FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

SERAPIS 2 Ecore

Bridging Two Modeling Spaces in Eclipse

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Gerald Margreiter
Matrikelnummer 0728495

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Mitwirkung: Univ.-Ass. Mag. Dr. Manuel Wimmer
Univ.-Ass. Dipl.-Ing. Dr. Philip Langer

Wien, 01.07.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

SERAPIS 2 Ecore

Bridging Two Modeling Spaces in Eclipse

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Gerald Margreiter
Registration Number 0728495

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel
Assistance: Univ.-Ass. Mag. Dr. Manuel Wimmer
Univ.-Ass. Dipl.-Ing. Dr. Philip Langer

Vienna, 01.07.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Gerald Margreiter
Mariahilferstraße 156-158, 1150 Wien

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)

Acknowledgements

First of all, I would like to express my very great appreciation to my advisor Professor Gerti Kappel for her supervision. I am particularly grateful for the assistance given by my co-advisors Dr. Manuel Wimmer and Dr. Philip Langer, especially for the valuable technical discussions and the constant support.

Also I would like to acknowledge the help provided by Josef Weissinger and the Federal Ministry of the Interior allowing to study the SERAPIS modeling tool in a production environment. Furthermore, my special thanks are extended to Sphinx IT Consulting, especially Matthias Kammermann, for granting access to the internals of SERAPIS.

Abstract

As the approach of Model-Driven Engineering (MDE) is becoming mainstream in modern software development practices, there is a growing variety of tools to support the lifecycle of modeling artifacts. Standards such as Meta-Object Facility (MOF) defined by the Object Management Group (OMG) help to avoid information loss when trying to integrate multiple modeling tools across their technical boundaries.

The Eclipse-based modeling tool SERAPIS by Sphinx IT Consulting defines a proprietary meta-language which is not compliant to MOF or any other modeling standard. As a consequence, metamodels specified in this meta-language and therefore also the instantiations of these metamodels cannot be interchanged with existing tools based on standards such as MOF which results in a vendor-lock for customers.

The contribution of this master thesis is to develop a transformation approach allowing to translate metamodels and models from the SERAPIS technical space to the Eclipse Modeling Framework (EMF), which employs the meta-language Ecore as the de facto standard corresponding to MOF. The strategy to achieve this is based on an approach presented in the Ph.D. thesis by Dr. Manuel Wimmer who suggests a semi-automatic transformation for metamodels by mapping the corresponding meta-languages. Moreover, we specialize this approach to also enable the automatic transformation of SERAPIS models based on the mappings of their metamodels. The transformation approach developed in this work is implemented in the Eclipse IDE in order to prove its feasibility and to evaluate the generated results.

Kurzfassung

Da der Ansatz von Model-Driven Engineering (MDE) im Bereich der Softwareentwicklung immer breitere Anwendung findet, nimmt auch die Bandbreite der verfügbaren Werkzeuge zur Verwaltung der Modelle immer weiter zu. Standards wie Meta Object Facility (MOF) definiert durch die Object Management Group (OMG) zielen darauf ab, Informationsverluste beim Überbrücken technischer Grenzen im Rahmen unterschiedlicher Szenarien der Werkzeugintegration zu vermeiden.

Das auf Eclipse basierende Modellierungswerkzeug SERAPIS der Firma Sphinx IT Consulting verwendet eine proprietäre Metasprache, die weder zu MOF noch zu anderen Standards kompatibel ist. Folglich können weder Metamodelle, welche in dieser Metasprache spezifiziert werden, noch entsprechende Modelle mit anderen Modellierungswerkzeugen integriert werden, was den Anwender solcher Modellierungswerkzeuge an den Hersteller der Werkzeuge bindet.

Der Beitrag dieser Masterarbeit bezieht sich auf die Untersuchung eines Transformationsansatzes, der es erlaubt, Metamodelle und Modelle aus SERAPIS in korrespondierende Modelle des Eclipse Modeling Framework (EMF) zu transformieren, welches mit dem de facto Standard Ecore eine MOF-konforme Metasprache einsetzt. Der Transformationsansatz wird dabei an die Doktorarbeit von Dr. Manuel Wimmer angelehnt, der die halbautomatische Transformation von Metamodellen anhand der Korrespondenzen ihrer definierenden Metasprachen beschreibt. Nach demselben Prinzip wird auch eine Komponente entwickelt, die das Übersetzen von SERAPIS Modellen ermöglicht. Der in dieser Arbeit untersuchte Transformationsansatz wurde implementiert und in die Eclipse IDE integriert, um einerseits die Machbarkeit zu zeigen und um andererseits die Resultate der Transformation zu evaluieren.

Contents

CHAPTER 1

Introduction

1.1 Motivation

As the requirements to modern software systems increase in quantity and complexity, the need to adopt new approaches in software development emerges. Model-Driven Engineering (MDE) is expected to increase productivity of the development process, improve the overall quality of software and reduce both development time and costs.

Whether these claims are completely justified is not to be discussed in the scope of this work, but as a matter of fact the model-driven approach introduces a couple of helpful aspects. For once a model provides abstraction to the system under study and thereby improves communication between the stakeholders of a software project and helps to avoid misunderstandings in the assessment of requirements. A model is also comparably cheap to build and allows to take an early glance at the final result of a project. Therefore, design flaws can be detected at an early stage of the development process where the correction of defects is still inexpensive. As a model evolves over time, it also represents the current state of a software project and thereby serves documentation and planning purposes [21].

Once a model is created, the need for manipulation, refinement, and transformation emerges from the design process. Various development tools provide different strengths and weaknesses when it comes to these tasks which often makes it necessary to apply multiple tools in the development process. As a consequence, avoiding information loss in order to support seamless tool integration becomes a crucial objective in a model-driven software project [8].

On the sector of model-driven development a variety of tools is available for supporting the modeling lifecycle. These tools often introduce their own metamodels and techniques for model representation which is why the interchange of modeling data is complicated or not feasible at all. In order to overcome the lack of interoperability, methods to bridge divergent metamodels and technical spaces have to be applied. In the context of MDE, tool integration can be implemented by model transformation techniques [7].

The Object Management Group (OMG) [27] is a consortium with the goal to specify standards in the domain of information technology and software development. Next to others, stan-

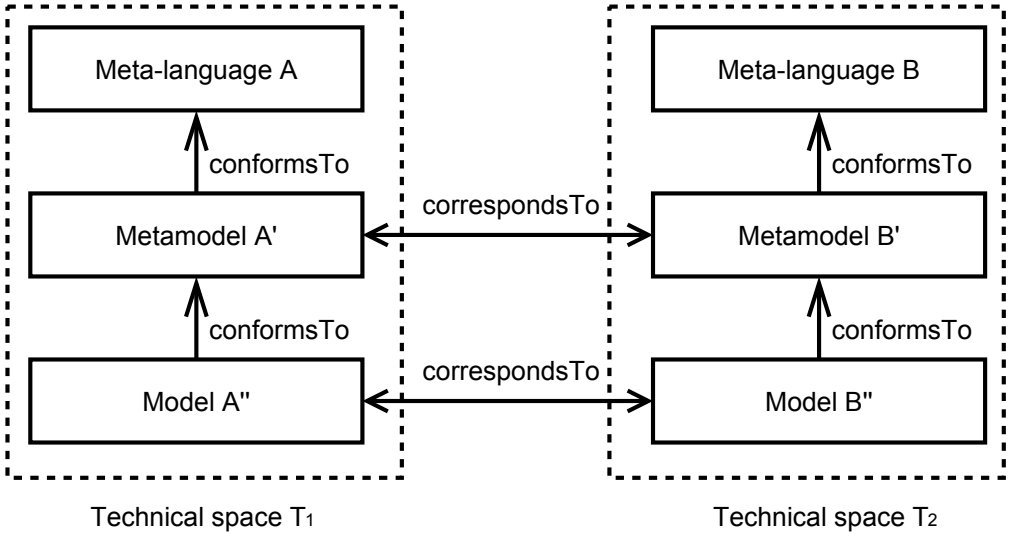

Figure 1.1: Outline of the transformation problem

dards such as Model-Driven Architecture (MDA) [22], Unified Modeling Language (UML) [24], Meta-Object Facility (MOF) [23], or XML Metadata Interchange (XMI) [26] aim to unify the model-driven development process and help to support tool integration.

As a framework based on the software development environment Eclipse [2], the Eclipse Modeling Framework (EMF) aims to support the model-driven engineering process. The framework adopts an XMI-based persistence strategy and, conforming to the definitions specified in MDA, it adheres to an architecture which comprises three meta-layers [30]. EMF employs the meta-language Ecore which is supported by a wide range of modeling tools and therefore can be considered as a de-facto standard in MDE.

The Austrian company Sphinx IT Consulting [5] provides a modeling tool called SERAPIS [11] which is based on Eclipse like EMF and also introduces three meta-layers in its architecture. Despite the similarities to EMF, SERAPIS does not utilize Ecore [30] and introduces a proprietary meta-language that is not compliant to existing standards instead. As a consequence, SERAPIS-based models cannot be integrated with other development tools which implies a vendor-lock.

The scope of this master thesis is to research and evaluate an approach to bridge the technical boundaries between SERAPIS and EMF. Therefore, a method has to be determined that allows to transform modeling artifacts such as the metamodel and the models from SERAPIS to EMF-compliant counterparts. The decision for Ecore as the target meta-language is primarily based on the fact that it is commonly supported by numerous tools, allowing a successful transformation to bypass the vendor-lock.

Figure 1.1 depicts the overall transformation problem and shows that source and target models reside in different technical spaces. Designing appropriate transformation mechanisms usually involves implementing adapters that allow to access the modeling data in these technical spaces. Transformation tools or frameworks, which might be used to implement these transfor-

mations, are therefore required to provide a high degree of flexibility which is why not all of them are suitable for this task.

The *conformsTo*-relation between model and metamodel depicted in Figure 1.1 states that the model is defined by the metamodel and underlines that the existence of a valid model presumes that it conforms to an adequate metamodel which resides within the same technical space as the model. As a consequence, it is not sufficient to transfer only the model information to a different technical space without taking the metamodel into consideration.

With respect to the implementation of a transformation process allowing to bridge a model from one technical space to another, in a first step it becomes necessary to bridge also the metamodel which the model conforms to. Transforming the metamodel implies to build a new metamodel within the target technical space which corresponds to the source metamodel. This step might entail a substantial amount of workload as the definition of the metamodel might become very extensive.

As depicted in Figure 1.1 the relation between metamodel and meta-language equals to the relation between model and metamodel which implies that the metamodel is defined by the meta-language the same way as the model is defined by the metamodel. Consequently, the target meta-language also plays an important role with respect to the transformation process in the sense that it is required to provide a set of language features which matches the features of the source meta-language. This is essential for it determines whether the target meta-language provides enough expressiveness in order to create a sufficient metamodel within the target technical space.

In his Ph.D. thesis [31], as well as in further scientific research [29], Wimmer describes a similar problem situation as depicted in Figure 1.1 in the context of Ecore and the XML technical space. His transformation approach is based on the fact that technical spaces usually adhere to an architecture with three meta-layers which corresponds to the problem discussed in this work.

Wimmer proposes to examine the source and target meta-languages first in order to detect correspondences. From these correspondences rules and heuristics can be derived to map the conforming source and target metamodels. Once this mapping is established, it can be applied to automatically transform models specified in the source metamodel to models that conform to the target metamodel. The approach of Wimmer still is semi-automatic as it suggests manual refinement after the transformation step.

As the approach by Wimmer [31] proposes to generate the metamodel in a semi-automatic process, it is not necessary to build the metamodel from scratch which would be error-prone and time consuming. Moreover the approach is generic enough to transform different metamodels defined by the same meta-language without having to adapt transformations. The approach is not restricted to any specific transformation tools which simplifies its adaptation to individual requirements. For these reasons the approach proposed by Wimmer is chosen as a model for the work discussed in this master thesis.

1.2 Contribution

The modeling tool SERAPIS provides no compliance to existing standards such as those specified by the OMG or with other widely used frameworks such as EMF and its meta-language

Ecore. The fact that SERAPIS employs its own technical space complicates tool integration which implies a vendor-lock.

In a similar problem area, Wimmer [31] proposes a strategy to bridge technical spaces in order to support tool integration scenarios. The contribution of this master thesis therefore is to evaluate whether this approach can be applied to bridge the SERAPIS technical space to EMF. Consequently, we identify and document possible adaptations of the approach by Wimmer to fit the individual problem requirements stated in this master thesis.

In order to validate the correctness and feasibility of the approach, we provide an implementation consisting of two components which are developed and presented in this master thesis. The first component is a generator that transforms metamodels specified in the SERAPIS meta-language to metamodels that comply to Ecore. The second component is a generator that translates models specified by a SERAPIS metamodel to models which conform to the generated Ecore metamodel.

As the meta-languages of both technical spaces are not compliant to each other, the most important concern that goes along with the implementation is how to map metamodels defined by SERAPIS to Ecore. As this issue is specific to the SERAPIS technical space, it is not covered by the approach of Wimmer and therefore an individual solution has to be found.

As the SERAPIS modeling information is very complex, a complete transformation is not an objective of the implementation. Instead a criteria catalogue is created to determine which data is to be mapped. The implementation is also restricted to the unidirectional transformation of metamodels and models, whereas further disciplines of model-driven development such as code generation and additional refinements are not intended.

1.3 Outline

After an introductory discussion of basic state-of-the-art terms and concepts on the field of model-driven software development, the master thesis starts with the presentation of the SERAPIS modeling tool. The presentation comprises the user interface, modeling concepts, and a detailed discussion of the models employed by SERAPIS. Subsequently, the results of a former tool evaluation are examined in order to identify the issues of the modeling tool.

The next chapter depicts the architecture of the transformation process designed to bridge models from SERAPIS to Ecore. Therefore, the semi-automatic metamodel transformation approach of Wimmer is presented and adapted to the SERAPIS technical space in form of a metamodel generator. In addition, the architecture describes another component referred to as model generator responsible for the transformation of models.

The following two chapters discuss the implementation of both metamodel and model generator in detail. The discussion comprises an evaluation of which modeling information is to be included in the transformation process, challenges with the implementation of the generators, and the solutions enabling a feasible transformation. At the end of each chapter the working mechanics of the generators are demonstrated by examples.

In the last chapter the result of the master thesis is evaluated with respect to the feasibility of the generation process and the quality of the generated modeling artifacts. The master thesis finally presents the conclusion and an outlook to further related topics of research.

CHAPTER 2

State of the art

The objective of this chapter is to take a glance at state-of-the-art concepts of Model-Driven Engineering (MDE) related to the context of this master thesis. Therefore, a brief definition of terms essential for the understanding of this work is provided. Subsequently, the results of an extensive literature study constituting the related work are discussed.

2.1 Metamodeling and Model-Driven Engineering

In general, a model is described as a simplified view of reality. Its purpose is to provide abstraction while still being accurate enough to represent the characteristics and behavior of the modeled system in order to make it understandable and predictive. In software development the approach of MDE takes into account these features, and exploits the fact that analyzing a simplified model is significantly cheaper than analyzing the complex system under study. Therefore, MDE suggests to develop the model initially and to subsequently generate software artifacts such as source code [13].

Metamodel and Meta-metamodel

Similar to the well-known object-oriented concepts of *inheritance* and *instantiation*, the model-driven approach also adheres to relations between models and the system they describe. The relation *representation* maps a model to a real-world object while *conformation* determines the conformity to another model [13].

The term metamodel is specified by the OMG as *a model that defines the language for expressing a model* [23, p. 346]. So the metamodel can be considered on a higher abstraction level providing the language to define a model. As a consequence, the model conforms to and can be validated by the metamodel.

In addition the OMG specifies the term meta-metamodel as *a model that defines the language for expressing a metamodel* [23, p. 346] and further states that *the relationship between a meta-metamodel and a metamodel is analogous to the relationship between a metamodel and*

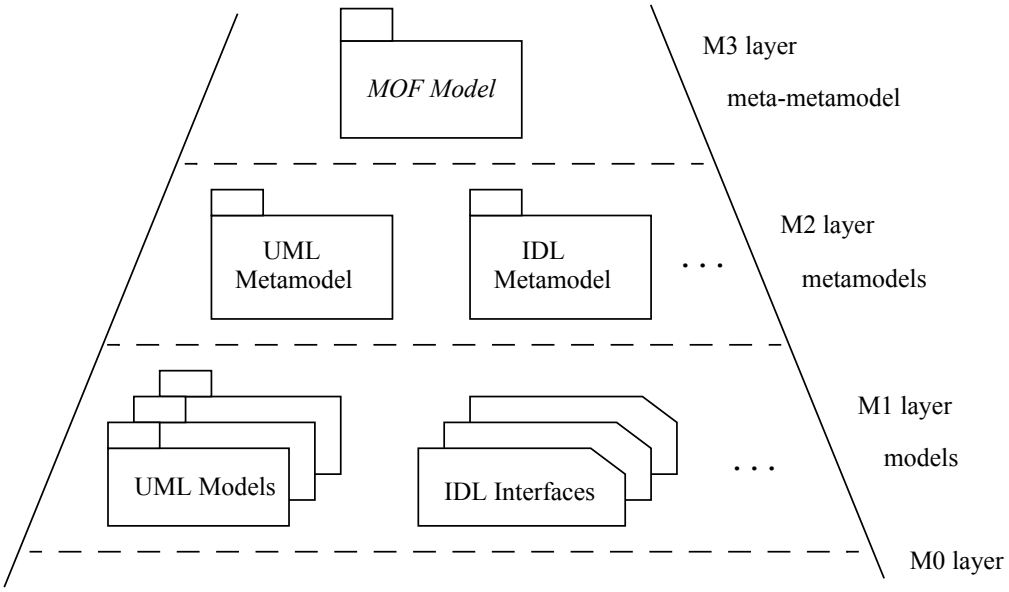

Figure 2.1: Four layer metadata architecture by the OMG [23]

a model [23, p. 346]. Theoretically, there might be an infinite number of abstraction levels but in practice further abstraction levels might not introduce significant benefits. As a reference, the classical metamodeling framework of the OMG outlined in Figure 2.1 is based on the following four layers:

M3 The meta-metamodel is an abstract language for describing metadata which is why in OMG terms it is also called meta-language. The OMGs architecture proposes MOF as hard-wired meta-metamodel.

M2 The metamodel layer is comprised of the metadata which are defined by the structure and semantics specified on the meta-metamodel layer. *A metamodel is an abstract language for describing different kinds of data; that is, a language without a concrete syntax or notation* [23, p. 34]. As depicted in Figure 2.1, besides the UML metamodel, MOF also supports an IDL (Interface Definition Language) metamodel which creates the specification for a CORBA metadata service allowing to map the metadata to objects specified by the CORBA IDL [23, p. 54].

M1 The model layer is comprised of the metadata described on the metamodel layer and is informally aggregated as models.

M0 The information layer contains the actual data to be described.

Model-Driven Architecture

Model-Driven Architecture (MDA) is an architectural framework by the OMG with the objective to support model-driven development. With the model-driven approach, the OMG aims to tackle

interoperability issues resulting from different software platforms. In order to achieve this, MDA suggests to distribute models to different abstraction levels so the development process can be focused on requirements on a higher order without the necessity of taking into account platform-specific details.

The Computational-Independent Model (CIM) presents the highest level of abstraction and corresponds to the domain model in software engineering. The Platform-Independent Model (PIM) contains no platform-specific information so its execution scope can be considered as virtual machine. Only the Platform-Specific Model (PSM) defines the details for the target platform which is why it is used to generate artifacts of software development such as executable code [13].

Meta-Object Facility

An important standard specified by the OMG in the context of MDA is the Meta-Object Facility (MOF). In MDE [13] MOF is defined as *an abstract, self-defined language and framework for specifying, constructing, and managing technologically independent metamodels* [13, p. 134]. The attribute of self-definition states that it is defined by its own language constructs. The fact that MOF is self-defined makes it easier to be extended for future tasks and also shows that it is expressive enough for practical metamodeling [23].

Initially MOF has been developed as an adaptation of the UML core in order to meet the requirements of MDA and can be considered as a small set of concepts used to define a metamodel. With the release of MOF 2.0 the two standards Essential MOF (EMOF) and Complete MOF (CMOF) have been introduced. EMOF is a small subset of MOF being easier to implement while CMOF is more expressive due to its more complex language features which makes it harder to implement [13].

XML Metadata Interchange

The XML Metadata Interchange (XMI) [26] standard is used to map meta-metamodels, metamodels and models to XML documents and schemas. The structure of these documents is straight-forward as it matches closely the hierarchy of the representing models which is probably the reason for its popularity [30]. XMI is supported by a significant number of modeling tools as an interchange format for MDE metadata [13].

Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is a framework for the Eclipse extensible development platform [30] with the purpose of supporting the model-driven software development process. EMF is developed parallel to MDA and shares the same basic idea, but introduces its own meta-language Ecore instead of MOF. As Ecore originates from UML like MOF, it is not surprising that they share common concepts [13].

Figure 2.2 displays an essential subset of the Ecore meta-language. The concepts expressed by this model might be recognized from object-oriented development [30].

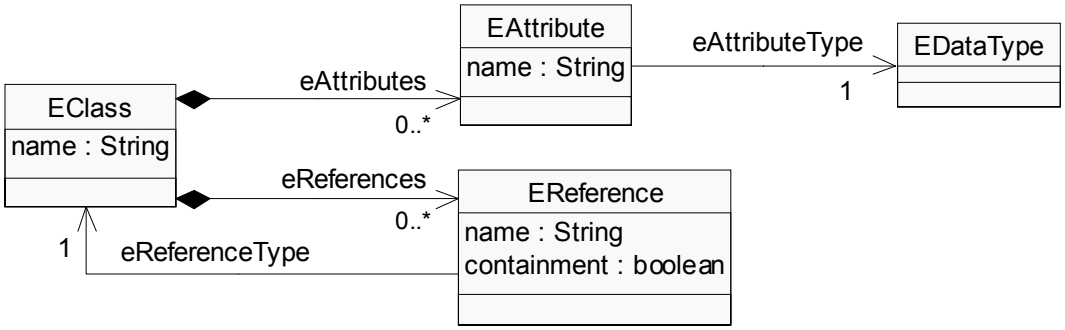

Figure 2.2: A subset of the Ecore meta-language [30]

EClass defines a modeled class and has an unbounded number of attributes an references. It can have multiple supertypes in order to support inheritance.

EAttribute represents the data of an object which is defined by type and name.

EDataType corresponds to a primitive or complex type which is not modeled as an `EClass`.

EReference represents one side of an association between two classes. If the association is bidirectionally navigable there is another reference at the opposite side. References are typed like attributes with the restriction of referring to `EClass` only. As known from UML they provide lower bounds and upper bounds in order to support multiplicities.

EStructuralFeatures

As elements of the Ecore meta-language, `EAttributes` and `EReferences` play an essential role by specifying the very basic definition of complex data types. Both language constructs are defined by name and type, and like `EReference` also `EAttribute` specifies multiplicities. Because of these similarities Ecore aggregates them under a common base called `EStructuralFeature`. The relationship between `EClass` and the `EStructuralFeatures` it defines is bidirectional, so references and attributes know which class they belong to [30].

EPackages and EFactories

Providing a unique URI for identification, `EPackages` are used in Ecore to group classes and data types that are related to each other which complies to the purpose of packages in Java. For serialization of Ecore models, the packages name is used as document root and its XML namespace is specified by the URI. Along with an `EPackage`, EMF provides an `EFactory` implementation in order to allow the instantiation of classes contained in the package [30].

EMF and Modeling Standards

As both EMF and standards of the OMG are subjects of this paper, their relationships are discussed in the following.

MOF Both Ecore and the Meta-Object Facility (MOF) are meta-languages and originate from UML. Therefore, they share some similarities when it comes to defining classes and their features. Both support inheritance, packages, and reflection but they differ in details of data types, relationships, and associations [30].

XMI The XML Metadata Interchange (XMI) standard is the default serialization format of the Eclipse Modeling Framework (EMF) as it provides an easily understandable document structure. Although using XMI every EMF-based model can be serialized, it is most appropriate for serializing metamodels such as Ecore itself in which case the resulting document format is called Ecore XMI and can be recognized by the *.ecore* file extension [30].

MDA EMF and the Model-Driven Architecture (MDA) standard share the key concepts of model-driven code generation and serialization for data interchange. Nevertheless, MDA is an architectural framework designed to work with multiple platforms while EMF is an actual framework based on Java [30].

Technical Spaces

In model-driven software development the design decision for a certain modeling framework also implies the technical space. The term technical space or technological space [19] is defined as *a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities* [19, p. 1]. In other words, a technical space determines which technology is available to represent the models. This includes file formats, data structures, and mechanisms to transform and manipulate models.

The decision for a modeling framework has to be taken very carefully as models are only processable within the technical space a framework provides. As a result, technical boundaries between different spaces have to be bridged if tool integration is a concern.

