13] and QueryInterface[14] need to be implemented. The PersistInterface is responsible for the persisting of model changes. The QueryInterface is responsible of loading models. These services should use an EDB implementation as persistence back end.

Step 3 - Implement Model Registry The Model Registry is the component responsible for recognizing models in the environment and is illustrated in more detail in section 6.2. In the prototype solution, this component gets its information about the models (which models are present in the project and if they are active) from the OSGi environment. If there is no OSGi environment, a different attempt need to be done.

A possible solution would be that a list of models is generated during the compile phase of the project and this list is passed to a custom implementation of the Model Registry. For a custom implementation of the Model Registry, a service implementing the ModelRegistry[15] interface must be written.

Step 4 - Perform model weaving with an agent In the presented prototype model objects get enhanced with model related functions at run-time through a mechanism of the OSGi environment. If there is no OSGi environment, this mechanism is not present, so the enhancement need to be done at compile-time.

The OpenEngSB provide a Java agent[16] which is also able to do this enhancement, so this agent needs to be added to the compile procedure instead of relying on the run-time mechanism.

Step 5 - Add EKB persistence and transformation solution The components of the transformation solution and the EKB persistence components need to be added to the project. If the project is an OSGi based project, then only the corresponding bundles of the OpenEngSB project have to be added to the project and all work here is done.

If the project is not OSGi based, then the bundles needed for the transformation solution need to be adapted for the project. In most cases this means that all classes need to be integrated into the project (in contrary to OSGi where they are a project for themselves) or sub projects.

Step 6 - Annotate models and define transformations If the project is OSGi based, then there are the following three steps that need to be done:

[13] https://github.com/openengsb/openengsb-ekb-api/blob/master/src/main/java/org/openengsb/core/ekb/api/PersistInterface.java

[14] https://github.com/openengsb/openengsb-ekb-api/blob/master/src/main/java/org/openengsb/core/ekb/api/QueryInterface.java

[15] https://github.com/openengsb/openengsb-ekb-api/blob/master/src/main/java/org/openengsb/core/ekb/api/ModelRegistry.java

[16] http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html

- Add the entry `ProvideModels` to the manifest header of the bundles that contain models.
- Annotate models indicating that they are models and that they are provided with the labs delegation project[17], with the context 'models'.
- Define transformations between the models where needed (how to define transformations is explained in section 6.3).

If the project is not `OSGi` based, then there are the following two steps that need to be done:

- Annotate models with the annotation for models.
- Define transformations between the models where needed (how to define transformations can be read in section 6.3).

After step 6 is finished, the transformation solution application finished and transformations can be performed.

[17] `https://github.com/openengsb/labs-delegation`

CHAPTER 7

Validation

7.1 Overview

This chapter presents the validation of the prototype in different aspects. The structure of the chapter is as follows:

Section 7.2 shows the comparison of the proposed Transformation Definition Language (*TDL*) with the `TDLs` explained in the related work (see section 3.4). As base for this validation, a list of requirements were defined, which are described in detail in this chapter.

Section 7.3 illustrates details about the validation of the prototype against the defined use-cases which are explained in chapter 4.

Section 7.4 provides information about the time needed to define transformations and shows performance measurements of the prototype.

7.2 TDL comparison

In this section, the comparison of the `TDLs` described in the related work and the `TDL` presented in this thesis is shown. This comparison is based on a list of requirements, which were gathered in the planning phase of the Open Engineering Service Bus (*OpenEngSB*) model transformation solution.

In the following, the list of requirements is explained in detail. It will be illustrated how the requirements have been formed and what they mean for the transformation solution. After that, an overview table is shown, that shows which `TDLs` fulfil which requirements. Finally for each `TDL` of this comparison, an own section is given which summarize the findings for the corresponding `TDL`.

7.2.1 Requirements Definition

The requirements for the TDL used in the OpenEngSB transformation solution were already briefly mentioned in chapter 5. This listing only gave an overview of these requirements, but no further explanation. In the following, a more detailed view on the requirements is given:

R1 Source and target object can be Java objects. One of the key requirements of the TDL is the possibility to transform Java objects into other Java objects. This means that the TDL needs to be able to operate on Java objects.

R2 No meta models needed. The TDL needs be able to operate in environments with no meta models. This is the case, since in the OpenEngSB models are normal Java objects and there is no knowledge about abstract layers above these models.

- **R2.1** Source and target object need no meta model. The source and the target object of a transformation do not need any meta model. These objects can be used exactly like they are received by the transformation engine.
- **R2.2** Transformation Description need no meta model. In many TDLs transformation descriptions are themselves models which need a meta model. Since the TDL should work with Java objects, the descriptions should not need meta models.

R3 Custom transformation operations supported. A requested feature of the TDL is the possibility to dynamically modify the set of possible transformation operations. This also includes the adaptation of existing and the adding of new transformation operations.

- **R3.1** Operations can be adapted. It should be possible to modify transformation operations to the needs of the user. This means that access to the code of the operations is needed and modification of the code should be possible.
- **R3.2** Custom operations can be altered. It should be possible to add new operations which can be used with the TDL to fulfil more complex scenarios.
- **R3.3** Custom operations can be written in Java. Ideally, custom transformation operations can be written in Java, since Java is the major programming language of the OpenEngSB project.

R4 Separation of definition from execution. The definition of the transformation between models should be separated from the execution of the the transformation. This mainly means, that there should be files containing the definitions, which are read by the execution component when needed.

R5 Meta-information about transformations easily accessible. The TDL transformation description should be defined in a way that meta-information about the description can be obtained in an easy way (e.g. by a well defined format or a not too bloated language).

- **R5.1** Which source fields influence which target fields. One of the previous mentioned meta-information which are interesting for retrieval is a summary of which properties of the target model are influenced by which properties of the source model.

R5.2 Definitions can easily be parsed. To be able to easy obtain meta-information from a description, it needs to be parse-able in a reasonable way (e.g. description is located only in one file, using a not proprietary format or a stable format).

7.2.2 Overview

The TDL of this thesis has been compared with the TDLs explained in the related work, based on the requirement list which has been defined before. Table 7.1 gives an overview over the result of the comparison and in the following the results are explained in more detail.

Requirement	QVT	ATL	Kermeta	Smooks	DTL
R1	~	~	~	✓	✓
R2.1	X	X	X	✓	✓
R2.2	X	X	✓	✓	✓
R3.1	X	X	X	X	✓
R3.2	✓	X	✓	~	✓
R3.3	✓	X	✓	~	✓
R4	✓	✓	✓	✓	✓
R5.1	X	✓	~	✓	✓
R5.2	~	~	~	✓	✓

✓ requirement fulfilled.
X requirement not fulfilled.
~ requirement fulfilled under specific circumstances.

Table 7.1: TDL requirements satisfaction comparison

The rows of the table are the requirements, which were described before. The columns are the different TDLs. Each cell has one of three possible values:

✓ This value means that this requirement is fulfilled.

X This value means that the requirement is not fulfilled.

~ This value means the requirement is fulfilled under specific circumstances which are explained in the detail sections; e.g. a workaround or a plug-in.

7.2.3 QVT

QVT (see section 3.4) supports the requirements as follows:

R1 ~. There exists a project called *jQVT*[1] which is able to transform a QVT transformation definition into a Java method. This Java method can then be used with Java objects.

[1] http://sourceforge.net/projects/jqvt/

R2.1 X . Source and target of a transformation need to be conform to a previously defined meta model.

R2.2 X . The transformation description is itself a model which needs to be conformed to a meta model.

R3.1 X . QVT is mainly declarative, but allows imperative elements in two possible ways: Either by using a Black Box approach which allows the call of external functions or by using a set of defined operations. These operation set can not be changed.

R3.2 ✓. QVT allows imperative functions to be called from a different source (e.g. an own script). In that way, custom functions can be used.

R3.3 ✓. There is an interface which needs to be implemented so that QVT can call a custom function in Java.

R4 ✓. All elements in the QVT environment are separated in different files. A performing unit then loads this files to work with them.

R5.1 X . There is no built-in solution to get the information which source properties influence which target properties. Additionally, it is not easy to retrieve this information since this information is spread around different files.

R5.2 ~. QVT files have a well defined specific layout. However, there are no parsers for this layout like e.g. for XML.

7.2.4 ATL

ATL (see section 3.4) supports the requirements as follows:

R1 ~. It is possible to use Java objects as source and as target object for a transformation in ATL through a work around. The ATL library for Java defines a Java object which can be used for this purpose. However, this Java object is just a 'container' for the properties of the model, which means that a model object would need to be converted to this Java object and vice-versa.

R2.1 X . Meta models are needed for the models involved in the transformations.

R2.2 X . ATL transformations need to be conformed to a meta model.

R3.1 X . There is a fixed set of operations[2] which can be used.

R3.2 X . There is a fixed set of operations defined for ATL.

R3.3 X . There is a fixed set of operations defined for ATL.

[2] `http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language`

R4 ✓. Transformation definitions are placed in separate files which then are called by the performing unit.

R5.1 ✓. This meta-information is very easy to obtain from the ATL transformations, since the rules are already defined in that way (targetProperty = ...).

R5.2 ~. ATL transformation files are not difficult to analyse. The input and the output model can be received (if the access to model definition files is given) and it is also not difficult to get meta information from the files. However, there are no parsers for this layout like e.g. for XML.

7.2.5 Kermeta

Kermeta (see section 3.4) supports the requirements as follows:

R1 ~. It is possible to use Java objects as source and as target object for a transformation in Kermeta through a work around. The Kermeta library for Java defines a Java object which can be used for this purpose. However, this Java object is just a 'container' for the properties of the model, which means that a model object would need to be converted to this Java object and vice-versa.

R2.1 X . Source and target model need a meta model.

R2.2 ✓. Transformation descriptions are relation definitions between models.

R3.1 X . There is a fixed set of imperative operations defined for Kermeta transformation descriptions.

R3.2 ✓. It is possible to call external Java functions.

R3.3 ✓. It is possible to call external Java functions.

R4 ✓. The transformation descriptions are written in the model relations which are located in files.

R5.1 ~. In general, the operations can be parsed similar to ATL, but it need to be known which operations are transformations.

R5.2 ~. In general, the operations can be parsed similar to ATL, but it need to be known which operations are transformations.

7.2.6 Smooks

Smooks (see section 3.4) supports the requirements as follows:

R1 ✓. Smooks is able to operate on Java objects.

R2.1 ✓. No meta models needed.

R2.2 ✓. No meta models needed.

R3.1 X . Smooks has a set of defined functions which can not be changed.

R3.2 ~. It is possible to add custom functions but not in a really easy way. It is possible to define functions via ruby scripts or to add them as SAX parsing functions to the Smooks engine. However, it is not trivial to implement functions via SAX parsing.

R3.3 ~. It is possible to write a function inside a SAX parser in Java and add that to the Smooks engine, but it does not work directly.

R4 ✓. The transformations are defined in a Smooks configuration XML file.

R5.1 ✓. The structure of Smooks transformation definitions makes it easy to obtain this information, because every step in the definition has a source and a target field property.

R5.2 ✓. The Smooks transformation definitions are written in XML which is an easy to analyse data format, since the tool support is huge.

7.2.7 DTL

Dynamo Transformation Language (*DTL*) (see section 6.3) supports the requirements as follows:

R1 ✓. `DTL` is able to operate on Java objects.

R2.1 ✓. No meta model for models needed.

R2.2 ✓. No meta model for transformation descriptions needed.

