EMF Ecore Based Meta Model Evolution and Model Co-Evolution

Article · January 2008

CITATIONS
4

READS
320

3 authors:

Moritz Eysholdt
TypeFox GmbH
10 PUBLICATIONS **399** CITATIONS
SEE PROFILE

Soeren Frey
Daimler TSS
25 PUBLICATIONS **521** CITATIONS
SEE PROFILE

Wilhelm Hasselbring
Christian-Albrechts-Universität zu Kiel
371 PUBLICATIONS **4,631** CITATIONS
SEE PROFILE

CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Master's Thesis

EMF Ecore Based Meta Model Evolution and Model Co-Evolution

Moritz Eysholdt

April 20, 2009

First examiner	Prof. Dr. Wilhelm Hasselbring
Second examiner	Heiko Niemann
First advisor	Sven Efftinge
Second advisor	Tammo Freese

Abstract

When algorithms work on information, there is data that follows a formally defined structure. If this structure definition changes over time, what must happen to the data to preserve the underlying information?

This is the fundamental question this thesis aims to solve for the Eclipse Modeling Framework (EMF), whereas EMF Ecore is the language to formally define data structures. This scenario occurs when an application, which persists data, is being evolved and needs to co-evolve the existing data. Furthermore, this scenario can be seen when applications exchange data over a network and implement different versions of the protocol/interface. The need to automatically migrate the data creates the need for a migration algorithm. This thesis focuses on creating such a model migration algorithm based on the differences between two meta models, which is introduced as the diff-approach. As opposed to the operation-approach, which records all editing steps the developer uses to modify the meta model, the diff-approach requires no editor integration.

The implementation provided by this thesis consists of two Xtext-based Domain Specific Languages (DSLs), the Epatch and the Metapatch, and the needed tooling for creating and interpreting instances of these languages. The *Epatch* resembles the patch format (which is known from the UNIX world) for models. Thereby, the Epatch is bidirectional, declarative, self-contained, and meta model agnostic. It can be created by comparing models as well as by recording changes which are applied to a model. By applying a patch, model B can be created from model A as well as model A can be recreated from model B. The *Metapatch* extends the Epatch via grammar inheritance: It restricts the Epatch to meta models (Ecore models) and additionally defines how to migrate models between the two meta models. For many situations, the correct strategy to migrate the models can be determined implicitly based on the mapping between the meta model elements. However, the Metapatch allows to call Java methods or integrate Xtend Expressions to customize the migration algorithm. Thereby, the Metapatch becomes a model to model transformation language, which is optimized for scenarios where models need to be migrated between similar meta models: Its size and thereby its complexity is proportional to the amount of changes – as opposed to being proportional to the amount of meta model concepts.

The thesis introduces models as an approach to generalize data structures by using the formalism of meta models to allow generic implementations for standard operations, such as serialization, deserialization, validation, transformation, comparison – and, as of this thesis: co-evolution. It furthermore introduces the usage of models in software engineering (MDSD, MDA, DSM, etc.) as a field of application for meta model evolution and as an approach which is applied when developing the Epatch and the Metapatch. This thesis explores the occurrence of evolution and co-evolution in biology and computer science and compares both fields. Then, it introduces related work and identifies challenges, classifications and approaches from the field of meta model evolution and model co-evolution. After introducing the Epatch and the Metapach with their requirements, design decisions, meta models, textual representations and implementation details, they are evaluated: The quality of the model migration algorithm is judged with regards to its runtime performance, the complexity the developer has to handle and the possible degree of automation in the creation process, as well as its completeness and correctness.

Contents

Declaration

This thesis is my own work and contains no material that has been accepted for the award of any other degree or diploma in any university.

To the best of my knowledge and belief, this thesis contains no material previously published by any other person except where due acknowledgment has been made.

Place, Date, Signature

Acknowledgement

I would like to thank the company itemis and Sven Efftinge for offering a great environment to work and to share expertise.

I would like to thank the following people for their professional advice and/or linguistic support[1].

- Constantin Radjapaksa
- Eliana de la Rosa
- Heiko Behrens, itemis
- Dr. Jan Köhnlein, itemis
- Markus Herrmannsdörfer, Technische Universität München, Germany
- Olesea Stirbu
- Said M. Marouf
- Sebastian Zarnekow, itemis
- Sven Efftinge, itemis
- Tammo Freese
- Torsten Krohn, itemis
- Prof. Dr. Wilhelm Hasselbring, Universität Kiel, Germany

Furthermore, I am thankful for the coincidence that Prof. Hasselbring moved to the University of Kiel, Germany, at the same time as I moved to Kiel for writing this thesis.

[1] The order of this list is alphabetical.

1 Introduction

When algorithms work on information, there is data that follows a formally defined structure. If this structure definition changes over time, what must happen to the data to preserve the underlying information?

This is the fundamental question this thesis aims to solve for the Eclipse Modeling Framework (EMF, [Ecle]), whereas EMF Ecore is the language to formally define data structures[1]. This introduction starts with a lightweight description of the problem and continues with outlining its context within research and with regards to existing technology. To provide an overview, the thesis' contents is outlined and it is explained what is not covered by this work and how it differs from existing work.

1.1 An Introductory Example

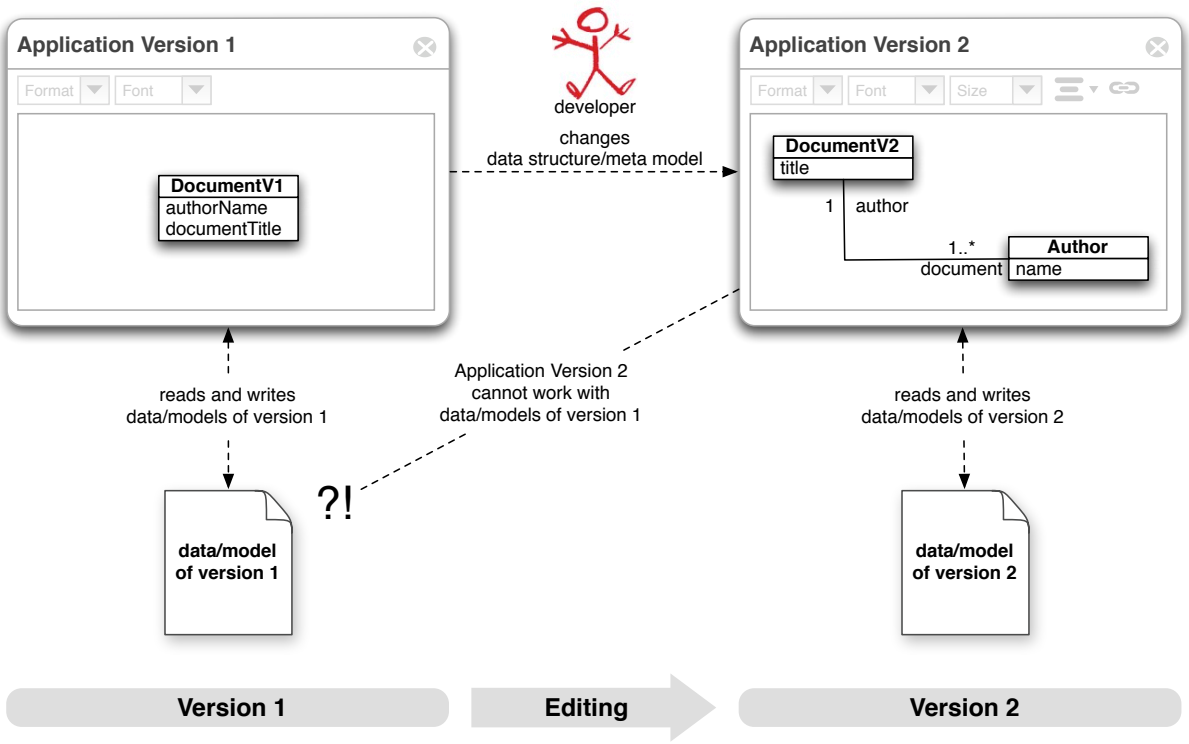

Figure 1.1: Model Breaks due to Meta Model Changes

[1]EMF Ecore is capable of more than what is mentioned in this thesis, such as defining operations and how an implementation can automatically be generated from an Ecore model. However, only the capabilities for defining data structures are relevant for this thesis.

An example scenario for the problem addressed by this thesis is illustrated in Figure 1.1. There is an application which stores data to a file using a specified format. Therefore, the data is structured and the specification of the file format includes the structure definition. The application relies on the data to comply with the structure definition, otherwise the application would be unable to read the files. In this example, the application in version 1 uses a concept named `DocumentV1` which holds an `authorName` and a `documentTitle`. Now the developer keeps on working on the application to fix errors, introduce new features, improve the quality of the code, etc. In this example he implements the requirement that a document must be able to have more than one author. Therefore, the developer *evolves* the application to version 2 by introducing a new concept named `Author`, which can hold a `name` for each author. Furthermore, he renames the field `documentTitle` to `title`. These changes lead to the concept `DocumentV2`.

However, by applying these changes, the developer *breaks* structural compatibility with the former file format. Since he has modified the structure definition from concept `DocumentV1` to `DocumentV2`, data complying with the former structure definition can not be read or written anymore by the new application version. Since in most real-world scenarios there is plenty of existing data which can not be discarded, a migration becomes necessary. This thesis introduces a strategy and tooling for developing the needed *migration algorithms*.

1.2 The Problem's Context

The motivation for *meta model evolution and model co-evolution* is based on two key-aspects: Using models to handle structured data and the necessity to continuously adapt and improve an application.

In software engineering there are and there have been plenty of computer-readable ways to define data structures. Examples are *structs* in C/C++, *records* in Pascal and *classes* in Object Oriented Programming (OOP). For persistence, Relational Databases use the SQL Data Definition Language (DDL) to define data structures, which are known as the database *schema*. For persistence and data exchange, the *XML Schema* language can be used to define the structure of XML files.

Nowadays, the concepts of *models* can be seen as a way to unify those approaches. Due to generalized implementations for serializing, deserializing, verifying, transforming and comparing of models, reuseability of software components, which can be integrated in the form of frameworks or libraries, has been strongly increased. Besides using models for persistation, as illustrated in Figure 1.1, they can be sent over a network, for example to implement Remote Procedure Calls (RPC). It is characteristical for models that their structure definition is a model as well, which makes it a *meta model*. Therefore, it can be said that a *model* is an instance of a *meta model*.

The need for continuous change arises from the fact of living in an environment of continuous competition. To adapt to these environmental conditions is essential to be successful – while the conditions of the environment change over time. This is true for software in general as well as for all the artifacts software consists of – and meta models are one kind of these artifacts. The field of *Software Evolution* investigates the change of software and the change of the software's properties over time. When one artifact is changed, a dependent artifact might need to be migrated, which leads to scenarios of *co-evolution*. Since models depend on their meta models, they need to be co-evolved (migrated) when their meta model

evolves. Migration generally becomes challenging when an artifact is not available to the developer, for example models stored on a customer's computer. Hence, the migration must be able to take place in absence of the developer, which creates the need for a *migration algorithm*, which can migrate the models automatically.

One field of application is Model Driven Software Development (MDSD, [SV06]), where models are used to describe a software's structure or behavior at a high level of abstraction, and where the models are then used to automatically derive artifacts of the software's implementation. Within recent time, there is a trend towards meta models which are specialized for a certain domain, which has led to the fields of Domain Specific Modeling (DSM, [TPK07]) and Domain Specific Languages (DSL, [MHS05]). However, since requirements of a domain change over time, the need for adapting the meta model and migrating existing models arises.

This thesis focuses on the Eclipse Modeling Framework (EMF) which has enjoyed an increasing popularity within the recent years. It could be argued that due to the need for change there should exist a concept and tooling for the migration of models. However, this was not the case when starting this thesis. In the meantime, the framework COPE ([HBJ08c]) has emerged, which follows an operation-based approach, while this thesis follows the diff-based approach. Since no tooling was available before, developers have tried to restrict their meta model changes to changes which do not *break* old models, or they have ignored backward compatibility, which [Mey96] refers to as "schema revolution!". Only to apply non-breaking changes strongly restricts the capability to change and ignoring backward compatibility destroys the value represented by the models. Therefore, these approaches may not be suitable. Alternatively, there is the concept of *deprecation*, which suggests not to remove the obsolete concepts from the meta model but to keep them until all users have migrated their models manually. While such a manual migration is not possible in all cases, the developer has the burden to support two versions of a concept at the same time.

This leaves implementing a model migration algorithm as the most promising approach. In research, this is being investigated in the field of *Meta Model Evolution* and inspiration is provided by related fields such as *Database Schema Evolution* and *XML Schema Evolution*. While a migration algorithm can be implemented in General Purpose Programming Languages (GPPLs) or a designated model transformation language[2], GPPLs tend to have an extremely verbose syntax for this task. Model transformation languages require to also implement transformations for the parts of the meta model which have not changed (identity transformation). A more specialized language is needed.

1.3 This Work

Within this thesis, a solution in form of concepts and implementation is being developed to semi-automatically derive a model migration algorithm from the differences between two meta models. This work focuses on the Eclipse Modeling Framework and on the way to a solution for this task, a Domain Specific Language (DSL) named *Epatch* is being developed which describes the differences between models. The Epatch language is then applied to describe the differences between meta models and is extended to include instructions for the migration of the models and thus leads to the *Metapatch* language.

[2]Such as Xtend, QVT, ATL, etc. For details, see subsection 2.1.2.