Different technical spaces such as MDA, EMF, XMI, and therefore XML have already been discussed in this paper. Besides them, also programming languages introduce their own technical spaces as they can be used to define and manipulate models. The main benefit for using programming languages in this context is that models can be actually executed while XML only provides a data structure. On the other hand, programming languages lack the ability to separate content and presentation. When it comes to model transformation, programming languages also provide only insufficient mechanisms to enable traceability which describes the ability to keep references between source and target models. Figure 2.3 compares some technical spaces in the context of a four layer metadata architecture. An in-depth look at transformations with respect to further technical spaces is provided by additional literature [10, 15, 16].

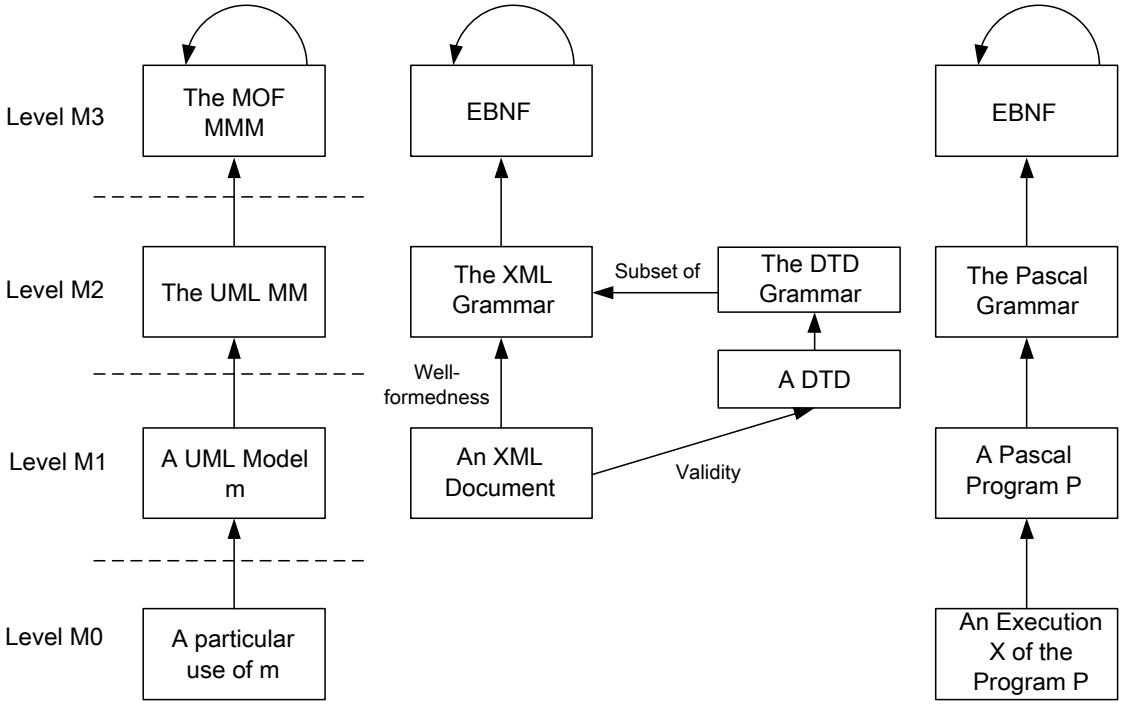

Figure 2.3: The technological spaces MDA, XML, and the programming language Pascal [19]

2.2 Metamodel Derivation and Model Transformation

The following section provides a brief overview of topics which are related to the problem described in the context of this master thesis. The overview starts with a short discussion of two model transformation tools in order to determine their applicability with respect to the implementation of the SERAPIS2Ecore framework proposed in this thesis. As SERAPIS provides a Java API for granting access to its modeling artifacts, a set of techniques for mapping Java to Ecore is presented as the section continues.

Subsequently, the DTD2Ecore framework is presented as it tackles a problem which is very closely related to the context of the SERAPIS to Ecore transformation. Finally recent research efforts with respect to the transformation of models between SERAPIS and Ecore are discussed.

Model Transformation Tools

There is a wide range of tools providing different model transformation mechanisms with individual strengths and limitations. Whether a model is to be synthesized for refinement, multiple source models have to be merged to one single target model, or a model has to be integrated with another tool, the choice of the adequate transformation tool depends on the desired outcome and therefore the problem the transformation process tackles.

In order to support the decision process, the literature study of Biehl [7] establishes a classification scheme summarizing the problems that can be addressed by model transformation. In the scope of this thesis not all of these classifications are relevant and therefore only the following are discussed.

Change of metamodels This problem classification basically describes whether the source metamodel is the same as the target metamodel. In this case these transformations are called *endogenous transformations* or *rephrasing transformations*, and are usually applied to change only specific parts of the processed model. If source and target metamodels do not match, the transformation is called *exogenous transformation* or *translation transformation*.

Supported technical spaces Source and target models may reside in different technical spaces and therefore the knowledge of which ones are supported by a transformation tool can be an important decision criteria.

The literature study [7] provides an overview of current model transformation languages, tools, and standards. As SERAPIS and EMF provide contrary technological implementations only transformation tools that allow exogenous transformation over different technical spaces are discussed in the following.

Henshin is a joint project by various European research institutions, and provides a transformation language and a tool environment based on the Eclipse Modeling Framework (EMF).

The Henshin transformation language conforms to an EMF-based metamodel and applies graph transformation concepts. For this purpose the language provides rules which consist of left-hand side (LHS) and right-hand side (RHS) graphs and their mappings. First order logical formulas over graph conditions can be defined in order to specify where the rules are applied. The transformation language also provides control structures to determine the application order of rules.

To support the transformation development, Henshin provides one tree-based editor generated by EMF and another graphical editor implemented using GMF. It also comes with a runtime component which takes an EMF model as input and directly applies transformations. Besides that, Henshin provides an analyzing tool which simulates the transformation steps on a given model and extension points for additional model checkers [6].

Epsilon Transformation Language The Extensible Platform of Integrated Languages for mOdelmaNagement (Epsilon) [17] is an Eclipse project that includes a set of task-specific languages covering modeling aspects such as transformation, code generation, comparison, merging, and validation. The Epsilon Object Language (EOL) is a general-purpose modeling language and provides a common base for all task-specific languages such as the Epsilon Transformation Language (ETL) which covers the transformation aspect. ETL is a hybrid model transformation language as it provides declarative rule-based mechanisms but also imperative features.

Besides the definition of source and target elements, ETL rules contain a block for EOL statements called *body* to specify the mapping logic. It is also possible to execute external code in order to access models that reside in different technical spaces. To control its application, a rule can specify a guard consisting of EOL expressions that determine whether its body is executed or not. Besides that, rules declared as *greedy* are applied whenever possible while *lazy* rules have to be called explicitly. ETL rules are organized in modules for reuse and support multiple inheritance [17].

According to the insight by this literature study, both Henshin and the Epsilon Transformation Language (ETL) seemingly have the potential to implement the SERAPIS2Ecore framework. But as both frameworks, and probably those that have not been covered by this literature study, would expect an implementation of adequate adapters for SERAPIS in the first place, their inclusion into the development of the SERAPIS2Ecore framework would provide no significant benefit compared to a manual implementation from scratch.

Mapping Java to Ecore

In order to implement the SERAPIS2Ecore framework, the first step is to create a corresponding metamodel for SERAPIS in Ecore. This is a sophisticated task as building the metamodel from scratch is to time-consuming and error prone due to its extent. As SERAPIS provides a Java API to access its metamodels, existing automatic approaches are discussed in this context.

Deriving EMF Models from Java Source Code

The primary aim of the work dedicated to the derivation of EMF models from Java source code [32] is to re-engineer user interfaces (UI) based on Java Swing API without having to change the underlying business logic. The basic idea is to read UI information of existing applications and transform them into EMF-based models. In the next step, tools of EMF can be used to apply modifications to the model and generate Java source code.

To allow model transformation, first an EMF metamodel that corresponds to the Java Swing API has to be established. To achieve this, an automatic approach for deriving the metamodel from the Java source code is described. The overall objective of the approach is simplicity in the sense that the transformation requires no further intermediate steps or additional external tools. For that reason, the article proposes to use the Java parser from Oracle which is utilized by *JavaDoc* in order to generate API documentation files. The parser builds an in-memory graph of the source code and executes *doclets* to inspect the graph and perform further actions.

The problem described in this article is tackled by implementing the transformation rules for the Java-to-Ecore mapping as *doclets*. The parser builds a graph representing the Java Swing API and executes the rules in order to determine the corresponding Ecore mapping. The *doclets* additionally work as generators which instantiate metamodels in the form of Ecore XMI.

Some relevant assumptions and considerations on the mapping of Java to Ecore are listed in the following for providing a general insight to this topic.

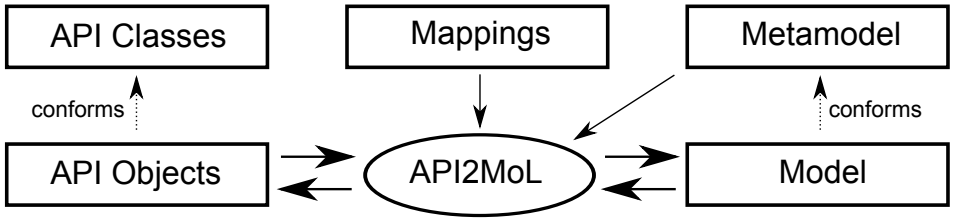

Figure 2.4: Conceptual overview of API2MoL [28]

- Visibility markers such as *private*, *protected*, or *public* do not need to be mapped as every Ecore element has public visibility.
- Both languages contain predefined base types but also allow to introduce complex types which makes the type mapping a sophisticated task.
- It is difficult to determine if an attribute in Java corresponds to an `EReference` or an `EAttribute` in Ecore. An approach to this problem is to map all base types to attributes and all complex types to references.
- For the sake of simplicity, getter and setter methods might not be included in the transformation process as they add no information value to the metamodel.
- Enumerations in Java can be detected easily when declared by the `Enum` language feature but, when implemented as string constants there exists no heuristic for detection.
- Some classes or interfaces referenced by the Java application might not be a direct part of the model. As they have to be represented in the model, in order to grant consistency they can be organized in separate packages.

API2MoL

API to Metamodel Language (API2MoL) [28] is a Java-based framework developed by AtlanMod [1] and Modelum [4] research groups and targets the integration of APIs into model-driven software engineering.

The general objective of API2MoL is to derive a model representation from an existing application based on the mappings between the API this application exposes and the metamodel the models conforms to as displayed in Figure 2.4. For this purpose API2MoL provides its own language to define the mappings for bridging the technical spaces. These mappings are designed to work bidirectional so it is also possible to generate software artifacts from existing models. The current implementation of the framework is restricted to Ecore metamodels and Java-based APIs but an extension to other object-oriented programming languages would be feasible.

In case an API has to be bridged without an appropriate metamodel existing, API2MoL provides a bootstrapping mechanism. Thereby, the framework discovers the API structure using the Java Reflection API and creates both metamodel and mapping definitions [14].

According to the API2MoL documentation [14] correctness, expressiveness, and completeness of the API2MoL framework has been validated with various APIs such as Swing, SWT, and JTwitter. As SERAPIS also provides a Java API in order to grant access to its metamodel, the applicability of the API2MoL framework has been evaluated within the scope of this master thesis. Unfortunately, this attempt has failed as the SERAPIS metamodel cannot be mapped to Ecore directly as described in Chapter 5.

Java2Ecore

Java2Ecore [9] is a one-way translator for Java applications to Ecore models. The idea is to derive Ecore conform models from Java classes by applying hard-wired Java-to-Ecore mapping rules. While classes and attributes in Java can be mapped directly to Ecore other language features have no counterpart which is why Java2Ecore only covers a subset of the Java language. Additional modeling information can be added by annotations which has the downside that the original Java code has to be manipulated in order to get translated.

Unfortunately, the project has been frozen and never exceeded the beta status which is why it did not qualify for further evaluation in the first place [9].

The DTD2Ecore Framework

Within the scope of his Ph.D. thesis [31] Wimmer proposes a set of strategies for different tool integration scenarios in the context of MDE. One of these scenarios describes a situation that requires two separate technical spaces, namely DTD and EMF, to be bridged in order to allow an interchange of models. Before the model transformation can be implemented, a metamodel the corresponds to the source metamodel has to be created in the target technical space. As in this case the metamodel cannot be recreated from scratch due to its extent, the work of Wimmer describes a semi-automatic approach to generate the target metamodel as part of the DTD2Ecore framework.

The approach suggests to use the correspondences of the meta-languages to derive transformation rules which are applied to automatically generate the target metamodel according to the specifications of the source metamodel. Due to lack of expressiveness provided by the DTD language constructs, the mapping of the meta-languages introduces ambiguities which can only be resolved manually. As a consequence, the DTD2Ecore framework employs heuristics to identify these ambiguities in the metamodel and mark the resulting elements for manual refinement.

The problem situation described in the Ph.D. thesis [31] matches the problem presented in the context of this master thesis in the way that both scenarios require to bridge the boundaries of different technical spaces and furthermore an automatic approach to generate the target metamodel is required. Although the approach of Wimmer is presented along the use case of the DTD2Ecore framework, it is to be considered as a more general approach for implementing bridges on the M3 layer which is why it is adopted to the mapping between SERAPIS and Ecore as described in Chapter 4.

SERAPIS and Ecore

According to Sphinx, the idea of bridging the models of the proprietary technical space of SERAPIS to a more common standard such as Ecore has already been discussed and evaluated in the recent years. EMF is very popular and has a strong supporting community which is why a wide variety of built-on tools exist. Gaining access to these tools is the overall benefit expected from the result of the bridging attempt.

The inability to map the SERAPIS metamodel directly to Ecore is considered as the major challenge when it comes to implementing a transformation approach, as this not only requires in-depth knowledge about both meta-languages but furthermore requires to research a mapping technique which exceeds the capabilities of most state-of-the-art transformation approaches. As this task introduces a high degree of uncertainty with respect to the technical feasibility and the amount of workload this effort entails, no actual attempt to design or implement a bridge to Ecore has been made yet.

CHAPTER 3

SERAPIS at a Glance

Sphinx IT Consulting located in Vienna is a medium-sized service contractor in the sector of software development. To aid the process of software development, Sphinx provides a proprietary tool called SERAPIS which follows the concept of model-driven software engineering. In that sense its architecture relies on three separate meta-layers including models, metamodels, and a meta-language.

Unlike other modeling tools and frameworks which simply provide a meta-language and leave the design of the individual metamodel to the customer, SERAPIS uses one core metamodel which is extended to meet the customer requirements. According to the business model of Sphinx, the product is not shipped to the customer in its entire scale, but instead it is tailored to the individual requirements and then deployed.

SERAPIS is based on the Eclipse Rich Client Platform (RCP) which is an open tools platform for supporting the development of desktop applications. The development tool comes with a set of convenience features such as a wizard that aides the creation of new projects, a graphical editor for model manipulation, and an explorer for presenting the project structure.

Besides Java code, SERAPIS can also generate SQL Data Definition Language (DDL) scripts that define the database outline according to the model. If provided with a database connection, it is also possible to have the database created automatically and evolved along with the model. It is to be mentioned that this process can be reversed using the so-called Schema Migrator. This feature analysis the database and derives models from the table definitions. This reverse engineering method can help to save time when migrating existing projects. Although this migration tool is designed to work fully automatically, it sure is advisable to put some effort into further manual refinement as some derived data types and references might still need a little adjustment. Besides that, SERAPIS also creates Entity Relationship (ER) diagrams along with the transformation process to depict the model.

The product is composed by two independent modules that cover different tiers of software development. The first one is focused on the presentation layer and therefore supports the design of graphical user interfaces including forms, dialogues, and controls for both web and rich client platforms. The second module allows to model the persistence layer by defining business ob-

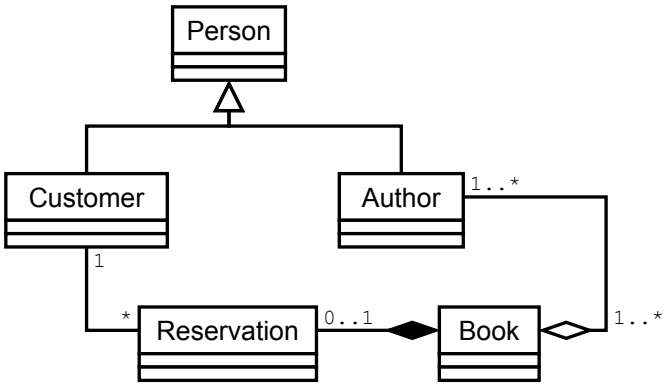

Figure 3.1: *Library* example model

jects, data transfer objects (DTO), and services. The persistence layer is expected to offer more potential for code generation because it usually includes no business logic and, unlike a user interface, it is mandatory for most software projects. For this reasons the scope of this master thesis is restricted to the second module [11].

The outline in Figure 3.1 shows a strongly simplified model of a public library. It serves as an example to illustrate different issues discussed in the scope of this master thesis. In this chapter it is used to present the SERAPIS development environment and to discuss advanced aspects of the modeling tool. Subsequently, the models of SERAPIS including the meta-language are examined in detail as they are the key to the transformation approach according to [31]. The chapter is concluded with the presentation of an evaluation of the SERAPIS modeling tool by the Vienna University of Technology.

3.1 SERAPIS Modeling Tool

As mentioned before, Sphinx tailors the SERAPIS modeling tool to individual requirements before its deployment, rendering the customer unable to autonomously apply further modifications. Nevertheless, SERAPIS still provides mechanisms to configure modeling projects in order to customize the outcome of code generation. As the project configuration presumes a certain amount of know-how, an in-depth look into this topic is to be provided here. Subsequently, the user interface of the development environment is presented.

Project configuration

For building a new modeling project from scratch, SERAPIS provides a custom wizard that creates an empty Eclipse project and assists the customer with the initial setup. Besides that, it is also possible to add the modeling nature to an existing Eclipse project which basically has the same outcome as the wizard to the effect that it creates a XML file for configuration named *.sxmemodel* which looks as follows. The existence of this file is necessary for the development environment to recognize the modeling aspect of the Eclipse project.

```
<sxme:sxmemodel xmlns:sxme="http://www.sphinx.at/sxme/model">
  <model folder="model" />
  <generated folder="src-gen" />
  <generatedjava folder="src-gen/java" />
</sxme:sxmemodel>
```

Listing 3.1: Content of the *.sxmemodel* file

Due to the volatile quality of generated code, isolation from manually written code is ensured in order to avoid information loss. Therefore, the configuration file specifies separate source folders for generated artifacts. In addition, a central directory to hold the model definition is declared leaving the original project unaltered. Besides this basic project configuration, SERAPIS offers more advanced mechanisms to customize the code generation listed as follows.

Storages With respect to the generation of database related scripts, storages can be defined to represent access to existing databases. When provided with valid user credentials, different database types such as IBM DB2, PostgreSQL, and Oracle can be accessed to automatically deploy changes made to the model.

Deployment environments Storages can be grouped to deployment environments with the purpose of testing, development, and production. Also database schemas can be declared here to distinguish different deployment environments.

Build configurations SERAPIS provides generators which iterate the models allowing to create a certain kind of output. For instance, the DDLBuilder is responsible for creating database scripts while others generate Java classes for the persistence layer. These generators, referred to as build steps, are grouped to build configurations so different types of artifacts can be generated at the same time. The scope of a build configuration is always restricted to a certain deployment environment.

Build strategies In order to generate code artifacts, build steps rely to additional parameters which are usually specific to a certain project and therefore defined by the customer. For instance, the DDLBuilder needs to be provided with naming conventions for tables and views. For this purpose build strategies can be considered as a set of value assignments to predefined parameters which are looked up by the build steps.

User Interface

As the SERAPIS modeling tool is based on Eclipse, it provides its own perspective to the customer allowing to access the modeling features. Within this perspective not only the model can be created and manipulated but it is also possible to adjust project configurations.

The model explorer in Figure 3.2 provides a tree view to the project structure of the *library* example. Here the aforementioned deployment- and storage-related configurations can be performed. Besides that, the explorer allows to manage the data types and the packages they are assigned to. These packages comply to the Java packages in which the generated classes are organized.

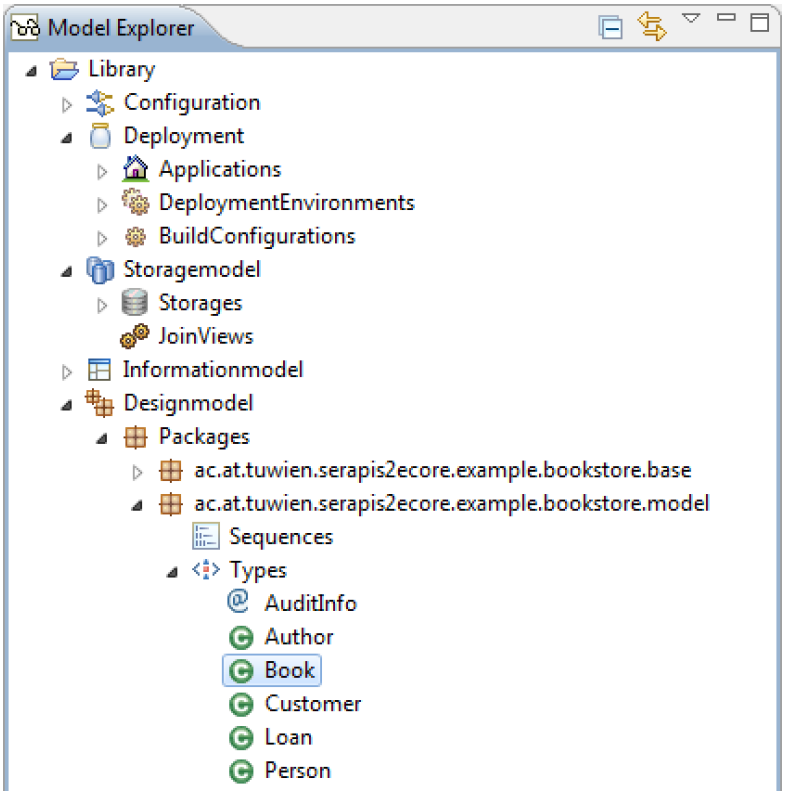

Figure 3.2: *Library* example project structure in the model explorer

The model editor is used to manipulate all data types available within a project. By default, the modeling environment provides a set of base types such as string, date, and integer. Complex types are defined by the user and referred to as entities which can be composed of base types and other complex types. They adhere to general object-oriented concepts such as inheritance or aggregation, but they can also provide more language-specific properties to the effect that they can be declared as serializable or marked as final.

Besides attributes, entities can also declare method signatures. In this case, the generated code is contained in one abstract class which inherits to a concrete class providing the method implementation. The abstract class is volatile, meaning that it is erased and rebuilt with every launch of the generation process and therefore is contained by a source path different to the location of the manually written code.

Figure 3.3 shows the entity `Book` of the *library* example represented in the model editor. The view outlines the attributes declared by this entity stating their name and type. While `Author` is a complex type and therefore an entity itself, `Title` and `Language` refer to the string base type. The digits at the end of the types name indicate the maximum length of the value a field can contain. These constraints are essential for the generation of database scripts while they have no relevance to the generation of Java code.

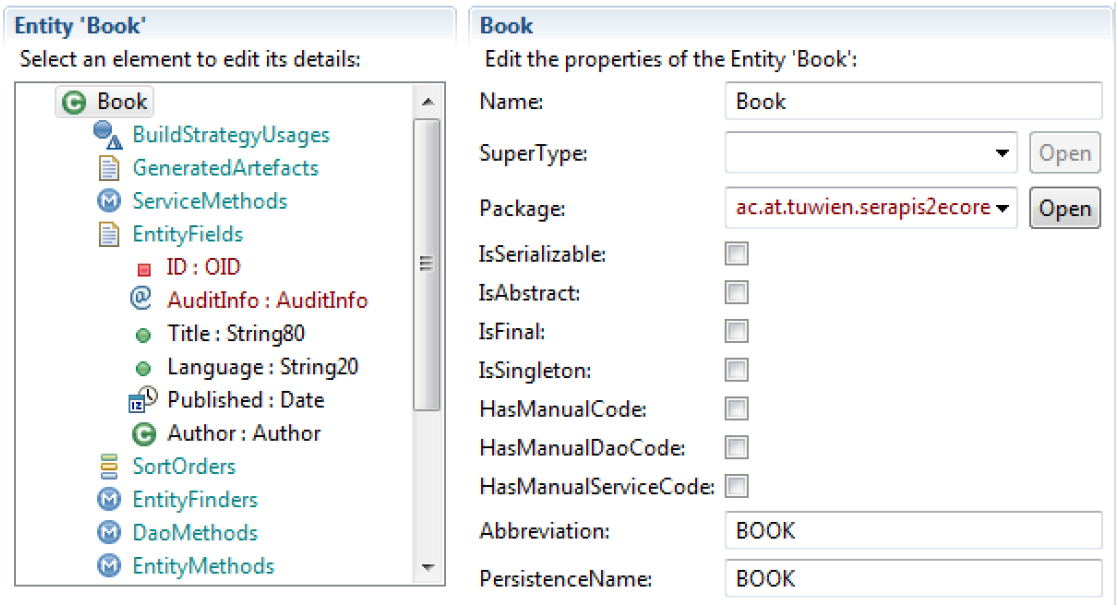

Figure 3.3: Entity `Book` of the *library* example in the model editor

`ID` and `AuditInfo` are technical attributes which are neither declared by the entity nor inherited from a super type. The term technical attribute here refers to fields which are intended to hold control information only, such as the time an entry has been created or which user has performed the latest modification. Both `ID` and `AuditInfo`, which in turn is only a container that holds a collection of technical attributes, are declared globally by a build strategy so they can be managed centrally without having to re-declare them for every entity. Entities can change the behavior of build strategies by overriding them.