R3.1 ✓. Every transformation operation can be renamed/changed/deleted, even at run-time.

R3.2 ✓. To add a new transformation operation, just implement a new class implementing the interface for a transformation operation and export it.

R3.3 ✓. Transformation operations are written in Java.

R4 ✓. Transformation descriptions can either be written in Java or in XML files.

R5.1 ✓. Since the transformation descriptions can be expressed in Java, such meta information retrieving needs only a new Java method analysing the description.

R5.2 ✓. The XML files are parsed with a normal XML parser which creates the Java equivalent for them.

7.2.8 Conclusion

The support of Java objects as input and output format of the transformations is an uneasy condition. Although theoretically it is possible with every technology, only DTL and Smooks are able to provide this out-of-the-box.

Since one of the most important requirements of the OpenEngSB transformation solution is, that there are no meta models for the models involved in the transformations, most of the TDLs compared in this section are not fitting, even though these existing solutions are more powerful than the introduced TDL. Like in requirement **R1** only Smooks and DTL are fulfilling the wanted behaviour.

The modification of the transformation operation set is a very useful feature. With custom functions, it is possible to extend the range of transformation possibilities to the extent of the language in which the custom functions are written. However, only DTL fulfils all needs of this requirement. QVT and Kermeta are almost as good as DTL, with the exception that it is not possible to rename already existing operations, which is only a minor complaint. While it is not possible at all in ATL, Smooks provides a possibility to allow custom functions, but they are rather hard to write and limited to the functions a SAX parser can provide.

The requirement of the separation from the transformation definition from the transformation execution is provided by all technologies. Every solution has a data format defined in which it is possible to describe the transformations without code in the project.

The easy parsing of the transformation descriptions for analysis of the transformation meta-data is not that clearly defined. For this comparison, the assumption of tool support has taken into consideration. So Smooks and DTL have a clear advantage of having a XML syntax, since there is a huge support of XML parsing in almost every programming language. The descriptions of the other technologies can also be parsed, but it is harder to analyse them. Smooks and DTL are using an XML based approach, which brings the benefit of a very clear structure of the descriptions. The other approaches allow a more free possibility to define the transformations in an own simple programming language providing all necessary operational constructs. Because of these unstructured definitions it is harder to analyse them.

7.3 Prototype validation by use-cases' development

This section validates the developed prototype by developing the use-cases described in chapter 4. It will be shown that these use-cases can be realised with the solution presented in this thesis.

The implementation of the use-cases is provided by the thesis author by an OSGi bundle which can be started in a running OpenEngSB instance. In this section, this bundle will be called **presentation bundle**. The models, operations and all other elements needed for the logic of this bundle is provided in a different bundle which is started automatically with the other bundle. In this section, this bundle will be called **presentation provider bundle**.

7.3.1 OpenEngSB Domains, transformation functionality

The use-case which is validated in this section is described in section 4.2. In order to validate the prototype with this use-case, the scenario defined in the use-case description is realised with

it. In the following, the set-up of the validation is explained, afterwards some interesting parts of the implementation are shown and finally a short conclusion is given, explaining the advantages of the prototype implementation for this use-case and the problems including their solutions found during the implementation.

Set-up

Before the realisation of this use-case can be shown, the set-up of the environment needs to be done. The first step to accomplish this, is to download and start an `OpenEngSB` instance. Since there is no official release of the `OpenEngSB` containing the *Transformation Environment* solution at the point of time this thesis was written, an unreleased version need to be installed.

This version of the project can be downloaded from the official Github repository[3]. As soon as there is a version of the `OpenEngSB` downloaded, it can be started with the start script, located under openengsb-folder/etc/scripts.

In the running `OpenEngSB` instance the bundles which providing the models for the use-case need to be added. In particular, there are four bundles that need to be added:

- The issue domain bundle[4] of the `OpenEngSB` project.
- The requirement domain bundle[5] of the `OpenEngSB` project.
- The bundle[6] containing the model coounterparts provided by the thesis author, the **presentation provider bundle**.
- The bundle[7] containing the logic for the scenario provided by the thesis author, the **presentation bundle**.

To install these bundles, the following steps need to be performed:

1. load the sources from the corresponding Github repositories.
2. run `mvn clean install` in each of the bundles folders to install them.
3. add them to the running `OpenEngSB` instance. For that a feature in the **presentation bundle** has been created. The two commands necessary to add all needed components are:

```
> feature:repo-add mvn:org.openengsb/org.openengsb.presentation/
  3.0.0-SNAPSHOT/xml/features
> feature:install openengsb-presentation
```

[3] `https://github.com/openengsb/openengsb-framework`
[4] `https://github.com/openengsb-domcon/openengsb-domain-issue`
[5] `https://github.com/openengsb-domcon/openengsb-domain-requirement`
[6] `https://github.com/Arcticwolf/openengsb-presentation-provider`
[7] `https://github.com/Arcticwolf/openengsb-presentation`

As soon as these steps are done, the models are installed in the Transformation Environment. The models are recognized by the `Model Registry` automatically. As examples, two of these models are shown in the appendix A. The **presentation bundle** also contains all *Transformation Descriptions* needed for the use-case.

Implementation

For this use-case four models have been defined, provided in the **presentation provider bundle** in the path 'src/main/java/org/openengsb/presentation/model':

- 'Issue' which resembles the model for the issue domain.
- 'ExternalIssue' which resembles the counterpart of the 'Issue' model.
- 'Requirement' which resembles the model for the requirement domain.
- 'ExternalRequirement' which resembles the counterpart of the 'ExternalRequirement' model.

The Transformation Descriptions which are used in this use-case are located inside the provided **presentation bundle** in the path 'src/main/resources/META-INF/transformations'. All files located there which have the suffix '.transformation' contain Transformation Descriptions and are added autonomously by the Transformation Environment at the start of the bundle.

Two of these Transformation Descriptions are given in appendix B of this thesis, namely the both Transformation Descriptions between the `OpenEngSB` issue domain model and its counterpart. The operations used there were almost all already explained in section 6.3.

```
package org.openengsb.presentation.operation;

public class PrefixChangeOperation implements TransformationOperation {
  private String oldPrefixParam = "oldPrefix";
  private String newPrefixParam = "newPrefix";

  @Override
  public Object performOperation(List<Object> input,
    Map<String, String> parameters)throws TransformationOperationException {
    String oldPrefix = parameters.get(oldPrefixParam);
    oldPrefix = oldPrefix != null ? oldPrefix : "";
    String newPrefix = parameters.get(newPrefixParam);
    newPrefix = newPrefix != null ? newPrefix : "";
    String result = input.get(0).toString();
    result = StringUtils.removeStart(result, oldPrefix);
    return StringUtils.join(new Object[] { newPrefix, source });
  }

  // remaining functions
}
```

Listing 7.1: Simplified digest of the 'changePrefix' operation

Three of the operations shown in these Transformation Descriptions are not part of the standard *Transformation Operation* set of the Transformation Environment. These operations are defined and exported by the provided **presentation bundle**. Listing 7.1 shows a simplified digest of the **changePrefix** method, which make explicit that it is not hard to write an own Transformation Operation. A short description of the additional operations are given in the following:

changePrefix This operation removes the string given in the parameter **oldPrefix** from the beginning of the value from the source field. After that, it writes the string given in the parameter **newPrefix** at the start of this value and returns the result.

Example: In the source field the value 'issue-1' is written, the **oldPrefix** parameter is set to 'issue' and the **newPrefix** is set to 'requirement', then the result of the operation would be the string 'requirement-1'.

flatList This operation flattens a list of strings into a string where the list values are concatenated. Between each element, the string defined in the parameter **concatString** is placed.

Example: In the list located in the source field, the set A, B, C, D is saved and the **concatString** parameter is set to '?' then the result of the operation would be the string 'A?B?C?D'.

createList This operation is the reverse operation to the **flatList** operation. It takes a string as input, splits the string based on the parameter **splitString**, fills the resulting set of elements into a list and returns this list.

Example: In the source field, the string 'A?B?C?D' is saved and the **splitString** parameter is set to '?' then the result of the operation would be a list with the element set A, B, C, D.

toString This operation calls the `toString` method of the object in the source field and returns this string. This method was needed for working with enumerations.

Example: In the source field, there is an enumeration object with the value `NEW`. Then the operation returns 'NEW'.

After the necessary set-up of the environment has been done and the additional Transformation Operations are explained, the interested reader should be able to understand and execute the use-case implementation. In the following some interesting code parts of the scenario implementation are shown. The interested reader may download the solution and the provided **presentation bundle** and try it out himself.

```
ModelDescription sourceModel = new ModelDescription(
  "org.openengsb.presentation.model.ExternalRequirement", "3.0.0.SNAPSHOT");
ModelDescription targetModel = new ModelDescription(
  "org.openengsb.domain.requirement.Requirement", "1.0.0.SNAPSHOT");

Requirement requirementInstance = (Requirement) transformationEngine.
  performTransformation(sourceModel, targetModel, extRequirementInstance);
```

Listing 7.2: Simple Transformation request

Listing 7.2 shows a simple initiation of a transformation through the *Transformation Engine* component. This transformation request is very simple, since there is a direct Transformation Description between both models and also no identifier is added to the graph search algorithm. The preconditions of this code listing are:

- `transformationEngine` is a reference to the `Transformation Engine` service.
- `extRequirementInstance` is an instance of the 'ExternalRequirement' model.
- There is a Transformation Description defined between the 'ExternalRequirement' and the 'Requirement' model.

```
ModelDescription sourceModel = new ModelDescription(
  "org.openengsb.domain.requirement.Requirement", "1.0.0.SNAPSHOT");
ModelDescription targetModel = new ModelDescription(
  "org.openengsb.domain.issue.Issue", "3.0.0.SNAPSHOT");

String identifier = null;
if (isAlsoDeveloper(requirementInstance.getAssignedTo())) {
  identifier = "RequirementToIssueForDeveloper";
} else {
  identifier = "RequirementToIssue";
}

Issue issueInstance = (Issue) transformationEngine.performTransformation(
  sourceModel, targetModel, requirementInstance, Arrays.asList(identifier));
```

Listing 7.3: Transformation request with identifier

Listing 7.3 shows a transformation request with an additional identifier. As already explained in section 6.4, this identifier array influences the transformation graph path search. It means that in the resulting *Transformation Path* a Transformation Description must be involved with the given identifier. The preconditions of this code listing are:

- `transformationEngine` is a reference to the `Transformation Engine` service.
- `requirementInstance` is an instance of the 'Requirement' model.
- `isAlsoDeveloper` is a method which returns true if the requirement creator is also an internal developer.
- There are Transformation Descriptions defined between the 'Requirement' and the 'Issue' model with the shown identifiers.

```
ModelDescription sourceModel = new ModelDescription(
  "org.openengsb.presentation.model.ExternalRequirement", "3.0.0.SNAPSHOT");
ModelDescription targetModel = new ModelDescription(
  "org.openengsb.presentation.model.ExternalIssue", "3.0.0.SNAPSHOT");
```

```
ExternalIssue result = null;

try {
  result = (ExternalIssue) transformationEngine.performTransformation(
    sourceModel, targetModel, extRequirementInstance);
  System.out.println("UNWANTED - transformation possible");
} catch (IllegalArgumentException e) {
  System.out.println("WANTED - transformation not possible");
  // comes here because the 'Requirement' model is gone
}

TransformationDescription desc = getNewTransformationDescription();
transformationEngine.saveDescription(desc);

try {
  result = (ExternalIssue) transformationEngine.performTransformation(
    sourceModel, targetModel, extRequirementInstance);
  System.out.println("WANTED - transformation possible");
  // comes here because the new description fills the "gap" between
  // requirements and issues
} catch (IllegalArgumentException e) {
  System.out.println("UNWANTED - transformation not possible");
}
```

Listing 7.4: Transformation request repairing

Listing 7.4 shows the code passage which handles the adding of a new Transformation Description after the shut down of the bundle containing the 'Requirement' bundle. The direct transformation is not possible any more because of the loss of this model. A possibility to fix this problem is shown in this listing. A new Transformation Description is inserted into the Transformation Environment via the `Transformation Engine`. This is no problem, since a Transformation Description can also be written as Java object. The preconditions of this code listing are:

- `transformationEngine` is a reference to the `Transformation Engine` service.
- `extRequirementInstance` is an instance of the 'ExternalRequirement' model.
- `getNewTransformationDescription` is a method which returns a Transformation Description between the 'ExternalRequirement' and the 'Issue' model.
- There is a Transformation Description defined between the 'Issue' and 'ExternalIssue' model defined.