The thesis is structured as follows: It begins with introducing models and meta models (section 2.1) to establish the needed theoretical background, point out existing tools and frameworks, and clean up some common misconceptions[3]. The section points out typical operations on models, e.g. (de)serialization, transformation, validation, etc., distinguishes syntax from semantics and models from languages. Furthermore it points out that models can have multiple representations, such as graphical and textual. Then, the thesis elaborates on how models are used in software engineering (section 2.2) and introduces Model Driven Software Development (MDSD), Model Driven Architecture (MDA) and Domain Specific Modeling (DSM). The techniques of MDSD are applied when implementing the solution suggested by this thesis, and models in software engineering are a field of application for meta model evolution.

On the way to meta model evolution, section 2.3 identifies the characteristics of evolution in biology and in computer science and correlates these fields. Based on this, section 2.4 introduces co-evolution in biology and explores where scenarios of co-evolution can be seen in computer science. In section 2.5 the thesis dives deeper into the field of co-evolution and explores scenarios where data structure definitions change and instances have to be co-evolved. Furthermore, it gives an overview over the existing work from the related fields, such as Database Schema Evolution, XML Schema Evolution, Evolution in UML and Evolution in Domain Specific Languages (DSLs). Since describing differences between meta models is needed, section 2.6 introduces the concept of patching. Then, chapter 3 analyzes the challenges of meta model evolution and model co-evolution. It evaluates how the changes between two meta model versions can be obtained an how they can be enriched to build the model migration algorithm. The challenges and possible solutions of this process are identified and useful classifications are introduced based on literature and own observations. At the end of chapter 3, the diff-approach and the operations-approach are introduced and compared with each other.

The following chapters explain and evaluate the software that has been developed within the context of this thesis.

In chapter 4 the *Epatch* language is being developed to be able to describe differences between meta models. After defining the requirements (section 4.1) and listing intended use cases (section 4.2) it is evaluated whether existing formats could qualify for the task (section 4.3). Then, after explaining the design decisions (section 4.4), Epatch's meta model (section 4.5) and its textual representation are explained (section 4.6) based on examples. The chapter finishes with explaining the *Diff-to-Epatch Converter* (section 4.7), the *Epatch Recorder* (section 4.8) and the *Epatch Applier* (section 4.9).

Based on the Epatch language, the *Metapatch* language is developed in chapter 5, which is the suggested format to implement model migration algorithms. After listing the requirements (section 5.1), explaining the design decisions (section 5.2) and evaluating existing formats (section 5.3), section 5.4 explains Metapatch's meta model. Before explaining Metapatch's textual representation (section 5.7), it describes the `MetapatchMigrater` (section 5.5), which implements a generic model migration algorithm that interprets Metapatches (section 5.6). The chapter finishes with an example Java code snippet which shows how to invoke the `CopyingEpatchApplier` to recreate an old meta model version and how to invoke the `MetapatchMigrater` to migrate instances of the old meta model to the current meta

[3]Common misconceptions are, for example, that a model is required to have a graphical representation or that there are no models beyond the Unified Modeling Language (UML).

model (section 5.8).

The suggested concepts and implementations are evaluated in chapter 6 with regards to performance, complexity, completeness, automatability and correctness. Furthermore, it is discussed how a developer can evaluate his custom model migration algorithms based on these qualities. Finally, in chapter 7 this work is summarised and concluded.

1.4 Demarcation

This thesis does not explicitly explain the utilized tools and frameworks (Eclipse [Eclg], EMF [Ecle], Xtext [Eclf], Xtend [oAWd], EMF Compare [Eclh], etc.). Instead, their functionality is roughly outlined where needed to explain the decisions made in this thesis. The reader is asked to refer to the official documentation and tutorials, if necessary.

The approach followed by this thesis to co-evolve models is the diff-approach which derives the migration algorithm from the differences of two meta models – alternatively to recording the operations the developer uses to edit the meta model. The latter is the operation-approach, which is followed by the majority of the existing work. The details are explained in section 3.10 and section 3.11.

Furthermore, this thesis does not provide a graphical user interface for the implementation. The developed software has the format of a framework which can easily be integrated into existing applications or be called from JUnit tests. However, since the Epatch language and the Metapatch language are being developed using Xtext, there is a "smart" editor including syntax highlighting, outline view and go-to-declaration navigation available for these languages.

2 Fundamentals and Related Work

This section introduces the conceptual work from the fields of computer science and information technology that this thesis is based on. It points out the related work and emphasizes the fundamentals. In cases where definitions are fuzzy in the common conception, this section provides a precise definition for the context of this thesis. In areas where implementations are available, these are mentioned.

2.1 Fundamentals of Models

The term *model* has been widely used in computer science. Usually, it refers to an abstract, simplified description of a system which can be used to simulate, document or derive properties of a system [SA00]. This thesis perceives *models* as an equivalent to *structured data* and implies, that the structure itself is defined using a model, which thereby becomes a *meta model*.

This section outlines the common characteristics of models in order to establish a clear impression of the possibilities and alternatives the developer gains by using models.

Besides having a meta model (subsection 2.1.1), there are characteristical operations that can be executed on models (subsection 2.1.2) and which are well supported by existing frameworks[1]. The operations build the foundation for typical use cases (subsection 2.1.3). Furthermore, it is characteristical for models that there can be multiple representations of a model, and that there can be multiple views on a model, which is described in subsection 2.1.6. When dealing with models it is essential to be able to judge the quality of models. The common criteria are listed in subsection 2.1.7.

2.1.1 Models, Meta Models, and a Meta Meta Model

Meta Model	Meta Meta Model	Example Concepts
Ecore instance	Ecore	EClass, EDataType, EReference, EAttribute
XML Schema	XSD Schema	ComplexType, SimpleType, Element, Attribute
MOF instance	MOF	Class, DataType, Property
Java Classes	Java Type System	Class, Primitive Type, Member Variable

Table 2.1: Meta Models and their Concepts

The *meta model* defines a *model*'s structure. Alternatively, to describe it from a model's perspective, a *model* is a *meta model*'s instance. Except for methods and operations, this is the same as in Object Oriented Programming (OOP): a *class* defines the structure for an *object* and an *object* is a *class*'s instance.

[1] The frameworks are named in subsection 2.1.2, where the operations are described.

To define a model's structure, meta models consist of structure describing elements, which are also referred to as *concepts* (Meta Programming System (MPS) [Jet], Metaedit+ [Met, TPK07]). Depending on the type of meta model, there are different types of concepts, Table 2.1 provides an overview. Common concepts are to have data structures with properties (Eclipse Modeling Framework (EMF) [Ecle]: EClass, XML Schema Definition (XSD) [W3Cb]: ComplexType), atomic datatypes (EMF: EDataType, XSD: SimpleType) and semantics to declare properties (EMF: EAttribute and EReference, XSD: Attributes and Elements).

It is common to define these concepts by using a model, which is usually referred to as a *meta meta model*. Table 2.2 lists a set of commonly known meta meta models. In some popular implementations (MOF [OMGd], Ecore [Ecle] [BSM+03], XML Schema [W3Cb]), the *meta meta model* is capable of describing its own structure, so there is no necessity for a "meta meta meta model". For example, in the case of XML, there is an XML Schema which defines the concepts available in XML Schema, which this thesis refers to as the *XSD Schema*.

When designing meta models, the meta meta model defines the power of expression that is available, which manifests itself in the concepts that are allowed. Thereby it becomes important for a meta model designer to know the meta meta model. For data formats that are intended as exchange formats and which are intended to be supported by various implementations, it is common to publish the meta models (for examples see Table 2.2).

Meta Meta Model	Example Meta Models
Ecore	Diff[CB07], ChangeDescription, BPMN[Eclb], XSDSchema[W3Cb]
XSD Schema	SVG, XHTML, AUTOSAR, HL7, WSDL, SOAP
MOF	Class Diagrams, Use Case Diagrams, State Machine Diagrams

Table 2.2: Example Meta Models

Naming Meta Levels: OMG's M0..M3

The Object Management Group (OMG [OMGb]) has specified names for the different meta levels as visualized in Figure 2.1. An absolute number (M0..M3) is given to every level. Starting with M3 being the meta meta model, M2 is the meta model, and M1 the model. It is important to note that this naming schema assumes that the models are used in the context of Model Driven Software Development (MDSD[2], subsection 2.2.1). In this field the models have instances as well (M0), which are the instances of the implemented software, also known as launched applications.

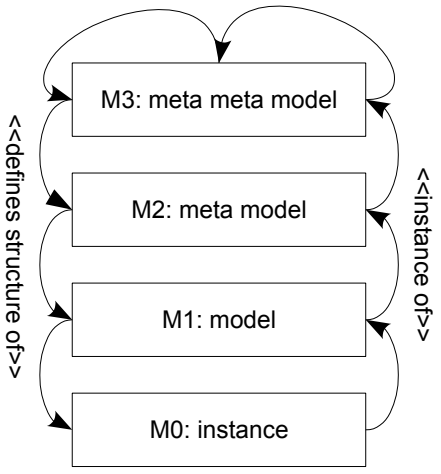

Figure 2.1: Meta Levels

The relationship between meta model and model reflects in this naming scheme in the way that M_n defines the structure of M_{n-1} and M_n is an instance of M_{n+1}, with the exception that

[2]When following the specifications of the OMG, this is Model Driven Architecture (MDA). This thesis sees MDA as one way to apply MDSD.

M3 defines its own structure and is its own instance.

Naming Meta Levels: EMF's Instances, Models and Ecore

To keep things simple for those not familiar with the "meta" terminology, the EMF community sometimes refers to M3 as *Ecore*, M2 as *EMF model*, and M1 as *instance* and thereby elegantly avoids the word "meta". This is useful for scenarios in which an application's persistence layer is based on EMF and the instances are plain application data that cannot be "launched".

However, this does not apply to MDSD, since there even those instances have instances. Therefore, this terminology is not used in this thesis. Instead, *Ecore* will be referred to as *meta meta model*, the Ecore instances as *meta models* and their instances as *models*. In cases where it is not relevant that meta models have instances, they may be referred to as models.

2.1.2 Typical Operations with Models

Besides the relationship between model and meta model, it is characteristical for models that the following operations can be executed on them. The applicability of these operations is not a requirement for data to be classified as a model since it mainly depends on the availability of implementations. These operations can be seen as powerful tools for the developer to increase the efficiency in crafting software. Since the operations are generalized tasks, there are frameworks, libraries and specialized languages available which can be integrated to the to-be-developed application. The following sections describe these operations.

Validation

Models can be validated using a set of rules which are evaluated against the model. This can be used to derive properties from the model or, which is the primary intention, to restrict the state space of the meta model to only those instances (models) which are valid to a certain domain. Since the rules operate on a model they require it to have a certain structure, which typically makes rules specific to a meta model. Known implementations are the openArchitectureWare's *Check*-language ([oAWa]) and the OMG's Object Constraint Language (OCL, [OMGa]).

Model to Model (M2M) Transformation

Models can be transformed from one format to another. The transformations can be classified to the following categories.

In-Place Transformation The transformation modifies the model. This requires the source and target meta model to be the same and has as a consequence that during the modification process the source model becomes the target model. The source model is not available anymore after the transformation has finished.

Pure Transformation The transformation derives the target model from the source model without modifying the source model, and as a consequence the target model does not reference the source model. Pure transformations can be distinguished based on whether their source and target models are instances of the same meta model.

- **Same Meta Model** The transformation's source and target model are instances of the same meta model. In [CS03], this is classified as an *endogen* transformation.
- **Different Meta Model** The transformation's source and target model are instances of different meta models. This kind of transformation is commonly used to transform models from one format to another. In [CS03], this is classified as an *exogen* transformation.

In the same way as validation rules, M2M transformations are usually implemented for certain meta models. Designated programming languages for this task are the openArchitectureWare's Xtend [oAWd], the OMG's Query View Transformation Language (QVT, [Obj07]) and the Atlas Transformation Language (ATL [Ecla]).

Model to Text (M2T) Transformation

Models can be transformed to text. The text is not restricted to a special purpose, however there are common scenarios: In MDSD, where models describe architectural or behavioral properties of a software system, the result can be source code, configuration files, or documentation. If the model describes a document (e.g. DocBook), it can be transformed to a document markup language (LaTeX, HTML etc.). In contrast to persisting a model, an M2T transformation does not need to store all of the model's data to its textual output and nothing is implied about whether the textual output is machine readable. As M2T transformations require models to have a certain structure, they are usually specific to a meta model.

Languages for M2T transformations are in most cases template languages. Common examples are the openArchitectureWare's Xpand ([oAWc]), the W3C's Extensible Stylesheet Language Transformations (XSLT, [W3Cf]) and the OMG's Models to Text Transformation Language (MTL [OMG07]).

Serialization and Deserialization

Models can be serialized (written) to a byte stream or deserialized (read) from a byte stream. This is commonly used to persist a model to a hard disk or to send it over a network. While M2T transformation often only transform parts of the model to text, the process of serialization and deserialization requires that all information is preserved. In the case

of XML and XMI the serialization/deserialization formats and implementations are usually generic, which means they are not specific to a meta model, but need the meta model to be accessible while the serialization/deserialization process. Generic implementations have advantages with regards to being exchangeable and/or reusable.

Meta Model	Example Serialization Formats
Ecore Instances	XMI
XML Schemas	XML
MOF Instances	XMI
Java Classes	Binary
SQL DDL	SQL-Dump, Binary
Grammar	Language, Code [1]

Table 2.3: Meta Models and their Serialization Formats

Models can be serialized, stored to a persistent storage or send over a network. Table 2.3 shows examples of common formats.