3.2 Models in SERAPIS

SERAPIS adheres to a three-layer meta architecture including meta-language, metamodel, and model. In order to evaluate whether the approach of Wimmer [31] can be applied to transform the models, it is necessary to understand these meta layers.

Meta-language

The metamodel used by the SERAPIS modeling environment is based on a proprietary language definition here referred to as the SERAPIS meta-language. Unlike Ecore, which provides a rich set of language features allowing high flexibility with respect to the definition of metamodels, the SERAPIS meta-language is tailored to the needs of the proprietary product.

The essential elements of the SERAPIS meta-language are outlined in 3.4. The core of the language definition is presented by `Element` which can be compared to the `EClass` construct of Ecore. An `Element` is able to contain an unbound set of `Properties` which is a gener-

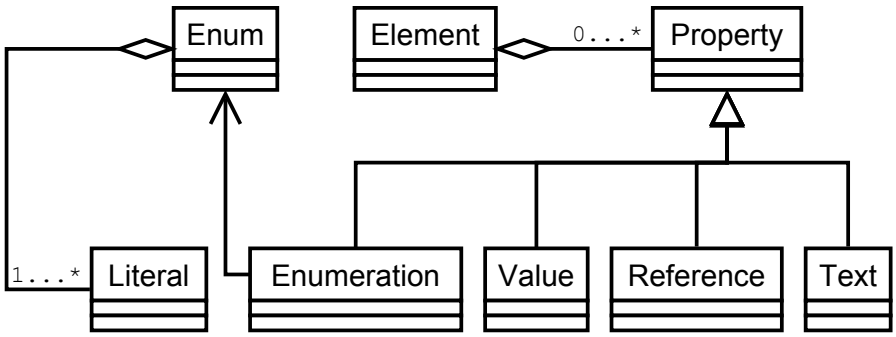

Figure 3.4: SERAPIS meta-language

alization for all field types an `Element` can have, such as `Value`, `Reference`, `Text`, and `Enumeration`. `Values` can be considered as attributes such as the `EAttribute` in Ecore used to characterize and define the state of complex types. `References` on the other hand represent compositions and relationships between complex types similar to `EReference` in Ecore. Unlike `Values` and `References`, which can be matched directly to object-oriented concepts, the `Text` property has to be considered separately for it defines information which serves primarily documentation purposes and therefore has more relevance to the representation inside the modeling tool. For instance, entities inherit `Text` properties for specifying comments and descriptions so the model editor 3.3 provides input boxes allowing to assign values to these documentation properties.

In the SERAPIS meta-language the functionality of enumerations is provided by (1) an element called `Enum` for constructing enumerations by defining `Literals` and (2) a field type named `Enumeration` which is used to reference `Enums`. Originally in the SERAPIS meta-language the construct `Enum` is also named `Enumeration` but for reasons of clarity here the name `Enum` is introduced. However the functionality of enumerations in the SERAPIS meta-language basically complies to the functionality of enumerations known from various programming languages.

The characteristics of all language features provided by the SERAPIS meta-language, consisting of elements such as `Element` and `Enum`, or field types such as `Value`, `Reference`, `Text`, and `Enumeration`, are defined by an individual set of attributes listed in Table 3.1. The attributes shared by the field types are aggregated in the abstract field type `Property`.

Metamodels

The SERAPIS modeling environment is based on a single metamodel which can be considered as the default metamodel. It defines a consistent set of elements in order to cover general aspects of development such as the definition of business objects, data types, and packages. Before the modeling tool is deployed within a production environment, the default metamodel is modified in order to meet specific requirements of the customer. Due to the fact that the metamodel is tailored to meet individual domain-specific requirements, the language it specifies can be considered as a Domain-Specific Modeling Language (DSML).

Name	Attribute	Type	Description
Element	name	String	Name of the `Element`
	super	Element	Declares a superclass
	class	Class	Class representing the `Element` in Java
	isAbstract	Boolean	Declares the `Element` as abstract
	isSingleton	Boolean	Sets the instantiation behavior
	isDisplayOnly	Boolean	Sets the `Element` editable in the modeling tool
	isHidden	Boolean	Visibility setting for the modeling tool
	isPSM	Boolean	Declares the `Element` as platform-specific
	transformer	Class	Java class for transforming the `Element` to its platform-specific representation
	icon	URL	Icon representing the `Element` in the modeling tool
Property	name	String	Name of the `Property`
	isRequired	Boolean	Declares the `Property` as mandatory
	isDisplayOnly	Boolean	Sets the `Property` editable in the modeling tool
	isHidden	Boolean	Visibility setting for the modeling tool
	isPSM	Boolean	Declares the `Property` as platform-specific
	isUserSetting	Boolean	Determines the `Property` as a tool-specific setting
Text	lines	Integer	Number of lines in the content
	editorId	String	The associated editor in the modeling tool
	isLocalized	Boolean	Determines if the content is localized
	isDescription	Boolean	Determines display settings in the modeling tool
Value	class	Class	Class representing the `Value` in Java
	default	String	Default value
	isUpperCase	Boolean	Determines whether string values are upper-case
	hasListOfValues	Boolean	States if the `Value` refers to a list
	unit	String	Unit for measuring the `Value`
Reference	element	Element	The referenced `Element`
	isComposition	Boolean	States if the target `Element` is a composition
	delete	String	Sets the delete strategy
Enumeration	enumeration	Enum	References an `Enum`
Enum	name	String	Name of the `Enum`
	literalList	List<Literal>	Associated `Literals`
Literal	name	String	String representing the `Literal`

Table 3.1: SERAPIS meta-language attributes

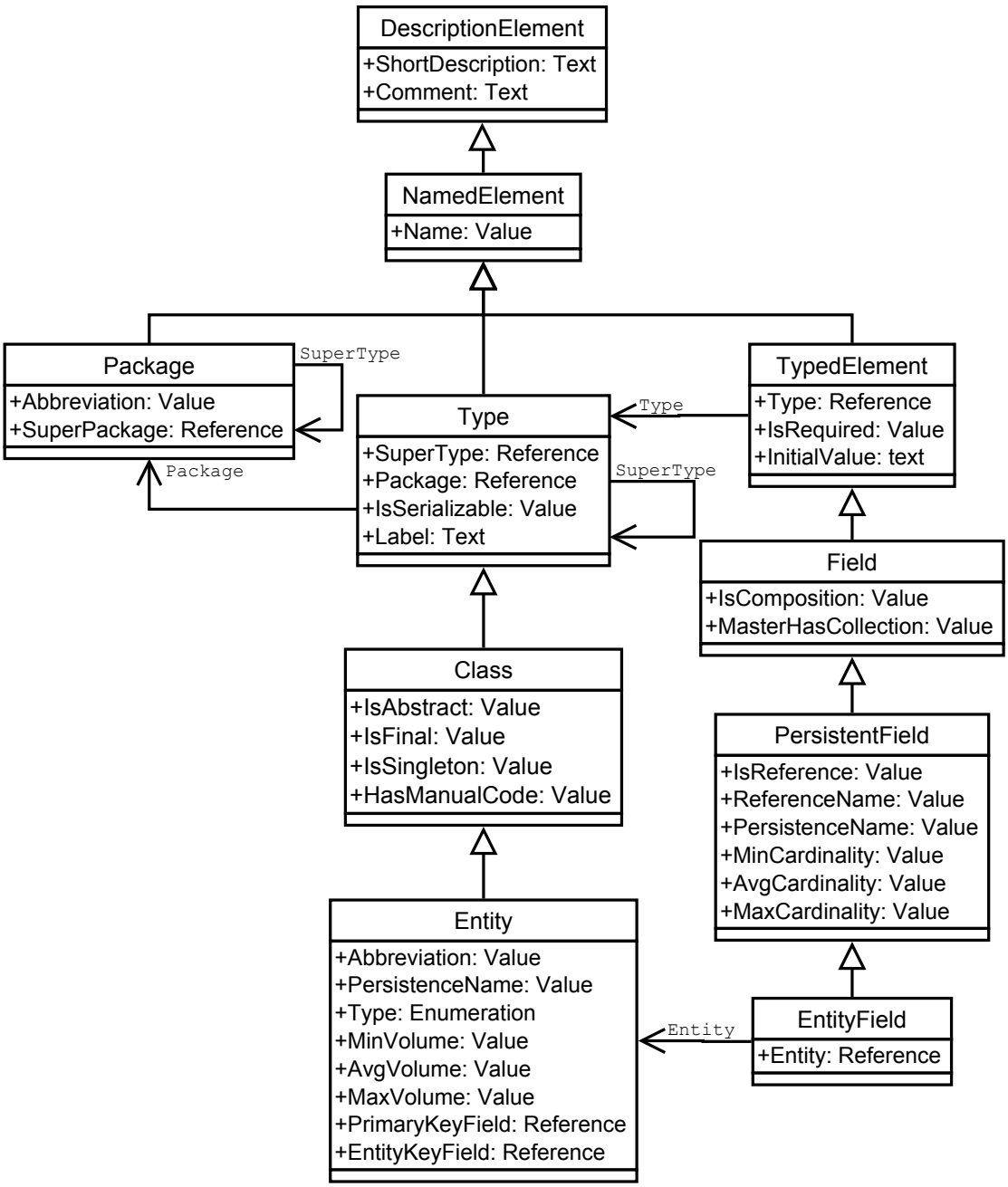

Figure 3.5: SERAPIS metamodel

The modeling features of SERAPIS are implemented and distributed to the customer in form of Eclipse plugins, whereas the elements of the default metamodel and the elements individually tailored to the domain-specific requirements are encapsulated in separate plugins. The elements of each plugin are defined as extensions [3] in the *plugin.xml* file. In Eclipse, extensions are used to specify contracts for providing access to contents between plugins. As a SERAPIS plugin is loaded, the element definitions are de-serialized in order to allow access during runtime. The elements designed to meet the individual requirements are configured to overwrite the elements of the default metamodel in order to provide the domain-specific functionality without having to alter the default metamodel. As an example, the code snippet in Listing 3.2 shows the definition of the element `Entity` and three of the `Properties` it defines. The diagram in Figure 3.5 shows a small overview of the default metamodel element definitions. As the hierarchy contains over 200 elements, only the most essential elements for specifying entities and their relationships are depicted. There is no visual editor for manipulating the metamodel as known from EMF, instead element alterations need to be performed directly in the *plugins.xml* file.

Models

Models created by the SERAPIS modeling tool reside in the folder *model* inside the Eclipse project specified during project configuration. For each instantiation of an element defined in the metamodel, a separate file is created in the file system. As a consequence, all methods and fields of an entity are persisted in self-contained files. As a new element is instantiated and persisted, the modeling environment generates a unique identifier which is used as reference by related elements. This identifier is also included in the naming convention for the generated files along with the extension *esx*. Although SERAPIS adheres to the XML notation in terms of persistence, the fact that entities are dispersed among multiple files and referenced by artificial identifiers practically makes the model unreadable for humans.

The code extraction in Listing 3.3 displays the XML notation for the element `Book` included in the *library* example. Except for the primary key field and the other technical attributes, the entity has no direct reference to its functional attributes and methods, instead it declares a Universally Unique Identifier (UUID) in order to be referred to. The listing further shows some fields of type `Value`, `Enumeration`, and `Reference` which have been specified by the metamodel respectively the meta-language. Here the field `ShortDescription` gives an example how `Text` answers the purpose of documentation. In this case, the value of the field is displayed as additional information textually describing the current entity inside the model editor.

3.3 Issues of SERAPIS

In the year 2011 the Austrian Federal Ministry of the Interior (BMI) assigned the task of evaluating the versatility of the SERAPIS modeling tool to the Vienna University of Technology. The resulting document presented a criteria catalogue which was subsequently used to evaluate the SERAPIS modeling tool. After a comparison with other state-of-the-art modeling tools, recommendations in terms of possibilities for future evolution steps have been made.

```
<element
    name="Entity"
    super="EntityBase"
    class="at.sphinx.sxme.informationmodel.
      model.entity.IMEntity"
    transformer="at.sphinx.sxme.informationmodel.
      transformer.IMEntityTransformer"
    icon="10_Source/icons/elements/entity.gif"
    isAbstract="false"
    isSingleton="false">
  <value
      name="Abbreviation"
      class="at.sphinx.sxme.core.value.StringValue"
      isRequired="true"
      isDisplayOnly="false"
      isHidden="false"
      isUpperCase="true"/>
  ...
  <enumeration
      name="Type"
      enumeration="EntityType"
      isRequired="true"
      isDisplayOnly="false"
      isHidden="false"
      default="ProductionData"/>
  ...
  <reference
      name="PrimaryKeyField"
      element="EntityField"
      isComposition="false"
      isRequired="false"
      isDisplayOnly="true"
      isHidden="false"
      isPSM="true"
      delete="nullify"/>
  ...
</element>
```

Listing 3.2: Code excerpt of the *plugin.xml* file

```
<?xml version="1.0" encoding="windows-1252"?>
<sxme:elements xmlns:sxme="http://www.sphinx.at/sxme">
  <element display="Book"
      model="ba67a429_dd55_40ad_ab46_08646049d1f0"
      type="Entity"
      uuid="e9164599_527e_40fd_a8b4_8a1eecd8c24b"
      psm="false">
    <localizedtext
        name="ShortDescription"
        computed="false">
      <textstring
          language="a36eeef6_5e2e_4de4_8bf2_60ff2698dcce">
        Short description of the Book entity
      </textstring>
    </localizedtext>
    ...
    <value name="IsAbstract"
        value="false"
        computed="true" />
    <value name="Abbreviation"
        value="BOOK"
        computed="false" />
    ...
    <enumeration
        name="Type"
        literal="ProductionData"
        computed="true" />
    ...
    <reference name="PrimaryKeyField"
        display="ID : OID"
        model=""
        type="EntityField"
        uuid="6535c580_5344_4fba_a9d8_53be2b6c5e38"
        index="-1"
        computed="true" />
    ...
  </element>
</sxme:elements>
```

Listing 3.3: Excerpt of the `Book` element definition

The evaluation is significant to this master thesis as it not only provides an in-depth look into the mechanics of the modeling tool, but it also indicates the competitiveness on the sector of MDE tools. Therefore the outcome of the evaluation and the resulting recommendations are summarized as follows.

Criteria catalogue

The criteria catalogue covers the aspects of supported modeling languages, interoperability, and code generation. In order to keep this summary compact, the criteria definition is presented along with the tool assessment.

UML version Crucial to any modeling tool is the question which modeling languages it supports. Here the OMG suggests to use MOF-based languages such as UML. If this is the case also the UML version is significant due to the improvements introduced with UML 2.0. The SERAPIS metamodels tailored to customer requirements are not necessarily compliant to any meta-language standard which causes a vendor-lock. The metamodel customized for the BMI is only vaguely related to a subset of UML 1.4 while on the other hand its language features introduce very little overhead.

Constraint languages Constraint languages such as the Object Constraint Language (OCL) [25] allow to define requirements and restrictions to models in order to validate and ensure their correctness. The SERAPIS modeling tool provides no support for languages of that kind.

UML profiles UML profiles represent a light-weight approach to extend the UML metamodel by domain-specific modeling concepts. Due to the lack of compliance to UML, the modeling tool developed by the Sphinx IT-Consulting features neither UML profiles nor any comparable mechanisms.

Metamodeling If UML profiles are not applicable, other mechanisms might allow the extension of an existing metamodel. As the capability to perform manipulations to the metamodel is reserved to Sphinx IT-Consulting there is no way to apply extensions of any kind.

Abstraction levels In order to grant separation of platform-independent and platform-specific definitions, modeling tools are suggested to adopt the paradigm of MDA which distinguishes between PIM, PSM, and code. SERAPIS adheres to no such distinction as the metamodel is already platform-specific due to the customization process. So on one hand it accurately meets specific requirements, but on the other hand the adaptation to other platforms probably results in much effort.

Markers Markers can be used to enrich modeling elements with platform-specific information. As all platform-specific details are included in the metamodel it is not necessary for SERAPIS to support appropriate mechanisms.

Verification and validation In order to detect errors in a model at an early stage of development, modeling tools are expected to provide mechanisms for verification and validation. While verification checks the syntactical correctness of a model, validation refers to functional requirements. SERAPIS provides features to ensure the correctness of the syntax but offers no validation mechanism.

Modeling test cases In order to conduct software tests it is desirable to have the option to model test cases. Although it would be feasible to have this feature implemented by Sphinx, there is no out-of-the-box approach for this.

UI modeling Using models to specify user interfaces introduces an additional abstraction level which allows to automatically generate different variants of implementations such as web-based or rich clients. Here the evaluation by the Vienna University of Technology originally states that SERAPIS provides no such features because the product deployed with the BMI includes only the license for the persistence layer. As mentioned at the beginning of this chapter, SERAPIS comprises a second module which actually allows to model user interfaces. As that module is not in the scope of this master thesis, it has not been evaluated yet which is why only its existence is to be mentioned here for the sake of completeness.

Model interchange Modeling tools are required to comply to well-defined interfaces such as the XMI format in order to allow the loss-free interchange of model information. SERAPIS provides no XMI support for importing or exporting models.

Multi-user support To enable concurrent development, modeling tools would benefit of features to support version control and conflict management of models. Although the files created in order to persist model information can be managed by external version control software, due to the fragmentation and structure of the SERAPIS modeling artifacts it is virtually not feasible to merge conflicting changes without being at risk to corrupt the model.

Forward engineering This term describes the transformation of the abstract specification made by the model to the corresponding target language, usually resulting in executable code. The SERAPIS modeling tool provides generators for Java and SQL DDL.

Reverse engineering In order to integrate existing projects, a modeling tool is required to extract model definitions from code. The Schema Migrator integrated with SERAPIS can be used to derive models from DDL definitions.

Transformation languages MDA suggests to specify the transformation logic from models to code but also between models using model transformations usually based on specialized transformation languages. SERAPIS features no support for such transformations, instead it exposes the models using a Java API.

Protection mechanisms Modeling tools necessarily allow partial generation leaving room for manual implementations. To preserve these implementations from getting lost due to further code generation, modeling tools are required to provide adequate protection mechanisms. SERAPIS tackles this problem by initially creating specific implementation classes which are not overwritten by further iterations of code generation.

Documentation A convenient modeling feature is the automatic generation of documentation including textual descriptions and diagrams. With respect to this requirement SERAPIS allows to generate ER diagrams depicting the model.

Comparison

Table 3.2 shows the comparison with other state-of-the-art modeling tools according to the presented criteria catalogue. This selection of reference software is only a subset of available tools which implement the MDA standard.

Although SERAPIS lacks the ability to satisfy some evaluation criteria such as constraint languages or the modeling of test cases, the fact has to be considered that it would be technically feasible to remedy a major part of these deficiencies.

The conclusion of the evaluation points out potential for improvements with respect to future evolution of the SERAPIS modeling tool. The first issue refers to the absence of mechanisms to adapt the development environment to domain-specific modeling concepts. As the customer is not provided with a light-weight approach such as UML profiles, the only way to achieve this is to directly manipulate the metamodel which is reserved to Sphinx IT Consulting.

Basically the same problem can be observed with the code generation. The need to change the output of the code generation process can emerge from various events such as the migration to a new database system or programming framework. In this case, the customer is also required to add changes directly to the source code which would be neither intuitive nor sustainable. Instead, the evaluation document suggests to deploy a template-based approach as a common solution to this problem.

The lack of interchangeability of the modeling information resulting in a vendor lock is the main issue which is pointed out by the study. This problem can be tackled either by adopting a standardized interchange format such as XMI or by creating new models from existing definitions using model transformations. The observation concerning this issue is most significant to this master thesis as it confirms the need for the transformation approach evaluated in this work.

Criteria	SERAPIS	BridgePoint 6.1	iUML/iCCG 2.20	OptimalJ 3.1	ArcStyler 4	Objecteering 5.3	AndroMDA 3.4
UML version	Subset 1.4	1.5	1.5	1.4	1.4	1.4	1.4
Constraint languages	-	OAL	ASL	-	OCL	-	OCL
UML profiles	-	-	-	-	+	+	+
Metamodeling	/	-	-	+	-	-	-
Model interchange	-	+	+	+	+	+	-
Multi-user support	+	+	+	+	+	+	-
Verification	+	+	+	-	+	/	-
Validation	+	+	+	-	+	-	-
Documentation	+	/	/	/	/	+	-
Modeling test cases	-	/	/	/	+	+	-
UI modeling	+	-	-	/	/	/	-
Abstraction levels	PSM	aPIM	aPIM	PIM, PSM	aPIM	aPIM	aPIM
Markers	+	+	+	+	+	+	+
Forward Engineering	Java, SQL DDL	C, C++	C, C++, Ada	J2EE	J2EE, .NET	C++, Java, CORBA	Struts, Java, EJB
Reverse Engineering	SQL	-	-	COBOL, SQL, IDL	EJB	COM	-
Protection mechanisms	+	-	-	+	+	+	+
Transformation languages	Java	OAL	ASL	IC TPL	JPython	J	VTL

Table 3.2: Comparison of SERAPIS and MDA modeling tools

CHAPTER 4

Metamodel Bridging

This chapter briefly presents different integration strategies suggested by the Ph.D. thesis of Wimmer [31] for bridging metamodels. The approach is discussed by taking a glance at the different phases he introduces and by clarifying the terms of transformation rules and heuristics. According to the insight gained from the suggestions, the approach is adapted in order to provide an architecture for solving the transformation problem presented in the context of this master thesis.

4.1 Metametamodel-based Transformation

The Ph.D. thesis of Wimmer [31] proposes strategies for tool integration with respect to model interchange in the context of MDE. The contribution comprises approaches for integration scenarios based on mapping the concepts of different metamodels as depicted in Figure 4.1.

Part I of the contribution proposes a mining pattern for metamodels and models, whereas the term mining pattern refers to *the process of generating model-based representations out of text-based descriptions* [31, p. 10]. The mining pattern can be deployed on the M3 meta-layer in order to bridge different meta-languages. The DTD2Ecore framework presented in the Ph.D. thesis [31] implements this mining pattern to create Ecore-based metamodels from Document Type Definitions (DTD).

Part II introduces a framework which allows to define mapping operators with the purpose of increasing the level of abstraction when building metamodel bridges. The framework provides a high-level mapping view to describe the correspondences of metamodel elements, while the transformation logic of the mapping operators is described at a more detailed level. As an example, mapping operators resulting from the application of the framework are employed by the CAR mapping language presented by Wimmer which is intended to resolve structural heterogeneities of different metamodels.

Part III of the contribution proposes two concepts to implement roundtrip transformations without taking the risk of information loss. The term roundtrip here refers to the bidirectional

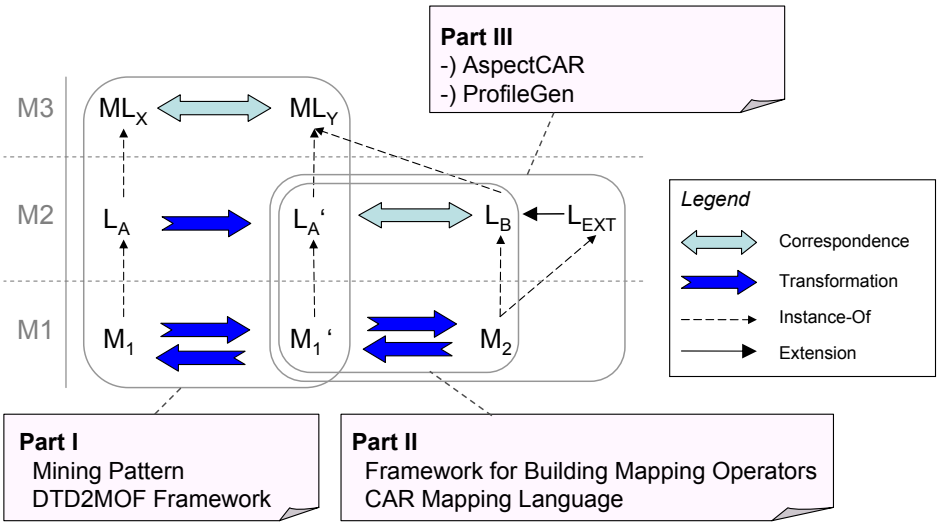

Figure 4.1: Contribution and supported integration scenarios by Wimmer [31]

bridging of metamodels. At first, the thesis describes AspectCar which is an aspect-oriented extension for the CAR mapping language. The second approach called ProfileGen particularly focuses on an integration scenario that features the bridging of domain-specific languages to UML as these transformations are especially exposed to information loss.