In this thesis was explained that a model can consist of more than one Java class. The key functionalities to allow such model class sets, are the **instantiate** Transformation Operation and the possibility of nested field access:

instantiate operation With this Transformation Operation it is possible to create instances of Java classes. This is important because this can be used to create instances of model classes other than the 'master' class.

nested field access The nested field access allows the access to properties of objects which are properties of the 'master' class. So this can be used to get information from the nested classes to e.g. feed new created object instances.

```
...
<instantiate>
  <source-fields>
    <source-field>priority</source-field>
    <source-field>status</source-field>
    <source-field>type</source-field>
  </source-fields>
  <target-field>metaInfo</target-field>
  <params>
    <param key="targetType"
      value="org.openengsb.presentation.model.IssueMetaInfo" />
  </params>
</instantiate>
...
```

Listing 7.5: DTL multi-class model example 1: Issue → IssueExternal

Listing 7.5 shows a part of a DTL description, which uses the **instantiate** operation to create an instance of a sub class. The model IssueExternal consist of the master class IssueExternal and the additional class IssueMetaInfo. In the shown example, an instance of this additional class is instantiated and filled with the information of **priority**, **status** and **type** from the Issue model. This new created instance is then set to the property **metaInfo** of the ExternalIssue class.

```
...
<instantiate>
  <source-field>metaInfo.priority</source-field>
  <target-field>priority</target-field>
  <params>
    <param key="targetType"
      value="org.openengsb.domain.issue.model.Priority" />
    <param key="targetTypeInit" value="valueOf" />
  </params>
</instantiate>
...
```

Listing 7.6: DTL multi-class model example 2: ExternalIssue → Issue

Listing 7.6 shows again a part of a DTL description (in fact from the inverse description used for listing 7.5). The example creates an instance of the Priority enumeration class which is part of the Issue model. As base for this enumeration class instantiation, a string value of the ExternalIssue model is used.

However, the access to this string value is a nested field access. It takes the object saved in the **metaInfo** property (which has the type IssueMetaInfo as seen in the last example) and gets the value of the **priority** property of this object.

Conclusion

With the implementation of the described use-case it has been shown that it is possible to use the presented transformation solution in environments where the need of model transformations is given. The prototype is a possibility to add a transformation solution to a Java project which is easy to use and understand by Java developers. There is no need for meta models and time-consuming format transformations between Java objects and a format readable for the transformation solution.

These models are only connected through the Transformation Descriptions, so the models do not need to know anything from each other. The possibility to add custom Transformation Operations allows a wide variety of transformation applications, since the code of the custom Transformation Operations are written in Java. This means that the complete programming language power can be used for a custom Transformation Operation. It does not matter if the Transformation Operation uses simple string operations or if it calls web services, load data from a data source or even start a different process.

With the possibility of performing transformations based on Transformation Paths, the `Transformation Engine` caller do not need to know how the internal transformation graph structure looks like. The only thing the caller needs to know is where the transformation should start and where it should end. No knowledge about the intermediate models in the Transformation Path is necessary.

Additionally, with the **instantiate** Transformation Operation and the feature of nested field access, it is possible to allow model class constructs, instead of only supporting pure class to class model transformations. This allows a wider range of supported transformation scenarios and can be combined with the powerful mechanism of creating custom operations.

During the first implementation of this use-case two problems were found, which were fixed afterwards in the transformation solution:

1. The models of the use-case define separate Java objects as properties to structure their information (for example `Person` in the 'Requirement' model). At the first iteration, there was no generic possibility present to instantiate these objects. Thus, it was only possible to create instances of these separate objects (`Person`) by adding an own custom function to do this.

   To fix this problem, the **instantiation** Transformation Operation was extended. Before the extension, this operation was only able to instantiate objects located in the Java development kit, so the Java objects used in the models to structure their information could not be instantiated through this operation. After the extension, this Transformation Operation tries to instantiate the object also by trying to load the defined class through the `Model Registry` service (see section 6.3 for details).

2. After the problem explained in point 1 was resolved, the first version of the Transformation Descriptions were created. These Transformation Descriptions were rather complicated, due to the fact that there was no support for nested field access. That means, that every time someone wanted to get data from a nested object, this nested object first needed to be assigned to a temporary field, which then be accessed in the usual way. To simplify

this procedure and to make the Transformation Descriptions more readable, the support for nested field access was added (see section 6.3 for details).

7.3.2 Andritz Hydro, model update propagation

The use-case which is validated in this section is described in section 4.3. In order to validate the prototype with this use-case, the scenario defined in the use-case description is realised with it. In the following, the set-up of the validation is explained, afterwards some interesting parts of the implementation are shown and finally a short conclusion is given, explaining the advantages of the prototype implementation for this use-case and the problems with their solutions found during the implementation.

Set-up

The set-up of this use-case is the same like the set-up of the transformation functionality use-case explain previously. The models and the Transformation Descriptions used in this use-case are all located in the **presentation provider bundle** and the **presentation bundle** bundle.

Implementation

For this use-case four models have been defined, provided in the **presentation provider bundle** in the path 'src/main/java/org/openengsb/presentation/model':

- 'Eplan' which resembles the model for the electrical engineering domain.
- 'Logicad' which resembles the model for the software engineering domain.
- 'Zuli' which resembles the model for the hardware configuration domain.
- 'Signal' which resembles the common concept for the other three models.

In the path 'src/main/resources/META-INF/transformations' of the **presentation bundle** bundle in the files with the '.transformation' suffix the Transformation Descriptions are located.

This use-case uses also the custom transformation operations added by the **presentation bundle** which were explained in the set-up of the transformation functionality use-case.

For the implementation of the use-case scenario, the *`Engineering Object`* concept was used (see section 6.7). By using this concept, it is essential that instances of the `Engineering Objects` are instantiated and saved in the `OpenEngSB` project. Only if the instances are present, the automatic model propagation is able to be executed.

However, the creation of these instances was one of the key architectural decisions for the use-case implementation. There are several ways to create the instances of the `Engineering Objects` and also there are some architectural positions where the code for this instantiation can be placed (a more detailed discussion about that topic can be seen in section 8.4).

The creation of the `Engineering Object` instances is implemented in an `EKBPre-CommitHook`. A hook is a piece of software that is executed at a specific time point in a process, which enhances the power of this process. Specifically, an `EKBPreCommitHook` is

called before a commit by the Engineering Knowledge Base (*EKB*) is executed. In such hooks, the commit object can be modified in order to add/update/delete elements of this commit.

Similar to the custom transformation operations, hooks are defined via interfaces. If a new hook should be installed, a class which implements the corresponding interface need to be exported via Open Services Gateway initiative framework (*OSGi*). After the exportation the hook is considered in every future commit procedure.

Listing 7.7 shows how an `EKBPreCommitHook` is registered in the `OSGi` environment with Apache Aries Blueprint. The `SignalCreationHook` class is implementing the `EKBPreCommitHook` interface and needs the `QueryInterface` service so that it can check for the existence of the corresponding `Engineering Objects`.

```
<service interface="org.openengsb.core.ekb.api.hooks.EKBPreCommitHook">
  <bean class="org.openengsb.presentation.hook.SignalCreationHook">
    <argument ref="queryInterface"/>
  </bean>
</service>
```

Listing 7.7: Register an EKBPreCommitHook

Like already mentioned, the `SignalCreationHook` class has the job to check for every commit, if there are models committed for which there is no 'Signal' instance present. This instance could either be saved in the Engineering Database (*EDB*) or it is present in the same commit. If both cases are not fulfilled, a 'Signal' instance is created. Listing 7.8 shows the basic logic in the `SignalCreationHook`. The preconditions of this code listing are:

- `needsToBeChecked` is a method which returns true if the model is an 'Eplan', a 'Logicad' or a 'Zuli' model instance and returns false otherwise.
- `signalExists` is a method which returns true if a 'Signal' model instance for the given model is either in the `EDB` saved, or is located in the actual commit and returns false otherwise.
- `createSignal` is a method which returns for the given model instance the corresponding 'Signal' model instance. In this instance, the id of the 'Signal' and all references to the other model instances are set.

```
public class SignalCreationHook implements EKBPreCommitHook {

  @Override
  public void onPreCommit(EKBCommit commit) throws EKBException {
    for (OpenEngSBModel model : commit.getInserts()) {
      if (needsToBeChecked(model) && !signalExists(model, commit)) {
        commit.getInserts().add(createSignal(model));
      }
    }
  }

  // remaining functions
}
```

Listing 7.8: Base logic of the SignalCreationHook

After the question about the creation of the `Engineering Object` instances was answered, the next interesting part was how the 'Signal' is structured. In particular it is worth mentioning how the connection to the model instances is defined.

```
public class Signal {
  @OpenEngSBModelId
  private String id;
  @OpenEngSBForeignKey(modelType = Eplan.class,
                       modelVersion = "3.0.0.SNAPSHOT")
  private String eplanId;
  @OpenEngSBForeignKey(modelType = Logicad.class,
                       modelVersion = "3.0.0.SNAPSHOT")
  private String logicId;
  @OpenEngSBForeignKey(modelType = Zuli.class,
                       modelVersion = "3.0.0.SNAPSHOT")
  private String zuliId;

  // other fields, getters and setters
}
```

Listing 7.9: Connections from 'Signal' model to other models

Listing 7.9 shows this part of the 'Signal' model. All four properties shown in this listing are important for the `Engineering Object` concept. The **id** property defines the identifier for the 'Signal' model instance. This is important for internal reasons. The **eplanId**, **logicId** and **zuliId** are needed for the linking of the other model instances with the 'Signal' instance. As values, there need to be an identifier string placed in each of these properties which refer to a model instance of the type defined in its annotation.

Regarding the Transformation Descriptions between the 'Eplan', 'Logicad' and 'Zuli' models and the 'Signal' model, they are defined in exactly the same way as they would have been written for a direct transformation request. The only difference in this situation is that the model update propagation performs the transformations in background (see section 6.7).

After the models are defined, the `Engineering Object` connections to the other models are set, the transformations between the models and the `Engineering Object` model are defined and the `EKBPreCommitHook` which does the 'Signal' model instantiation is implemented and registered, the complete construct is ready to use.