Addressing/Querying/Navigation

Model elements can be selected using path-like expressions or a query language. Mechanisms can be distinguished by whether they allow to address one element at a time or whether they can select multiple elements with one query. Common examples are the openArchitectureWare's Expression Language ([oAWb]), the OMG's Object Constraint Language (OCL, [OMGa]), Xpath [W3Ce], EMF Query [Ecld], EMF FragmentURI ([Ecle]), and the Structured Query Language (SQL).

2.1.3 Typical Use Cases with Models

With the previously defined set of operations that can be executed on models, several use cases become possible. This section lists the common ones.

Data Storage

Since models can be serialized and deserialized without the loss of information (subsection 2.1.2) these serialization formats can be used to persist the model to a storage medium. In Table 2.3, common formats are listed. In the fields of XMI and XML it is popular to store the model to one or more files in the file system. In the case of using multiple files, the mechanism of addressing is used (subsection 2.1.2) to reference model elements from another file.

Data Exchange

Besides storing serialized models to disk it is common to send serialized models over a network. Popular are file based and Remote Procedure Call (RPC) based patterns. Java's

[1] Traditionally, grammars are unidirectional (parser: text to model), but it is possible to derive a model to text transformation from a grammar, and thereby make it bidirectional.

Remote Method Invocation (RMI) and the XML-based SOAP ([W3Cc]) belong to the latter group.

Models in Software Engineering

In Software Engineering, models can be used to describe architectural or behavioral properties of software and thereby become a first-class citizen of the software development process. For more details, see section 2.2.

2.1.4 Syntax vs. Semantics

Languages have a syntax as well as semantics. This section starts with explaining how semantics are defined since this is a prerequisite for differentiating between abstract and concrete syntax.

Semantics

A meta model's semantics defines how information which is represented its models is to be applied. It defines the actual meaning of a model's contents. There are different ways to define semantics:

Informal A human-readable and human-understandable but not machine-readable document in spoken language defines the meaning of the model's contents. This is usually the documentation or specification of a model.

Formal To define semantics in a formal way, a human-readable formal language (e.g. pseudo code) can be used, but it is more common to define the semantics implicitly by having an interpreter for the model or by having a mechanism to transform the model to another existing formal model/language. This is typically machine code or a format which can be compiled to machine code. Thereby, semantics are defined by the interpreter or the code generator. A requirement for this is to have semantics without ambiguities.

Syntax

The syntax defines a notation or a format in which the model is represented. Since a model can have multiple representations, there may be multiple concrete syntaxes for the same model (subsection 2.1.6). When looking at a model's representation one can distinguish between information that has a semantic meaning and information that is just needed for layout and formatting purposes.

Abstract Syntax covers all the information that has a semantic meaning, stripped from layout and formatting information. M2T and M2M transformations as well as validation operate (subsection 2.1.2) on the model's abstract syntax.

Concrete Syntax additionally covers the data related to human-readable representations of a model. This is typically layout and formatting information. In the case of a textual representation, these are typically white-spaces and line-breaks[3]. In the case

[3]Exception: the programming language Python, where the level of indentation determines code blocks.

of a graphical representation, these describe positions, sizes and optionally colors and shapes of graphical elements, etc.

In the field of compiler construction, the in-memory data structure which holds the abstract syntax is known as the Abstract Syntax Tree (AST). For graphical modeling, the Eclipse Graphical Modeling Framework (GMF, [Eclc]) stores abstract syntax and concrete syntax in two different files.

2.1.5 Meta Models vs. Languages

A *language* is defined in this thesis as a meta model combined with a syntax and semantics. Depending on whether the semantics are formal or not, the language can be considered formal or informal. If the representation is graphical or textual, the language is called a textual or graphical language.

2.1.6 Typical Representations

To work with a model, different representations of a model are needed. For persisting a model or for sending a model over a network, a *serialized representation* is needed (subsection 2.1.2, subsection 2.1.3). Program implementations that work with a model need to hold an *in-memory representation* of the model. But most importantly the user needs a representation of the model he/she can view and/or edit. Table 2.4 provides an overview of common representations.

This section focuses on representations that are intended for a user to work with. This brings several challenges: A user should understand the model and he should not be overwhelmed with unneeded details, while maintaining the availability of all information he/she needs. Since these are very subjective requirements, there is no general solution. It mostly depends on who the users are, what they want to do with the model, which syntaxes they are familiar with and how much detail they need. These qualities are further explored in subsection 2.1.7.

First, this section points out that a model's representation can be a view on the model, which only presents a small part of the models contents. Then it explores popular representation types: *graphical*, *textual*, *trees & tables*, and *mixed*.

It is important to understand that a model can have multiple co-existing representations. It is an application's responsibility to keep the representations synchronized, in case one is being edited.

Meta Model	Model Representation		
	In-Memory	Persisted	Human Viewing/Editing
XML Schema	DOM	XML	Trees, Text, WYSIWYG, etc.
MOF Instances	UML	XMI	Trees, Diagrams, etc.
SQL DDL	Relational Data	SQL-Dump, binary	Tables, Forms, etc.
Grammar	AST	Text	Language/Code
Ecore Instances	EObjects	XMI	Trees+Properties, Diagrams

Table 2.4: Model Representations

Complete Models vs. Views on a Model

At first it has to be decided whether a model's representation represents the complete model or just part of the model, which is also called a view on the model or a perspective on the model. There can be multiple views on one model.

In case the representation is complete, this format could also be the format for persisting the model and the viewer/editor has the confidence that no detail of the model is hidden.

In case the representation is a view on the model, this view may provide a summary, an overview or a representation which is specialized on a certain concern of the model (e.g. show only classes and associations of an UML Class Diagram, but not properties and operations; show the inheritance hierarchy of one class). Having views on a model helps in handling large models and allows developers to separate the concerns of a model.

Graphical

Graphical representations are usually diagrams, consisting of nodes and edges (also known as box and line diagrams), but other formats are possible as well. Well known examples are Entity-Relationship (ER) diagrams and the OMG UML's ([OMGc]) Class Diagram, State Machine Diagram and Use Case Diagram.

The Eclipse Graphical Modeling Framework (GMF) allows to define a graphical editor using models and to generate the editor's implementation from these models. The software Metaedit ([TPK07]) provides a graphical editor to generically define a graphical representation of a model.

Textual

Textual representations may look like (or even be) source code, and can be considered a formal language if there are semantics defined and it has a machine-readable syntax.

Reading a textual representation is handled by lexers and parsers that have been researched in the field of compiler construction. To write a textual representations an M2T transformation can be used. A more elegant solution is to "invert" the parser and match the grammar rules (which are the basis for the parser) against the model and thereby recreate the textual representation. This process is supported by Eclipse TMF Xtext ([Eclf]) and the Textual Editing Framework (TEF [Sch08]).

If there is a textual representation, this format is usually also chosen for persisting the model. XML [W3Ca] is an interesting example since it is a compromise between being human-readable and machine-parseable.

Trees and Tables

Models which have a hierarchical structures can be presented as trees, as seen for example in the EMF's sample Ecore editor ([Ecle]). The nodes typically have properties which can be edited using table controls.

Mixed

The examples listed above can be combined, and further representations can be added. Examples are lists and editors for mathematical formulas.

One example for the combination is a UML editor, which uses a tree view to hierarchically represent all model elements, diagrams (to have several views on these model elements) and textually expressed (using OCL) statements which define a behavior for the model. The model element's properties are edited in tables.

Another example is the *Domain Workbench* [SCC06], which can look to the user similar to an office application, but provides the environment for a domain expert to specify a software's properties and behaviors for his domain. For this he might define rules, mathematical formulas, and design surfaces using WYSIWYG (What You See Is What You Get) editors.

2.1.7 Criteria for Representations

If representations of models are meant to be viewed and edited by a human being (the user), all common rules for communicating with human beings apply. One can think of the representations as a language that is meant to be understood by both human beings and computers. Users usually want the communication to be efficient and fault tolerant. Therefore, they only want the information they need and not the information they do not consider relevant for them, and they want the information they get to be easily understandable for them.

For representations of models that are not meant to be viewed by users, for example a format to persist the model, an exchange format or the in-memory representation, there are still the developers who need to create and maintain the software that handles these representations. So there is communication as well.

Appropriate Level of Abstraction

Generally speaking, the level of abstraction determines the level of detail covered by the representation or the entire model. For software development, common levels are:

- Leave out all information that is specific to a certain platform (e.g. Java, .NET). This information can either be added later in a transformation step or be expressed more generally. The advantage is not only increased portability, but also that working with the model does not require platform specific knowledge.
- Leave out computational information (e.g. the description of algorithms) which improves readability for users that are not from the field of computer science.

For more examples see subsection 2.2.2.

Domain Specific vs. General Purpose

To increase the level of abstraction it is often required to design the representation in a way that it becomes specific to a certain domain. This basically means to leave out detail and expressiveness that is not needed for this field of application and to introduce constructs that are specialized to this domain. This leads to Domain Specific Languages (DSLs: [MHS05], [Spi01]) and Domain Specific Modeling (subsection 2.2.2).

The opposite are General Purpose (Programming) Languages (GPLs, e.g. Java, C++, etc.)

Intuitive Readability

The model's representation has to be readable, and ideally, also writable by the user it is designed for. Usually, users already have a notation for their domain which they use when communicating within their domain (e.g. verbally with their colleagues, graphically on the flip-chart, textually in specifications). This already-established notation can be a good start point for designing a model's representation.

Conciseness as Opposed to Syntactic Noise

When reading, the human brain has to tell apart relevant from irrelevant information. With experience, this process of filtering becomes easier, but will always be exhausting.

This point is about avoiding unnecessary syntactic elements to keep the representation concise and to help the reader focus on what is relevant. It assumes the representation is at the appropriate level of abstraction (subsection 2.1.7).

To reach this goal, it helps to identify what the user has to specify explicitly and what can be determined implicitly. The latter can require reasonable default values. For graphical/table-based representations, it helps to hide fields/elements which are not suitable in the current situation. For example, when working on a UML Class Diagram for code generation the usual UML Editor provides more than a dozen property fields for a class, but the code generator might only make use of three of them. Therefore, all the other fields can only lead to distraction and confusion of the user.

2.2 Models in Software Engineering

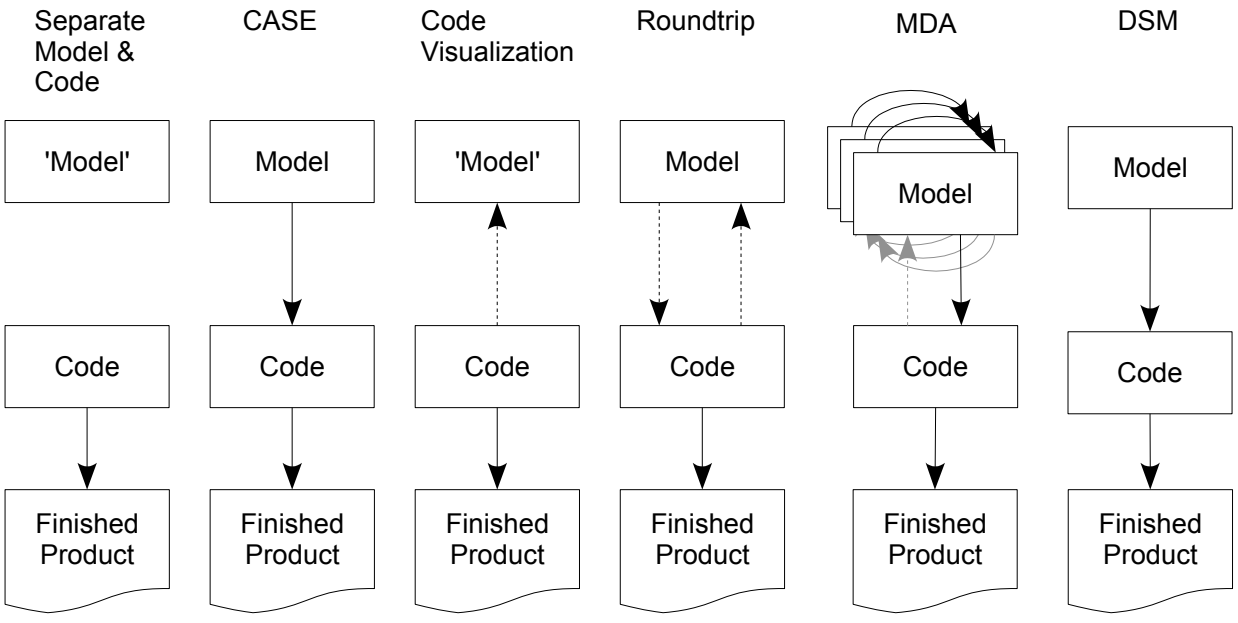

Figure 2.2: Models in Software Engineering [Tol08]

Models are a valuable tool for software engineering, however, there is still much misconception about what models are and how they can be used. After section 2.1 has sharpened

the definition of what models are, this section clarifies the value of models for software engineering.

For this thesis, Model Driven Software Development is an important field of application for meta model evolution. Furthermore, the concepts of MDSD are applied for the software which is implemented in the context of this thesis.

Figure 2.2 provides an overview how models and things that are not actual models, but are sometimes perceived as models, are involved in the process of developing software.

Separate Model and Code Since the "models" are neither interpreted nor transformed to other models or code, they are *disconnected* from the actual software. Whether they have a meta model and how formal they are is not required by tools, but depends on the taste of the author. Depending on their content, they can be seen as a kind of documentation or specification. Due to the fact that they are disconnected, it is the responsibility of the software developer to keep them synchronized with the software. In practice, this easily leads to the scenario that the software evolves, but the "model" is not updated and thereby loses its value completely.