As depicted in Figure 4.1, the framework for building mapping operators as well as the suggested approaches for implementing roundtrip transformations refer to integration scenarios which presume source and target metamodels to be defined by the same meta-language. The mining pattern on the other hand, presented as a part of the contribution, actually allows to bridge different meta-languages rendering it a potentially applicable strategy for tackling the problem discussed in this master thesis.

From DTDs to Ecore-based Metamodels

Wimmer describes his metamodel bridging approach by the example of WebRatio, which is a commercial tool with the focus on the development of web-based applications using the modeling language WebML. The language concepts of WebML are partially defined in XML DTDs and partially hard-wired in the WebRatio modeling tool. WebRatio uses an XML structure for the persistence of models and consequently applies Extensible Stylesheet Language Transformations (XSLT) in order to generate code from these models.

DTDs provide less expressiveness, extensibility, and readability than MOF with respect to describing modeling languages. Besides that, the development of XSL transformations with respect to code generation is a sophisticated task. A metamodel-based approach adhering to the concept of MDE would address these problems by allowing to apply adequate modeling techniques such as QVT and ATL which is why the Ph.D. thesis of Wimmer [31] suggests to establish a bridge between WebML and MOF.

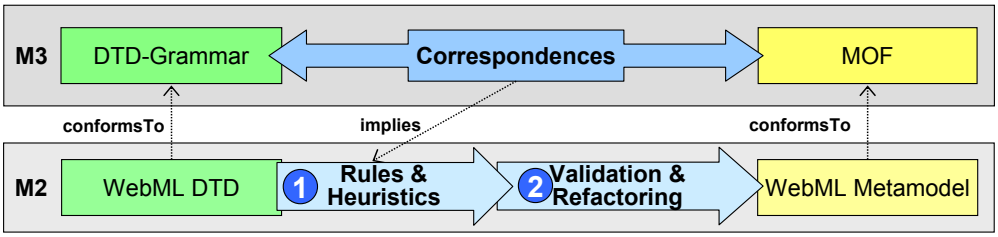

Figure 4.2: Two phase semi-automatic transformation approach [31]

In order to implement this bridge, a MOF-based metamodel for WebML has to be created. Due to the extent of the language, building the metamodel from scratch would be error-prone and time-consuming which is why an automatic approach is needed to create the metamodel. To bypass these deficiencies, the thesis [31] describes a mining pattern allowing to generate a MOF conform metamodel from the existing language definitions of WebML.

The principle of the mining pattern is based on the fact that the architectures of the related technical spaces employ at least three meta-layers including meta-languages which reside on the M3 layer. From the correspondences of these meta-languages mappings can be derived which are used to transform the source metamodel defined in the XML technical space to the target model defined in the MOF technical space.

The mining pattern depicted in Figure 4.2 is described in terms of a semi-automatic approach which is based on two phases. During the first phase a component referred to as metamodel generator applies the non-ambiguous transformation rules resulting from the meta-language correspondences to automatically generate an initial version of the WebML metamodel. Due to deficiencies resulting from the limited expressiveness of DTD, transformation rules are not sufficient to map the meta-languages. Therefore, an additional set of heuristics is applied along with the generation process. The second phase allows to manually validate the resulting metamodel in order to apply refinements if necessary.

Rule	**DTD Concept**	**Ecore Concept**
R2	XMLAttribute	EAttribute
	XMLAttribute.name	EAttribute.name
(1)	XMLStringAtt, NMTOKEN(S), IDREF(S)	EAttribute.eAttributeType=EString
(2)	ID	EAttribute.eAttributeType=EString, EAttribute.id=true
(3)	XMLEnumAtt	**add** EEnum EEnum.name= XMLEnumAtt.name+"_ENUM" **for each** XMLEnumLiteral **add** EEnumLiteral EAttribute.eAttributeType=EEnum

Table 4.1: Attribute rule of the DTD2Ecore framework [31]

As an example, Table 4.1 displays the rule for mapping attributes in the DTD2Ecore framework. `XMLAttributes` and their names can be mapped directly using `EAttributes`. `XMLStringAtt`, `NMTOKEN`, `NMTOKENS`, `IDREF`, `IDREFS`, and `ID` are handled as string types (1) whereas the corresponding `EAttribute` for an `ID` is additionally tagged as such (2). As both DTD and Ecore allow to specify enumerations and literals, the definition of an adequate rule is straightforward (3).

DTD Concept	**Ecore Concept**
If XMLEnumAtt has two XMLEnumLiterals and XMLEnumAtt is one of {true, false}, {1, 0}, {on, off}, {yes, no}	**then** EAttribute.eAttributeType=EBoolean annotate with «Validate Boolean»
else if XMLEnumAtt has two XMLEnumLiterals	**then** annotate EEnum with two EEnumLiterals with «Resolve possible Boolean type manually»

Table 4.2: Example heuristic of the DTD2Ecore framework [31]

As mentioned before, the expressiveness of DTD is limited in comparison with Ecore which is why the mapping of these meta-languages cannot be expressed completely using transformation rules. These deficiencies for instance include the absence of a grouping mechanism and the inability to define inheritance relations. In order to address these deficiencies, Wimmer introduces heuristics to be applied in addition to the transformation rules.

Table 4.2 displays a heuristic that is based on the fact that DTD only provides a limited set of data types. While `Strings` are the most common data types, a `Boolean` can only be expressed using an `Enumeration` with two `Literals` to define a negative and a positive value. The heuristic applies to each `Enumeration` in order to examine the number of `Literals` and their naming conventions. Once a potential `Boolean` definition is recognized, the corresponding Ecore element is annotated to tag it for further refinement.

This example shows how heuristics use annotations in order to mark the need for manual intervention during the second phase of the transformation process. Furthermore, it shows how the effectiveness of heuristics depends on the adequate application of naming conventions.

4.2 Transforming SERAPIS to Ecore

The initial situation described in this master thesis is related to the problem approach of the DTD2Ecore framework presented in the Ph.D. thesis of Wimmer [31]. The persistence strategies for models of both SERAPIS and WebRatio are based on proprietary XML notations, and the metamodels of both modeling tools are to extensive in order to be recreated from scratch. As the main difference, SERAPIS introduces a custom meta-language while WebRatio employs DTD in order to define metamodels. However, this has no effect on the transformation approach as both meta-languages are not compliant to MOF which is why each tool effectively introduces

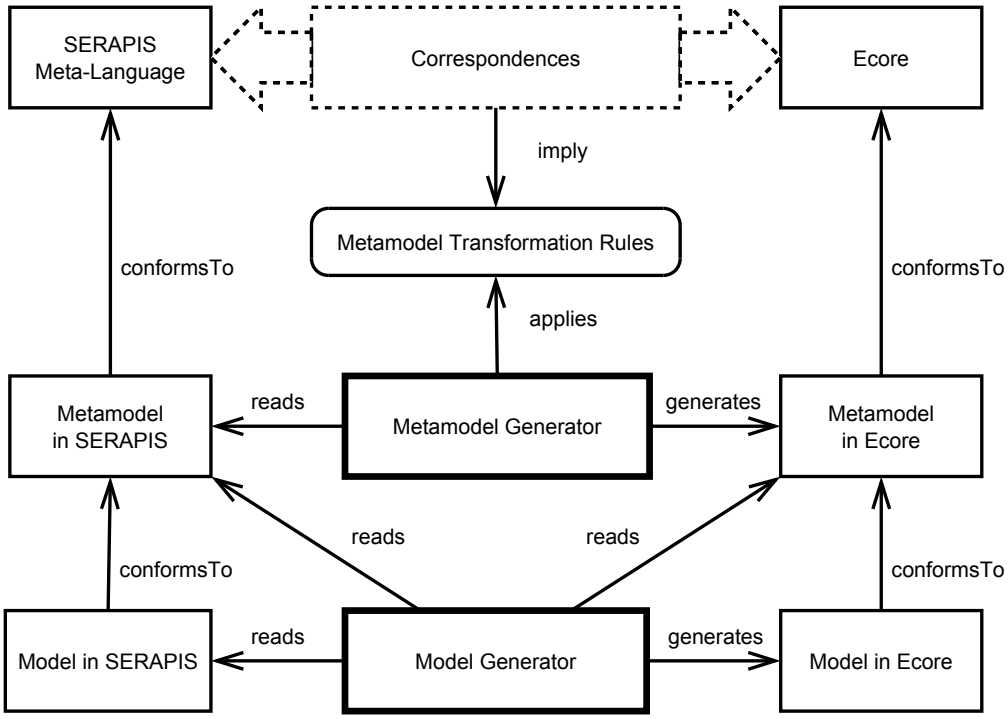

Figure 4.3: SERAPIS2Ecore framework

its own technical space. As a result, both commercial modeling tools suffer from a lack of compliance to existing standards in terms of tool integration.

The goal of this master thesis is to research and evaluate an approach for establishing a bridge from the SERAPIS technical space to the EMF technical space. As the transformation approach described in terms of the DTD2Ecore framework applies to the problem situation presented in this master thesis, the approach of Wimmer will be adopted as a model to this work.

With respect to the problem discussed in this master thesis, the main achievement of the DTD2Ecore framework is to provide a generic approach which allows to semi-automatically create a target metamodel corresponding to a source metamodel by identifying correspondences between the source and the target meta-languages they conform to. As mentioned before, the component responsible for the generation of the target metamodel is the metamodel generator.

Once the metamodel has been created successfully, it can be used to generate conforming models from existing source models. In order to provide this functionality, the framework will be extended by another component consequently referred to as model generator. The design of this component is related to the basic concept for model transformation as described in the work of Czarnecki and Helsen [12].

Figure 4.3 depicts the architecture of the SERAPIS2Ecore framework as an adaptation of the DTD2Ecore framework to the SERAPIS technical space. According to the approach described in the Ph.D. thesis of Wimmer [31], the correspondences between the SERAPIS meta-language and

Ecore are identified in order to derive transformation rules. The metamodel generator reads the SERAPIS metamodel and applies these transformation rules in order to generate a corresponding metamodel which conforms to Ecore.

The metamodel generator is also designed to apply heuristics allowing to tag elements with annotations indicating the need for further refinement. The necessity to identify heuristics depends on whether it is possible to map the language concepts of SERAPIS and Ecore by applying non-ambiguous transformation rules only. In this case, there would be no need to include heuristics which would consequently render the phase for manual validation redundant. The mapping of the meta-languages introduces the main uncertainty as it directly affects the feasibility of the transformation approach which is why the following discussion of the metamodel generator in Chapter 5 emphasizes this issue.

The model generator has been introduced to the SERAPIS2Ecore framework to read models from existing SERAPIS projects and to generate corresponding models which comply to the metamodel provided by the metamodel generator. In order to implement this transformation, the component first determines which language concepts have been specified by the source metamodel to define the elements of the source model. Afterwards, the corresponding language concepts specified by the target metamodel have to be identified to instantiate model elements corresponding to the source model elements. Chapter 6 provides a more detailed view on the mechanics of the model generator.

CHAPTER 5

Metamodel Transformation

Before a model defined by the SERAPIS modeling tool can be transformed to an Ecore-conform model, an adequate metamodel which is used to instantiate the target model has to be created using the Ecore language features. In this context, the Ph.D. thesis of Wimmer [31] describes a semi-automatic approach that suggests to provide mappings between the corresponding meta-languages which are employed to generate the target metamodel according to the existing definitions of the source metamodel.

This chapter describes how the approach suggested by Wimmer is employed to generate an Ecore-based metamodel from the metamodel defined by the SERAPIS meta-language. Before the transformation process can be examined in detail, the elements of the SERAPIS meta-language which should be included in the transformation have to be identified. Therefore, a set of modeling concepts is discussed which is used to evaluate the relevance of the language elements with respect to the target metamodel.

The mapping of the SERAPIS meta-language to Ecore comprises the major part of the whole transformation process. This chapter presents a first approach for the language mapping and consequently identifies its flaws. By applying the concept of EMF Profiles [20] the mapping approach is enhanced in order to provide a mapping for the complete set of SERAPIS meta-language features. Subsequently, the effects of the enhanced mapping approach to the architecture of the metamodel transformation and the component responsible for the implementation, referred to as metamodel generator, are presented briefly. Finally, the transformation mechanism of the metamodel generator is demonstrated by an example.

5.1 Type Mapping

Compared to Ecore, the meta-language employed by the SERAPIS modeling tool provides a rather small set of language concepts. The core element which contains all other elements specified in the language definition is referred to as `Element`. The main characteristics of `Elements` are specified by the `Properties` they declare. `Property` is a common su-

perclass for `Value`, `Reference`, `Text`, and `Enumeration`. `Values` can contain primitive data types while `References` are used to map relationships to complex types such as `Elements`. `Text` elements are special `Properties` which allow to declare textual information usually for documentation purposes. Furthermore, the SERAPIS meta-language allows to define `Enums` which are equal to the eponymous concept of other languages such as Ecore and Java. `Enumerations` are declared by `Elements` in order to reference `Enum` definitions.

Besides the relationships between these elements, the SERAPIS meta-language furthermore defines individual attributes for each element to specify the semantics. For instance, `Element` introduces an attribute called `super` which optionally points to another `Element` in order to map an inheritance-relation. On the M3 meta-layer the language elements `Element`, `Enum`, `Enumeration`, `Value`, `Reference`, and `Text` can be considered as classes known from object-oriented programming languages while their attributes match the fields they declare.

Once a metamodel is created using the SERAPIS meta-language, the elements are instantiated and their attributes are assigned with concrete values. As mentioned before, the customization of the SERAPIS modeling tool for a certain case of application does not require a new metamodel to be built from scratch, instead the default metamodel is extended and modified to meet the customer requirements.

An examination of the default metamodel reveals that the definitions it provides are not fully restricted to the specification of models, furthermore it comprises information that is specific to the modeling environment. Probably this information has been introduced for practical reasons as the modeling tool has evolved over time, but with respect to the original purpose of a metamodel, which is to provide a specification for models, this information can be considered as redundant.

In the context of bridging the SERAPIS metamodel to the EMF technical space it is recommendable to exclude this information from further processing. As the definition of this information results from the meta-language, this can be achieved with little effort simply by eliminating it from the language mapping. In this context, the general question emerges which language concepts of the SERAPIS meta-language should be considered in the transformation process.

Answering this question calls for a detailed examination of the available language concepts which is why a short clarification of modeling concepts is to be presented as follows. In order to determine whether a language element should be considered in the mapping process, it is sufficient to evaluate the attributes it defines according to the presented concepts as the absence of any significant attributes would also render the corresponding element obsolete.

Redundant modeling concepts As a rule, the metamodel is designed with the goal to include essential information only, in order to keep it simple and therefore maintainable and understandable. The SERAPIS meta-language provides a small set of attributes which are mainly used to enrich the resulting models with information that could be derived from other attributes. This information is actually added for reasons of convenience and can be considered as redundant. This aspect determines the redundancy of an attribute by examining whether it is possible to derive its value from other attributes.

Tool-specific modeling concepts The SERAPIS meta-language defines a complete set of elements necessary to create a metamodel which in turn provides a specification for models.

As mentioned before, not all of the attributes defined by the SERAPIS meta-language are explicitly necessary to provide further specifications for models, instead they provide functionality which is specific to the modeling tool only. This aspect tries to determine the significance of an attribute with respect to metamodeling, assuming that tool-specific attributes are to be excluded from the transformation process.

Functional significant modeling concepts Some few attributes are neither redundant nor completely tool-specific, but still it is necessary to examine whether these attributes can be expected to introduce essential information with respect to tool integration. This aspect focuses on the functional significance of attributes in order to determine whether the provided information could be of interest for further modeling applications.

Table 5.1 shows the evaluation of the attributes in order to determine whether they are to be considered with respect to mapping the meta-languages. The result shows that every element of the SERAPIS meta-language is necessary to create a complete metamodel while a considerably high number of attributes has been skipped for providing only tool-specific functionality. `Lines` and `IsUpperCase` on the other hand are the only attributes to be declared as redundant for this information can be retrieved by examining the actual value assigned to these fields.

In further detail, the result has shown that the functional significance of an attribute correlates with the assessment of whether it is tool-specific. Although this outcome was to be expected, the attributes `Lines` and `IsDescription` are exceptions for they cannot be considered as tool-specific but still lack functional significance.

5.2 Mapping the SERAPIS Meta-language to Ecore

The feasibility of finding correspondences between two meta-languages depends on whether their language elements can be mapped to each other. For some elements it might be possible to identify clear and distinct counterparts within the target language. In this case, non-ambiguous transformation rules can be derived in order to implement the mapping. Some elements on the other hand might not be expressed in another language without introducing ambiguities which is why Wimmer suggests in his Ph.D. thesis to apply heuristics followed by manual validation in order to resolve the resulting conflicts.

For the sake of simplicity, straight-forward mappings between SERAPIS and Ecore are preferred which is why subsequently the potential transformation rules *R1* to *R4* are identified. Therefore, the SERAPIS meta-language is examined in order to find matching language constructs in Ecore.

R1 `EClass` is the obvious choice for mapping `Element` as both elements inherently share a common set of attributes and can be considered as main elements in terms of building a metamodel. Both elements are instantiated to specify complex types which are defined by their fields and the references they define to qualify the relationships between each other. For this purpose, `Element` defines `Properties` in SERAPIS while `EClass` defines `EStructuralFeatures` in Ecore. With respect to inheritance, Ecore exceeds the requirements of the SERAPIS meta-language by allowing `EClass` to have multiple

Attribute	**Element**	***Redundant***	***Tool-specific***	***Functional significant***	**Included in mapping**
name	Element, Property, Enum, Literal	no	no	yes	yes
super	Element	no	no	yes	yes
class	Element, Value	no	yes	no	no
isAbstract	Element	no	no	yes	yes
isSingleton	Element	no	no	yes	yes
isDisplayOnly	Element, Property	no	yes	no	no
isHidden	Element, Property	no	yes	no	no
isPSM	Element, Property	no	yes	no	no
transformer	Element	no	yes	no	no
icon	Element	no	yes	no	no
isRequired	Property	no	no	yes	yes
isUserSetting	Property	no	yes	no	no
lines	Text	yes	no	no	no
editorId	Text	no	yes	no	no
isLocalized	Text	no	no	yes	yes
isDescription	Text	no	no	no	no
default	Value	no	no	yes	yes
isUpperCase	Value	yes	no	yes	no
hasListOfValues	Value	no	no	yes	yes
unit	Value	no	yes	no	no
element	Reference	no	no	yes	yes
isComposition	Reference	no	no	yes	yes
delete	Reference	no	yes	no	no
enumeration	Enumeration	no	no	yes	yes
literalList	Enum	no	no	yes	yes

Table 5.1: Evaluation of the attributes provided by SERAPIS meta-language

Rule	SERAPIS Concept	Ecore Concept
R1	Element	EClass EClass.interface=false
	Element.name	EClass.name
	Element.super	EClass.eSuperTypes
	Element.isAbstract	EClass.abstract
R2	Reference	EReference EReference.upperBound = 1
	Reference.isRequired	**if** Reference.isRequired EReference.lowerBound = 1 **else** EReference.lowerBound = 0
	Reference.name	EReference.name
	Reference.isComposition	EReference.containment
	Reference.element	EReference.eType
R3	Enum	EEnum
	Enum.name	EEnum.name
	Enum.literalList	**for each** Literal **add** EEnumLiteral
R4	Literal	EEnumLiteral
	Literal.name	EEnumLiteral.name
		EEnumLiteral.value = # of Literal

Table 5.2: Mapping between the SERAPIS meta-language and Ecore

super types while `Element` only supports single inheritance. Unlike Ecore, the SERAPIS meta-language does not master the concept of interfaces which is why the corresponding attribute of `EClass` is always set to *false*.

R2 `Reference` is used to point to complex types namely `Elements` in terms of the SERAPIS meta-language. This behavior corresponds to the characteristics of `EReference` in Ecore, as implied by the naming convention. A `Reference` can only point to one single `Element` at a time as the main difference to `EReference` which supports various combinations of multiplicities. Of course this has no effect to the mapping as Ecore provides at least the same level of expressiveness, but as a result the upper bound of `EReference` can be initially set to 1. The `isRequired` attribute is mapped to `EReference` by setting its `lowerBound` to 1 in case it is *true*.

R3 and R4 The mapping between `Enumeration` and `EEnum` becomes obvious considering the fact that they serve the very same functional purpose. For representing values, both language elements employ literals namely `Literal` respectively `EEnumLiteral` which allow for a non-ambiguous mapping. The main difference between both enumeration concepts is that `EEnumLiteral` allows to explicitly assign an integer representation to its value while `Literal` only allows to assign its name as a string value. In order to

provide a valid value assignment to `EEnumLiteral`, its integer representation is chosen according to the position of the corresponding `Literal` occurrence in the definition of the `Enumeration` it belongs to.

The mapping of the SERAPIS meta-language and Ecore is implemented by applying the transformation rules described in Table 5.2. These rules are applied during the phase of automatic generation due to their non-ambiguous nature. Unfortunately, the rules cover only a small set of the SERAPIS language concepts so far. The major part of the attributes is still missing, and no adequate mapping mechanisms for the language elements `Enumeration`, `Value`, and `Text` have been found so far.

At a first glance, modeling the missing language elements in Ecore appears to be the most natural approach to tackle this problem. As an example, the following code snippet shows the Ecore definition of the language element `Value`. To provide the missing counterpart, the approach employs an instance of `EClass` with the same name as the corresponding element. The attributes `name` and `hasListOfValues` are mapped by `EAttribute` conforming to specifications of the SERAPIS meta-language in terms of name and data type.

```
<eClassifiers xsi:type="ecore:EClass"
    name="Value">
  <eStructuralFeatures xsi:type="ecore:EAttribute"
      name="name"
      eType="ecore:EDataType
      http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  <eStructuralFeatures xsi:type="ecore:EAttribute"
      name="hasListOfValues"
      eType="ecore:EDataType
      http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
  ...
</eClassifiers>
```

Listing 5.1: Ecore definition of the language element `Value`

Due to this simple and intuitive approach, it becomes possible to create the missing elements including all their attributes. Clearly the main benefit in this case is that the mapping between both meta-languages introduces no ambiguities as the design of the elements in Ecore exactly matches the specifications of SERAPIS. With respect to implementing the transformation mechanism, this approach still has three major flaws.

Missing attributes in `EClass` and `EReference` The transformation rules in Table 5.2 show the mapping of the language elements `Element`, `Reference`, `Enum`, and `Literal`. While `Enum` and consequently `Literal` can be expressed entirely in Ecore, the mappings in *R1* and *R2* are missing attributes such as `IsSingleton` and `isComposition`. The reason for this is that the language definition of Ecore obviously provides no adequate language features to map these attributes. Apparently the only way to add the missing attributes would be to manipulate the Ecore language which is actually no option.

Meta-layer transition As mentioned before, the mapping of SERAPIS and Ecore is based on the correspondences between both meta-languages. In a three meta-layer architecture these correspondences conceptually refer to the M3 layer while the actual transformation targets the metamodel residing on the M2 layer. Accordingly, the rules displayed in Table 5 map elements of both meta-languages while the application of the rules refers to the instances of those elements. Consequently, the language elements specified on the right-hand side of the rule are instantiated in order to create the target metamodel. The instantiation implies the transition of a model from one abstract meta-layer to another more concrete layer. With respect to this transition of meta-layers, the aforementioned approach to create the missing language elements by instantiation of `EClass` introduces a conceptual issue, as the generated metamodel consequently resides within the wrong meta-layer. The aforementioned approach already results in one meta-level transition while the transformation process requires to pass another abstraction level. As a consequence, the resulting metamodel actually corresponds to a model on the M1 meta-layer.

Meta-layer merge Another concern that goes along with the *meta-layer transition* problem is that Ecore language elements are used to map one set of elements specified by the SERAPIS meta-language while at the same time their instances provide the mapping for the rest of the SERAPIS language elements. As a result, the mapping spans two meta-layers within the architecture of Ecore, rendering it also impossible to match the generated metamodel to one certain meta-layer.

Please note that addressing all of these issues is beyond the scope of this master thesis. Solving the problem in terms of the *missing attributes in `EClass` and `EReference`* for once is not necessarily essential to the feasibility of the metamodel transformation. Nevertheless, a solution to this issue would be desirable as the objective of the transformation is to avoid information loss.