From now on, every time a model is committed, the `OpenEngSB` takes care for the model update propagation. This process is running completely in the background, so the user does not need to be aware of it any more. Models are just committed in the `OpenEngSB` as they are used to.

```
EKBCommit commit = createCommitObject();
commit.addInsert(eplanInstance).addInsert(logicadInstance);
persistInterface.commit(commit);
// do some modifications
commit = createCommitObject();
commit.addUpdate(eplanInstance);
persistInterface.commit(commit);
```

Listing 7.10: Committing models to the OpenEngSB

Listing 7.10 shows how models are committed in the OpenEngSB. To accomplish this, the PersistInterface is used, which is a part of the EKB (see section 2.2) and is responsible for the persisting of models. The preconditions of this code listing are:

- persistInterface is a reference to the PersistInterface service.
- createCommitObject is a method which returns a new EKBCommit object.
- eplanInstance is an instance of the 'Eplan' model.
- logicadInstance is an instance of the 'Logicad' model.

Conclusion

In the implementation of the Andritz Hydro use-case it has been shown that it is possible to use the automated model update propagation via the Engineering Object concept introduced in this thesis in a real world scenario. While it has some pre-conditions, like the correct creation of Transformation Descriptions which do not mess up the data integrity of the model instances, automated model update propagation is a useful feature which can help reducing the amount of errors caused by communication mistakes between employees.

Another practical side-effect of this feature is that the employees which work on the same product do not need to take care about the model synchronization any more. This synchronization is automatically done by the framework and human errors are reduced.

During the implementation of this use-case some problems have been found which have been corrected in the final implementation:

- During the planning of the Engineering Object concept, the detail where the instances will be created was overseen. In discussions with the OpenEngSB research and development team there were different solutions for that problem found (see section 8.4). For this use-case the solution based on an EKBPreCommitHook was chosen because this solution is the most dynamic one and will most probably be the default way to go for this problem.
- During the first invocations of the use-case implementation, there were some irregularities found where sometimes models were not correctly updated. The reason for that was that the algorithm for the update propagation was not written in a recursively manner, which had the result that already created and updated models in a commit were ignored (i.e. the changes were not seen) since they got loaded every time from the EDB. This has been fixed by rewriting the algorithm in a recursive style.

7.4 Prototype benchmarks

An important indicator for the usability of a new application/tool is the performance. For the proposed prototype, two aspects have been taken into account:

Transformation definition time defines the time which is needed in order to define transformations.

Transformation execution defines the performance of the execution of a transformation at runtime.

7.4.1 Transformation definition time

This section illustrates the time needed to define transformations with the introduced prototype. The process to define transformations can be separated into three sub processes:

Model definition. This sub process is normally the most time consuming part if there are no data models already defined. Since the extraction and planning of data models is not in the scope of this thesis, they are not considered in the time needed for transformation definition. However, if the model was already defined and there exist a Java object for it, it takes a few minutes to define it as a model. It is just needed to add the required annotations (see section 6.6) and this sub process is finished.

Transformation operation definition. This sub process is only needed if there are *Transformation Steps* required for which there exist no Transformation Operation to fulfil their needs. In this case new operations need to be added. Since the proposed prototype is implemented in order to allow an easy way to add new operations this is not difficult to do.

There exist an interface which needs to be implemented to realise a new operation. After the implementation, this implemented class needs to be exported via `OSGi` in order to provide it for the transformation definition (see section 6.3.2). Depending on the complexity of the new operation and the experience of the developer implementing in Java this sub process takes between twenty minutes and a few hours.

Transformation definition. In this sub process the actual transformations between the models need to be defined. Most of the time spent in this sub process is the planning of the transformation definition. It needs to be planned which properties of the source model shall be written in which order, with which operations, to which properties of the target object. Additionally, planning which models shall be connected through transformations need to be done.

As soon as the planning has been completed, the transformation itself needs to be written. This can be done either in XML or in Java code, but the XML way is the preferred way since it separates the transformation definition from the project code. In the following, the explanations will be referring to the XML approach. The user has to write down the previously defined transformation in `DTL` format. To accomplish this, the user needs to know about the structure of the involved models, the `DTL` operations and the parameter of the operations (see section 6.3).

Someone new to `DTL` needs about two to three hours to get familiar with the language. However, this step is only required for the first usage of `DTL`. The transformation writing takes (dependent on the amount of transformation steps per transformation definition)

about half an hour per definition. This time can be improved by providing a proper graphical user interface which supports the user (see section 9.4.1).

As the list of the steps point out, it is not a hard task to define transformations for the user. The most time consuming steps are the planning of the models and the transformations. However, this time would also be needed by other transformation solutions, since no tool is able to spare the user a planning phase.

7.4.2 Transformation execution

This section illustrates the time spent by the prototype to actually perform transformations. It has already been explained which benefits the usage of the introduced prototype delivers to a project. Additionally, this section shows that the added intermediate step of calling the prototype's transformation capabilities does not decrease the system overall performance in a noticeable amount.

To accomplish this, a small benchmarking tool has been developed, which performs transformations in specific set-ups and returns the average time amount spent for performing the transformation requests. Every set-up is tested in four different modi, which differ on the instances count that shall be transformed. The modi are one instance, ten instances, hundred instances and thousand instances. The set-ups are defined as follows:

direct simple transformation. A transformation from one model into another model to which a direct transformation has been defined. The search for a Transformation Path is very easy. The involved models consist each of only one Java class. Additionally, the transformation itself mainly consist of string manipulating Transformation Steps and almost no difficult Transformation Operations are used.

direct complex transformation. A transformation from one model into another model to which a direct transformation has been defined. The search for a Transformation Path is very easy. However, the transformation itself is harder. The models consist of a class structure, so object instantiations need to be performed which need the most performance of the pre-defined Transformation Operations.

indirect simple transformation. Very similar to **direct simple transformation**, but there is the need to find and perform a Transformation Path from the source model to the target model, with 2 intermediate models. All intermediate models are in the same complexity class like in **direct simple transformation**.

indirect complex transformation. Very similar to **direct complex transformation**, but there is the need to find and perform a Transformation Path from the source model to the target model, with 2 intermediate models. All intermediate models are in the same complexity class like in **direct complex transformation**.

For every of these set-ups there were 4 modi performed which differ in the amount of instances that shall be transformed. So in total there where sixteen scenarios tested where each scenario (combination of set-up and amount of instances to be transformed) was performed five times and the average transformation execution time of this five performances has been taken as

a result. The average has been taken because there exist some slight fluctuations in the service call performance.

Figure 7.1 show the result of the benchmark process. The x-axis shows the number of transformations that were performed and the y-axis show the milliseconds the transformations needed for execution. It can be seen that even at an amount of one hundred transformations, the prototype needs way less than one second to perform all transformations. At an amount of thousand transformations, the differences between the set-ups are in an area so that the user can notice the difference without measuring it.

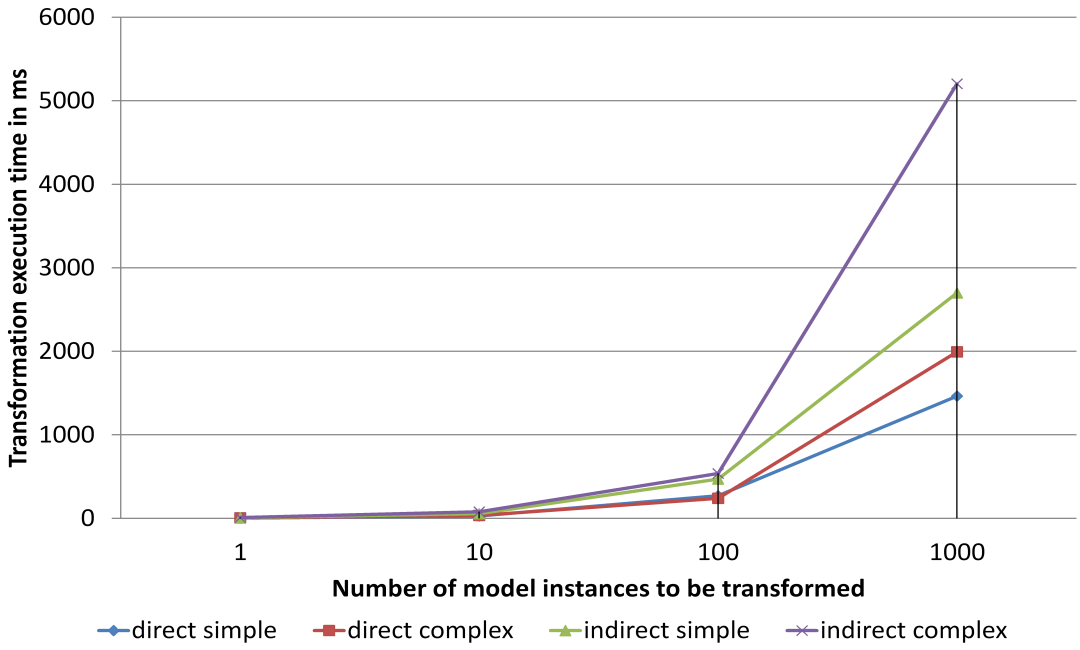

Figure 7.1: Transformation Performance Benchmark Results

Two important notes on the benchmark:

- The run time of of the transformations rely mainly on the operations that need to be performed. If there are more complex operations which need more computation and communication power (e.g. call a web service for a specific value) these operations slow down the performance.

- The transformations have been performed in a serial way (one after another). If they would be performed in parallel, the performance would be better, since there is a new performer created for every transformation which operate independently from the others.

Table 7.2 shows the absolute time values of the benchmark in a textual form, since the concrete numbers can not be seen from figure 7.1.

	direct simple	**direct complex**	**indirect simple**	**indirect complex**
1 transf.	4 ms	5 ms	7 ms	10 ms
10 transf.	28 ms	35 ms	55 ms	79 ms
100 transf.	271 ms	240 ms	471 ms	537 ms
1000 transf.	1464 ms	1994 ms	2699 ms	5204 ms

Table 7.2: Transformation Performance Benchmark Results Absolute Numbers

The results show that the proposed transformation solution does not decrease the system performance in a drastic way: A normal transformation usually takes about 6 milliseconds. This value is rather low compared to other service calls present in the `OpenEngSB`, e.g:

- A typical workflow (some service calls and calculations): about 1 second.
- Saving a model in the `EDB`: about 100 milliseconds - 200 milliseconds.
- Loading a model from the `EDB`: about 30 milliseconds.
- Calling a logging service: about 5 milliseconds.

7.4.3 Conclusion

The test of both aspects of the prototype performance have been checked in this section.

- It has been shown that the definition of the transformations does not take a lot of time for the user. Precisely, not more than any other transformation solution would take, since these need also some time for planning and actual writing the rules.
- The run time performance check shows that the provided performance is acceptable and so it can be concluded that it can also be used in other domains than the `OpenEngSB` which have similar response time requirements.

The summary of the validation states out, that the proposed prototype provides a transformation solution that,

- makes the communication between components easier.
- needs no meta modelling experts.
- provides a good performance.
- does not slow down the system dramatically.

CHAPTER 8

Discussions

8.1 Overview

In this chapter, some examples are listed for which the transformation solution presented in this thesis can be used to improve current problem solving approaches in the software engineering context. Additionally, a discussion point was added to this chapter, which deals with an aspect of the architectural planning for using the model update propagation solution proposed in this thesis.

8.2 Model version migration

The Open Engineering Service Bus (*OpenEngSB*) uses a versionised model store, the Engineering Database (*EDB*). Since the `EDB` is able to load model instances from an earlier point in time, it is possible that the model which was used to save the model instance does not exist any more in that form, e.g. if the model has been changed.