CASE Computer-Aided Software Engineering was an early field where the concept of describing software in models and generating source code from them was applied. However, the majority of CASE tools used to be proprietary and tightly tied to their domain. So they appeared as black boxes for the developers which could not be adopted to the custom needs. For example, meta models were not extensible and code generation facilities not customizable. This led to a bad reputation of CASE tools in general which lasts until today. However, nowadays there are tools for modeling UML, Structured Analysis and Structured Design (ARIS: EPC [Sch98]) and Entity-Relationship Models of which some are open source and which are sold under the tag of being a CASE tool.

Code Visualization Information is extracted from existing source code and presented graphically. Popular examples are package structures and dependency graphs of components. There are also approaches to visualize more sophisticated metrics about code quality ([WL08]). The "model" part of this process is the interchange format between the code analyzation and the visualization component. Techniques introduced in section 2.1 are not required to be applicable. It is important to note that code visualization analyses existing software, and thereby aids the developer to understand its structure and functionality, but does not contribute fragments (code/models) to the development process.

Roundtrip In the approach of roundtrip engineering there is a model which can be edited and which is automatically synchronized with source code. When the model is modified the source code is updated as well as the model is updated when the source code is modified. This approach works good for UML Class Diagrams, but generally, there are two technical challenges: At first, the model of the source code holds more information than the source code itself. This leads to the scenario that when the model is modified, existing source code has to be merged with the updates from the model. The second challenge is that to increase the level of abstraction the model typically represents structural properties of the source code in a simplified way. This requires the source code always to be structured in a way so that it can be represented in this simplified

way. This can be very restrictive for the developer who edits the source code. As a result, roundtrip engineering is hardly combineable with the criteria for a good representation (subsection 2.1.7).

MDA For Model Driven Architecture (MDA) see subsection 2.2.2.

DSM For Domain Specific Modeling (DSM) see subsection 2.2.3.

2.2.1 Model Driven Software Development (MDSD)

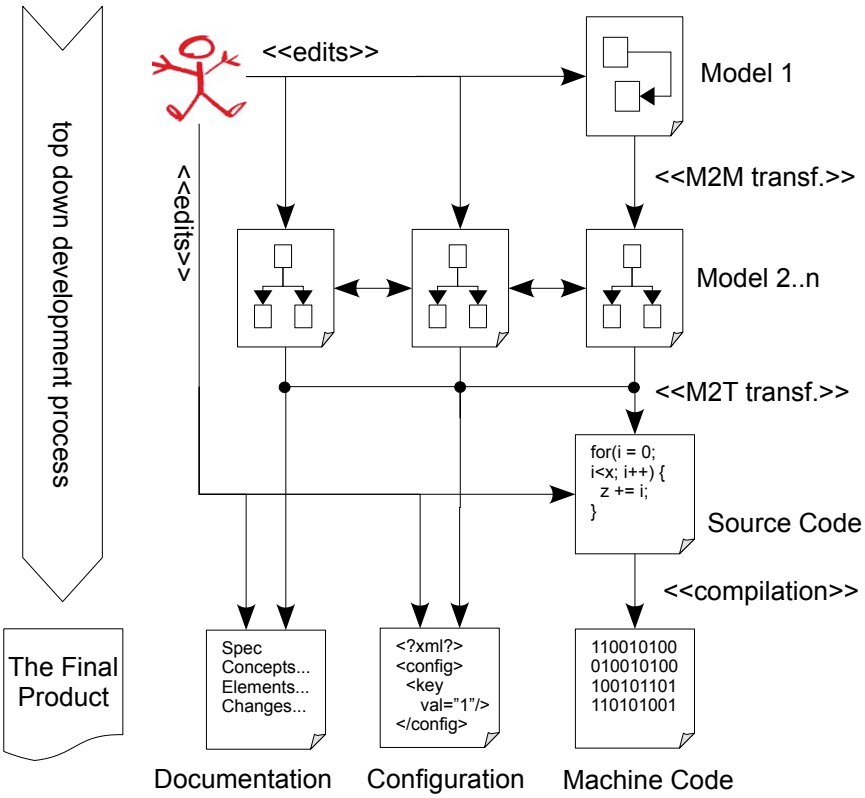

Figure 2.3: Models in Model Driven Software Engineering (MDSD)

"The history of software development has shown a trend towards higher levels of abstraction" [OSBE01]. Model Driven Software Development (MDSD) continues to increase the level of abstraction to any level that is useful for the developer.

Commonly software is implemented in General-Purpose Programming Languages (subsection 2.1.7), and experience shows that well-structured software tends to have repetitive patterns in its code. To increase the level of abstraction means to express these repetitive patterns in a more concise form and to hide unnecessary information. Frameworks and internal DSLs ([FB]) can achieve some increase of the level of abstraction, but are restricted by the limits of the host language. More flexibility is provided when expressing the needed information using models and then to generate the source code from these models.

This approach is called Model Driven Software Development (MDSD, [SV06]), whereas the term *driven* emphasizes that source code is generated from models as opposed to derive

models from source code. In MDSD it is common to design custom meta models (subsection 2.1.1) with custom representations (subsection 2.1.6) which are tailored for the needs (subsection 2.1.7) of the people who work with the models. These people can be developers as well as domain experts. The models can define structure and architecture as well as behavior or configuration of the software.

As shown in Figure 2.3, there can be multiple models which can be at different levels of abstractions, which can reference each other and which can be instances of different meta models. Models can be crafted by hand or be derived via a model to model transformation (subsection 2.1.2). For a model that is partly edited and partly derived, a clear concept of how to avoid that hand crafted elements are overwritten by the transformation, is required.

When models are available, source code, configuration and documentation can be generated via model to text transformations (subsection 2.1.2) from the models. MDSD does not demand that any of the mentioned fragments are generated to a hundred per cent. Even though this can be possible for some domains, it is common that because of the increased level of abstraction the models are not intended to cover all aspects of the software. Therefore, the generated source code needs to be extended manually. As with derived models, a clear concept to separate hand crafted and derived (generated) source code is needed to avoid that the former is being overwritten in the generation process. A common pattern for object oriented target languages is to separate generated and manually written source code in different source folders and then to automatically generate abstract base classes which all have a manually implemented subclass. This subclass can just be empty or override methods of the generated class as needed for customization. Another approach is to use *protected regions* ([DW04]), which indicate to the code generation utility not to overwrite certain sections within a file.

Summarizing, it can be said that model driven development is a straight forward top down development process where fragments at a higher level of abstraction are transformed to fragments at a lower level of abstraction. Generated elements and hand crafted elements should be cleanly separated but interact with each other.

2.2.2 Model Driven Architecture (MDA)

The Model Driven Architecture (MDA, [Obj03]) can be seen as a variation of MDSD (subsection 2.2.1) which is based on conceptional and technical standards defined by the Object Management Group (OMG, [OMGb]). The best-known is probably the Unified Modeling Language (UML, [OMGc]) and its diagram types: Class Diagrams, Use Case Diagram, Sequence Diagram, Activity Diagram, etc. Each diagram type has its own meta model which is an instance of the Meta Object Facility (MOF, [OMGd]). Therefore, MOF is the meta meta model of all UML models.

To work with models, the OMG has specified a number of specialized languages: There is Query View Transformation (QVT, [Obj07]) for uni- and bidirectional model to model transformations, the Object Constraint Language (OCL, [OMGa]) which can be used for constraint checking and as an expression language and there is the Model to Text Transformation Language (MTL, [OMG07]). For model transformations and validation, see subsection 2.1.2.

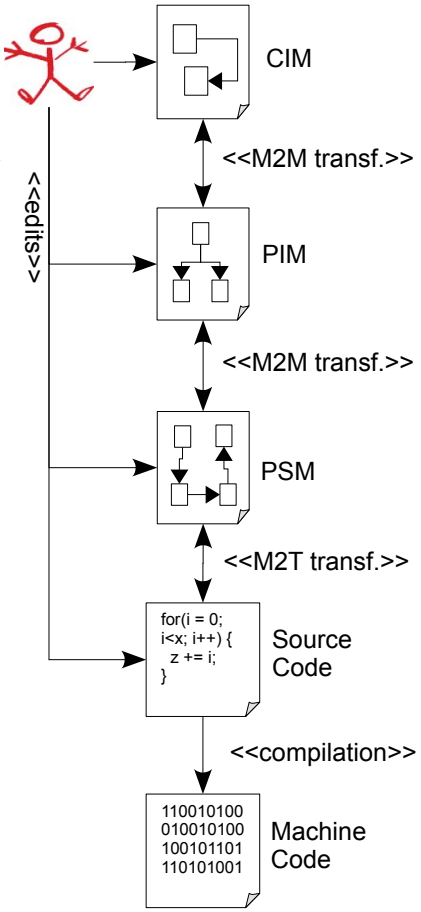

Figure 2.4: Models in Model Driven Architecture (MDA)

Furthermore, the OMG has defined three levels of abstraction for models which are used in the context of MDA. They are visualized in Figure 2.4. Since the bidirectionality of QVT and the concept of roundtrip engineering, the model to model and model to text transformations in Figure 2.4 are allowed to propagate changes upwards in the hierarchy of abstraction, which leads to the challenge of maintaining hand crafted elements along with derived elements discussed in subsection 2.2.1.

CIM The Computation Independent Model holds all descriptions of the to-be-implemented software that are made in a notation that does not imply assumption about the software structure or algorithms. This can be requirements or the analyses of the problem domain.

PIM The Platform Independent Model describes the software structure and functionality without being specific to a certain platform or technology, such as Java, J2EE, Spring, .NET, SOAP, CORBA, etc. For example, this requires the PIM to have platform independent datatypes (e.g. “String” and “URL”) and stereotypes (e.g “Entity”).

PSM The Platform Specific Model can be derived from the PIM with the help of a Platform Definition Model (PDM), which defines the mapping of platform independent

concepts to concepts of a certain platform. For example, for the Java platform, the mentioned platform independent datatypes would be mapped to "java.lang.String" and "java.net.URL" and the stereotype could be mapped to "J2EEEntityBean".

2.2.3 Domain Specific Modeling (DSM)

The Domain Specific Modeling (DSM) can be seen as a specialization of MDSD, which emphasizes the creation of meta models and model representations which are specialized to a certain domain (subsection 2.1.7). Examples for common domains are business processes, rules to calculate a person's pension, or the page flow of a mobile phone application. By being domain specific the models have the potential to be understood and eventually to be editable by domain experts, which have no understanding of the underlying software technology. Furthermore, being specific provides the flexibility to have models at the optimal level of abstraction (subsection 2.1.7).

2.3 Fundamentals of Evolution

Evolution is a concept which is probably best known from the field of biology with Charles Darwin [Dar59] as the one who shaped the term the most. Therefore, this section starts with quoting the essential ideas of evolution in biology and moves on by identifying where these ideas apply in computer science as well. After having explored evolution, it points out how data structure definitions and their instances can co-evolve and what similarities exist in biology.

2.3.1 Evolution in Biology

The book "Evolution" by D.J. Futuyma introduces the term evolution as follows:

> The word *evolution* comes from the Latin *evolvere*, "to unfold or unroll" – to reveal or manifest hidden potential. Today "evolution" has come to mean, simply "change". It is sometimes used to describe changes in individual objects such as stars. **Biological** (or **organic**) **evolution**, however, is *change in the properties of groups of organisms over the course of generations.* The development, or *ontogeny*, of an individual organism is not considered evolution: individual organisms do not evolve. Groups of organisms, which we may call **population**, undergo *descent with modification.* Populations may become subdevided, so that several populations are derived from a *common ancestral population.* If different changes transpire in several populations, the populations *diverge.* [FEJ05]

It is interesting to see that the term "evolution" is being used to describe "change in the properties groups over the course of generations" as well as the development of "individual objects", whereas in the field of biology only the former is considered evolution. Besides describing and explaining change, evolution has two fundamental elements in biology:

> Evolution can be seen as a two-step process. First, hereditary variation arises by mutation; second, selection occurs by which useful variations increase in frequency and those that are less useful or injurious are eliminated over the generations. "Useful" and "injurious" are terms used by Darwin [Dar59] in his definition

> of natural selection. The significant point is that individuals having useful variations "would have the best chance of surviving and procreating their kind". As a consequence, useful variations increase in frequency over the generations, at the expense of those that are less useful or injurious. [Aya07]

To summarize, it is important to see that evolution requires *variation* based on random changes (mutation) and *natural selection* of those candidates that are the fittest to the environmental requirements.

When looking at the characteristics of individuals, it is furthermore important to distinguish between the *genotype* and *phenotype*. While the genotype describes the genetic makeup of an individual, the phenotype refers to properties of the individual which result from interaction of the genotype with the individuals environment.

2.3.2 Evolution in Computer Science

In Computer Science there are several fields where evolution occurs. Among them are the emergence and the disappearance of individual software implementations and concepts, the change of software implementations over time and the usage of the concepts of evolution in algorithms. This section gives an overview of these fields, it shows a classification of them based on scope and selection-kind and furthermore relates them to evolution in biology.

Speaking about software, there are two general things to consider: The most remarkable difference to the field of biology is that software does not reproduce by itself. All emerging variety has to be created intentionally by human beings. This may change at the day when artificial intelligence advances to the point where it is able to implement software on its own, but it is not possible to foresee when, or if at all, this will ever happen.

The second difference is that for a software implementation it can not be distinguished between a genotype and a phenotype. While in biology the genotype can not be changed after the birth of a being, there is nothing of a software that can not be changed – assuming an unlimited amount of effort is allowed. Frameworks, libraries and algorithms can be exchanged, parts can be reimplemented in another programming language, GUI and public API can be modified and users as well as developers can come and go. In practice, of course it is not reasonable to apply modifications that's cost exceed the cost of a complete rewrite of the software implementation.