Also the issue with respect to the *meta-layer merge* is a conceptual deficiency which has no effect to the feasibility of the language mapping. From the architectural point of view however, this might have a negative effect to the understandability of the transformation approach which is why a solution would still be desirable.

Clearly the problem concerning the *meta-layer transition* appears more severe as the absence of an appropriate metamodel renders the whole model transformation impossible. As the nature of this problem is not related to potential ambiguities that might have been introduced along with the mapping of the meta-languages, the problem cannot be tackled by heuristics or manual validation. In a more general sense, a possible solution needs to provide a way to lift the instantiated Ecore language elements by one meta-layer. Here the EMF Profiles [20] project might provide an approach to achieve this.

5.3 EMF Profiles

Domain-Specific Modeling Languages (DSMLs) are modeling languages which are tailored to the domain-specific requirements in order to support the process of software development by introducing an additional level of abstraction. In terms of evolution, DSMLs provide only a

small degree of flexibility because changes in the requirements usually result in modifying both the syntax and the components of the modeling environments which are specific to language.

One of the main reasons for the success of UML is that it provides a profile mechanism to dodge this problem. A lot of UML tools support this lightweight and language-inherent extension mechanism by allowing the user to create custom profiles as well as by providing predefined profiles. Due to the success of this mechanism, some of the profiles have been specified as standard by the OMG.

On the one hand, DSMLs can be built from scratch and therefore allow full flexibility with the design of the modeling language, but it also becomes necessary to provide an adequate modeling environment which has to be considered in terms of evolution. On the other hand, the lightweight approach of UML profiles provides only an extension mechanism without actually allowing to change an existing metamodel. The extension mechanism suggested by EMF Profiles [20] adapts UML profiles to extend existing DSMLs in order to combine the advantages of both languages aiming for the benefits described as follows.

Lightweight language extension The UML profiling mechanism is referred to as lightweight, as it allows to introduce new language features without the need to re-create the existing modeling environment or the modeling language from scratch. Using UML profiles, new language concepts can be created and evaluated quickly while preserving the structure of the modeling language they extend.

Dynamic model extension UML profiles allow to dynamically extend existing models while preserving the extended model elements. As the additional profile information is stored aside the model, the original model instance is not polluted. One model element can even be extended by multiple stereotypes at the same time which conforms to having multiple types.

Preventing metamodel pollution Modeling information that is not domain-specific can be separated by using UML profiles in order to keep the original domain metamodel clean. As an example, the elements of a domain model are extended with the results of a model review which is used to evaluate the domain model. In this case, the model review is not a part of the domain model which is why this information is not supposed to be mixed into the domain model.

Model-based representation Information introduced by a profile can be accessed and processed like regular model information. Therefore, it is possible to reuse model engineering technologies on profile applications and to check their validity against the profile definition which is equal to checking the validity of models with the metamodels they conform to.

The objective of EMF Profiles is to adapt the idea of UML profiles to other modeling languages in order to provide extension capabilities with the aforementioned benefits. Although the contribution mainly focuses on EMF for currently being one of the most popular modeling framework, the approach is designed to be applicable to arbitrary modeling languages.

From UML Profiles to EMF Profiles

The UML package `Profiles` is a part of the UML specification and resides on the M3 meta-layer as depicted in Figure 5.1. Instances of the meta-language package `Profiles` are defined on the same meta-layer as the UML metamodel which is M2. As a consequence, `Profiles` can be instantiated in order to build a profile application the same way as the UML metamodel is instantiated.

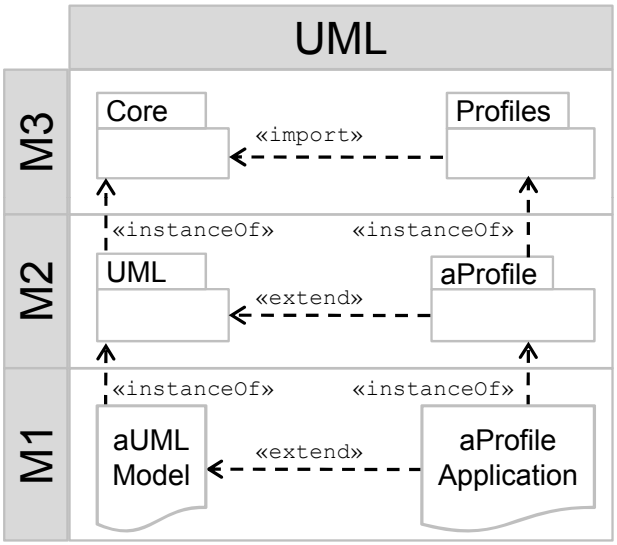

Figure 5.1: UML architecture [20]

To adapt the idea of the UML profiling mechanism to EMF, a metamodel referred to as `Profile MM` (Figure 5.2) first is designed in terms of Ecore. This metamodel corresponds to the UML package `Profile` in the sense that it also allows to specify profiles including stereotypes and tagged values. In order to apply a specific profile (in the figure referred to as `aProfile`) to arbitrary models, a profile application (`aProfile Application`) is created by instantiating the profile and assigning concrete values to the tagged values it defines.

Unlike UML, the meta-language Ecore, which resides on the M3 meta-layer of EMF, provides no native support for a profiling mechanism. In order to instantiate profiles in EMF like in UML, it would be necessary to extend Ecore on the same meta-layer which is no option as this would entail an intervention with the implementation of EMF. Therefore, `ProfileMM` is defined on the M2 meta-layer while `aProfile` consequently is defined on M1 as depicted in Figure 5.2. The main problem here is that EMF does not allow to instantiate the instance of a metamodel. As a result, it is not possible to instantiate the stereotypes defined by `aProfile` as the profile resides on M1.

In order to tackle this problem, providing `aProfile` with the capability of being instantiated, it is necessary to lift `aProfile` to the M2 meta-layer. To achieve this, the extension mechanism [20] suggests two strategies described as follows.

Meta-level lifting by transformation This strategy, as depicted in column (a) in Figure 5.2, suggests to automatically create a metamodel (in the figure referred to as `aProfile as MM`) on M2 corresponding to the profile on M1 by applying model-to-model transformation based on the mapping of the language concepts between EMF Profile and Ecore. The generation process creates a corresponding `EClass` for each `Stereotype` while each `TaggedValue` is mapped to an `EStructuralFeature`. The generated metamodel can be instantiated as is usual for being a regular instance of Ecore.

Meta-level lifting by inheritance The second strategy is based on the instantiation capabilities of the language element `EClass` residing on the M3 meta-layer. Instances of `EClass` are the only elements of a metamodel which can be instantiated in order to create an object on the M1 meta-layer. As depicted in column (b) in Figure 5.2, this capability is exploited in order to allow the direct instantiation of specific profiles residing on M1. In particular, `Stereotype` being a part of `ProfileMM` is specified as a specialization of `EClass`. As a consequence, the instantiation capabilities are granted to `aProfile` which in turn can be instantiated to create stereotype applications.

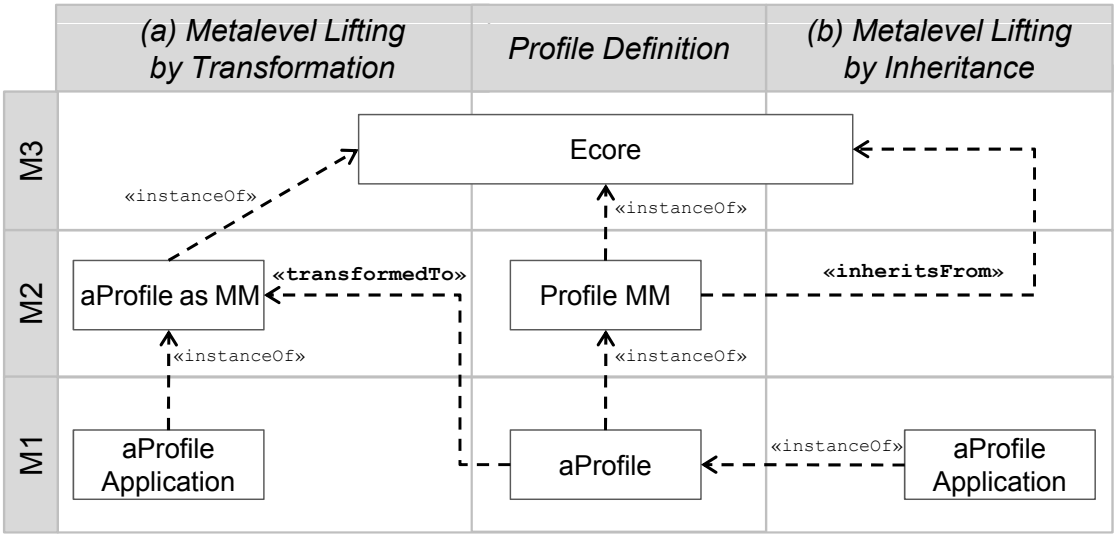

Figure 5.2: EMF profile architecture strategies [20]

According to the second strategy, `aProfile` and its instance `aProfileApplication` reside within the same meta-layer which implies that a profile can be considered as an entity with two facets with respect to meta-layers. Therefore, the EMF profiles framework favors the second strategy as this approach allows both to define and instantiate the profile by using the same artifact. When considered as an instance of `ProfileMM`, a profile is assigned to the M1 meta-layer. At the same time it is also plausible to locate a profile on the M2 meta-layer, for the stereotypes it contains are indirect instances of `EClass`.

The EMF Profile Metamodel

Figure 5.3 shows the metamodel for specifying the profile modeling language of the EMF Profiles framework. The language features are divided among packages which are subsequently merged to build the complete EMF Profile language.

Standard EMF Profile

As suggested by the second strategy for meta-level lifting, `Stereotype` is specified as a subclass of the `EClass` language element to inherit its instantiation capability. Due to this inheritance relation, `Stereotype` furthermore is allowed to contain `EAttributes` and `EReferences` which is why no additional element has to be introduced to the profile metamodel in order to implement the concept of tagged values.

`Extension` allows to apply stereotypes to metaclasses by specifying an `EClass` as the base meta-class for the stereotype. `LowerBound` and `UpperBound` can be used to define restrictions defining how many instances of the base meta-class a specific stereotype must be applied to. The relationships `redefined` and `subsetted` are employed by `Extension` in order to map the UML language concept of *Associations*.

In order to provide stereotype applications with a reference to the model elements they are applied to, the metamodel package `ProfileApplication` introduces the language element `StereotypeApplication`. This class is automatically set as a superclass for each stereotype whenever a profile is saved. Consequently the reference `appliedTo` is inherited allowing to refer arbitrary `EObjects`.

Generic Profiles

This language feature allows to reuse a profile specification for several DSMLs. Instead of extending concrete metaclasses, the approach of generic profiles suggests stereotypes to extend so-called generic types. These types can be considered as placeholders which makes them independent of a concrete meta-language. Generic types are bound to concrete meta-classes according to the actual DSML the profile is applied to. As a profile can provide an arbitrary number of bindings, the specification can be applied to a set of DSMLs.

Meta Profiles

Unlike generic profiles, this approach aims to reuse profiles for all DSMLs without the need to specify an extension for each DSML. Therefore, stereotypes must not refer to a specific metamodel, instead they must be applicable to any given model element independent of its metaclass. To achieve this, instead of directly extending a meta-class, stereotypes refer to `EClass` for all model elements are indirect instances of `EClass` [20].

5.4 Extending Ecore with Additional SERAPIS Concepts

To recall, the main problem concerning the mapping between the meta-languages of SERAPIS and Ecore is that the language elements provided by Ecore are not sufficient to cover all ele-

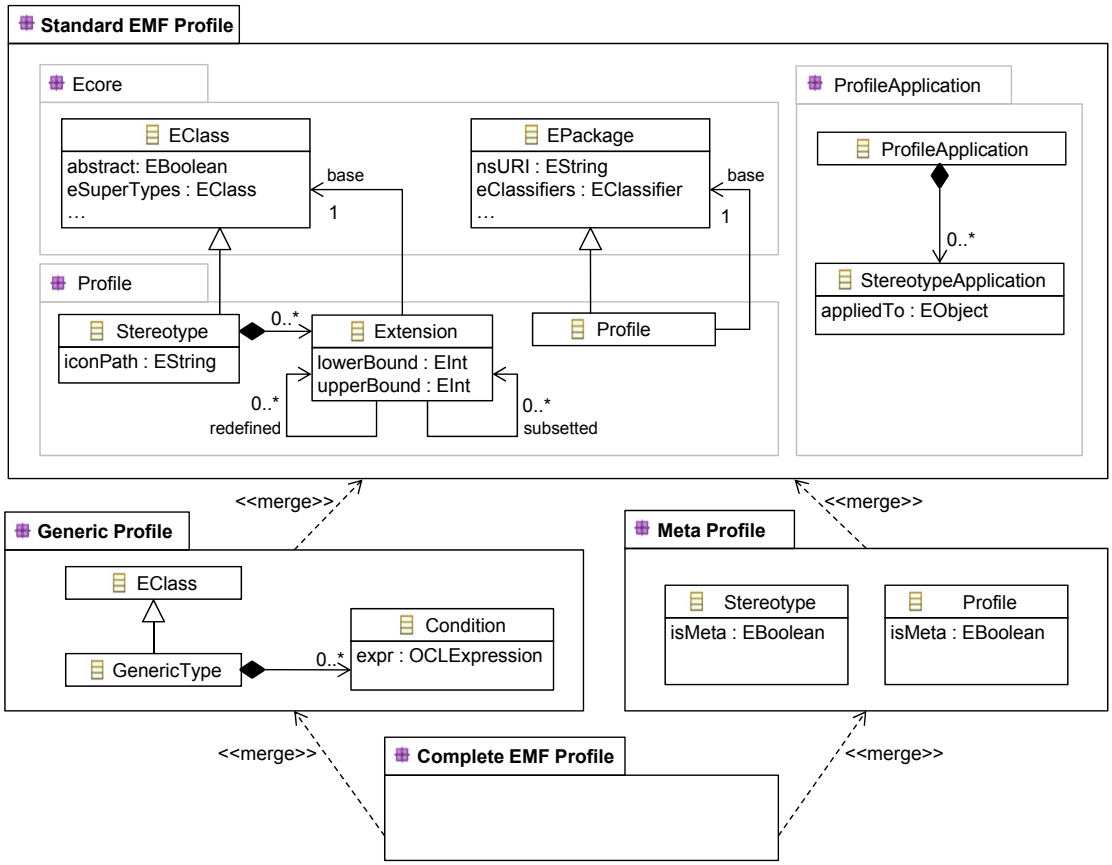

Figure 5.3: Excerpt of the EMF Profile metamodel [20]

ments of the SERAPIS meta-language. The intuitive attempt to tackle this problem is to model the missing elements by employing instances of `EClass`. As a result, the missing language definitions are specified by a metamodel on the M2 meta-layer, instead of the M3 meta-layer where Ecore resides. As a consequence to this problem, an approach is needed to lift the metamodel to the M3 meta-layer.

As mentioned before, the goal of the EMF Profiles framework is to provide a lightweight extension mechanism for Ecore-based domain-specific modeling languages. Analogously to the problem discussed in terms of mapping the meta-languages, the approach initially suggested to create the desired extensions by modeling them in Ecore using instances of `EClass`. As a result, the EMF Profiles framework faces the same problem as the mapping between SERAPIS and Ecore. To overcome this problem, the framework presents two strategies to lift the metamodel.

As both problem domains are strongly related to each other, it seems obvious to employ the EMF Profiles framework to implement an extension for Ecore in order to map the missing SERAPIS language features. The modeling language provided by the framework not only allows to model the desired extensions by establishing profiles, furthermore it provides features to reuse these profiles with different modeling languages. However, as the work presented in this master

thesis is scoped to a single modeling language, namely the SERAPIS meta-language, these additional features for reusing profiles are not subject of further research.

Due to the fact that an additional metamodel, such as the one provided by the EMF Profiles framework, would introduce to much complexity to the SERAPIS2Ecore framework, it is not desirable to apply the EMF Profiles framework with its full functionality. As a consequence, only the concept presented along the framework is adopted in this master thesis, instead of employing the framework itself. This has no effect to the solution of the mapping problem, as the presented strategies for meta-level lifting can be considered as the essential contribution.

The documentation of the EMF Profiles framework [20] presents two strategies for meta-level lifting, namely by transformation and by inheritance. For the implementation of the framework the latter strategy is favored, as it allows the specification of the profile and its instantiation to be defined within the same artifact. Although it is not essential to the SERAPIS2Ecore framework, this quality helps to keep the implementation of the framework compact and therefore benefits maintainability and understandability. As specializations of `EClass` can contain `EAttributes` and `EReferences`, the favored strategy furthermore allows to map the UML concept of tagged values without additional effort. Considering the attributes of the SERAPIS meta-language to be mapped by Ecore as tagged values, this strategy clearly benefits the architecture of the SERAPIS2Ecore framework in terms of simplicity. Taking these benefits into account, the strategy favored by the EMF Profiles framework also seems the suitable mapping strategy providing a solution to the problem addressed in this thesis.

The EMF Profiles framework introduces a profiling mechanism to modeling languages which are based on Ecore. Consequently, the `Profiles` metamodel, as depicted in Figure 5.3, is designed to provide this feature to a set of languages covering their individual requirements. The architecture of EMF Profiles therefore adheres to a generic approach in order to be employed by a set of potentially unknown modeling languages. The mapping of meta-languages as discussed in this chapter however concerns only two concrete languages, namely the SERAPIS meta-language and Ecore. Both languages provide well-known features and as the overall transformation process is conceived to work unidirectional, it suffices to extend Ecore with the missing language elements of the SERAPIS meta-language. As a result, there is no need to take into account the applicability to other languages which is why the mapping approach discussed in this chapter can remain less complex than the concept suggested by the EMF Profiles framework.

Due to the insight gained, the first approach to map the missing elements of the SERAPIS meta-language as described in the Section 5.2 is enhanced with the second meta-level lifting strategy presented by the EMF Profiles framework. Consequently, the SERAPIS meta-language elements modeled in Ecore are specified to directly extend `EClass` with the effect that these elements are granted with the instantiation capability. Due to the fact that the language elements conceptually reside on the same meta-layer as the Ecore meta-language, the *meta-layer transition* issue can be can considered as solved.

According to the rules presented in Table 5.2, some elements of the SERAPIS meta-language can be mapped directly to corresponding elements of Ecore while other SERAPIS language elements are modeled using instances of `EClass`. As described in the context of the *meta-layer merge* issue, the metamodel spanning two meta-layers is not a desirable outcome from the architectural point of view. Due to the enhancement introduced by the EMF Profiles framework,

this issue is also solved as the modeled elements henceforth conceptually reside on the same meta-layer as the Ecore meta-language, thanks to meta-level lifting strategy. Consequently, it is possible to reuse some of the rules depicted in Table 5.2 while also modeling additional elements in Ecore.

The approach also allows to solve the issue of the *missing attributes in `EClass` and `EReference`* as the subtypes of `EClass` can be defined to contain an arbitrary set of attributes. With the remaining issues solved, a metamodel which maps the elements of the SERAPIS meta-language is designed in Ecore. The class diagram in Figure 5.4 depicts the SerapisEcore metamodel extending the original Ecore metamodel.

```
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    name="serapisEcore"
    nsURI="http://tuwien.ac.at/serapisecore/extension"
    nsPrefix="serapisEcore">
  <eClassifiers xsi:type="ecore:EClass"
      name="Element"
      eSuperTypes="platform:[...]//EClass">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="Singleton"
        eType="ecore:EDataType [...]#//EBoolean"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
      name="Property"
      abstract="true"
      eSuperTypes="platform:[...]//EAttribute">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="isRequired"
        eType="ecore:EDataType [...]#//EBoolean"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
      name="Value"
      eSuperTypes="#//Property">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="Default"
        eType="ecore:EDataType [...]#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="ListOfValues"
        eType="ecore:EDataType [...]#//EBoolean"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
      name="Text"
```

```
      eSuperTypes="#//Property">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="Localized"
            eType="ecore:EDataType [...]#//EBoolean"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
        name="Reference"
        eSuperTypes="#//Property">
    <eStructuralFeatures xsi:type="ecore:EReference"
        name="Element"
        eType="#//Element"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
        name="Composition"
        eType="ecore:EDataType platform:[...]//EBoolean"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass"
      name="Enumeration"
      eSuperTypes="#//Property">
    <eStructuralFeatures xsi:type="ecore:EReference"
        name="Enumeration"
        eType="ecore:EClass platform: [...]//EEnum"/>
  </eClassifiers>
</ecore:EPackage>
```

Listing 5.2: SerapisEcore: Ecore definition of the SERAPIS metamodel

The XML notation presented in Listing 5.2, showing the Ecore definition of the SerapisEcore metamodel, has been stripped of namespaces and package definitions for reasons of clarity. As in the code definition depicted, `Element` has been declared to be a subtype of `EClass` in order to gain its instantiation capability. `Properties` on the other hand are specializations of `EAttribute` as they can not only be instantiated, furthermore they are `EStructuralFeatures` and as such `Properties` can be added to `EClasses` respectively `Elements`. This behavior is intended as `Properties` answer the same purpose in the SERAPIS meta-language as `EAttributes` in Ecore.

The new mapping approach theoretically provides enough flexibility to model the complete set of SERAPIS meta-language features in Ecore, including the attributes which have not been determined by the clarification of modeling concepts as necessary for tool integration. Nevertheless, it is still intended to take into account the native attributes provided by the Ecore meta-language with respect to the language mapping. For instance, inheritance relations in Ecore are indicated by the `eSuperTypes` attribute of the `EClass` element. Instead of modeling a corresponding counterpart for the `super` attribute in `Element`, it obviously is preferable to match it to the existing `eSuperTypes` attribute for it actually allows to provide a correct inheritance mapping. This also applies to the `name` attribute which is used to identify types and `isAbstract` indicating whether a type can be instantiated. As the functionality of the `Enum` element provided by both meta-languages overlap in functionality, the metamodel also

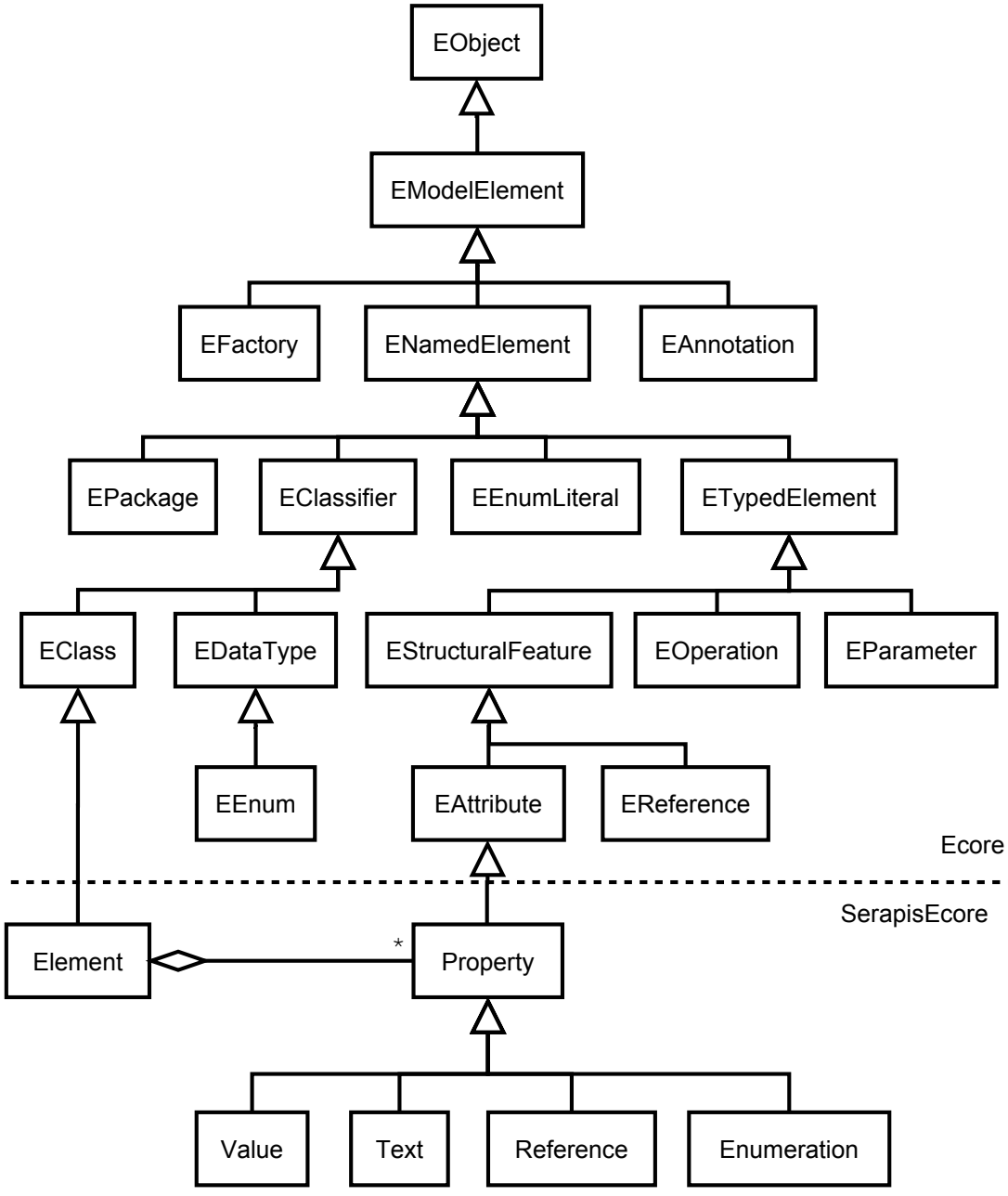

Figure 5.4: The SerapisEcore metamodel extending Ecore

provides no additional definition. Table 5.3 summarizes the new transformation rules affected by the improvements introduced along with the meta-level lifting approach of the EMF Profiles framework. It is to be noticed that the revised transformation rules target the SerapisEcore metamodel extending Ecore in order to access the new modeling features.