A change in a model could be that additional properties have been added (the smallest possible problem in the model migration), a property has been renamed, a properties type has been changed or a property was deleted. Since all these changes (except the first one) makes the loading of an old model instance problematic, a solution for that need to be found.

A quite common approach performed in many companies is to update the old data instances to the new format in the data source itself. An example for this procedure is the migration of old entries in a database table, when a new column has been added. In many cases this migration can be performed automatically, but often human interaction is needed.

The typical procedure in this common approach is the generation of an update script which is executed against a data source. The updated data entries are either directly updated in the data source, or can be reviewed in a migration tool before the updated entries are persisted. Often a mix of both approaches is defined, where most of the entries are updated automatically and only

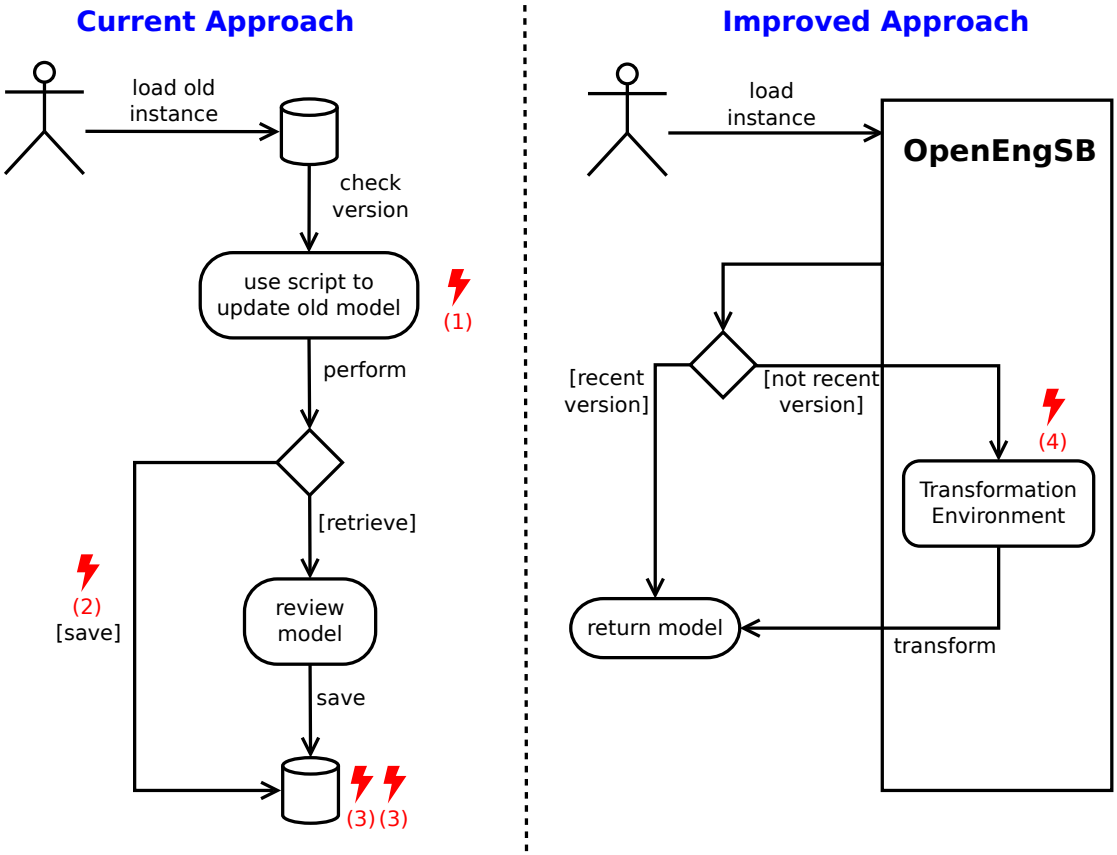

Figure 8.1: Current model migration vs. improved model migration

the entries which cause problems (however this detection is implemented) need to be reviewed by a human.

This procedure is graphically illustrated on the left side of figure 8.1. The lightning bolts which have been added to the figure visualize steps that usually are error-prone and can cause problems. The number under the lightning bolts is used as identification number for the problems which are described here:

1. At this point an error can be introduced in the script. E.g. a typo, a logical error or an overseen exception for a specific property value. An error at this point can cause severe problems in the next steps.

2. Saving an updated model instance directly in the data source can be quite a big problem, especially if at point 1 an error happened. The updated model instance is saved without human review possibility, so a sanity check for them need to be added before or do not happen at all.

3. The updated model instance is now persisted in the data source. If an error happened at point 1 or 2, this error is now persisted and has possibly overwritten the old values. If a mistake is found out after this point, only a data source backup can restore the old model instance.

By using `OpenEngSB` with the proposed transformation solution, the error-prone steps can be minimized. The solution approach here is, that the persisted old model instances are never changed. They stay untouched in the data source, but only the instances get transformed to the newest version of its model.

Precondition for this procedure is only that there are *Transformation Descriptions* defined between the old model versions and their newer versions. Thus, the loading mechanism of the `OpenEngSB` can be altered as follows:

1. A model instance is loaded from the `OpenEngSB`. The instance is checked if there is a newer version of the corresponding model.

   a) If there is no newer version of the model the instance is returned.

   b) If there is a newer version of the model, the instance is passed to point 2.

2. The model instance gets transformed to the newest model version and the result is returned.

The right side of figure 8.1 illustrates this behaviour graphically.

The only problem which may happen here is located inside the `OpenEngSB`, namely if a Transformation Description is incorrect. However, this problem is only a marginal one since the updated models get newer persisted. The only thing that needs to be done here is to correct the Transformation Description. The next time an old model instance is loaded which need the formerly incorrect Transformation Description is now returned correctly.

8.3 (Traceable) Tool Chains

Whenever more than one employee is working on a model instance, it is often needed to pass the instance from one employee to another. Often the outcome of the work from employee is needed by another employee to start or continue his part of the work. Another example would be if one employee found an error in a model instance and wants to inform another employee that there is an instance that need to be checked.

Such scenarios where several tools are involved in a development process, where each tool need the outcome produced by other tools, is called a *Tool Chain*. Such situations provide some challenges, like the question how models are converted so that another tool can use them, or how it is possible to reconstruct the way a model instance has taken in the process.

A common current approach is to pass the instances per mail or any other direct way to spread models between employees. E.g. an employee (Alice) uses Tool A to produce some outcome. This outcome can be of any type, e.g. a CSV or Microsoft Word file.

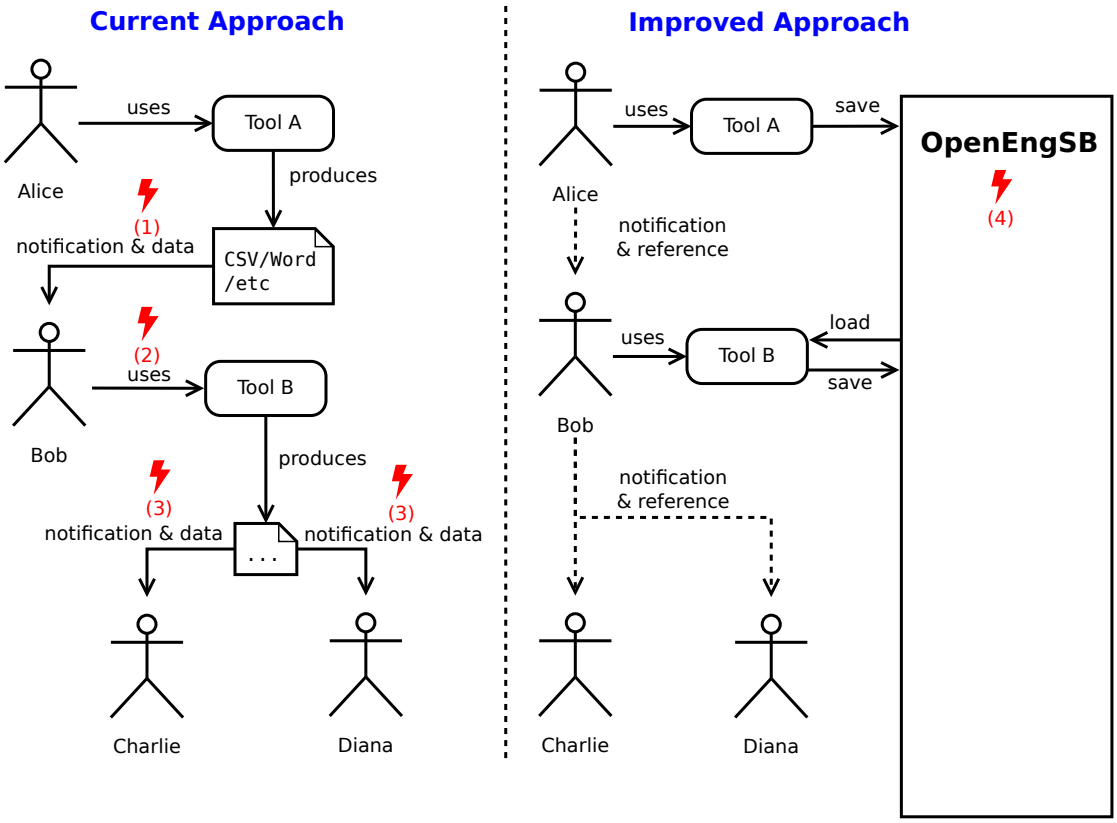

Figure 8.2: Current Tool Chain approach vs. improved Tool Chain approach

Bob needs this outcome to work further on it, so Alice sends the file to Bob. Bob transforms this file to a format he needs and starts his work with the transformed file. When he is finished, the next outcome is generated and is sent to Charlie and Diana. They again transform the file and so on and so forth.

This procedure is graphically illustrated on the left side of figure 8.2. The lightning bolts which have been added to the figure visualize steps which are error-prone and/or may cause problems. The number under the lightning bolts is used as identification number for the problems which are described here:

1. If the notification is in an informal way or if there is no tracking of this notification, it is very difficult to reconstruct the trace of a model instance.

2. Bob now transforms the received model into a format he can work with. If the transformation of the outcome of the tool used by Alice to the format Bob needs is not defined very clearly, this can cause problems. Of course a wrongly performed transformation can happen here too.

3. In principle, the same problem like in point 1. However, if an error happened in point 2, the wrong instance is passed to the next employees in the *Tool Chain*.

This approach gets even more problematic if the transformation definitions between the models are not defined by a central instance, so that there is no common transformation approach. The missing traceability in this scenario makes it very difficult to locate the failing *Tool Chain* element.

With the `OpenEngSB` and its *Transformation Environment*, the problems in this context can be decreased (see right side of figure 8.2). The models and the Transformation Descriptions between the models are persisted in the `OpenEngSB` environment. Employees do not need to send the instances directly, but only send the other employees references to models they added/updated/etcetera.

The next employee is loading this model and let the Transformation Environment transform it to a format he can work with. The Transformation Descriptions are saved inside the Transformation Environment, so they are saved in a central place. If a Transformation Description changes, the change is only needed at one point in the interaction chain, but no employee need to be explicitly informed.

This infrastructure introduces two big benefits:

- The employees do not need to worry about formats, they get the model instances in the format they need directly from the `OpenEngSB` environment.
- Whenever model instances are saved/loaded and whenever instances get transformed, the `OpenEngSB` can log this action. So the *Tool Chains* can be traced.

The only problem which can happen here, is located at the `OpenEngSB` namely if Transformation Descriptions are incorrect. However, since the steps are traced, the instances can be reset to the state they had before they were wrongly transformed. Especially with a versioned model persistence solution like the `EDB` this is an easy task.