When humans modify software and if change is only limited by the workload staying reasonable, change could speed up seriously compared to evolution in biology. While in biology evolution happens relatively slow, most times too slow to be spectateable, software users can enjoy new features and bugfixes of open source software multiple times a year.

Scope: Populations; Selection: Natural

Within a group of individuals (which can be software implementations or concepts) that fulfill the same needs of a user there is evolution. In contrast to biology, the lifespan of individuals is potentially endless and their survival depends on whether people choose to use them.

Thereby, a *population* consists of individuals of a certain *species* whereas a species is defined as all individuals which satisfy the same need of a user. Furthermore for evolution it is required that the user is not bound to single individuals, he must be able to choose the

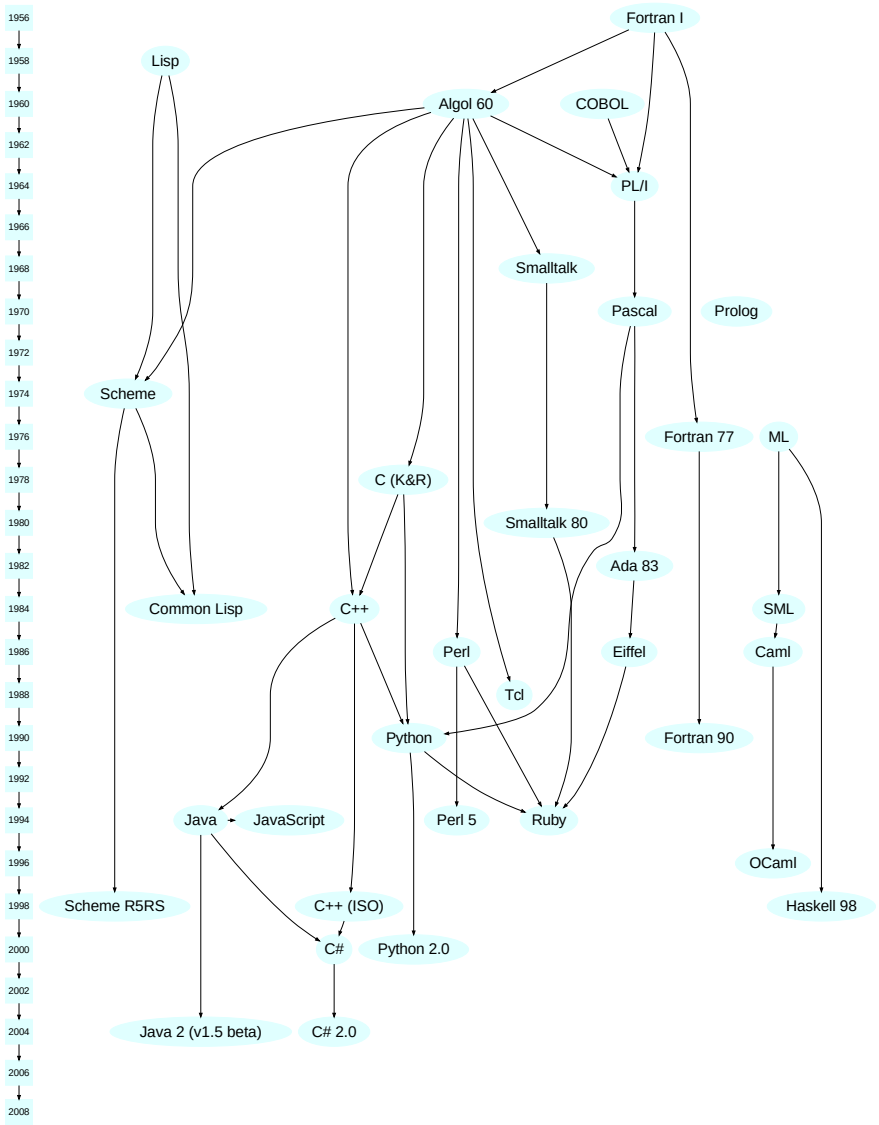

Figure 2.5: History of Programming Languages [Rig]

one that fits him best. Otherwise there would be no *selection*. This kind of selection is close to *natural selection* in biology, since it does usually not depend on a single person's decision whether an individual is successful, but on the amount of persons which find that individual useful. Typical examples of individuals are:

Concepts, such as *algorithms*, *design patterns*, *process models* and *architectures*, which exist as specification or documentation and manifested in implementations. Every time a new concept is invented, the inventor copies and eventually modifies parts of existing concepts and optionally adds new own ideas. Thereby the mixing of the "ingredients", the genes, as required for *mutation*, happens. Every time a developer creates or modifies a software he can choose from existing concepts that he might use and thereby participates in the process of *selection*.

Software Implementations, such as *programs* and *applications* exist as a sequence of machine executable bytes, and for the sake of being modifiable, it is preferable for these bytes to be source code rather than binary code. It is reasonable to have a distinguished look at

open source and proprietary software. The reader might keep in mind that the following two positions are the extremes and that most real-life scenarios are mixtures. In *open source*, *mutation* happens by developers who write new source code, modify existing source code, and copy and eventually reuse code snippets from other software. Since all source code is available in public, the "pool of genes" is constantly mixed and extended. *Selection* happens fairly natural, since a successful software attracts more users and more developers. More users help to spread the word and more developers help to grow and improve the software. On the other hand there is software that nobody needs or that has a code quality so bad that it scares away developers and will become orphaned and forgotten. For proprietary software, *mutation* is hindered, since source code is kept secret to avoid competition, which decreases the size of the available "pool of genes". *Selection* happens in two ways: At first, somebody must be willing to pay developers for working on the software and secondly, there must be users willing to use and to pay for the software. To prevent selection on the side of the users, some companies use so-called *vendor lock-ins* to make it difficult for a user to switch to a different vendor.

The "Programming Languages History" (Figure 2.5) is an example for the evolution of programming languages. The languages can be considered a combination of the concept of the language (syntax and execution model) and sometimes a central implementation, such as a compiler or interpreter. The figure shows not only the emergence and disappearance of languages, but it also shows which other languages have inspired the newly created languages. This visualizes the "flow of genes", the concepts of language design, which are copied from one language to another.

Since meta model evolution refers to the change of an individual and not of a population, this scope is not of further relevance for this thesis.

Scope: Individuals; Selection: Manual

Besides populations, individuals can change. *Mutation* and *selection* happens for the single elements an individual consists of. This leads to a different situation: In contrast to the scope of populations, mutation and selection happen manually as intended activities of the developer. Furthermore, there is no group of different elements which fulfill the same needs and which compete with each other, but instead there is a smaller group of elements that all have distinct tasks and which cooperate with each other. If the developer decides to exchange or modify one of them, he does so to improve the individual's quality, which can be to implement a stakeholders requirement.

The changing individuals can be *concepts* as listed in the last section, but for marketing reasons it seems to be more common to publish a modified concept under a new name instead of a new version number. On the other hand the programming languages shown in Figure 2.5 are a good example for concepts changing over time.

Furthermore, the changing individuals can be *software implementations*, as listed in the last section. The changing elements are all fragments and concepts the implementation consists of or makes use of. Examples are data structures, public API, GUI, algorithms, libraries, frameworks, programming languages, etc. While in biology the change of an individual is not considered evolution at all (subsection 2.3.1), in computer science this is the field of *Software Evolution*, which is further investigated under subsection 2.3.4. Meta Model Evolution is a sub-field of Software Evolution.

Scope: Dataset; Selection: Algorithmic

Algorithms have been developed based on the concepts of evolution for purposes of simulation and optimization. In this scope, the individuals, that the population consists of, are sets of data, whereas it is not of importance for the algorithm what these sets of data represent. However, the algorithm must be able to *mutate* the data sets and to choose the most promising ones as the basis for the next generation. This *selection* is usually done using a *fitness function*. This field is further investigated in subsection 2.3.5.

2.3.3 Terminology: Version, History, Evolution

This thesis will make use of the words *version*, *evolution* and *history* in the same way as [DGF05] defines them: A *version* is a snapshot of an entity at a particular moment in time. The *evolution* is the process that leads from one version to another. A *history* as the reification which encapsulates knowledge about evolution and version information. According to these definitions, we say that we use the history to understand the evolution (i.e., history is a model of Software Evolution).

2.3.4 Software Evolution

In Software Engineering, the term *Software Evolution* refers to the change happening to a software during the development phase as well as the maintenance phase. In Software Evolution, software or its components are looked at as a whole concerning the changes over time with regards to size (growth), complexity, correctness, maintainability, etc.

Since the work of Meir M. Lehman [LB85] in the 70's the importance of observing and modeling software evolution started to be recognized. He distinguishes *S-Type* and *E-Type* software, whereas he defines the former as "A program is defined as being of type S if it can be shown that it satisfies the necessary and sufficient condition that it is correct in the full mathematical sense relative to a pre-stated formal specification" and the latter as "programs that mechanise a human or societal activity" [LR02]. Based on the *E*-type, he has declared eight laws of Software Evolution, which are cited in Table 2.5. While these laws are believed to mainly apply to monolithic, proprietary software, there are evaluations for open source software as well: [GT00].

Meta Model Evolution is put into the context of Software Evolution in subsection 2.3.6.

2.3.5 Evolutionary Algorithms

In Computer Science, the field of *Evolutionary Algorithms* (EAs), is a sub-field of *Evolutionary Computation* [Ash06], which is further a sub-field of *Artificial Intelligence*.
Evolutionary Algorithms are introduced by Dr. Thomas Bäck in his book "Evolutionary Algorithms in Theory and Practice" as follows:

> *Evolutionary Algorithms* (EAs), [...], is an interdisciplinary research field with a relationship to biology, Artificial Intelligence, numerical optimization, and decision support in almost any engineering discipline. [As an introduction], it is sufficient to know that these algorithms are based on models of organic evolution, i.e., nature is the source of inspiration. They model the collective learning process within a *population* of *individuals*, each of which represents not only a search

No	Brief Name	Law
I 1974	Continuing Change	*E*-type systems must be continually adapted else they become progressively less satisfactory.
II 1974	Increasing Complexity	As an *E*-type system evolves its complexity increases unless work is done to maintain or reduce it.
III 1974	Self Regulation	*E*-type system evolution process is self regulating with distribution of product and process measures close to normal.
IV 1980	Conservation of Organisational Stability (invariant work rate)	The average effective global activity rate in an evolving E-type system is invariant over product lifetime.
V 1980	Conservation of Familiarity	As an *E*-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
VI 1980	Continuing Growth	The functional content of *E*-type systems must be continually increased to maintain user satisfaction over their lifetime.
VII 1996	Declining Quality	The quality of *E*-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
VII 1996	Feedback System (first stated 1974, formalised as law 1996)	*E*-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

Table 2.5: Lehman's Laws of Software Evolution [LRW+97]

> point in the space of potential solutions to a given problem, but also may be a temporal container of current knowledge about the "laws" of the environment. The starting population is initialized by an algorithm-dependent method, and evolves towards successively better regions of the search space by means of (more or less) randomized processes of *recombination*, *mutation*, and *selection*. The environment delivers a quality information (fitness value) for new search points, and the selection process favors those individuals of higher quality to reproduce more often than worse individuals. The recombination mechanism allows for mixing of parental information while passing it to their descendants, and mutation introduces innovation into the population. This process is currently used by three different mainstreams of Evolutionary Algorithms, i.e. *Evolution Strategies* (ESs), *Genetic Algorithms* (GAs), and *Evolutionary Programming* (EP). [Bäc96]

One field of application for Evolutionary Algorithms is to write programs:

> *Genetic programming* is a systematic method for getting computers to automatically solve a problem starting from a high-level statement of what needs to

> be done. Genetic programming is a domain-independent method that genetically breeds a population of computer programs to solve a problem. Specifically, genetic programming iteratively transforms a population of computer programs into a new generation of programs by applying analogs of naturally occurring genetic operations. [KR92]

Since meta model evolution is about co-adapting models to modified meta models, it is not relevant whether the meta model has been modified by a human being or by an evolutionary algorithm. Furthermore it has to be evaluated whether genetic programming could potentially lead to reasonable meta models or meta model modifications, but that is not the scope of this thesis.

2.3.6 Evolution in Model Driven Software Engineering (MDSD)

When the changes observed in the field of *Software Evolution* originate from applying techniques of Model Driven Software Development, new dimensions of evolution become visible. In [vDVW07], they are listed as follows:

- In *regular evolution*, the modeling language is used to make the changes; the development platform, that is, the set of domain specific and general purpose languages, is fixed.
- In *meta model evolution*, changes are required to the modeling language to improve its expressiveness. Such changes may require migration of models.
- In *platform evolution*, the underlying infrastructure, such as the code generators and the application framework, is required to change, because of new requirements in the target platform. Existing models may remain unaffected by such changes, if the modeling language abstracts over the specifics of the target platform.
- In *abstraction evolution*, new modeling languages are added to the set of (modeling) languages to reflect increased understanding of a technical or business domain. After introducing new languages, the old system should be migrated to make use of it.

It can be added that besides requiring the migration of models, *meta model evolution* may also require the migration of other dependent artifacts, such as code generators, interpreters and model editors. Furthermore, *abstraction evolution* is related to *meta model evolution* in the sense that abstraction evolution describes a change in the level of abstraction in the modeling languages. Since a language consists of a meta model, a concrete syntax and a semantic (subsection 2.1.5), an evolved meta model can imply *abstraction evolution*.

For this thesis, only the dimension of meta model evolution is relevant.