For the sake of simplicity, the names of the elements modeled in SerapisEcore correspond to the element names of the SERAPIS meta-language. As a main benefit, it becomes easier to identify the corresponding counterpart for an element in the context of the transformation process. Due to the fact that every element of the SERAPIS meta-language can be non-ambiguously matched to another element of the metamodel defined in Ecore which adheres to the same naming convention and data structure, there is no further need to explicitly define transformation rules.

The transformation approach of Wimmer suggests the definition and application of heuristics to respond to ambiguities in the language mapping. As the approach based on EMF Profiles allows a complete non-ambiguous mapping, there is no need to define heuristics. Consequently, it is not necessary to introduce a phase for manual intervention which strongly simplifies the transformation approach and allows for an automatic generation of the target metamodel.

Rule	**SERAPIS Concept**	**SerapisEcore Concept**
R0	*.name	*.name
R1	Element	Element*M* **extends** EClass Element*M*.interface=false
	Element.super	Element*M*.eSuperTypes
	Element.isAbstract	Element*M*.abstract
R2	Property	Property*M* **extends** EAttribute
	Property.isRequired	**if** Property.isRequired Property*M*.lowerBound = 1 **else** Property*M*.lowerBound = 0
R3	Enum	EEnum
	Enum.literalList	**for each** Literal **add** EEnumLiteral
R4	Literal	EEnumLiteral
		EEnumLiteral.value = # of Literal
R5	Enumeration	Enumeration*M* **extends** Property*M*
	Enumeration.enumeration	Enumeration*M*.enum

Table 5.3: Mapping between the SERAPIS meta-language and SerapisEcore

R0 As depicted in Figure 5.4, `EClass`, `EEnum`, and `EAttribute` extend `ENamedElement` which provides the `name` attribute. Due to the fact that the SERAPIS meta-language can be built entirely in Ecore either by mapping these elements or by extending them, every element of the SerapisEcore metamodel is granted with a `name` attribute. Consequently, the mapping of this attribute is aggregated into one single rule.

R1 This rule has changed only slightly, except that `Element` is mapped to `Element`*M* which is defined in the SerapisEcore metamodel to extend `EClass` in order to inherit its instantiation capability. To recall, the missing attributes of `Element` are not explicitly displayed among the transformation rules as they are mapped to their eponymous counterparts in `Element`*M*, rendering it redundant to introduce particular rules for them. Notice also that the prefix *M* on the right-hand side part of the rules has been introduced only for reasons of clarity.

R2 Instead of mapping `Reference` to `EReference`, this rule is revised to map `Property` to `Property`*M* which extends `EAttribute` in order to gain its instantiation capability and its ability to be attached to `EClass` respectively `Element`*M*. As `Property` is the common supertype for all field types in the SERAPIS meta-language, also their counterparts in the SerapisEcore metamodel inherit these capabilities. Notice that the field types `Value`, `Reference`, `Text`, and `Enumeration` are modeled entirely as new elements of the SerapisEcore metamodel which is why there is no need for additional transformation rules.

R3 and **R4** As the enumeration concept is sufficiently mapped by Ecore, there is no need for a modification of these rules.

R5 This rule has been added in order to avoid confusion resulting from naming conflicts, as the attribute used to specify which `Enum` an entity field of type `Enumeration` refers to is also named `enumeration`. Consequently, this rule maps the attribute `Enumeration.enumeration` to `Enumeration`*M*`.enum` whereas `Enumeration`*M* is a subclass of `Property`*M*.

5.5 Metamodel Generator

As the initial approach for meta-language mapping failed to provide a complete set of transformation rules for SERAPIS and Ecore, the meta-level lifting concept of EMF Profiles was introduced in order to provide an enhanced mapping strategy to cover all elements of the SERAPIS meta-language. An intermediate metamodel referred to as SerapisEcore extending the Ecore meta-language was employed allowing to design custom language elements with instantiation capabilities.

As depicted in Figure 5.5, the enhancement of the mapping approach presented in this chapter introduced a change to the initial architecture of the metamodel transformation approach. Instead of being a direct instance of the Ecore meta-language, the target metamodel generated by the transformation process now corresponds to the SerapisEcore metamodel. However, this does not affect the standard conformity as SerapisEcore conforms to Ecore for being an extension of the meta-language. Therefore, the generated metamodel is still indirectly conform to Ecore and EMF.

The metamodel generator is the component responsible for implementing the entire transformation process between the SERAPIS meta-language and Ecore. The component generates an Ecore-conform metamodel which is used by the model generator to transform model instances

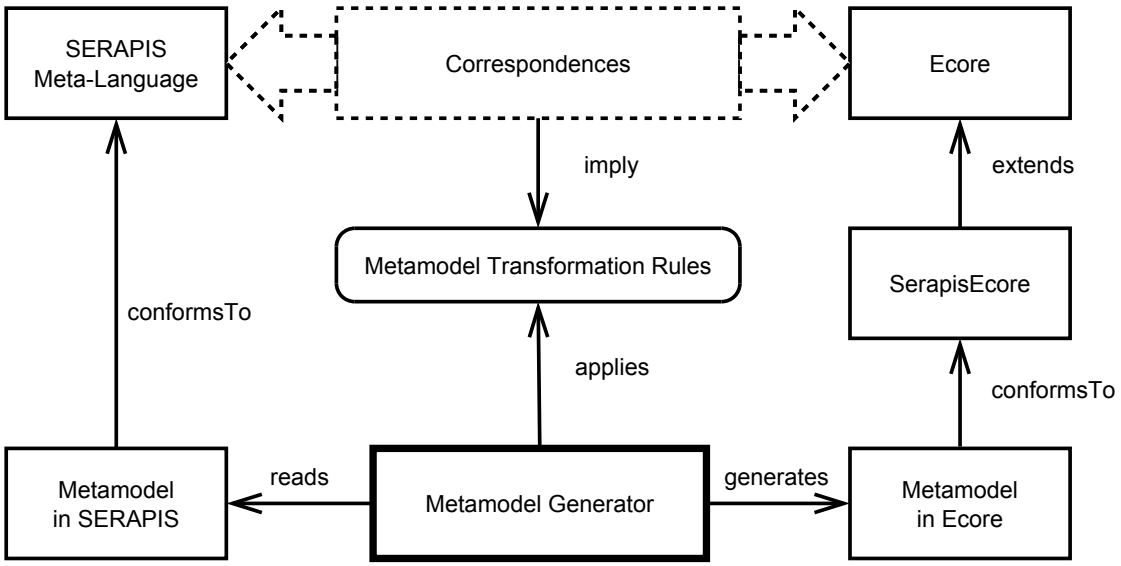

Figure 5.5: Enhanced metamodel transformation approach

of the SERAPIS metamodel. As mentioned before, the business model of Sphinx proposes to provide one default metamodel which is modified adhering to individual requirements. Consequently, each customer employs exactly one metamodel which is why it is sufficient to apply the metamodel generator only once within the development environment of the customer while the model generator is applied within the scope of a single modeling project.

As a first step of the transformation process, the metamodel generator reads the SERAPIS metamodel and examines the elements it provides. The transformation rules *R1* to *R5* are examined by the generator in order to identify a corresponding counterpart in Ecore for each source element of the SERAPIS metamodel. If no adequate rule can be retrieved, the generator assumes that the Ecore extension SerapisEcore provides a matching language element with the same name as the source element. The target element in Ecore then is instantiated according to the value assignments of the source element. The metamodel generator creates the target metamodel in one automatic generation step and needs no further input or intervention due to the complete non-ambiguous language mapping provided by the enhanced mapping approach.

In order to demonstrate the mechanics of the metamodel generator, the excerpt of the SERAPIS metamodel, defining the `Entity` element displayed in Listing 3.2, is used as an example for the transformation of elements from the SERAPIS meta-language to Ecore. For the sake of completeness, this example is enhanced by the definition of an `Enumeration` which is referenced by `Entity`.

Table 5.4 shows `Entity` as an instance of the SERAPIS meta-language element `Element` and its counterpart defined by the SerapisEcore metamodel in Ecore. As a first step of the transformation, the metamodel generator applies rule *R1* of Table 5.3 in order to create an instance of `Element`M which is modeled in the SerapisEcore metamodel. In the resulting Ecore

<element
  name="Entity"
  super="EntityBase"
  class="at.sphinx.sxme.
  informationmodel.model.
  entity.IMEntity"
  transformer="at.sphinx.
  sxme.informationmodel.
  transformer.
  IMEntityTransformer"
  icon="10_Source/icons/
  elements/entity.gif"
  isAbstract="false"
  isSingleton="false">
  ...
</element>
	<eClassifiers
	  xsi:type="s2e:Element"
	  name="Entity"
	  eSuperTypes="#//EntityBase"
	  Class="class at.sphinx.sxme.
	  informationmodel.model.
	  entity.IMEntity">
	  ...
	</eClassifiers>

Table 5.4: Transformation example for `Element`

definition the elements `EClass`, `EDataType`, and `EEnum` are notated as `EClassifier` for being extensions of this element as depicted in Figure 5.4, whereas the `type` attribute refers to the actual subtype. In this example the value of the `type` attribute declares the `EClassifier` as `Element`*M* for it is designed to extend `EClass`.

According to rule *R0*, the `name` attribute can be mapped directly while the supertype and the declaration of `Element`*M* as abstract is mapped to native attributes according to rule *R1*. As *false* is the default value of boolean fields in Ecore, there is no need for the resulting XML notation of the `EClassifier` to explicitly declare `Entity` as not abstract. The attributes `class`, `transformer`, and `icon` are excluded from the transformation process according to the evaluation presented in Table 5.1. `EClass` provides no native attribute allowing it to be declared as a singleton which is why `Element`*M* of the SerapisEcore metamodel introduces an additional attribute for this purpose. As the `Entity` element is not specified as a singleton, the corresponding attribute is not explicitly stated in the definition of the resulting `EClassifier`.

Table 5.5 displays an instance of `Value` named `Abbreviation`, which is a field of the `Element`-instance `Entity` depicted in Table 5.4 and its corresponding definition in Ecore. In the SERAPIS meta-language `Value` is defined as a subtype of `Property` which is the abstract supertype for all fields an `Element` can have. There is no need to define an explicit transformation rule for `Value` as it is modeled in Ecore and therefore mapped to the eponymous element of the SerapisEcore metamodel, here referred to as `Value`*M*. Rule *R2* describes the mapping of `Property` stating that `Value`*M* indirectly inherits from `EAttribute`.

<value name="Abbreviation" class="at.sphinx.sxme.core. value.StringValue" isRequired="true" isDisplayOnly="false" isHidden="false" isUpperCase="true"/>	<eStructuralFeatures xsi:type="s2e:Value" name="Abbreviation" lowerBound="1" eType="ecore:EDataType [...]//EString"/>

Table 5.5: Transformation example for `Value`

From the `Value` definition in the example, the metamodel generator creates an instance of `Value`*M* which is notated in XML as `EStructuralFeature` for it is the supertype of `EAttributes` and `EReferences`. Analogously to `EClassifier`, the actual type of an `EStructuralFeature` is distinguished by the `type` attribute which references `Value` for being a subtype of `EAttribute`. The `name` attribute is inherited from `EAttribute` and therefore can be mapped directly according to rule *R0*. The attribute `isRequired` on the other hand is mapped by setting the `lowerBound` attribute of the generated `EStructuralFeature` to 1. The rest of the attributes is not translated according to the evaluation depicted in Table 5.1.

<enumeration name="Type" enumeration="EntityType" isRequired="true" isDisplayOnly="false" isHidden="false" default="ProductionData"/>	<eStructuralFeatures xsi:type="s2e:Enumeration" name="Type" lowerBound="1" eType="ecore: EDataType [...]//EString" Enum="#//EntityType" default="ProductionData"/>

Table 5.6: Transformation example for `Enumeration`

In Table 5.6 an `Enumeration` named `Type`, which is a field of the `Element`-instance `Entity`, and the corresponding notation in Ecore is depicted. The field references an `Enum` named `EntityType` which is declared as depicted in Table 5.8.

`Enumeration` is mapped to its corresponding counterpart in the SerapisEcore metamodel referred to as `Enumeration`*M* which is modeled in Ecore. `Enumeration`*M* is designed to

provide exactly the same attributes as `Enumeration` with the only exception specified in rule *R5* which states that the attribute `enumeration` is mapped to attribute `enum` in the target element.

In the first step the metamodel generator creates an instance of `Enumeration`$_M$ which is also a subtype of `Property`$_M$ and therefore inherits from `EAttribute`. As a consequence, according to transformation rule *R2* the `lowerBound` attribute is set to 1 as the `isRequired` attribute of the `Enumeration` definition is set to `true`. The `name` attribute is set according to rule *R0*, and `default` is mapped directly while the rest of the attributes is excluded from the transformation process as stated in Table 5.1.

<*reference* name="PrimaryKeyField" element="EntityField" isComposition="false" isRequired="false" isDisplayOnly="true" isHidden="false" isPSM="true" delete="nullify"/>	<*eStructuralFeatures* xsi:type="s2e:Reference" name="PrimaryKeyField" eType="ecore:EDataType [...]//EString" Element="#//EntityField"/>

Table 5.7: Transformation example for `Reference`

Table 5.7 shows the definition of a `Reference` field declared by the `Element`-instance `Entity` and its corresponding definition in Ecore. As there is no specific transformation rule, the metamodel generator assumes the existence of an eponymous language element in the SerapisEcore metamodel, here referred to as `ReferenceM`, and instantiates it. As `Reference`$_M$ is a subtype of `Property`$_M$, `isRequired` is processed according to *R2* although it does not occur in the target definition for its value is set to *false*. The attributes `name` and `element` are mapped directly while the remaining attributes are not included in the transformation process.

The example in Table 5.8 shows the definition of an `Enum` named `EntityType` and its counterpart in Ecore. The metamodel generator first considers transformation rule *R3* and instantiates the corresponding `EEnum` using Ecore. For each `Literal` the `Enum` contains, the generator creates an `EEnumLiteral`. Rule *R4* specifies the construction of the `EEnumLiteral` whereas its `value` attribute is assigned according to at which position the `Literal` is defined by the `Enum`.

<enumeration name="EntityType"> <literal name="ConfigurationData"/> <literal name="MasterData" /> <literal name="ProductionData"/> </enumeration>	<eClassifiers xsi:type="ecore:EEnum" name="EntityType"> <eLiterals name="ConfigurationData"/> <eLiterals name="MasterData" value="1"/> <eLiterals name="ProductionData" value="2"/> </eClassifiers>

Table 5.8: Transformation example for `Enum`

CHAPTER 6

Model Transformation

At the beginning of this chapter, an adequate approach for the transformation problem with respect to SERAPIS and Ecore is researched. As a first step, essential features necessary for grouping and describing transformation approaches are presented. Based on these features, the requirements of the SERAPIS2Ecore model-to-model transformation approach are evaluated. With regard to the result of this evaluation, a set of existing transformation approaches is discussed and the most appropriate match for the requirements of the SERAPIS2Ecore framework is chosen as a foundation for the reference architecture.

The next section takes a glance at the structural design of the SERAPIS model definition in order to determine the expected outcome of the transformation process. The objective is to identify potentially redundant model information which can be excluded from the transformation process. The examination of the SERAPIS model definition also includes observations with respect to linguistic and ontological instantiation which is discussed within the scope of the subsequent section.

At the end of this chapter, the final architecture of the SERAPIS2Ecore model transformation is summarized. This enhanced version includes some minor improvements resulting from observations made with the architecture of the metamodel transformation, presented in the previous chapter. In the context of this summary, the model generator, which is the component responsible for the actual implementation of the model transformation, is discussed in detail. Additionally, the running example is used to demonstrate the exact working mechanics of the generator.

6.1 Model-to-Model Transformation

In order to find an adequate approach for the transformation of models defined by SERAPIS to Ecore-based models, the first objective is to understand the overall transformation problem. The transformation process generally refers to various modeling artifacts residing on different meta-layers and software components which are responsible for the execution of the actual

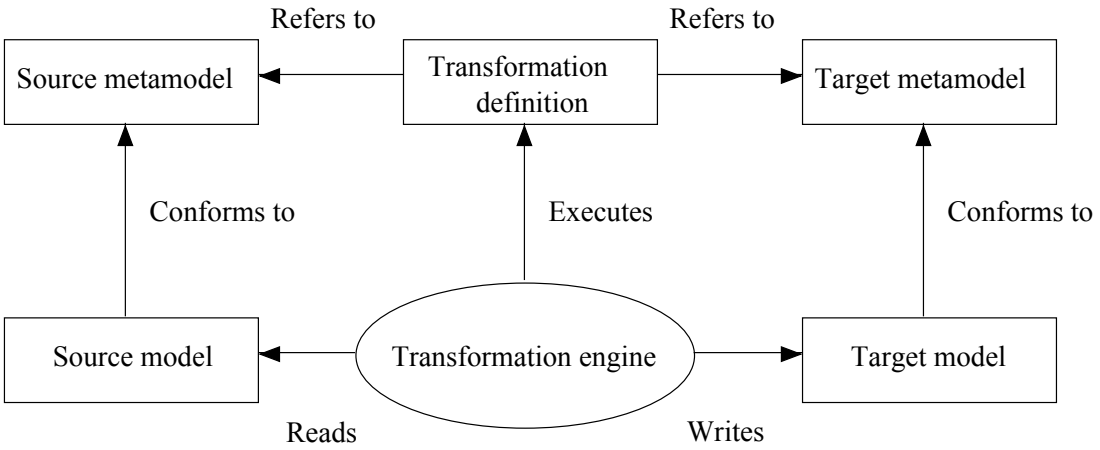

Figure 6.1: Basic concepts of model transformation [12]

transformation. The transformation problem comprises the question which of these modeling artifacts and transformation components are relevant for the transformation process and how the relationships between them are to be defined.

The work of Czarnecki and Helsen [12] describes a blueprint for a general model transformation approach, helpful to understand the transformation problem stated in the context of this master thesis. The architecture depicted in Figure 6.1 comprises the concrete source and target models residing on the M1 meta-layer and the metamodels they conform to. The transformation definition, stating the mapping between source and target elements, refers to the metamodel definition while the actual transformation is applied to the concrete model instances. The architecture specifies a single component referred to as transformation engine which is responsible for the execution of the transformation definition. In general, the transformation is designed to be applicable to multiple source and target models. Besides that, it is possible that the transformation definition refers to the same metamodel both as source and as target.

As the overall model transformation problem has been discussed, an actual approach which is to be applied in order to implement the transformation from SERAPIS models to Ecore-based models has to be chosen. Therefore, the various existing approaches are characterized in order to evaluate their applicability with respect to the transformation problem discussed in this master thesis. The work of Czarnecki and Helsen [12] furthermore presents the following set of features allowing to characterize a transformation approach.

Specification Specification refers to the existence of mechanisms for providing pre- and post-conditions such as those expressed by OCL. This mechanism describes relations between source and target models by defining functions which can be executable in some cases. In this context, the QVT specification distinguishes between the potentially non-executable relation definition and its implementation which can be executed in order to evaluate the relation.

Transformation rules Transformation rules can be considered as the most fine-grained unit of a transformation. A rule generally refers to specific *domains* which are responsible for accessing different models. As a transformation rule usually consists of a left-hand side (LHS) and a right-hand side (RHS) definition, it refers to at least two *domains*. Rules with multiple input *domains* are referred to as *n-way transformations*. Optionally, rules can be enhanced with application conditions which can be evaluated in order to determine their applicability.

Rule application control Rule application control covers the two aspects *scheduling* and *location determination*. *Scheduling* describes mechanisms to define the order of rule applications. Rules can be called implicitly when their application conditions evaluate positively, or explicitly to provide the user with the full control of their application. Additional scheduling strategies such as *rule iteration* mechanisms, providing recursion, looping, fixpoint iteration, or *phasing* which determines the applicability of transformation rules according to defined phases, can be applied to determine the application order. *Location determination* refers to a strategy for specifying the model element a rule is applied to. As a rule can have multiple matches within its source scope, it is necessary to apply a strategy for determining the exact location of an element within the model. For instance, such a strategy may traverse the model according to its containment hierarchy.

Rule organization The term rule organization refers to the structuring and composition of transformation rules. For once, this includes *modularity mechanisms* allowing to organize rules into packages which in turn can be imported in order to access their content. Furthermore, *reuse mechanisms* describe the ability to compose a rule of one or more other rules. This includes inheritance mechanisms between rules respectively packages. Finally, rules can also be organized according to an *organizational structure* which refers to the structure of the elements provided by a language definition. One approach for instance suggests to define one rule for each target element type and to nest the rules according to containment hierarchy specified by the target metamodel.

Source-target relationship This feature describes whether source and target of a transformation refer to the same model instance. While some transformation approaches propose the creation of new models as well as the update of existing models, other approaches are restricted to in-place transformation expecting both source and target model to be the same. ATL for instance mainly allows to create new target models as separate instances of the source model with the effect that in-place transformations need to be simulated by an automatic copy mechanism.

Incrementality *Target-incrementality* which is also referred to as *change propagation* defines the capability of a transformation approach to modify an existing target model according to updates of the source model. Potential implementations usually create the necessary target elements according to the definition of the source model. During further transformation iterations, differences in the source model are derived, for instance by examination of traceability links in order to consequently update the target model. The objective of a transformation approach in terms of *source-incrementality* is to minimize the extent of the

source model that has to be re-examined once it has changed. Enhancements with regard to this change impact analysis especially affect the transformation performance with large-scaled models. The final aspect of the incrementality feature refers to the preservation of user edits in the target. A common scenario requires to apply incremental transformations to target models which have been manually modified between transformation iterations. In this case, it is essential to preserve these modifications from getting lost during the transformation process.

Directionality This feature describes whether transformations can be executed in one direction (unidirectional) or multiple directions (multidirectional). Unidirectional transformations create or update one target model based on one source model. Accordingly, these transformations are based on unidirectional rules with one input model and one output model. Multidirectional transformations on the other hand can be implemented either by multidirectional rules or by multiple several complementary unidirectional rules, providing one rule for each direction.

Tracing The general objective of tracing is to track runtime information resulting from transformation execution. For instance, this includes *traceability links* which are used to track the mapping between source and target domains. To capture *traceability links*, the source elements necessary to create a target element are recorded for each transformation rule. Trace information is necessary to determine how modifications to one model would affect other related models which is referred to as impact analysis. Besides further uses for model synchronization, trace information is useful for debugging models by mapping the stepwise execution of a transformation implementation, and debugging transformation rules themselves. Although some transformation approaches provide a native tracing support, tracing information can be captured generally within any element of the target model which is why this feature can be implemented manually with any approach.

The approach necessary for the transformation from SERAPIS to Ecore can be characterized as a simple unidirectional transformation approach with the single objective to provide a direct mapping for elements specified in one meta-language to another meta-language. As the logic applied in the context of a transformation rule therefore basically is restricted to a copy mechanism, there is no need to express pre- and post-conditions with the effect that the feature of *specification* introduces no essential requirements with respect to the SERAPIS2Ecore model transformation.

As presented in the previous chapter, the meta-level lifting mechanism proposed by the EMF Profiles project allows for a complete and direct mapping between the meta-languages of SERAPIS and EMF. The metamodel generator creates an Ecore-based copy of a metamodel specified by the SERAPIS meta-language. This component therefore also determines the mapping assignments between the resulting metamodels residing on the M2 meta-layer. This mapping information is essential for the model-to-model transformation problem described in this chapter, as it defines the rules for the model transformation on the M1 meta-layer. Therefore, the outcome of the metamodel generation process comprises transformation rules which are applied in the context of the model-to-model generation. As a consequence, the feature of *transforma-*

tion rules only refers to the application of predefined rules, providing no mechanism to define individual ad-hoc rules.