8.4 Engineering Object Creation Possibilities

During the development of the *`Engineering Object`* concept (see section 6.7) the question was risen, where the best spot would be to instantiate the `Engineering Object` objects. There needs to be an entity defined, which is responsible for this task. This entity then, needs to be aware of all models which influence the `Engineering Object` instance and also needs to know about the possible connections.

In particular this means, the entity need to know which model instances are connected with each other through an `Engineering Object` instance. It also needs to know the identifier of these model instances to be able to fill the `Engineering Object` reference properties with the correct values. From a list of possible locations, the two most reasonable were chosen to be discussed shortly in this section.

Figure 8.3 shows the first possible location, which is at tool level. Here, the tool checks with every model instance insertion if the corresponding `Engineering Object` is created.

If this has already been done, the tool has nothing further to do. If not, the tool need to add the corresponding `Engineering Object` to the commit it is already performing.

There are certain ways to implement this. For example, one approach would be that every tool which commits model instances which are parts of `Engineering Objects` needs also to add this check. This approach has the drawback, that every tool need to be aware of the `Engineering Objects` and their influence on them.

Another approach would be to define that one tool does this check and is responsible for the `Engineering Object` concept interaction. In this way only this tool need to be aware of the `Engineering Objects`.

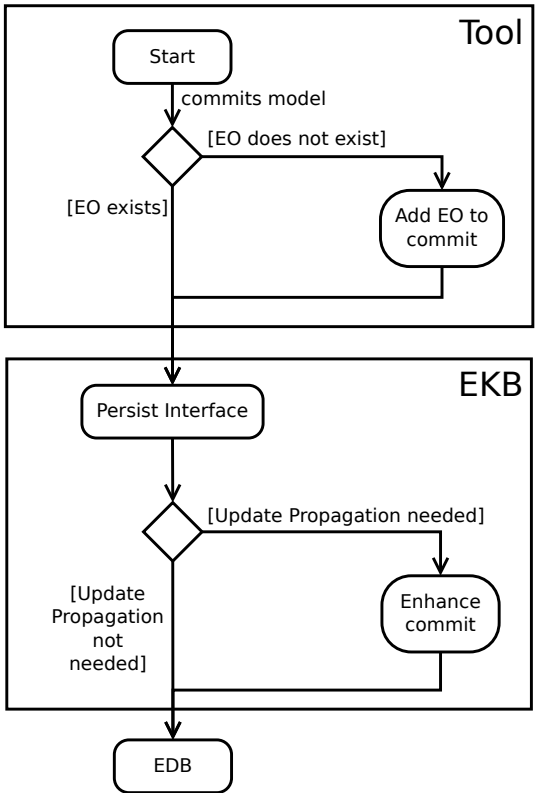

Figure 8.3: Tool based Engineering Object creation approach

This approach has the benefit of its simplicity. Also if there is a tool defined for the `Engineering Object` interaction, this tool is the single point of failure in this construct. However, it also has one really big disadvantage: the precondition that the tool has the knowledge about the `Engineering Object` concept and the interconnection of all involved models on tool level, so this approach is not very dynamic.

Figure 8.4 shows an alternate approach, which is more dynamic than the previous one. This approach operates at Engineering Knowledge Base (*EKB*) level. It is possible to put additional code to the commit procedure of the `EKB` via hooks. These hooks are Java classes implement-

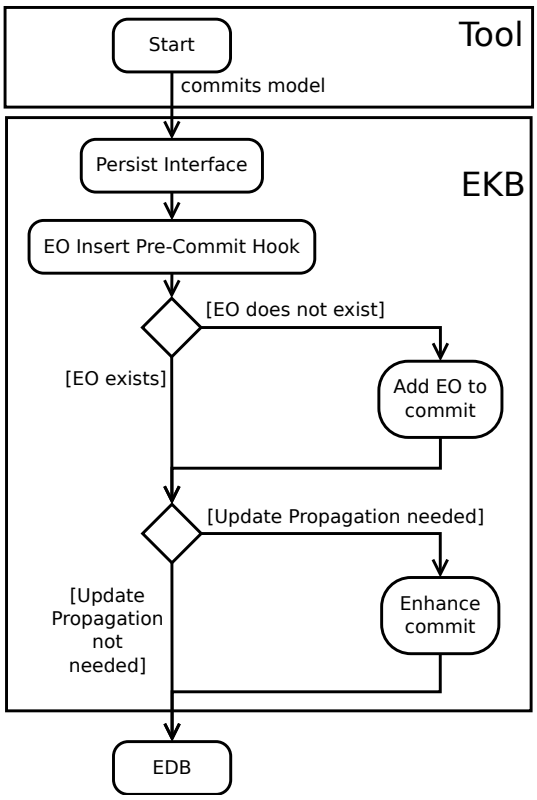

Figure 8.4: EKB based Engineering Object creation approach

ing a special interface which are registered in the Open Services Gateway initiative framework (*OSGi*) environment.

The persisting procedure of the `EKB` provides two kinds of these hooks:

PreCommitHook All hooks of this type are invoked before a commit is performed. In these hooks it is possible to modify the commit object, before it is actually persisted.

PostCommitHook All hooks of this type are invoked after a commit has finished. In these hooks it is possible to check afterwards how a commit has looked like, e.g. for logging or monitoring purposes.

The approach uses a `PreCommitHook` to modify the commit object. It scans the committed model instances, checks for the existence of the corresponding `Engineering Objects` and adds them to the commit if they do not exist in the commits point of time.

If for every `Engineering Object` a hook using this approach is added, the whole `Engineering Object` interaction is located at `EKB` level and can be modified, improved or removed without changing anything at tool level. This allows a complete separated development of the `Engineering Objects` from the tools and with it a separation of responsibility.

CHAPTER 9

Conclusion and Future Work

9.1 Overview

This chapter summarizes the thesis and its result. The limitations of the presented solution are described and a list of future work for improving the prototype is given.

9.2 Conclusion of the thesis

In every environment where a list of heterogeneous components need to communicate with each other, there is the problem to find a way to enable this communication. In particular, there is the need for a solution that enables the communication between the components, where every participant has its own models defined, even for things that are common with other participants.

The simplest solution to accomplish this, is to define common models inside an integration framework, where every component that wants to participate in a communication need to stick to the dictated format. However, this solution is only partly generic enough for modern integration frameworks, like for example Enterprise Service Bus (*ESB*) based frameworks (see section 2.3). In that way, for every component there is the need for connectors, which do the transformation work between the internal model(s) and the integration framework model(s).

The better option is to introduce a model transformation solution inside the integration framework. This solution is capable of performing the correct transformations in background. In such infrastructures there is often no need for connectors any more, since the transformations between two components can be done transparently. The only big problem here, is that the models and descriptions of the transformations need to be available to the transformation solution.

To overcome this problem, a common solution nowadays is, that the models are modelled in the means of Model Driven Engineering (*MDE*) and provided to the transformation solution (see section 3.4 and section 3.5). This enforces the integration framework user to deal with the

MDE methods and define the transformations accordingly. This is not an easy task and often too bloated for easy transformation requirements.

The prototype introduced in this thesis tries to improve the second option. Models are defined as normal Java classes, which get a special annotation (see section 6.6). These models are then provided to the transformation environment. After that, the last step is the definition of the transformations between the models and the set-up is completed (see section 6.8 for more information how to apply the prototype to a project). The prototype was developed and tested against the Open Engineering Service Bus (*OpenEngSB*) project, an open source tool integration solution (see section 2.2.

The advantages of this approach are:

- No MDE knowledge is needed.
- No meta models are needed.
- Easy to understand for Java developers.
- Easy adding and modification of transformation operations (see section 6.3).
- Easy to extend existing projects with the prototype.

In addition to that, the thesis presented an approach to perform automated model update propagation with the help of specific models, the *Engineering Objects* (see section section 6.7). With this concept it is possible to help a project to accomplish data integrity.

While the presented prototype is only capable of performing 1:1 (1 input model, 1 output model) transformations, it is already able to fulfil a list of typical use-cases. Especially with the dynamic transformation operation set, a lot of special needs can be fulfilled, like transformation operations which load data from external structures.

All in all, the presented solution provides a possible transformation solution alternative for projects which do not need the full function set provided by MDE based tools, but want an easy to use and adapt model to model transformation approach.

9.3 Prototype limitations

The presented transformation solution is not fully completed. There are some limitations for which there is no built-in solution until now realised. Built-in in that term means, that a requested feature can be expressed within the *Transformation Environment*.

These limitations are not final, it is possible to solve them in a built-in manner. So the limitations explained here are also points which are done in future work, but they were separated to specially point out that there is a work around for each limitation until the built-in solution is added.

However, the limitations which are presented in this section, can be solved by adding custom code outside the Transformation Environment. Here should be pointed out that the limitations can be solved built-in, but there is some further development needed, based on the presented solution.

9.3.1 Conditional transformations

There are situations in which there is the need that a model of a given source type should be transformed in different ways to another target model type. E.g. if the source model instance has a specific property set, then it should be transformed in a different way to the target model than in other situations. This feature is called *conditional transformations*.

Conditional transformations are a problem, because the `Transformation Engine` is a stateless component. The only thing it needs to know is at which model the transformation starts and which model should be the result of the procedure. To give the user the possibility to perform such kind of transformations, a workaround is added to the Transformation Environment. This workaround is based on identifiers for *Transformation Descriptions*.

Every Transformation Description has an optional identifier. This identifier needs to be unique in the Transformation Environment. The `Transformation Engine` methods have an optional list of identifiers as parameter. This parameter is a list of string objects, which should contain identifiers of transformations.

If identifiers are passed to a method, the search algorithm get an additional condition for the search result. This condition is, that all identifiers passed to the method, need to be in the path search result. If at least one identifier is not in the result, the search result is invalid (more details about the graph search algorithm can be read in section 6.4).

With this feature, the user has the possibility to set this additional parameter wherever the need of conditional transformations is given. The code for using this kind of transformations can be put e.g. in normal Java code or in a workflow.

Example workflow

This sub section gives an example where the need for a conditional transformation is given and how this feature can be implemented in a workflow so that it can be used in the `OpenEngSB` environment. Since the workflow code is very similar to the Java language it can be easily translated to Java functions.

Assume a company which builds some specific hardware elements for computers. This company develop everything for this elements on their own, also the electrical and information engineering part. Each engineering discipline uses its own models.

The electrical engineering tools use models which describe the circuits of the hardware elements. The information engineering tool models describe the hardware elements as variables the engineer can use (because of the structure of the electrical planning).

This company has a hierarchy of information engineers. The higher the hierarchy position of an engineer, the more rights for changes and more information are needed by this engineer.

If an electrical planning model needs to be transformed into an information planning model, the workflow need to check the hierarchy situation of the information engineer, since every information engineer get a different amount of information. Depending of the position of the engineer which should get the transformed model, a different Transformation Description need to be applied.

```
when
  e : NeedToTransformToInformationPlanningModelEvent()
```

```
then
  ElectronicalPlanningModel source = e.getModel();
  Recipient rec = e.getRecipient();
  InformationPlanningModel result;
  String identifier;
  if(rec.isHighHierarchy()) {
    identifier = "ElecToInfoHigher";
  } else {
    identifier = "ElecToInfoLower";
  }
  result = transformationEngine.performTransformation(
    EPModel, IPModel, source, Arrays.asList(identifier));
  // process the result
```

Listing 9.1: Example workflow for conditional transformations

Listing 9.1 shows how such a workflow could be implemented. However, this workflow is very easy and is mainly written here to clarify the process. The example workflow assumes that:

- The event, the models and the EPModel and IPModel are defined.
- All elements have the functions they use.
- In the Transformation Environment there are two Transformation Descriptions between the ElectronicalPlanningModel and the InformationPlanningModel, where one have the identifier 'ElecToInfoHigher' and the other one have the identifier 'ElecToInfoLower'. The Transformation Description with the identifier 'ElecToInfoHigher' processes more information into the resulting model than the one with the identifier 'ElecToInfoLower'.