2.4 Fundamentals of Co-Evolution

The field of *Co-Evolution* or *Coupled Evolution* investigates situations in which the modification of one fragment requires the adaptation of a second fragment. While Darwin observed the possibility of co-evolution for species (subsection 2.4.1), there are scenarios of co-evolution in computer science (subsection 2.4.2) as well, of which one is the meta model evolution and model co-evolution.

2.4.1 Co-Evolution in Biology

M. Wade summarizes in his article "The co-evolutionary genetics of ecological communities" that Darwin already noted the presence of co-evolution in biology:

> Darwin [Dar59] recognized that ecological interactions among species are the most important processes that drive the adaptive evolution and diversification of species: "I can understand how a flower and a bee might slowly become, either simultaneously or one after the other, modified and adapted in the most perfect manner to each other, by continued preservation of individuals presenting mutual and slightly favourable deviations of structure." Patterns of co-adaptation result from the process of co-evolution, which occurs whenever two ecologically interacting species exert reciprocal selection pressures on one another and the response is inherited. Although most interactions between species, including those between competitors, predator and prey, host and parasite, or host and symbiont, generate reciprocal selection pressures, the specific pattern that emerges over time varies with the nature of the ecological interaction, the genetic architecture of the co-evolving traits and the degree of co-transmission across generations. The degree of co-transmission alters the predictions and outcomes of the standard co-evolutionary models that assume random mixing of species. [WAD07]

It is interesting to see that the concept of co-evolution is not an invention of the field of computer science. Wade describes the process of co-evolution among species as to "generate reciprocal selection pressures". Therefore, the impact of two species on each other is bidirectional.

2.4.2 Co-Evolution in Computer Science

In Computer Science the pattern of *bidirectional co-evolution* as seen in biology (subsection 2.4.1) can be found for the scope of *populations* as well: For example, web technologies[4] need the support of web browsers[5] to be attractive for users. Therefore, it is advantageous for a web technology to be easily integratable into the web browsers. Web browsers on the other hand strive to implement the best web technologies in order to be attractive for the users themselves.

For the scope of *individuals*, however, developers prefer the pattern of having primary artifacts and dependent artifacts. When the primary artifact changes the dependent artifact has to be co-evolved. The dependent artifact is not allowed to populate changes to the primary artifact, which makes their relationship *unidirectional*, and which helps the developer to introduce change by having a clean pattern. For scenarios where the dependent artifact is an automatically derived artifact, the effort of co-evolution only consists of re-triggering the derivation process of the dependent artifact.

Co-Evolution in Computer Science describes the scenario in which a primary artifact changes and derived or dependant artifacts have to be co-adapted to these changes. Typical examples are:

[4]Such as HTML, XHTML, XML, CSS, PNG, JPEG, SVG, HTTP, AJAX, Adobe Flash.
[5]Such as Mozilla Firefox, Opera, Google Chrome, Safari, Konqueror, and the Microsoft Internet Explorer.

Data Structure Evolution

Depending on which format is used to define the data structure, there are various fragments which have to be co-evolved. Generally, data structures have instances and if the instances have to survive modifications of the data structure definition, the instances have to be co-adapted. This topic is known under several names, such as *schema evolution* and *meta model evolution* and is investigated in detail in section 2.5. However, there are more candidates for co-adaptation. Data Structures can have implementations in a programming language which can be automatically derived from the data structure definition, hand written, or a combination of both. In case of automatic derivation, the co-evolution is done by re-executing the generation process. In all other cases a more advanced tooling or manual co-evolution is required. If the data structure implementation is exported as public API, there is a case of *interface evolution* (see next description) as well. If frameworks from the field of modeling have been used, there can be model to model or model to text transformations (subsection 2.1.2) which depend on the data structure definitions. In case the data structure definition changes, the transformations have to be co-adapted as well.

Interface Evolution

Interface Evolution is also known as *API Evolution* and is closely related to *Library Evolution*. The evolving artifacts are sets of operation[6] signatures and their semantics. The artifact which is required to co-evolve is the source code which calls these operations. It is interesting to see, that the operation's parameter types and return types are data structure definitions as well. Therefore, Interface Evolution is related to *Data Structure Evolution*.

In [HD05], tool support for interface evolution, which is nowadays part of the Eclipse IDE, is introduced. The interfaces have to be modified using refactoring operations which are recorded and delivered in conjunction with the modified libraries. The maintainer of the source code which uses the library can then re-play the recorded refactorings and thereby automatically co-adapt his source code.

2.4.3 Co-Evolution and Refactoring

According to the definition of Martin Fowler,

> Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. [Fow99]

According to this definition it is important to note that a refactoring neither adds nor removes functionality from an existing system.

When looking at refactorings, it is possible for some refactorings to distinguish between a primary modification and dependent modification, whereas the latter is required to keep the source code consistent. For example: For a *rename refactoring* of a local variable the primary modification is to change the variables name at its declaration. To keep the code consistent, the variables name has to be changed at all places where the variable is being used. Another example is to modify a Java classes name in Eclipse. If the Class is referenced by a

[6] Depending on the programming language, this can as well be methods, procedures or functions.

`plugin.xml` because it contributes to an extension point, the class name in the `plugin.xml` has to be modified as well. To have an example that is not based on renaming: If a sub-class is introduced, some places in the source code may need to be adapted such that the sub-class and the super-class is used in the appropriate places.

As a result may be stated that some refactorings are scenarios of applied co-evolution. This suggests the idea to look at other cases of co-evolution as refactorings, which might be helpful for providing implementations – while keeping in mind that refactorings should neither add nor remove functionality.

2.5 Data Structure Evolution and Instance Co-Evolution

The field of *data structure evolution and instance co-evolution* has been widely investigated by the scientific community in various contexts and under various names, depending on how the data structures are defined. The most well-known are probably *SQL-* and *XML Schema Evolution* and *Meta Model Evolution*. This section gives an overview over the related work and the following section (chapter 3) analyzes the details in the context of meta model evolution and model co-evolution.
To judge about the comparability of the different fields and the transferability of the concepts to meta model evolution, it has to be taken into account that the concepts of each field depend on the expressiveness of each field's data structure definition language.

2.5.1 Relational Database Schema Evolution

Of all fields of schema evolution, this is probably the oldest. The Schema Evolution Bibliography [RB06] maintained by the University of Leipzig lists the oldest article on the topic to be from 1977 ([SHT+77]).

In Database Schema Evolution the data structure is defined using the SQL Data Definition Language (SQL DDL) using concepts such as *database*s, *table*s, *column*s, *type*s, *primary key*s, *foreign key*s, etc. The instance is the contents of the tables – the datasets stored in the table's rows. When changing the schema, there can be the need to co-adapt the instances to avoid the loss of information. Since many of today's Database Management Systems support advanced features such as *temporary tables*, *nested queries* and program code running inside a database (PL-SQL), the developer has powerful tools at hand to implement an instance co-adaptation algorithm.

In literature, [Rod95] gives a comprehensive introduction on the topic. He defines the terminology of *schema modification*, *schema evolution* and *schema versioning*. Furthermore, he looks at modeling and architectural issues as well as at impacts on existing SQL queries. In [Ber03] *model management* is described as an approach to manipulate models and mappings as bulk objects using operators such as *Match*, *merge*, *diff*, *compose*, *apply*, *copy*, *enumerate*, and *modelgen*. Then it is shown how to apply these operators to the problems of: *schema integration*, *schema evolution*, and *round-trip engineering*. In [CMTZ08] the changes of a popular open source application, the Media Wiki, the wiki application which is used to run Wikipedia, are analyzed. The changes are classified into Schema Modification Operations (SMOs), which are *create*, *drop*, *rename*, *distribute*, *merge* and *copy* for *tables*, and *add*, *drop*, *rename*, *copy*, and *move* for columns. The analysis includes the quantities of the SMOs.

2.5.2 XML Schema and DTD Evolution

Both XML Schema Definitions (XSDs) and Document Type Definitions are notations to define the structure of data which is persisted in an XML-based syntax. The concepts supported by XML Schema cover and outnumber the ones of DTD by far, therefore it is possible to transform a DTD to an XML Schema without losing information. Even though there is a transformation from XML Schema to EMF Ecore as well, there are concepts in XML Schema that Ecore does not support natively and which are implemented using `ExtendedMetaData`. Examples for XSD concepts that can be mapped to Ecore concepts are *complex datatypes*, *simple datatypes*, *elements* and *attributes* ([XSD04]). Examples for XSD concepts that are not natively supported by Ecore are complex datatypes restricting (instead of extending) other ones, simple datatypes that compose other simple datatypes, groups and choices. Even though there are powerful technologies like Xpath, XSLT, DOM, and SAX available, it is not trivial to implement a co-adaptation algorithm for XML files that's XSD or DTD has changed.

In literature, [SKC+01] introduces the concept of first looking at atomic primitives which describe atomic changes. It presents a list of theses changes for Document Type Definitions (DTDs), specifies preconditions for them and outlines their execution semantics. [GMR05] applies this idea to XML Schema and optionally composes the atomic change primitives to higher level primitives. [LL01] represents changes in the formats of documents as stepwise transformations on the underlying DTDs. The migration of the XML data is done by the DTD transformations which are implemented using XSLT.

[Kle07] introduces a custom meta model which they call "conceptual model" and which can be transformed to XML Schema. When editing the conceptual model, the editing steps are recorded and used to derive an algorithm for transforming the instances.

[Har07] looks at change operations modifying the XML Schema while distinguishing between primitive operations, complex operations and while taking into account the information capacity.

2.5.3 Evolution in UML

Looking at the field of UML there are three cases to distinguish – whether stereo types, standardized meta models or custom meta models are used. In research, the following articles may serve as entry points to the field.

In an early approach, [KR01] describes the change primitives *add*, *remove* and *replace* using XMI and applies them to XMI. Later, [HSE05] introduces an *instance model*, which connects the elements of the meta meta model, the meta model and the model. Furthermore, the paper strives to identify patterns in the kind of meta model changes and thereby ease the creation of the model co-adaption algorithm. In [SK04] a Domain Specific Visual Language (DSVL) is introduced which is based on the theory of *graph rewriting* to evolve UML class diagrams.

Profiles and Stereotypes

UML 1.x introduced the concept of applying custom Stereotypes on model elements. This is a less powerful substitute for custom meta models which can be considered as annotating model elements. Therefore, changing profiles and stereotypes does not fall into the field of

this thesis.

Standardized Meta (Meta) Models

Standardized means they are not intended to be modified by individuals. However, there are the rare cases when the OMG updates the standard. So the happening evolution has to be addressed by the UML tool developers, but not by the application developers (the users of the UML tools). For example, the specification of the MOF 2.0 standard ([OMGd]) has a chapter dedicated to "Migration from MOF, v1.4", which is an informal description of the migration process.

Custom Meta Models

It is possible to define custom meta models using the MOF for which it is likely to undergo modification over the time of their lifespan. This scenario falls into the context of this thesis, especially since the Ecore and MOF have have a similar subset of elements.

2.5.4 Textual Language Evolution

As introduced in subsection 2.1.5, a language consists of a meta model combined with a concrete syntax and a semantic. A grammar defines the concrete syntax for the meta model and a compiler, code generator or interpreter defines the semantic. Therefore, there can be evolution on all three dimensions: *meta model*, *syntax* and *semantic*. It is common that evolving a language involves all three dimensions and that the language designer tries to restrict his work to non-breaking changes for the syntax and the semantic.

Generally, co-evolving a language's instances can be addressed on two levels: The concrete syntax and the model. Both are further discussed in this section. But at first, there are, similarly to UML, and as already denoted in subsection 2.3.2, several scopes in which evolution for languages happens.

From General Purpose Languages to Domain Specific Languages

Formal (general purpose) languages have always been treated as something that does not change at all or just very slowly. This protects the existing implementations and the value of the developer's knowledge about a language. A popular example for a language modification breaking instances is *Java 1.4* introducing the `assert` keyword. One of *JUnit*'s central APIs was a method named `assert()`, thereby this method has been replaces with `assertTrue()` and JUnit in versions older than 3.7 wont work with Java 1.4 or newer.

In research, [Fav05] shows that – and how – programming languages have evolved. However, with the introduction of Domain Specific Languages [MHS05] the language's grammars and meta models become agile and existing DSL scripts (the instances of a grammar, source code) need to be co-adapted.

Textual languages have the speciality that their instances are usually persisted in the language's concrete syntax. Inspired by [SK04][7], this allows to distinguish between two

[7] [SK04] refers to *Concrete Syntax based Co-Evolution* as *Syntax Evolution* and *Model based Co-Evolution* as *Semantic Evolution*. The new names should avoid confusion since an evolved syntax can also be addressed with Model based Co-Evolution and an evolved meta model can be addressed with Concrete Syntax based Co-Evolution

techniques for Co-Evolution:

Concrete Syntax based Co-Evolution

When the grammar is being modified, the textual representation of the instances can be transformed by modifying the text so that it can be parsed again. This can be accomplished by applying "search and replace" operations or regular expressions. For carefully modified grammars this can be the easiest approach. However, this can be error-prone since the semantic is not taken into account and "search and replace" and regular expressions tend to match unintended regions when not verified manually. For example, using a simple "search and replace" for a certain text to exchange a keyword will also replace that text within string literals. This could be avoided by using a regular expression that excludes occurrences of the text within string literals. If the language supports the concept of comments as well, the regular expression would have to be extended not to match texts within comments. The more complex the language is, the more the regular expression-based matching is required to do be as powerful as the actual parser.