For the model-to-model transformation logic is reduced to a direct-copy mechanism, the model elements involved in the transformation process can be considered as equally biased in terms of their priority which is why there is no need for an explicit *scheduling* of rules. *Location determination*, being the second aspect of the feature *rule application control*, needs to be achieved by identifying the source elements according to their unique element names.

Especially when creating large-scale transformations, the feature of *rule organization* comprises valuable mechanisms to manage and reuse transformation rules in order to achieve understandability and avoid redundancies in rule definitions. As the model-to-model transformation approach of the SERAPIS2Ecore framework is essentially based on automatically generated rules, there is no need to provide readability by organizing transformation rules into modules or packages. For the same reason, reusing transformation rules is not a desirable objective as it would only introduce additional complexity to the generation of rules while providing no explicit benefit with respect to their application.

Both SERAPIS meta-language and Ecore rely on different modeling artifacts and implementations which are not necessarily compliant to each other. As a result, both meta-languages adhere to different technical spaces with the effect that complying models are also implicitly separated by these technical boundaries. With respect to the feature of *source-target relationship*, the SERAPIS2Ecore model transformation approach therefore only needs to consider transformations where source and target models are not the same.

The ability to update an existing target model according to modifications of the source model, while at the same time preserving manual changes in the target model, definitely can be considered as a desirable feature of any model-to-model transformation approach. Also to the `SERAPIS2Ecore` model transformation approach this ability would introduce a valuable benefit, as it seems to be very likely that changes, either resulting from the evolution of the source models or from improvements applied to the transformation component, need to be propagated. However, although it would be desirable to implement this feature, *incrementality* is not in the scope of this master thesis and therefore not included in the architecture of the transformation approach.

The SERAPIS meta-language and its related models are not compliant to any existing standards in model-driven software development which is why the objective of this master thesis is to provide a bridge to EMF allowing SERAPIS models to be accessed by a wider variety of modeling tools. On the other hand, a transformation in the opposite way, namely from EMF to the SERAPIS technical space, is expected to provide no practical benefit. Consequently, the *directionality* of the model-to-model transformation approach in terms of the SERAPIS2Ecore framework is defined to be unidirectional.

As mentioned before, it is not intended to provide an update mechanism for existing target models with the transformation approach researched in this master thesis. Consequently, features for model synchronization and impact analysis, which are strongly related to the implementation of the *traceability* feature, can be considered as redundant. The current version of the transformation is not provided with a *traceability* feature, but an adequate implementation could be realized at any later point of time.

As the features essential to characterize the approach for the SERAPIS2Ecore model transformation have been discussed and evaluated, the next step is to provide an overview of actual transformation approaches in order to pick an approach which is applicable to the transformation problem presented in this chapter. Therefore, the work of Czarnecki and Helsen [12] outlines and discusses various approaches for model-to-text and model-to-model transformation. As the transformation problem tackled in this chapter is scoped to model-to-model transformation, this is the only category to be presented as follows.

Direct-manipulation approaches Direct-manipulation approaches provide an internal model representation and the capability to manipulate the model by exposing an API. Usually, these approaches are designed as object-oriented frameworks which also provide the capability for organizing transformation rules. The implementation of the rules and features such as scheduling or tracing is not within the scope of these frameworks and therefore remains to the user. Besides building required features from scratch, external libraries and frameworks may be included in order to adapt the transformation process. For direct-manipulation provides only very limited support to the user, it can be considered as the most low-level approach.

Structure-driven approaches The execution of the structure-driven approach comprises two phases. During the first phase the elements of the target model and their hierarchical structure are created while the second phase is responsible for setting the attributes and references. The strategy for both application and scheduling of transformation rules is provided by the framework while the user can focus on implementing the rules. Structure-driven approaches have been developed with respect to certain kinds of applications, such as the generation of database schemas from UML models which (1) employ transformations with a one-to-one and one-to-many mapping between source and target elements and (2) require no iteration in the context of the rule application strategy.

Operational approaches Operational approaches are similar to direct-manipulation approaches except that they provide more support for model transformation. Therefore, usually metamodeling formalisms, such as the query language OCL, are extended with imperative constructs in order to allow for expressing computations. The resulting extended executable language becomes an actual object-oriented programming system once combined with a meta-language such as MOF or Ecore.

Template-based approaches This approach is based on templates which are models with embedded metacode. This metacode is specified by annotations which are used to define expressions, conditions, and iterations. Once a template is instantiated, the variable parts of the target model are computed by the metacode. Templates are defined by the concrete syntax of the target language which helps the user to predict the outcome of the transformation. As another benefit, template-based approaches enable an iterative development process by providing the user with the capability to create a sample of the target model in order to derive a template. Although these transformation approaches provide no native tracing mechanisms, it is easily possible to encode tracing information within the templates.

Relational approaches Relational approaches employ constraints to define relations between source and target element types. Although these relations are non-executable by definition, the declarative constraints can be extended with executable semantics. Relational approaches generally require source and target models to be separated strictly. Unlike imperative transformation approaches such as direct-manipulation, elements are created implicitly by these approaches. Relational approaches provide multidirectionality, different types of incrementality, and are especially suitable for model synchronization scenarios.

Graph-transformation-based approaches This category of transformation approaches is based on graph transformation theory and employs typed, attributed, and labeled graphs. These graphs can be considered as a formal representation of simplified class models. Rules employed by graph-transformation comprise a LHS pattern, which is matched to the source model, and a RHS pattern replacing the matched result in order to create the target model. For computing target attribute values, rules are required to provide additional logic. As models can be considered as graphs, these approaches seem to be an obvious choice for model transformations.

Hybrid approaches Hybrid approaches allow to mix the techniques of different transformation approaches in order to meet individual requirements. The assembly of different techniques can be achieved either by combining composites of several approaches, for instance relations and operational mappings, or on the level of transformation rules. Considering the numerous different applications of model-to-model transformation, it is very likely for an approach to be hybrid.

Considering the evaluation of the feature requirements for the SERAPIS2Ecore model transformation, it becomes clear that the model transformation problem presented in this chapter can be tackled with a very small extent of functionality. The source models need to be accessed somehow while the target technical space is expected to provide a mechanism for creating models. Here the SERAPIS modeling environment conveniently provides an API in order to access modeling artifacts. Also EMF provides an API to dynamically create and modify metamodels and models based on the Ecore meta-language. As the approach furthermore is required to employ only a simple direct-copy mechanism, no complex logic or pre- and post-conditions need to be expressed. Considering the absence of any further sophisticated feature requirements, such as traceability or incrementality, the *direct-manipulation* approach seems the best fit for its simplicity and the minimal functional overhead.

In order to provide an implementation for the SERAPIS2Ecore model transformation, the existence of an actual framework which adheres to the concept of *direct-manipulation* transformation is to be identified. A potentially applicable framework is required to provide adapters to access both source and target technical spaces in order to read the artifacts resulting from the SERAPIS modeling tool and to create Ecore-based artifacts, so custom transformation rules based on these adapters can be implemented. Due to the lack of support for the SERAPIS technical space with existing model-driven software development tools, it is not very likely to find a framework which provides the desired capabilities. As a consequence, the absence of a suitable existing framework requires the direct-manipulation approach for the `SERAPIS2Ecore` model transformation to be implemented from scratch.

6.2 Transformation Objectives

As mentioned in the introduction, the objective of this master thesis is not to implement a complete transformation of all modeling information provided by the SERAPIS modeling tool. As described in Chapter 5, partial information of the metamodel lacks significance with respect to a possible migration to other modeling tools which is why this redundant information has been eliminated from the transformation process.

Also in the context of model transformation neither it is intended nor does it make sense to provide a complete mapping for the modeling information. However, while the modification in terms of the metamodel transformation referred to the limitation of certain elements and attributes defined by the metamodel, the modification with the model transformation is more focused on a structural level. The following paragraphs describe these modifications in the form of transformation objectives in detail.

Simplifying References

Models created by the SERAPIS modeling tool are persisted as XML strings in the file system, whereas each instantiated element of the metamodel is stored within a separate file. Consequently, even elements which functionally belong together, such as an `Entity` and all the `EntityFields` it comprises, are spread among different files. For referencing each other, a unique identifier referred to as UUID is assigned to each element. These identifiers also determine the name of the file used to persist an element in order to avoid naming conflicts.

As a result, models can grow very big with respect to the number of files they allocate in the file system as they evolve. Certainly, due to low prices of storage the extent of a modeling project is not to be considered as an essential factor, but this circumstance inconveniently affects the practical application of the SERAPIS modeling tool in another way. Due to the distribution of interrelated elements, a modification in one model element can cause changes in numerous other files. Considering that not only the model definition files but also the source files resulting from code generation reside in the same modeling project, a slight change of a the model can result in hundreds of modified files. Additionally, due to the fact that the model files are named by UUIDs, which have no functional meaning, it becomes virtually impossible to manually merge multiple changes within a model.

The SERAPIS modeling tool assigns unique identifiers in order to avoid confusion due to naming conflicts when referencing model elements. In a SERAPIS model, `Entities` are always assigned to a single package and it is not allowed for multiple eponymous `Entities` to reside within the same package. The name of a package combined with the name of an `Entity` therefore must suffice to uniquely reference an `Entity` instead of using the UUID.

As a consequence, the objective of the model transformation is to simplify references by replacing the UUID assignments with a unique combination of the name of the target `Entity` and the package it resides in. Additionally, the model in the target technical space resulting from the transformation process is designed to contain all model elements within a single file. These measures aim to make models more understandable and consequently enable a manual merge of model changes.

Removing Redundant Model Information

Not all of the model elements defined by the SERAPIS metamodel are actually persisted as modeling artifacts, as the majority of the elements inside the inheritance hierarchy are specified as abstract. From the remaining elements a substantial part of the metamodel serves the purpose to specify tool-specific model information. Among others, this includes build configurations defining the usage of the code generators shipped with the modeling tool and model configurations which are used to store additional meta-information about the model.

Examining the model definition of the *library* example presented in Chapter 3 shows that basically all information which is not tool-specific is comprised by instances of the metamodel elements `Entity` and `EntityField`. This is not surprising, as the instances of `Entity`, such as `Book` described in Listing 3.3, represent the business objects in models of the SERAPIS modeling tool. `EntityFields` are instantiated in order to describe the properties of business objects such as the example depicted in Listing 6.1 representing the property `Title` of `Book`.

With respect to identifying essential information for the model transformation, the tool-specific model information is not expected to provide any benefit in the context of tool-integration. Furthermore, the model-specific information is comprised by the instances of `Entities` and `EntityFields`. As a consequence, the implementation of the model transformation is designed to focus on instances of the mentioned metamodel elements.

Eliminating Attributes Resulting from Linguistic Instantiation

Before describing the next transformation objective, it is necessary to look more closely at the term instantiation. According to the article of Kühne [18], instantiation relationships can be differentiated between ontological instantiation and linguistic instantiation. Ontological instantiation describes an instantiation-relation between two model elements based on their meaning. Linguistic instantiation on the other hand refers to an instantiation-relation between a model element and a linguistic type which specifies a language to define this element.

The example displayed in Figure 6.2 shows the different relationships based on ontological and linguistic instantiation. The model displayed on the lower left is an ontological instance of the model shown on the upper left for the elements of both models are related in terms of their meaning as Frankfurt, Munich, Darmstadt, and Nürnberg are concrete `Cities` connected by the `Roads` A3, A5, and A9. The upper right model on the other hand describes linguistic types specifying a language allowing for an arbitrary `Class` to have `Associations`. Consequently, the element `City` defined in the upper left model inherits the capability of specifying associations referred to as `Roads`. Clearly the elements described by both upper models imply no relationship in terms of their meaning.

Models created by the SERAPIS modeling tool are also the result of both ontological and linguistic instantiation. This becomes clear when examining the fields declared by the `EntityField`-instance in Listing 6.1. As an example, the `EntityField`-instance `Title` contains a `Value` in order to specify its instance-name. This instance-name is indicated by the string assigned to the attribute `name` provided by the `Value`. As depicted in Figure 6.3, the metamodel element `EntityField` inherits this `Value` from the element `NamedElement`. In addition, the same `Value` also provides the attribute `computed` in order to determine

```
<?xml version="1.0" encoding="windows-1252"?>
<sxme:elements xmlns:sxme="http://www.sphinx.at/sxme">
  <element
      display="Title : String80"
      model="ba67a429_dd55_40ad_ab46_08646049d1f0"
      type="EntityField"
      uuid="cb591076_bd70_4c2a_9ccc_f87dd7c8d500"
      psm="false">
    ...
    <value
        name="Name"
        value="Title"
        computed="false" />
    ...
    <reference
        name="Type"
        display="String80"
        model=""
        type="StringType"
        uuid="01a032f1_d39e_41c2_b549_78d6f72f5874"
        index="-1"
        computed="false" />
    ...
    <reference
        name="Entity"
        display="Book"
        model=""
        type="Entity"
        uuid="e9164599_527e_40fd_a8b4_8a1eecd8c24b"
        index="5"
        computed="true" />
    ...
  </element>
</sxme:elements>
```

Listing 6.1: Excerpt of the `Title` model element definition

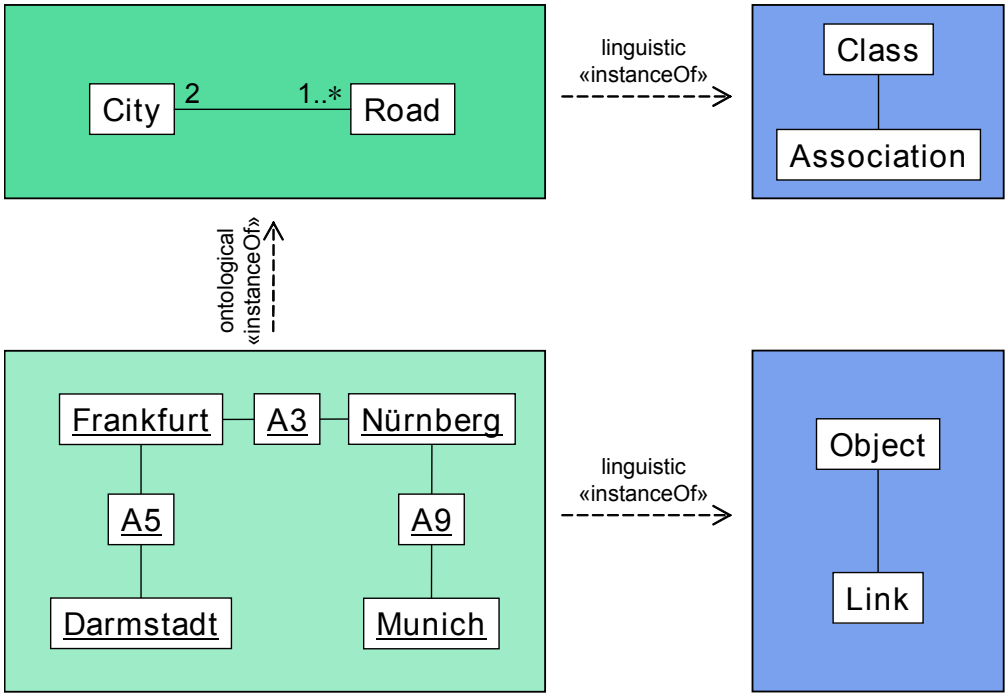

Figure 6.2: Example for linguistic and ontological instantiation [18]

whether its contents are derived from another `Value`. As depicted in the figure, the `Value` also is an instance of the Java class `ModelValue` from which it indirectly inherits this attribute.

The instantiation-relation between `EntityField` and `Title` is ontological as the meaning of an `EntityField` is to be an actual field of an `Entity`, the same way as `Title` is a field of `Book`. On the other hand, the `Value` provided by `Title` is a linguistic instance of `ModelValue`, as this class is a Java type and therefore only specifies the language to create a valid `Value` without considering its meaning.

As the attributes of model elements resulting from linguistic instantiation are not defined in the SERAPIS metamodel, it is also not possible for the metamodel generator to consider them when creating the target metamodel in Ecore. Therefore, instances of the target metamodel lack of these attribute definitions which is why they cannot be included in the model transformation as well. Consequently, this transformation objective states that only attributes resulting from ontological instantiation are considered in the context of model transformation.

6.3 Model Generator

The model generator is a component which implements the model transformation between SERAPIS and EMF. The generator reads an existing SERAPIS model and creates a corresponding counterpart in the target model for each element of the source model. In order to determine which type is to be instantiated, the generator looks up the type of the source element in the

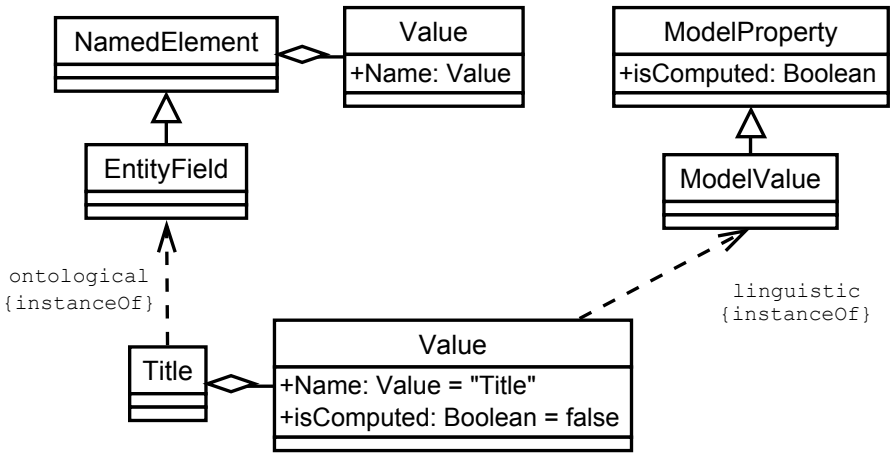

Figure 6.3: Linguistic and ontological instantiation in SERAPIS

source metamodel and matches it to the target metamodel. Therefore, the metamodels in both technical spaces need to be mapped to each other, allowing the generator to retrieve the matching type for each source element.

As discussed in the previous chapter, the meta-languages SERAPIS and Ecore are mapped by both transformation rules and naming conventions allowing the metamodel generator to find the matching correspondence for each type. As the model transformation basically faces the same problem on a different abstraction layer, it is also feasible to adopt this strategy for mapping the metamodels. However, the mapping problem is strongly simplified with respect to the model transformation due to the fact that the elements created by the metamodel generator are instantiated exactly with same names as the elements in the source metamodel. Consequently, it is possible for the model generator to map the elements of both metamodels only by their `name` attributes without the need for transformation rules.

In the SERAPIS modeling tool, instances of metamodel elements provide a `type` attribute in order to keep track of the metamodel element they adhere to. As an example, the `type` attribute of the `Title` element depicted in Listing 6.1 reveals that it is an instance of `EntityField`. As a benefit, the model generator can directly query the model instance for its type in order to find the matching counterpart in the target metamodel without the need to consult the source metamodel. Due to the fact that the model generator needs to consider neither the SERAPIS metamodel nor any transformation rules, its architecture can be simplified as depicted in Figure 6.4. According to the final architecture, the model generator is implemented to start the transformation process with reading the necessary model elements. As required by the transformation objectives, only the elements of type `Entity` and `EntityField` are considered. Depending on the value of the `type` attribute, the model generator then instantiates the corresponding element in the target metamodel SerapisEcore. In order to simplify the references between model elements, all occurrences of UUIDs are replaced by a combination of the package and the name of the target element. All generated elements are contained within a single artifact in order to avoid the distribution of the model over multiple files.

```
<serapisEcore:Entity
  ShortDescription=
    "Short description of
     the Book entity"
  Name="Book"
  Package=
    "ac.at.tuwien.serapis2ecore.
     example.bookstore.model"
  IsAbstract="false"
  IsFinal="false"
  IsSingleton="false"
  Abbreviation="BOOK"
  PersistenceName="BOOK"
  Type="ProductionData"
  PrimaryKeyField="Book.ID"
  AuditInfoField="Book.AuditInfo"
  ...
  />
```

Listing 6.2: The model element `Book` in Ecore

```
<serapisEcore:EntityField
  Name="Title"
  IsRequired="true"
  Type=
    "ac.at.tuwien.serapis2ecore.
     example.bookstore.base.String80"
  PersistenceName="TITLE"
  Entity=
    "ac.at.tuwien.serapis2ecore.
     example.bookstore.model.Book"
  ...
  />
```

Listing 6.3: The model element `Title` in Ecore

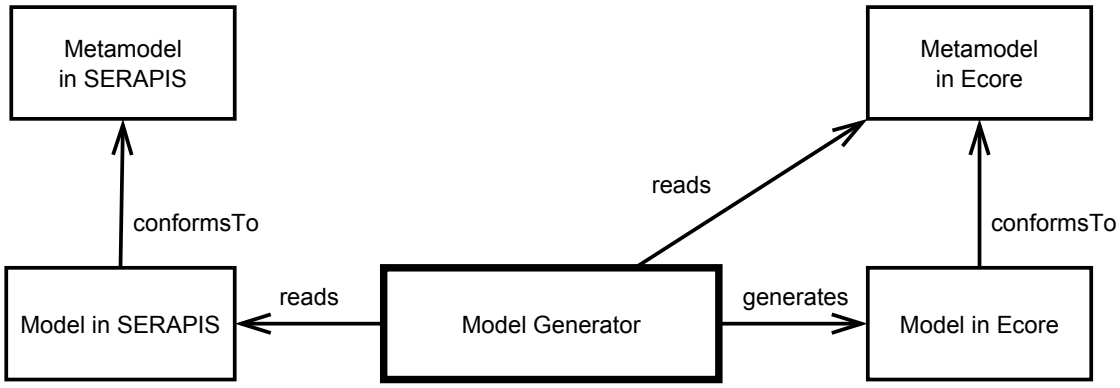

Figure 6.4: Simplified architecture of the model generator

To demonstrate the working mechanics of the model generator, Listing 6.2 shows the transformation result of the `Entity`-instance `Book`, and Listing 6.3 displays the transformation result of `Title` which is an instance of `EntityField` representing a field of `Book`. Due to the extent of the definitions, the listings present only a subset of the attributes provided by each element. The following paragraphs describe the exact transformation logic for each field type an `Entity` can declare.

Value

In SERAPIS models `Values` such as `name`, `isAbstract`, `isFinal`, `isSingleton`, `abbreviation`, and `persistenceName` are designed to represent simple types such as strings or boolean variables. `Value` definitions as the one depicted as follows consist of a `name`, an actual `value`, and an additional attribute `computed` resulting from linguistic instantiation. As the transformation objectives suggest to eliminate attributes such as the latter one, only `name` and `value` need to be transformed. The transformation of `Values` therefore is simple as the remaining two attributes build a natural key-value pair and therefore can be used to specify an attribute with the same name and value in the generated target element as depicted in Listings 6.2 and 6.3.

```
<value
  name="IsAbstract"
  value="false"
  computed="true" />
```

Reference

`References` represent pointers to complex types such as `Entities` in order to map relationships between elements. The following example displays a `Reference` defined by the `EntityField`-instance `Title` referring `Book` as the `Entity` it belongs to. `References`

such as package, primaryKeyField, auditInfoField, Title.type, and entity are using UUIDs in order to specify their target. According to the transformation objectives, the unique identifier is to be replaced by the name of the Entity combined with the name of the package in which it resides. Unfortunately, the package is not included in the reference, and the name of the target Entity is stored in the display attribute which results from linguistic instantiation. Consequently, the implementation of the model generator is required to handle the transformation of references different from other attributes. The generator uses the UUID to look up the target Entity, retrieves its package, and combines the names of both elements in order to provide a string which can be used as the new unique identifier. The model generator specifies the generated element to provide an attribute named after the reference and assigns the derived identifier as value.

```
<reference
  name="Entity"
  display="Book"
  model=""
  type="Entity"
  uuid="e9164599_527e_40fd_a8b4_8a1eecd8c24b"
  index="5"
  computed="true" />
```

Enumeration

Enumerations such as Book.type are very similar to Values with respect to the transformation process as they also provide a natural key-value pair by the attributes name and literal.

```
<enumeration
  name="Type"
  literal="ProductionData"
  computed="true" />
```

Text

Text provides additional documentation and comments such as the attribute shortDescription of Book. The definition depicted as follows seems complex at a first glance, but both attributes computed and language result from linguistic instantiation and therefore are not considered with respect to the transformation. Consequently, the model generator only needs to consider the name and the content of textstring in order to create a corresponding attribute within the generated target element.

```
<localizedtext
    name="ShortDescription"
    computed="false">
  <textstring
```

```
      language="a36eeef6_5e2e_4de4_8bf2_60ff2698dcce">
    Short description of the Book entity
  </textstring>
</localizedtext>
```

CHAPTER 7

Evaluation

The aim of this chapter is to evaluate the outcome of this master thesis which comprises the model transformation architecture and the implementations of both model and metamodel generator. The first question to be researched refers to the feasibility of the metamodel generation. The transformation approach proposed by Wimmer allows for an automatic generation of the metamodel but requires both source and target meta-languages to be mapped at first. The mapping of the SERAPIS meta-language to Ecore is considered as the most significant challenge in the context of this master thesis due to the differences with respect to features and expressiveness provided by both meta-languages. Therefore, the first question examines which measures were adopted in order to achieve a feasible mapping between both meta-languages.