9.3.2 Transformation cardinality (1:n, n:1, m:n)

The transformation solution presented in this thesis is able to handle 1:1 transformations. These kind of transformations have exactly one source model and exactly one target model. But there are specific situations where this kind of transformations can not fulfil all needs of the functionality.

The other transformation cardinalities are:

1:n The 1:n transformation cardinality, means that there is one source model and the result of the transformation are n models. An example for this transformation cardinality is given in the example given in the next subsection.

n:1 The n:1 transformation cardinality, means that there are n source models and the result of the transformation is 1 model. An example for this transformation cardinality is given in the example given in the next subsection.

m:n The m:n transformation cardinality, means that there are m source models and the result of the transformation are n models. This transformation cardinality is rather a theoretically one and an example where such transformations are needed is not easy to find.

Transformation cardinalities are a problem since the Transformation Descriptions are point to point oriented and a change here makes the transformation graph realisation more complicated. Also the `Engineering Object` concept would not work that well with transformation modi. However, until the cardinalities are added to the transformation solution itself, this limitation can be dealt with through custom code, e.g. in the tools or in a workflow. There the user can perform transformations multiple times and modify the results. Similar to the conditional transformation limitation, also transformation identifier can be used to overcome this limitation.

Example workflow

This sub section gives an example where the need for transformation cardinalities is given and how this feature can be implemented in a workflow, so that it can be used in the `OpenEngSB` environment. Since the workflow code is very similar to the Java language it can be easily translated to Java functions.

For this example the same company defined in the example of the conditional transformations is used. For security reasons, the electrical engineers need to add redundant elements, which overcome a total failure of a component if a specific element fails to operate. The information engineer does not know and does not need to know about this, since the element is a redundant one and makes no difference for the software.

As a result of this, a transformation from the information engineering model into an electrical engineering model is performed, it is possible that the result of this transformation would be two models, namely if it is a model which has redundant elements.

The other way round it could be that there is the need to transform a list of redundant elements into an information engineering model, if there is the need to check if all redundant elements are built correctly.

```
when
        e : NeedToTransformToElectronicalPlanningModelEvent()
then
  InformationPlanningModel source = e.getModel();
  List result = new ArrayList();
  if(source.hasRedundantElement()) {
    result.add(transformationEngine.performTransformation(
      IPModel,EPModel, source));
    result.add(transformationEngine.performTransformation(
      IPModel,EPModel, source, Arrays.asList("InfoToElecSecondary")));
  } else {
    result.add(transformationEngine.performTransformation(
      IPModel,EPModel, source));
  }
  // process the result
```

Listing 9.2: Example workflow for transformation modi

Listing 9.2 shows how such a workflow could be implemented. However, this workflow is very easy and is mainly written here to clarify the process. The example workflow assumes that:

- The event, the models and the EPModel and IPModel are defined.

- All elements have the functions they use.

- In the Transformation Environment there are two Transformation Descriptions between the InformationPlanningModel and the ElectronicalPlanningModel, where one of them have the identifier 'InfoToElecSecondary'.

9.4 Future Work

In this section, a list of possible improvements of the presented prototype solution is given, which are not dealing with a limitation of the prototype.

9.4.1 Transformation GUI + Transformation Validation

Until now, the Transformation Descriptions need to written manually by the users of the Transformation Environment. This approach works fine, but also require some knowledge preconditions for the developers, like:

- The fully qualified model names and their versions.

- The Dynamo Transformation Language (*DTL*) syntax.

- All available *Transformation Operations*.

- All parameters of the Transformation Operations with their exact names.

This circumstance gives some room for errors, since there can always be simple errors happen like an incorrect operation name or simple mistype errors. Also it is not the most user friendly way to write Transformation Descriptions.

The best solution to make the Transformation Description writing easier and more user friendly is the provision of a Graphical User Interface (*GUI*). In this `GUI`, the user can retrieve the available models with their versions and retrieve the available Transformation Operations.

The definition of the Transformation Operation interface is already defined in the means of using them in a `GUI`. It defines methods which return meta-information about the Transformation Operation, like the name and a description. In that way, an Transformation Operation provides dynamically all information about itself. This means that the `GUI` do not need to be updated when the Transformation Operation set changes.

As an additional feature, a `GUI` would enable an easy way to validate Transformation Descriptions: It would be possible to create a virtual instance of the source model, perform the Transformation Description against it and check if the results are according to the expectations. For now, this is done in the tests of the Transformation Environment, but this approach is not sufficient for a high percentage of users.

9.4.2 Bidirectional Transformation Descriptions

The Transformation Descriptions presented in this thesis are unidirectional, which means they work only in one direction, namely from the defined source model to the defined target model (see section 6.3).

It would be a nice feature of the transformation solution, if this unidirectional style can be replaced with a bidirectional style. In this way, the often repetitive Transformation Descriptions between two models could be unified in one Transformation Description. However, to accomplish this it need to be possible provide a reverse function for every operation, which is not possible at the prototype status when this thesis has been written.

9.4.3 Using Smarter Graph Search Algorithm

The presented solution uses a modified depth first search algorithm to find a fitting *Transformation Path* for a transformation request in the transformation graph. This algorithm is an uninformed path search solution, which means that there is no knowledge about the graph structure collected before the algorithm execution.

These approaches are normally slower, since they need to guess which path could be the correct one, by adding one node after another to the path search result until a path was found. Also, the algorithm used in the presented solution does not guarantee that the path search result is the shortest possible. In addition to that, it would be better if a Transformation Description has a cost value. In that way it is possible to prefer some Transformation Descriptions over other. For example the amount of *Transformation Steps* could be taken as a cost function.

There are already some solutions that tried to deal with smart path searching in a graph. However, there was no existing solution found during the implementation of the prototype which also can handle the special requirements this path search algorithm needs (see section 6.4).

APPENDIX A

Model Definitions

Definition of Model 'Issue'

```
package org.openengsb.domain.issue;

@Provide(context={Constants.DELEGATION_CONTEXT_MODELS})
@Model
public class Issue {
  @OpenEngSBModelId
  private String id;
  private String summary;
  private String description;
  private String owner;
  private String reporter;
  private Priority priority;
  private Status status;
  private Type type;
  private String dueVersion;
  private List<String> component;

  // list of getters and setters
}

public enum Priority {
  IMMEDIATE, URGENT, HIGH, NORMAL, LOW, NONE
}

public enum Status {
  NEW, ASSIGNED, CLOSED, UNASSIGNED
}

public enum Type {
  BUG, TASK, NEW_FEATURE, IMPROVEMENT
```

```
}
```

Listing A.1: Definition of model 'Issue'

Definition of Model 'ExternalIssue'

```
package org.openengsb.presentation.model;

@Provide(context={Constants.DELEGATION_CONTEXT_MODELS})
@Model
public class ExternalIssue {
  @OpenEngSBModelId
  private String internal_id;
  private String title;
  private String summary;
  private String assignee;
  private String reporter;
  private IssueMetaInfo metaInfo;
  private String due;
  private String involved;

  // list of getters and setters
}

public class IssueMetaInfo {
  private String priority;
  private String status;
  private String type;

  // list of getters and setters
}
```

Listing A.2: Definition of model 'ExternalIssue'

APPENDIX B

Transformation Descriptions

'Issue' → 'ExternalIssue'

```
<transformations>
  <transformation id="NormalIssueToExternalIssue"
  source="org.openengsb.domain.issue.Issue;3.0.0.SNAPSHOT"
  target="org.openengsb.presentation.model.ExternalIssue;3.0.0.SNAPSHOT">
    <prefixChange>
      <source-field>id</source-field>
      <target-field>internal_id</target-field>
      <params>
        <param key="oldPrefix" value="issue"/>
        <param key="newPrefix" value="externalIssue"/>
      </params>
    </prefixChange>
    <forward>
      <source-field>summary</source-field>
      <target-field>title</target-field>
    </forward>
    <forward>
      <source-field>description</source-field>
      <target-field>summary</target-field>
    </forward>
    <forward>
      <source-field>reporter</source-field>
      <target-field>reporter</target-field>
    </forward>
    <forward>
      <source-field>owner</source-field>
      <target-field>assignee</target-field>
    </forward>
    <forward>
      <source-field>dueVersion</source-field>
      <target-field>due</target-field>
```

```
    </forward>
    <instantiate>
      <target-field>metaInfo</target-field>
      <params>
        <param key="targetType" value=
          "org.openengsb.presentation.model.IssueMetaInfo;3.0.0.SNAPSHOT" />
      </params>
    </instantiate>
    <toString>
      <source-field>priority</source-field>
      <target-field>metaInfo.priority</target-field>
    </toString>
    <toString>
      <source-field>status</source-field>
      <target-field>metaInfo.status</target-field>
    </toString>
    <toString>
      <source-field>type</source-field>
      <target-field>metaInfo.type</target-field>
    </toString>
    <flatList>
      <source-field>components</source-field>
      <target-field>involved</target-field>
      <params>
        <param key="concatString" value=", " />
      </params>
    </flatList>
  </transformation>
</transformations>
```

Listing B.1: Transformation Description between model 'Issue' and 'ExternalIssue'

'ExternalIssue' → 'Issue'

```
<transformations>
  <transformation id="ExternalIssueToNormalIssue"
  source="org.openengsb.presentation.model.ExternalIssue;3.0.0.SNAPSHOT"
  target="org.openengsb.domain.issue.Issue;3.0.0.SNAPSHOT">
    <prefixChange>
      <source-field>internal_id</source-field>
      <target-field>id</target-field>
      <params>
        <param key="oldPrefix" value="externalIssue"/>
        <param key="newPrefix" value="issue"/>
      </params>
    </prefixChange>
    <forward>
      <source-field>title</source-field>
      <target-field>summary</target-field>
    </forward>
    <forward>
      <source-field>summary</source-field>
      <target-field>description</target-field>
```

```
    </forward>
    <forward>
      <source-field>reporter</source-field>
      <target-field>reporter</target-field>
    </forward>
    <forward>
      <source-field>assignee</source-field>
      <target-field>owner</target-field>
    </forward>
    <forward>
      <source-field>due</source-field>
      <target-field>dueVersion</target-field>
    </forward>
    <instantiate>
      <source-field>metaInfo.priority</source-field>
      <target-field>priority</target-field>
      <params>
        <param key="targetType" value=
          "org.openengsb.domain.issue.Priority;3.0.0.SNAPSHOT" />
        <param key="targetTypeInit" value="valueOf" />
      </params>
    </instantiate>
    <instantiate>
      <source-field>metaInfo.status</source-field>
      <target-field>status</target-field>
      <params>
        <param key="targetType" value=
          "org.openengsb.domain.issue.Status;3.0.0.SNAPSHOT" />
        <param key="targetTypeInit" value="valueOf" />
      </params>
    </instantiate>
    <instantiate>
      <source-field>metaInfo.type</source-field>
      <target-field>type</target-field>
      <params>
        <param key="targetType" value=
          "org.openengsb.domain.issue.Type;3.0.0.SNAPSHOT" />
        <param key="targetTypeInit" value="valueOf" />
      </params>
    </instantiate>
    <createList>
      <source-field>involved</source-field>
      <target-field>components</target-field>
      <params>
        <param key="splitString" value=", " />
      </params>
    </createList>
  </transformation>
</transformations>
```

Listing B.2: Transformation Description between model 'ExternalIssue' and 'Issue'

Bibliography

[Aldazabal et al., 2008] Aldazabal, A., Baily, T., Nanclares, F., Sadovykh, A., Hein, C., and Ritter, T. (2008). Automated Model Driven Development Processes. In *Proceedings of the ECMDA workshop on Model Driven Tool and Process Integration*.