Model based Co-Evolution

The more solid approach is to parse the old instances of the grammar, use a model to model transformation on the resulting model and serialize it back to its textual representation using the new grammar. In [PJ07] an implementation for this approach is introduced. Three languages are defined for evolving languages: The first is to describe a list of operations which modify the grammar (Grammar Evolution Language, GEL), the second one is to list operations which co-evolve the syntax tree (Word Evolution Language, WEL) and the third one combines both as the Language Evolution Language (LEL).

Grammar Evolution

The term *Grammar Evolution* has been claimed by [RCN+98] for a genetic algorithm (subsection 2.3.5) and ins thereby not in the scope of this thesis.

2.5.5 Ontology Evolution

Gruber describes ontologies in theory as "formal, explicit specification of a shared conceptualisation" [Gru93]. These concepts are used to define how real life knowledge can be modeled. However, there are many ontology languages, a popular example is the W3C's Web Ontology Language (OWL, [W3Cd]) and each one needs to be investigates separately in the context of data structure evolution and instance co-evolution. In [NK04] the differences between schema evolution and ontology evolution are outlined.

2.5.6 Serialized Objects

Many programming languages (e.g. Java, PHP) provide mechanisms to serialize their objects to byte streams and to deserialize these byte streams to objects. When the classes, which define the structure of the serialized objects, change, the process of deserialization may fail. In [Mey96] the ideas of database schema evolution are adapted to co-evolve serialized objects when their classes change.

2.5.7 Ecore Model Evolution

Ecore as a modeling language unites the common concepts of XML Schema, MOF and annotated Java, such as `EClass`, `EDataType`, `EAttribute` and `EReference`. `EAttribute` and `EReference` desribe the properties of an `EClass` and inherit from `EStructuralFeature`.

Since the growing user base both in academics and industry of the EMF (which Ecore is part of) and the availability of EMF as open source, there have been publications about evolving meta models and co-evolving models. In [BGGK07] a process model for semi-automatic model instance migration in face of meta model changes is suggested. In this process the detected changes between meta model versions are distinguished into *non-breaking*, *resolvable* and *breaking* changes which is further explained in section 3.9. Furthermore it is considered to gather user input before the model migration. In [Den08] *meta model alternation traces*, which are created using EMF's `ChangeRecorder` are refined with information how to co-adapt the models. Furthermore, in [HBJ08c, HBJ08a, HBJ08b] an extensive framework named COPE is introduced which implements the *operation-based* approach (section 3.10) and integrates as plugin into the Eclipse Workbench.

2.6 Patching

Patching has its origins in the UNIX-world ([Ray04]), where a patch is used to describe the differences between two text files. There, a patch is a text file itself which contains a standardized description of how those two files differ. Since the text fragments for both sides are stored, the patch is bidirectional: One patch file, which describes the differences between text file `A` and `B` can be used to recreate `A` from `B` as well as `B` from `A`. Furthermore, the patch contains intended redundancy: Text lines that surround the modified fragments. This improves readability for humans and when applying the patch it helps to find the correct region which has to be modified. This patch format has built the root of various Software Configuration Management (SCM) systems, such as CVS. Nowadays it is the defacto standard for exchanging modifications to text (source code) regardless of the used SCM.

Another field where the term *patching* is used is when fixing bugs or vulnerabilities in installed software programs. Some software vendors (with Microsoft as the most prominent example) regularly release patches which customers are encouraged to install on their computer systems. Such a patch has a binary format and modifies the software program executable and related files.

2.6.1 Patching Models

The UNIX' world's patch (UNIX-patch in the following) can be used for models as well, however, it operates on the model's textual concrete syntax. This has several disadvantages:

- *Bad Readability*: Having the differences serialized as a UNIX-patch adds additional complexity for the human reader to the model's concrete syntax. For example, when a model is stored in XMI, the reader has to be familiar with the syntax of XMI as well as with the UNIX-patch's syntax.
- *Unneeded Information*: A model's textual representation can be modified without modifying the model: For example, when adding whitespaces, linebreaks, indentation

or exchanging XMI-attributes with XMI-elements of the same name[8]. Therefore, a UNIX-patch can describe many changes which have no relevance for the model. This increases the patches size to the limit where it reaches the size of $size(model_1) + size(model_2)$. Furthermore, this increases the difficulty for the human reader to differentiate between relevant and unnecessary information, additionally to the challenging syntax.

- *Bad Analyzability*: If the UNIX-patch for a model is available, but not the to-be-patched models, the complete list of model elements which are added, modified or removed, can not be determined. Since the UNIX-patch describes the modifications in the textual representation, the complete model needs to be available to determine the full impact on the model for the textual modification.

2.6.2 Related Work

In academia, patching models becomes interesting when model differences need to be analyzed or re-applied. [Kön08] uses a patch format for models to analyze the changes that are being applied to models in the context of MDSD to detect common patterns and provide better tooling to support these patterns. For viewing, [CDRP07] introduces a meta model independent graphical representation for patches.

2.6.3 Patch Models vs. Model Transformations

While a patch defines modifications for a certain model, a model to model transformation defines modification for all instances of a certain meta model. This leads to the situation that while in a patch each instruction corresponds to exactly one model element, a model to model transformation consist of instructions how to modify instances of a certain meta model element, which is typically a data type or class.

2.6.4 Patch Creation

The process of creating a patch can be split into two steps: The first is to obtain the differences between two artifacts, the second is to store these changes as a patch format.

Obtaining Changes: Diffing vs. Recording

To obtain the changes between two artifacts, there are generally two options to consider: Two existing artifacts can be compared to obtain a difference or change-operations can be recorded while an artifact is edited. UNIX-patches are usually created using the comparison approach. For details about the advantages and disadvantages related to models, please see subsection 4.2.1.

The diff-approach can furthermore be split up in two sub-parts: First, model elements or text-sections of the two to-be-compared artifacts have to be matched with each other. Second, the mapping resulting from the matching can be used as a basis to determine the

[8] All these examples only apply for languages in which the suggested changes have no impact on the semantic meaning of the model.

actual differences. The matching phase can be called the "weakness" of the comparison-approach, since except for the scenario where model elements have instance-based IDs[9], the matching algorithm has to use fuzzy logic to determine the likeliness for two elements or text sections to match. Therefore, the matching becomes more a guess than a definite decision.

Creating the Patch

After having obtained the changes between two artifacts, they need to be stored in an appropriate patch format. Besides the change information, the patch format may contain redundancy to improve readability for humans and to be more fault-tolerant when being applied.

2.6.5 Patch Execution: Applying

Before an artifact can be modified, the locations within the artifact (text sections or model elements) for which the patch describes differences have to be identified. For UNIX-patches, the offset within the text is used for this identification and for model patches, unique elements IDs or path-expression (subsection 2.1.2) can be used. The challenge in applying the patch lies in identifying the right places for the modifications, even though the to-be-patched model or text file is not exactly in the state expected by the patch, but has been slightly modified in the meantime. This is where redundancy becomes useful. Since the UNIX-patch does not only store the offset of the to-be-modified text sections, but also some lines of text right before and after these section, they can still be found even though the offset has turned incorrect over time. For a change-proof identification of model elements, it is possible to store attribute values of unmodified attributes as well, path-expressions, element IDs (if available), container-relationships (if available) and type information.

After having identified the relevant text sections or model elements, these can be modified according to the information stored in the patch.

[9]For example UUIDs for XMI.

3 Analyzing the Challenges

After having explored the related fields of meta model evolution and model co-evolution, this section introduces the basic challenges, classifications and approaches in the context of meta model evolution and model co-evolution.

3.1 Context

To keep this section pragmatic, it is reasonable to look at the areas in which models co-evolve to changes of meta models. The two common scenarios for model usage which are already named in subsection 2.1.3 are now inspected for the occurance of meta model evolution:

3.1.1 Data Persistence

This is the well-known scenario: A user's data is stored in a file through an application, the application is then updated and has a new file format, but it is supposed to be able to read the old files. Therefore, the application needs to upgrade the files automatically to the current version, which is a scenario of co-evolving the file's content to the modified structure definition. While an upgrade is needed at this time, a downgrade can be desirable as well: When the user stores the application's data, he might want to have the option to choose an older format to keep the compatibility with the older version of the application. The probably most famous example is MS Word and all its document format versions.

3.1.2 Data Exchange

In this scenario, two applications have to exchange data, which is usually related to *Interface Evolution* (subsection 2.4.2). The exchange can be done locally (message bus, Inter Process Communication (IPC), etc.) or over a network (SOAP, RMI, etc.). For this communication to be successful, both applications need to support the same data format. In case one application updates the data format and the other one stays with the old one, an interpreter is needed which translates the one version of the data format to the other version. This interpreter actually co-evolves the exchanged messages to the modification of the data format. The interpreter could be integrated into one application or act as a dedicated service that passes through/migrates the messages. An example for the need of migrating messages is the evolution of interfaces in Service Oriented Architectures (SOA).

3.2 The Process

Now that the two common scenarios of applicability of meta model evolution have been introduced, this section describes the common process of how to create model migration algorithms and how to execute them. This process is intended to provide a general overview

and introduction for this topic. Figure 3.1 shows the process which is an extended version of the process suggested by [BGGK07].

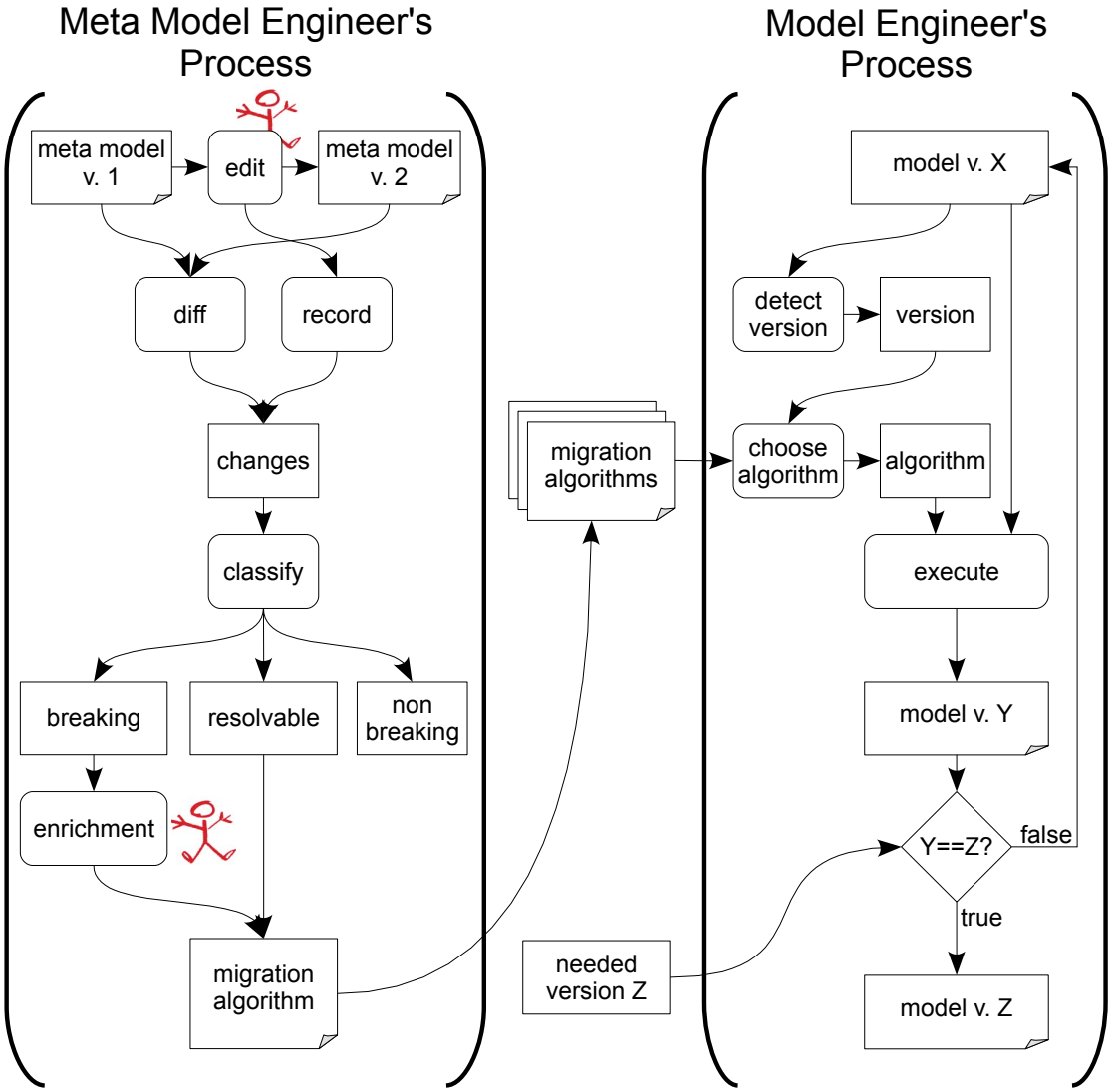

Figure 3.1: Meta Model Evolution and Model Co-Evolution Process Big Picture

For a quick and simple start, it is helpful to reduce the process to two black boxes and look at their input and output parameters. The first black box is the process of the *meta model engineer*, who edits the meta model and creates a model migration algorithm. Therefore, the input is the set of changes between the two meta model versions and the output is the *migration algorithm*. The second black box is the process of the *model engineer*, who (knowingly or unknowingly) executes one or more model migration algorithms and thereby updates the model from some version `X` to the needed version `Z`. Therefore, the input parameters are the to-be-migrated model, all available migration algorithms, and the needed target version of the model. The output is the updated model. The need to distinguish between this two roles and this two processes is motivated in section 3.3.

After this introduction, there is no further need to look at the processes as black boxes – instead they are going to be explained in detail.