The second question aims to examine the quality of the generated artifacts. Here the term quality refers to the information comprised by the generated metamodels and models rendering the avoidance of information loss as the highest priority of the transformation process. Consequently, source and target models are compared in order to determine the ability of the implemented generators to transform model information without losing significant details.

7.1 Feasibility

In order to evaluate how this master thesis contributes a feasible mapping for the SERAPIS meta-language to Ecore, the steps of finding a technical solution to the mapping problem are depicted in this section. The mapping of the SERAPIS language element `Element` to the Ecore element `EClass` provides a running example to illustrate this process based on trial and error.

The first and most obvious step with finding a feasible mapping for both meta-languages is to determine whether it is possible to identify direct correspondences between the elements both languages provide. Table 7.1 shows that `EClass` natively provides attributes to match a subset of the features required by `Element` such a naming, inheritance, and abstraction. However, the ability to specify the attribute `isSingleton` is restricted to the SERAPIS meta-language element `Element` only.

SERAPIS	Ecore
Element	EClass EClass.interface=false
Element.name	EClass.name
Element.super	EClass.eSuperTypes
Element.isAbstract	EClass.abstract
Element.isSingleton	-

Table 7.1: Correspondences between `Element` and `EClass`

SERAPIS	Ecore
Element	Element*M*
Element.name	Element*M*.name
Element.super	Element*M*.super
Element.isAbstract	Element*M*.isAbstract
Element.isSingleton	Element*M*.isSingleton

Table 7.2: Correspondences between `Element` and `Element`*M*

In order to avoid information loss with respect to the transformation process, an adequate approach to map the `isSingleton` attribute is to be found. Due to the fact that `EClass` provides no native candidate, the only option is to model an element here referred to as `Element`*M* which is specified to provide the missing attribute. As depicted in Table 7.2, the modeled element `Element`*M* is defined to represent an exact mirror for the attributes of `Element`.

Although this approach allows for a complete mapping of `Element` to Ecore, it introduces a major flaw as `Element`*M* is modeled by instantiating `EClass` and therefore resides on the wrong meta-layer. Instead of M3, where also the SERAPIS meta-language resides, the instance of `EClass` is located on the M2 meta-layer whereas, from the architectural point of view, it is considered as a part of the metamodel instead of the meta-language.

As EMF adheres to a three-layer architecture, it only supports the instantiation of the meta-language Ecore to create a compliant metamodel and another instantiation of the metamodel to derive a model. As the approach of mapping `Element` to the modeled `Element`*M* already requires an instantiation of `EClass`, metamodel elements instantiated from `Element`*M* would respectively lose their instantiation capability in order to create compliant models.

SERAPIS	Ecore
Element	Element*M* **extends** EClass
Element.name	EClass.name
Element.super	EClass.eSuperTypes
Element.isAbstract	EClass.abstract
Element.isSingleton	Element*M*.isSingleton

Table 7.3: Correspondences between `Element` and `Element`*M* extending `EClass`

Here the introduction of the meta-level lifting approach proposed by EMF Profiles helps to overcome this problem for it allows to architecturally lift `Element`*M* to the M3 meta-layer. This is achieved by specifying `Element`*M* to extend `EClass` allowing to inherit its instantiation capabilities. Consequently, `Element`*M* resides on the same meta-layer as `Element` and therefore allows for a valid mapping of the SERAPIS meta-language element as depicted in Table 7.3. As a further benefit, `Element`*M* also inherits the naming, inheritance, and abstraction features which can be used for the mapping instead of modeling them.

The sum of all steps described in this section lead to an approach capable of sufficiently mapping the SERAPIS meta-language to Ecore. Although it is not possible to prove that this is the only feasible mapping approach or the most efficient one, each step towards the mapping solution is plausible and comprehensible. This renders the approach not only a good choice for solving the problem described in the context of this master thesis, but furthermore allows for a potential adaptation to similar problem scenarios.

7.2 Quality

The quality of the generated artifacts can be determined by the completeness of information transformed by the generators. The first artifact to be examined is the metamodel resulting from the metamodel generator. In the next step, a model derived from the metamodel and generated by the model generator is inspected. As both artifacts result from an automatic generation process, a manual review is considered as the most adequate tool for the evaluation which is performed based on a representative case study.

Metamodel

In order to evaluate the completeness of the generated metamodel, it is neither possible nor necessary to examine each of the over 200 elements defined by the SERAPIS metamodel. Instead, it is sufficient to compare only one element of the SERAPIS metamodel with its generated counterpart as the mechanics and the rules applied by the metamodel generator have the same impact to every element.

The element `Type`, depicted in Figure 3.5 as a superclass of `Entity`, is used as an example to evaluate the completeness of the metamodel transformation. Subclasses of the abstract class `Type` inherit the ability to define inheritance-relations, specify packages they belong to, state whether they are serializable, and define additional textual attributes for documentation purposes.

Table 7.4 shows the definition of `Type` and the resulting notation in Ecore. The name, the superclass, and the declaration as abstract are mapped to the native counterparts provided by `EClass` while `isSingleton` is mapped directly to the eponymous attribute. As `class` and `icon` are intentionally excluded from the transformation process, the generated result suffers no loss of information.

```
<element
  name="Type"
  super=
    "GeneratedArtefactElement"
  class=
    "at.sphinx.sxme.
     informationmodel.
     model.IMType"
  icon=
    "10_Source/icons/
     elements/type.gif"
  isAbstract="true"
  isSingleton="true">
  ...
</element>
```

```
<eClassifiers
  xsi:type=
    "serapisEcore:Element"
  name="Type"
  abstract="true"
  eSuperTypes=
   "#//GeneratedArtefact
    Element"
  isSingleton="true">
  ...
</eClassifiers>
```

Table 7.4: Transformation of `Type`

```
<reference
  name="SuperType"
  element="Type"
  isComposition="true"
  isRequired="false"
  isDisplayOnly="false"
  isHidden="false"
  delete="inhibit"/>
<reference
  name="Package"
  element="Package"
  isComposition="false"
  isRequired="true"
  isDisplayOnly="false"
  isHidden="false"
  delete="inhibit"/>
```

```
<eStructuralFeatures
  xsi:type=
    "serapisEcore:Reference"
  name="SuperType"
  eType="ecore:EDataType
    [...]//EString"
  Element="#//Type"
  IsComposition="true"/>
<eStructuralFeatures
  xsi:type=
    "serapisEcore:Reference"
  name="Package"
  lowerBound="1"
  eType="ecore:EDataType
    [...]//EString"
  Element="#//Package"/>
```

Table 7.5: Transformation of `SuperType` and `Package`

<value name="IsSerializable" class= "at.sphinx.sxme. core.value.BooleanValue" isRequired="true" isDisplayOnly="false" isHidden="false" *default*="false"/>	<*eStructuralFeatures* xsi:type= "serapisEcore:Value" name="IsSerializable" lowerBound="1" eType="ecore:EDataType [...]//EString" Default="false"/>

Table 7.6: Transformation of `IsSerializable`

The notations depicted in Table 7.5 refer to `SuperType` and `Package`, two `References` defined by `Type`. In both elements the attributes `name` and `element` are mapped while `isDisplayOnly`, `isHidden`, and `delete` are eliminated during the transformation process. The attribute `isComposition` is only defined by the `Reference` named `SuperType` for it has assigned the value *true*. `IsRequired` on the other hand has no direct counterpart in the generated target definitions, instead the attribute `lowerBound` is set to 1. As all attributes which have been determined to be involved in the generation process are transformed properly, there is no indication of information loss with respect to `References`.

Table 7.6 shows the source definition of a `Value` in the SERAPIS metamodel and the generated target definition in Ecore. The transformation of the `Value` named `IsSerializable` seems complete for it misses only the attributes `class`, `isDisplayOnly`, and `isHidden` which are specified to be ignored by the metamodel generator.

The `Text` elements named `Label` and `Tooltip` depicted in Table 7.7 are defined by `Type` for the purpose of documentation. Except for `name`, `isLocalized`, and `isRequired`, no further attributes are defined to be covered by the metamodel generator which is why the transformation of `Text` elements can be considered as complete.

In conclusion the manual review has shown that the transformation process causes no loss of information, assuming that this information has been determined to provide a benefit to the resulting metamodel. Although the evaluation presented in this section provides only one element as a running example, the manual review has become an integrated part of the implementation process of the metamodel generator which is why in practice the evaluation covers numerous elements in order to ensure the completeness of the metamodel.

Model

The evaluation of the generated model follows the same principle as the evaluation of the generated metamodel. As the model generator also transforms each model element according to the

<text name="Label" lines="1" isLocalized="true" isDescription="false" isRequired="true" isDisplayOnly="false" isHidden="false" isPSM="false"/> <text name="Tooltip" lines="1" isLocalized="true" isDescription="false" isRequired="false" isDisplayOnly="false" isHidden="false isPSM="false"/>	<eStructuralFeatures xsi:type= "serapisEcore:Text" name="Label" lowerBound="1" eType= "ecore:EDataType [...]//EString" Localized="true"/> <eStructuralFeatures xsi:type= "serapisEcore:Text" name="Tooltip" eType= "ecore:EDataType [...]//EString" Localized="true"/>

Table 7.7: Transformation of `Label` and `Tooltip`

same rules, it is sufficient to manually review only one element in detail. Listing 7.1 depicts the definition of the model element `Customer` which is a part of the *library* model and serves as an example to evaluate the completeness of the transformation.

At a first glance, the generated Ecore-definition of `Customer` notated in Listing 7.2 appears relatively small compared to the extent of the element definition in SERAPIS. This is due to the fact that a substantial part of the attributes defined in the SERAPIS model results from linguistic instantiation and therefore is eliminated by the transformation process as these attributes are not specified in the metamodel.

The `type` attribute in the outer XML tag `element` in the SERAPIS definition of `Customer` specifies the model element to be an `Entity` which is why the model generator instantiates an `Entity` of the SerapisEcore metamodel as depicted in the first line of Listing 7.2. None of the other attributes specified in this XML tag are considered in the transformation process as they result from linguistic instantiation.

The element contains numerous `Properties` of type `Text`, `Value`, `Reference`, and `Enumeration`. For each of these `Properties` the generated target `Entity` is specified to provide an attribute according to the `name` of the `Property` it is mapping. The retrieval of the values assigned to these attributes depends on the type of the `Properties`.

The values assigned to `Properties` of type `Text` such as `ShortDescription` are stored within the `textstring` tag. The attributes `computed` and `language` are not speci-

fied to be a part of the target model which is why the transformation process loses no information as only the content of `textstring` needs to be transformed.

`Values` such as `Name` provide an attribute `value` to hold their actual value while `Enumerations` such as `Type` specify `literal` for the same purpose. Processing these attributes is sufficient to allow for a complete transformation as the attribute `computed` is ignored with both `Values` and `Enumerations`.

`References` are not transformed based on a single attribute which is responsible for holding the actual value, instead the UUID is used to look up the target element. The value assigned to the mapped attribute is derived from the name of the target element and the package it resides in. As this value is sufficient to unambiguously qualify a reference to another element, the transformation with respect to `References` is considered to lose no information.

Although the generated model appears much simpler than the source model, the manual review indicates that the transformation process provides a complete mapping for information considered as essential.

```
<?xml version="1.0" encoding="windows-1252"?>
<sxme:elements xmlns:sxme="http://www.sphinx.at/sxme">
  <element display="Customer"
      model="ba67a429_dd55_40ad_ab46_08646049d1f0"
      type="Entity"
      uuid="52cb7a20_dbf5_44d5_96e9_5b88a7d5e303"
      psm="false">
    <localizedtext name="ShortDescription"
        computed="false">
      <textstring
          language="a36eeef6_5e2e_4de4_8bf2_60ff2698dcce">
      Library Customer
      </textstring>
    </localizedtext>
    <localizedtext name="AnalysisDescription"
        computed="true" />
    <localizedtext name="DesignDescription"
        computed="true" />
    <localizedtext name="Comment" computed="true" />
    <value name="Name" value="Customer"
        computed="false" />
    <reference name="SuperType" display="Person" model=""
        type="Entity"
        uuid="21eb8177_3970_4e2a_a8f2_0ae5dc2db988"
        index="-1" computed="false" />
    <reference name="Package"
        display="ac.at.tuwien.serapis2ecore.
```

```
      example.bookstore.model"
    model="" type="Package"
    uuid="20056580_178d_41b1_a4c6_6e14cb439772"
    index="-1" computed="true" />
<value name="IsSerializable" value="false"
    computed="true" />
<localizedtext name="Label" computed="true" />
<localizedtext name="Tooltip" computed="true" />
<value name="IsAbstract" value="false" computed="true" />
<value name="IsFinal" value="false" computed="true" />
<value name="IsSingleton" value="false" computed="true" />
<value name="HasManualCode" value="false"
    computed="true" />
<value name="HasManualCodeHidden"
    value="false" computed="true" />
<localizedtext name="LabelPlural" computed="true" />
<value name="HasManualDaoCode"
    value="false" computed="true" />
<value name="HasManualDaoCodeHidden"
    value="false" computed="true" />
<value name="HasManualServiceCode"
    value="false" computed="true" />
<value name="HasManualServiceCodeHidden"
    value="false" computed="true" />
<value name="Abbreviation" value="CUST"
    computed="true" />
<value name="PersistenceName"
    value="CUSTOMER" computed="true" />
<enumeration name="Type"
    literal="ProductionData" computed="true" />
<value name="IsTimeDependent" value="false"
    computed="true" />
<value name="IsHistorized" value="false"
    computed="true" />
<value name="MinVolume" value="" computed="true" />
<value name="AvgVolume" value="" computed="true" />
<value name="MaxVolume" value="" computed="true" />
<value name="VolumeVolatility" value="" computed="true" />
<reference name="PrimaryKeyField" display="" model=""
    type="" uuid="" index="-1" computed="true" />
<reference name="EntityKeyField" display="" model=""
    type="" uuid="" index="-1" computed="true" />
<reference name="VersionNoField" display="" model=""
```

```
        type="" uuid="" index="-1" computed="true" />
    <reference name="AuditInfoField" display="" model=""
        type="" uuid="" index="-1" computed="true" />
    <reference name="HistoryIntervalField" display="" model=""
        type="" uuid="" index="-1" computed="true" />
    <reference name="ValidIntervalField" display="" model=""
        type="" uuid="" index="-1" computed="true" />
  </element>
</sxme:elements>
```

Listing 7.1: Definition of model `Customer` in SERAPIS

```
<serapisEcore:Entity
  ShortDescription="Library Customer"
  AnalysisDescription=""
  DesignDescription=""
  Comment=""
  Name="Customer"
  IsSerializable="false"
  Label=""
  Tooltip=""
  SuperType="ac.at.tuwien.serapis2ecore.
    example.bookstore.model.Person"
  Package="ac.at.tuwien.serapis2ecore.
    example.bookstore.model"
  IsAbstract="false"
  IsFinal="false"
  IsSingleton="false"
  HasManualCode="false"
  HasManualCodeHidden="false"
  LabelPlural=""
  HasManualDaoCode="false"
  HasManualDaoCodeHidden="false"
  HasManualServiceCode="false"
  HasManualServiceCodeHidden="false"
  Abbreviation="CUST"
  PersistenceName="CUSTOMER"
  Type="ProductionData"
  IsTimeDependent="false"
  IsHistorized="false"
  MinVolume=""
  AvgVolume=""
  MaxVolume=""
  VolumeVolatility=""
```

```
PrimaryKeyField=""
EntityKeyField=""
VersionNoField=""
AuditInfoField=""
HistoryIntervalField=""
ValidIntervalField=""/>
```

Listing 7.2: Definition of model `Customer` in Ecore

CHAPTER 8

Conclusion

This final chapter concludes this master thesis as it summarizes the contribution of this thesis and gives a brief outlook to further related topics of research.

8.1 Contributions of the Thesis

In this master thesis, we presented the SERAPIS modeling tool and the three-layer architecture it is based on, comprising meta-language, metamodels, and models. We discussed issues of the proprietary tool, underlining the necessity to bridge models to the more standard-conform technical space of EMF in order to overcome the vendor-lock. With respect to building this bridge, a semi-automatic approach allowing to transform metamodels between two technical spaces based on the correspondences of the meta-languages both metamodels comply to was discussed. We adopted this approach in form of a metamodel generator allowing to transform metamodels specified by the SERAPIS meta-language to Ecore. The adoption of this semi-automatic approach proposed by Wimmer proved a valuable asset as it allowed to significantly streamline the process of creating the target metamodel.

The major challenge of this master thesis was to find correspondences between the SERAPIS meta-language and Ecore with respect to the design of the metamodel generator. Due to the distinct expressiveness of the features provided by both meta-languages, it was impossible to achieve a direct mapping between the elements of both languages. Only the insight gained from the meta-level lifting approach contributed by the EMF Profiles project rendered the establishment of an adequate mapping technique possible.

Subsequently, we extended the architecture of the bridge between SERAPIS and EMF with a model generator responsible for transforming existing models to the target technical space. Although the working principle of the model generator corresponds to the metamodel generator, except for concerning a different meta-layer, the design of the model generator was much simpler as it was possible to directly match the metamodels of both technical spaces.

A successful implementation of both model and metamodel generator in Java helped to evaluate the feasibility of the designed transformation architecture. In addition, we reviewed the

generated modeling artifacts manually in order to determine the completeness of the transformed information. The reviews revealed that the quality of the generated results matches the objectives established in the context of the design process.

In conclusion, the insights gained from the metamodel transformation approach of Wimmer and the meta-level lifting approach of the EMF Profiles project constitute a valuable knowledge in the field of model-driven engineering allowing a wide variety of applications with respect to tool integration scenarios as presented in the context of the SERAPIS modeling tool. The major contribution of this master thesis is constituted by the combination of the knowledge gained from these existing approaches in order to enable a successful bridging of two different technical spaces such as SERAPIS and EMF.

8.2 Outlook

With the models successfully transformed to the EMF technical space, the only remaining issue concerns the functionality of generating code from the transformed artifacts. The SERAPIS modeling tool comes with a set of code generators allowing to generate Java code and SQL DDL definitions from models. Usually a substantial amount of effort is put into developing these generators as they need to be adapted to the individual requirements of a customer. For instance, special code generators for producing Java code might be developed allowing to generate certain programming constructs for converting, retrieving, or instantiating business objects. Consequently, the logic merged into the implementation of the generators can be considered as something that is worth to be preserved in a tool integration scenario as presented in this master thesis.

The code generators of the SERAPIS modeling tool are designed to inspect the Java objects, representing the model elements at runtime, in order to derive executable code. The rules for deriving this code are hard-wired into the logic of the generators which are implemented in Java as well. As EMF is also based on Java, it would be possible to import the libraries containing the generators in order to reuse their functionality with the models transformed to the EMF technical space. As the code generators are expecting the processed Java objects to represent model elements of the SERAPIS technical space, converters would be needed to match the structure of the Java objects representing the Ecore model elements to their counterparts in SERAPIS.

The approach described to reuse the SERAPIS code generators could be implemented without much effort as a converter would be required to provide only a one-to-one mapping between the attributes of two Java objects without considering any kind of business logic. Alternatively, due to the simple nature of the converting mechanism it would also probably be feasible to create these converters along with the transformation process of SERAPIS models to Ecore.

Reusing the functionality of the code generators by applying this approach preserves customers from losing substantial effort spent into the adaptation of the SERAPIS modeling tool in the case of an integration with EMF. This approach especially provides benefits on a short-term basis as it allows to quickly get back into production after the tool integration.

In the long term, reusing the code generators provided by SERAPIS however could cause inconveniences with respect to future modifications of the generation process. Changing the out-

come of the generation results in the manipulation of the rules implemented in the Java source code of the generator. This interference with the interiors of the generation logic is less flexible than applying a template-based code generation approach for instance. Therefore, it is recommendable to take into account the limited flexibility when reusing the code generators.

Bibliography

[1] AtlanMod. http://www.emn.fr/z-info/atlanmod/index.php/Main_Page, 2012. Online; accessed 01-07-2012.

[2] Eclipse. http://www.eclipse.org, 2012. Online; accessed 15-09-2012.

[3] Eclipse resources. http://www.eclipse.org/resources/?category=Extension%20points, 2012. Online; accessed 01-10-2012.

[4] Modelum. http://www.modelum.es/, 2012. Online; accessed 01-08-2012.

[5] Sphinx IT Consulting. http://www.sphinx.at, 2012. Online; accessed 01-09-2012.

[6] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *13th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2010.

[7] M. Biehl. Literature Study on Model Transformations. Technical report, Royal Institute of Technology, 2010.

[8] M. Biehl, C. Sjöstedt, and M. Törngren. A modular tool integration approach: Experiences from two case studies. In *3rd Workshop on Model-driven Tool and Process Integration (MDTPI)*, 2010.

[9] A. Breslav. Java2Ecore. http://code.google.com/p/java2ecore/, 2012. Online; accessed 01-09-2012.

[10] H. Bruneliere, J. Cabot, C. Clasen, F. Jouault, and J. Bézivin. Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools. In T. Kühne, B. Selic, M. Gervais, and F. Terrier, editors, *European Conference on Modelling Foundations and Applications (ECMFA)*, volume 6138 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2010.

[11] Sphinx IT Consulting. MDSD und SERAPIS. http://www.sphinx.at/fileadmin/downloads/factsheets_whitepaper_folder/SERAPIS_MDSD_Whitepaper.pdf, 2011. Online; accessed 01-07-2012.

[12] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3), 2006.

[13] D. Gasevic, D. Djuric, and V. Devedzic. *Model Driven Engineering and Ontology Development*. Springer, 2nd edition, 2009.

[14] J. Izquierdo, F. Jouault, J. Cabot, and J. Molina. API2MoL: Automating the building of bridges between APIs and Model Driven Engineering. *Information and Software Technology*, 54(3), 2012.

[15] H. Kern, H. Kremß, and S. Kühne. Modellinteroperabilität zwischen Microsoft Visio und Eclipse EMF als Mittel zur modellgetriebenen Integration. In S. Fischer, E. Maehle, and R. Reischuk, editors, *GI Jahrestagung*, volume 154 of *Lecture Notes in Informatics*, pages 3303–3307. GI, 2009.

[16] H. Kern and S. Kühne. Model Interchange between ARIS and Eclipse EMF. In J. Tolvanen, J. Gray, M. Rossi, and J. Sprinkle, editors, *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM)*, number TR-38 in Computer Science and Information System Reports, Technical Reports, pages 105–114, Finland, 2007. University of Jyväskylä.

[17] M. Kolovos, R. Paige, L. Rose, and F. Polack. The Epsilon Book. `http://dev.eclipse.org/svnroot/modeling/org.eclipse.epsilon/trunk/doc/org.eclipse.epsilon.book/EpsilonBook.pdf`, 2012. Online; accessed 01-12-2012.

[18] T. Kühne. Matters of (Meta-) Modeling. *Software & Systems Modeling*, 5(4), 2006.

[19] I. Kurtev, J. Bezivin, and M. Aksit. Technological spaces: An initial appraisal. In *International Conference on Cooperative Information Systems (CoopIS)*, 2002.

[20] P. Langer, K. Wieland, M. Wimmer, and J. Cabot. EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, 11(1), April 2012.

[21] S. Liddle. *Model-Driven Software Development*, pages 17–54. Springer, 2011.

[22] OMG. *Model Driven Architecture (MDA) Specification*, 2000.

[23] OMG. *Meta Object Facility (MOF) Specification*, 2002.

[24] OMG. *UML 2.0 Infrastructure Specification*, 2003.

[25] OMG. *Object Constraint Language (OCL) Specification*, 2010.

[26] OMG. *XML Metadata Interchange (XMI) Specification*, 2010.

[27] OMG. Object Management Group. `http://www.omg.org/`, 2012. Online; accessed 01-09-2012.

[28] AtlanMod Research Group and Modelum Research Group. API2MoL. `http://code.google.com/a/eclipselabs.org/p/api2mol/`, 2012. Online; accessed 01-15-2012.

[29] A. Schauerhuber, M. Wimmer, E. Kapsammer, W. Schwinger, and W. Retschitzegger. Bridging WebML to model-driven engineering: from document type definitions to meta object facility. *IET Software*, 1(3), 2007.

[30] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[31] M. Wimmer. *From Mining to Mapping and Roundtrip Transformations - A Systematic Approach to Model-based Tool Integration*. PhD thesis, Vienna University of Technology, 2008.

[32] A. Wolff and P. Forbrig. Deriving EMF Models from Java Source Code. In *Working Conference on Reverse Engineering Models from Software Artifacts (WCRE)*, 2009.