[Biffl and Schatten, 2009] Biffl, S. and Schatten, A. (2009). A Platform for Service-Oriented Integration of Software Engineering Environments. In *Proceedings of the 2009 conference on New Trends in Software Methodologies, Tools and Techniques*, pages 75–92, Amsterdam, The Netherlands, The Netherlands. IOS Press.

[Breest, 2006] Breest, M. (2006). An Introduction to the Enterprise Service Bus. Online: `http://www.cin.ufpe.br/~in1062/intranet/bibliografia/fose-the_enterprise_service_bus.pdf`. Last access: 2012-11-22.

[Chappell, 2004] Chappell, D. A. (2004). *Enterprise Service Bus*. O'Reilly, Beijing.

[Chauvel et al., 2007] Chauvel, F., Drey, Z., and Fleurey, F. (2007). Kermeta Language Overview. Online: `http://www.kermeta.org/docs/KerMeta-MetaModel.pdf`. Accessed: 2012-11-20.

[Czarnecki and Helsen, 2006] Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645.

[Dijkstra, 1959] Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271.

[Erl, 2005] Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Hein et al., 2009] Hein, C., Ritter, T., and Wagner, M. (2009). Model-Driven Tool Integration with ModelBus. In *Workshop Future Trends of Model-Driven Development*.

[Huber, 2008] Huber, P. (2008). The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches. Master's thesis, Vienna University of Technology.

[Jouault et al., 2006] Jouault, F., Bézivin, J., and Team, A. (2006). Km3: a dsl for metamodel specification. In *In proc. of 8th FMOODS, LNCS 4037*, pages 171–185. Springer.

[Jouault and Kurtev, 2006] Jouault, F. and Kurtev, I. (2006). On the architectural alignment of ATL and QVT. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1188–1195, New York, NY, USA. ACM.

[Kasanen et al., 1993] Kasanen, E., Lukka, K., and Siitonen, A. (1993). The Constructive Approach in Management Accounting Research. *Journal of Management Accounting Research*, 5:241–264.

[Mayerhuber, 2011] Mayerhuber, F. (2011). Versioning of tool data in service orientated architectures. Bachelor thesis, Vienna University of Technology.

[Mens and Van Gorp, 2006] Mens, T. and Van Gorp, P. (2006). A Taxonomy of Model Transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142.

[Modelbus Team, 2011] Modelbus Team (2011). `http://www.modelbus.org/modelbus/`. Accessed: 2012-11-20.

[Moser, 2010] Moser, T. (2010). *Semantic Engineering Environment Integration Using an Engineering Knowledge Base*. PhD thesis, Vienna University of Technology.

[Moser and Biffl, 2010] Moser, T. and Biffl, S. (2010). Semantic tool interoperability for engineering manufacturing systems. In *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pages 1 –8.

[Muller et al., 2005a] Muller, P., Fleurey, F., and Jézéquel, J. (2005a). Weaving executability into object-oriented meta-languages. In *in: International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 3713 (2005*, pages 264–278. Springer.

[Muller et al., 2005b] Muller, P., Fleurey, F., Vojtisek, D., Drey, Z., Pollet, D., Fondement, F., and Jézéquel, J. (2005b). On executable meta-languages applied to model transformations. In *In Model Transformations In Practice Workshop (MTIP*.

[Noy, 2004] Noy, N. F. (2004). Semantic integration: a survey of ontology-based approaches. *SIGMOD Rec.*, 33(4):65–70.

[Oldevik et al., 2005] Oldevik, J., Neple, T., Grønmo, R., Aagedal, J., and Berre, A.-J. (2005). Toward Standardised Model to Text Transformations. In Hartman, A. and Kreische, D., editors, *Model Driven Architecture – Foundations and Applications*, volume 3748 of *Lecture Notes in Computer Science*, pages 239–253. Springer Berlin Heidelberg.

[OMG, 2008] OMG (2008). `http://www.omg.org/spec/MOFM2T/1.0/`. Accessed: 2012-11-16.

[OMG, 2011a] OMG (2011a). `http://www.omg.org/spec/QVT/`. Accessed: 2012-11-18.

[OMG, 2011b] OMG (2011b). `http://www.omg.org/spec/XMI/`. Accessed: 2012-11-16.

[OSLC Team, 2011] OSLC Team (2011). http://http://open-services.net/. Accessed: 2013-02-08.

[Papazoglou and Heuvel, 2007] Papazoglou, M. P. and Heuvel, W.-J. (2007). Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415.

[Pieber, 2010] Pieber, A. (2010). Flexible Engineering Environment Integration for (Software+) Development Teams. Master's thesis, Vienna University of Technology.

[Reason, 1990] Reason, J. (1990). *Human Error*. Cambridge University Press.

[Sindhgatta and Sengupta, 2009] Sindhgatta, R. and Sengupta, B. (2009). An extensible framework for tracing model evolution in SOA solution design. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 647–658, New York, NY, USA. ACM.

[Smooks developers, 2011] Smooks developers (2011). http://www.smooks.org. Accessed: 2012-11-19.

[Stevens, 2009] Stevens, P. (2009). A Simple Game-Theoretic Approach to Checkonly QVT Relations. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, ICMT '09, pages 165–180, Berlin, Heidelberg. Springer-Verlag.

[Tarjan, 1972] Tarjan, R. (1972). Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160.

[The Eclipse Foundation, 2006] The Eclipse Foundation (2006). http://www.eclipse.org/acceleo/. Accessed: 2013-02-08.

[The Eclipse Foundation, 2012a] The Eclipse Foundation (2012a). http://wiki.eclipse.org/ATL/Concepts. Accessed: 2013-03-01.

[The Eclipse Foundation, 2012b] The Eclipse Foundation (2012b). http://www.eclipse.org/atl/documentation/. Accessed: 2012-11-16.

[The OSGi Alliance, 2011] The OSGi Alliance (2011). OSGi Service Platform Core Specification Version 4.3. Online: http://www.osgi.org/download/r4v43/osgi.core-4.3.0.pdf.

[The OSGi Alliance, 2012] The OSGi Alliance (2012). http://www.osgi.org/Specifications/HomePage. Accessed: 2012-11-21.

[Triskell Team, 2007] Triskell Team (2007). http://www.kermeta.org/documents/. Accessed: 2012-11-20.

[Uschold and Gruninger, 2004] Uschold, M. and Gruninger, M. (2004). Ontologies and semantics for seamless connectivity. *SIGMOD Rec.*, 33(4):58–64.

[Waltersdorfer, 2010] Waltersdorfer, F. (2010). Data Integration in Software-Intensive Systems Engineering- The Engineering Data Base Concept and Applications. Bachelor thesis, Vienna University of Technology.

[Waltersdorfer et al., 2010] Waltersdorfer, F., Moser, T., Zoitl, A., and Biffl, S. (2010). Version management and conflict detection across heterogeneous engineering data models. In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pages 928 –935.

[Winkler et al., 2011] Winkler, D., Moser, T., Mordinyi, R., Sunindyo, W. D., and Biffl, S. (2011). Engineering object change management process observation in distributed automation systems projects. In *Proceedings of 18th European System & Software Process Improvement and Innovation (EuroSPI 2011)*, pages 1–12. Vortrag: 18th European System & Software Process Improvement and Innovation (EuroSPI 2011), Roskilde University, Denmark; 2011-06-27 – 2011-06-29.

[Xiong et al., 2007] Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., and Mei, H. (2007). Towards automatic model synchronization from model transformations. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 164–173, New York, NY, USA. ACM.

[Zhibin et al., 2006] Zhibin, M., Gang, W., Bingyin, R., and Jingfeng, W. (2006). Research on OWL-based process model transformation for interoperability. *IET Conference Publications*, 2006(CP524):1546–1552.

Glossary

Common Concept It is a model which provides a common view of the data used by two or more components involved in a communication. Every component has its own view (represented as a model) of this common concept, which filters and transforms the set of information described in the Common Concept Model to a smaller set of information to work efficiently with it.

Dynamo Transformation Language (DTL) It is the name of the transformation definition language introduced in this thesis.

EKB Graph DB It is one of the core components of the presented transformation solution. Its main responsibility is the creation of the transformation graph and the generation of transformation paths for transformation requests.

Engineering Object It is the realisation of a Common Concept as a model. In this thesis, Engineering Objects are used to give a solution approach for the challenge of automated model update propagation.

Model It is a structural representation of concrete data (e.g. Java classes, class diagrams or ontologies). A model in the context of this thesis is a Java class structure that contains one or more Java classes, where one of these is the so-called 'master' class. This 'master' class enables navigation to the rest of classes that the model contains and therefore, it is the entry point for transformation processes.

Model Registry It is one of the core components of the presented transformation solution. Its main responsibility is the awareness of all models in the currently running environment.

Transformation Description It represents the definition of a transformation. It contains information like the source model, the target model and the operations which need to be performed.

Transformation Engine It is one of the core components of the presented transformation solution. Its main responsibility is the maintenance of transformation descriptions and the execution of transformations.

Transformation Environment The three main components of the presented transformation solution (Model Registry, Transformation Engine and the EKB Graph DB) including their functionality as an overall concept is called Transformation Environment.

Transformation Graph It is the graph which is built inside the EKB Graph DB component. In this graph, models are represented as nodes and Transformation Descriptions are represented as edges. It is used to find Transformation Paths for Transformation Requests..

Transformation Operation It represents an operation that should be performed in a Transformation Step. Such an operation could be for example a simple forward mapping, a string operation or a custom operation.

Transformation Path It is a series of Transformation Descriptions which need to be performed in order to transform a model instance from its source model to a target model.

Transformation Performer It is a sub component of the Transformation Engine. This component is executing the actual transformations (loading of Transformation Operations, executing of Transformation Steps and Temporary Field support).

Transformation Request It is a request for the Transformation Environment to execute a transformation.

Transformation Step It is the lowest level operation in the Transformation Environment and describes an instance of a Transformation Operation. It defines which properties are the source elements and which property is the target element of the operation.

Acronyms

DTL Dynamo Transformation Language.

EDB Engineering Database.

EKB Engineering Knowledge Base.

ESB Enterprise Service Bus.

GUI Graphical User Interface.

M2M Model to Model.

M2T Model to Text.

MDE Model Driven Engineering.

MOM Message Oriented Middleware.

NoSQL Not only SQL.

OMG Object Management Group.

OpenEngSB Open Engineering Service Bus.

OSGi Open Services Gateway initiative framework.

OSLC Open Services for Lifecycle Collaboration.

SOA Service Oriented Architecture.

TDL Transformation Definition Language.

UML Unified Modeling Language.

UUID Universally Unique Identifier.

XML Extensible Markup Language.