3.2.1 The Model Migration Algorithm

The model migration algorithm, which is a result of the meta model engineer's process and an input parameter of the model engineer's process, is technically a model to model transformation operating on two different meta models (subsection 2.1.2). The two different meta models are two different versions of the same meta model, but technically, this makes them two distinct meta models. The task of the model migration algorithm is into transform instances of the one meta model version to instances of the other meta model version. This can be applied as an upgrade as well as a downgrade.

There are different approaches for implementing the migration algorithm. The model can be migrated in one big step, which is typical for the diff-approach (section 3.10) or by applying many smaller change operations which modify the model as well as the meta model and create many intermediate versions of both (section 3.11). Furthermore, it has to be considered, that, depending on the used framework, a meta model may be read-only (frozen) while it has instances (section 3.6).

3.2.2 The Meta Model Engineer's Process

The meta model engineer's process starts with determining the changes between the two meta model versions. There are two approaches to do so: He can record all editing operations executed to modify the meta model or he can compare the two versions and create a diff of the two versions after he has finished editing them. While the recording approach requires editor integration, the diff approach can have difficulties to correctly match the two meta model version's elements with each other. The details about this and other advantages/disadvantages are evaluated in subsection 4.2.1.

After having obtained the changes, every single change is classified with regards to whether or not it breaks compatibility with existing instances. As determined by [BGGK07], there are three categories: *breaking*, *breaking* but automatically resolvable and *non-breaking*. section 3.9 explains the details. For the breaking changes which are not automatically resolvable the meta model engineer will have to specify the migration algorithm's behavior manually. Depending on the implementation of the tooling, he may choose from predefined behavior or provide code snippets which implement the desired functionality.

3.2.3 The Model Engineer's Process

The model engineer's process defines how an old model, which is an instance of meta model version `X`, can be transformed to an instance of meta model version `Z` by executing one or more model migration algorithms. At the beginning, the meta model version of the input model has to be detected. This can be a non-trivial task, which is described in detail in section 3.4. Based on the version, an appropriate model migration algorithm is chosen from a pool of algorithms which has to be supplied by the meta model engineer. By executing this algorithm the model is transformed to be an instance of meta model version `Y`. Depending on whether this version `Y` is the needed version `Z`, this process has to be repeated until the desired version is reached.

3.3 Challenge: Model and Meta Model not Available Simultaneously

When looking at the two scenarios of *data persistence* and *data exchange*, it is easy to notice that at the time when the developers modify the meta model, the model is usually not available. This happens because it is stored at a different computer or because it does not exist yet and it will be created to a later point. It may seem like a trivial observation, but the consequences are interesting: If the model would be available during the modification process of the meta model, the model could be co-adapted in place. The impact of every operation the developer uses to change the meta model could be applied to all models as well. A workflow could look similar to a rename-refactoring to the developer: When he renames a method the method-calls calling this method are co-adapted automatically everywhere within his workspace.

When the model is not present while the meta model is modified, this has the consequence that all information needed for the migration process has to be encapsulated in a *migration algorithm* which can be deployed separately or with the application. Furthermore this requires to distinguish between the person who modifies the meta model (the *meta model engineer*) and the person who executes the migration algorithm (the *model engineer*) since they are not necessarily the same. In the scenario of data exchange it is likely that there is no person in the role of the model engineer and therefore the migration algorithm is expected to do its work quietly in the background. In any case, it is valuable for the migration algorithm not to require user interaction.

3.4 Challenge: Identify Correct Version

The very first thing an application has to do when it accesses a model is to determine the version of the corresponding meta model. Assuming that multiple versions of the model can exist, this is required to choose the appropriate migration algorithm. Furthermore, knowing the correct version can be required to deserialize the model: section 3.5.

In the world of XML and XMI meta models (which can be in the form of XML Schemas), versions are identified by *namespace URI*s. By requiring that every meta model version must have a unique namespace URI, the detection of the version becomes trivial. Luckily, using unique namespace URIs for different meta model versions is already established best practice[1].

The process of detecting the version for a textual representation of a model, which is defined by a custom grammar, can not be solved by a generic approach based on the file contents. Every version of the language has to include some indicator to specify the correct version of the grammar. Since choosing the right version of the grammar is needed for the parsing process, it is required to identify the version before the deserialization.

A general approach which avoids accessing a file's contents is to have the file extension indicate the version. While this is an easy-to-implement approach, it requires the application to register itself for all these file extensions and it might be considered a too prominent place for version information since as well people who do not read the file's contents are exposed

[1] Meta models like AUTOSAR, HL7, the ones specified by the W3C and Ecore: all have a version number or release date included in their namespace URIs.

to the version information.

3.5 Challenge: Deserialize old Models

This includes two sub-challenges: First, the old meta model must be available, and second, the implementation of the deserializer must be able to deserialize the model based on the old meta model.

3.5.1 Have Old Meta Model Available

Surely the easiest solution to have the old meta model available is to deploy it along with the application. But on the other hand, it has been shown that the instance migration algorithm is based on the changes between two meta model versions. In the case that these changes need to be stored it can be possible to recreate the old meta model version by applying the changes to the current meta model. This would require the change description to be cumulative and applicable in the opposite direction of the migration algorithm. In the standard scenario where an old model should be processed, at first the old meta model needs to be recreated by *downgrading* the current meta model and then the migration algorithm would *upgrade* the instances. In cases where the recreation of the old meta models is applicable the explicit deployment of the old meta models could be avoided.

3.5.2 Generic Deserializer vs. Version Specific Deserializer

The requirement for the deserializer is that it must be able to deserialize models of all possible meta model versions. For deserializing XML/XMI, this is trivial since there are deserializer implementations available which are *generic* in the sense that they do not need any source code specific to the meta model. Instead, it is sufficient to provide the meta model as the configuration for the deserializer. For Ecore models, this functionality is called *Dynamic EMF*.

The scenario is less trivial for textual languages. They are commonly based on a grammar which is used to generate a parser, which then becomes the fundamental part of the deserializer[2]. Since the parser is generated, it is *specific to one version* of the language and most likely the meta model. Therefore, the application is either required to use a generic parser or to deploy a parser for every version of the language.

3.6 Challenge: The Frozen Meta Model

To preserve consistency, it is common that a meta model is read-only (*frozen*) while it has in-memory instances. For EMF, this is true for `EPackages`, which represent the root-element of an EMF based meta model. This has consequences for the migration algorithm: For the scenario in which the algorithm intends to co-evolve meta model and model as it is the case for the operation-based approach (section 3.11), it would have to create a complete meta model for each intermediate version to be able to have an intermediate model of this version. Since the changes between intermediate versions tend to be relatively small compared to the

[2] Additionally, a lexer and a linker can be required.

total size of the meta model, this adds a lot of overhead. However, there are two strategies to avoid this problem:

3.6.1 Migrate in One Step

The migration algorithm can do the migration in one step and avoid the creation of intermediate (meta) model versions. This procedure works well with the diff-approach (section 3.10) since it does not split up the changes made between two meta model versions based on the order in which they were made.

3.6.2 Use Intermediate Generic Datastructure

To do a coupled evolution of meta model and model by executing single operations and creating intermediate versions, the migration algorithm can detach itself from the constraints of the modeling framework and use a custom generic data structure. Then, modifying the meta model becomes possible as well as creating only the actually modified parts of a model and the meta model for intermediate version. However, this makes consistency a responsibility of the migration algorithm.

3.7 Classification: Construct, Refactor, Destruct Meta Model Concepts

As suggested by [Wac07], a first start to classify changes applied to a meta model is to distinguish *construction*, *refactoring* and *destruction* of meta model elements. The process of refactoring includes *moving* and *modifying* elements. This classification happens purely at the meta model level and therefore there are no assumptions about the migration of the models.

3.8 Classification: Expand, Preserve, Reduce Information Capacity

To classify the changes applied to a meta model with regards to models, it is important to look at the changing information capacity. Information capacity is defined as the potential information that can be expressed using instances (models) of a certain meta model. It might seem obvious that a construction of a meta model concept leads to an increase of the meta model's information capacity, a refactoring preserves the information capacity and a destruction reduces the information capacity. However, this is not true for all cases: In Ecore, an `EStructuralFeature` has the property `upperBound` which indicates how many values a feature may hold. `upperBound > 1` or `upperBound == -1` indicate[3] that the feature can hold more than one value and thereby holds a list. When modifying `upperBound` from 1 to -1, this is classified as a refactoring but since the model can now hold an arbitrary amount of values in this feature instead of one, it also has to be classified as an expansion of the information capacity.

[3]For -1, the list's size is unlimited.

3.8.1 Definition: Preservation of Information

When transforming model 1 (M_1) to M_2, all information is preserved when it is possible to transform M_2 to M_3 with M_3 being equal to M_1.

3.8.2 Expanded Information Capacity

When the modification of Meta Model 1 (MM_1) *expands* the information capacity and leads to MM_2, then

- all instances of MM_1 must be migrateable to MM_2 while preserving all information.
- there has to be at least one instance of MM_2 which can not be migrated to MM_1 while preserving information.

3.8.3 Preserved Information Capacity

When the modification of Meta Model 1 (MM_1) *preserves* the information capacity and leads to MM_2, then

- all instances of MM_1 must be migrateable to MM_2 while preserving all information.
- all instances of MM_2 must be migrateable to MM_1 while preserving all information.

3.8.4 Reduced Information Capacity

When the modification of Meta Model 1 (MM_1) *reduces* the information capacity and leads to MM_2, then

- there has to be at least one instance of MM_1 which can not be migrated to MM_2 while preserving information.
- all instances of MM_2 must be migrateable to MM_1 while preserving all information.

3.9 Classification: Non-Breaking, Resolvable, Breaking Changes

Introduced by [BGGK07], this classification distinguishes how meta model changes are reflected in the migration algorithm and which level of automation can be archived when creating the migration algorithm. The titles of these classes relate to whether the modification of the meta model implicitly requires migration of the models to preserve the models as valid instances.

3.9.1 Non-Breaking Changes

Non-Breaking changes do not require any migration of instances and thereby do not need to be considered in the migration algorithm. An example for this is the construction of optional `EStructuralFeature`s.

3.9.2 Breaking, but Resolvable Changes

Breaking, but *Resolvable Changes* require a migration of the instances and the migration algorithm can be derived automatically from these changes. An example is the renaming of `ENamedElement`s, which are for reasons of inheritance all `EClass`es, `EStructuralFeature`s, etc.

3.9.3 Breaking Changes

Breaking Changes require a migration of the instances and the migration algorithm can not be derived automatically. This requires the meta model engineer to intervene manually in the migration algorithm creation process and make choices between different strategies for the migration algorithm or to supply an implementation which solves this very problem. An example is setting `EStructuralFeature.unique` from `false` to `true` for a list-feature. A choice has to be made about what should happen with duplicate items. One option is to delete them, another one is to rename them. For the renaming scenario a strategy is required to calculate the new names. One strategy which can be chosen by the meta model engineer is to query the model engineer for a decision during the migration process.

3.9.4 Applicability of this Classification

How changes are classified strongly depends on the concepts which are provided by the meta meta model and on the tooling for the creation of the migration algorithm. While a change can be classified as *resolvable* in theory, the lack of support by the tooling for this change can require manual intervention and thereby make the change *breaking*.

3.10 Approach: Diff-Based

The *diff-based*, or *declarative* approach which the *metapatch* introduced in chapter 5 is based on, compares two versions of the meta model and uses the observed differences to semi-automatically derive a model migration algorithm. In literature, [dGSA07] suggests this approach for evolving DSLs based on the Microsoft DSL Tools.

Compared with the operation-based (section 3.11) approach, the diff-approach promises the following advantages:

- No editor integration is needed for obtaining the changes. For details, see subsection 3.2.2 and subsection 4.2.1.
- Since no editor integration is needed, the diff-based approach promises to be usable for scenarios where the meta model is not the primarily edited artifact – but derived from other formats, for example XML Schema or annotated Java classes.
- Since the migration algorithm (subsection 3.2.1) can do the migration in one step, this has advantages for performance and avoids the problems that can be caused by a read-only meta model (section 3.6).
- Complexity is reduced in the sense that since the final versions of meta models are compared with each other, editing operations that cancels each other out are automatically ignored.

For the disadvantages of the diff-based approach, please refer to the advantages of the operation-based approach section 3.11.

3.11 Approach: Operation-Based

The *operation-based*, or *imperative* approach records the change-operations the meta model engineer uses to modify the meta model. These change operations can be enriched by the meta model engineer to resolve *breaking* (section 3.9) changing. This leads to a sequence of migration steps, each co-evolving meta model and model, which add up to the migration algorithm. In literature and practice, the COPE framework [HBJ08b, HBJ08a, HBJ08c] follows this approach.

Compared with the diff-based (section 3.10) approach, the operation-based approach promises the following advantages:

- As discussed in subsection 4.2.1, editor integration allows to obtain accurate information about the movement of model elements. Furthermore, editing operations which apply multiple modifications in one step, for example refactorings, can be recognized as single operations.
- It is possible to create tooling which queries the meta model engineer for information about the model migration at the time when the engineer executes operations to modify the meta model.
- Complexity is reduced to two aspects: At first, for each editing operation the needed model migration has to be created. At second, the right sequence of editing operations has to be found.

For the disadvantages of the operation-based approach, please refer to the advantages of the diff-based approach section 3.10.

4 Prerequisite: Epatch

The Epatch format is a model which describes the differences between two models and which has been inspired by the UNIX-world's patch (section 2.6). When two models are available, the Epatch can be derived. When one model and the Epatch are available, the other model can be reconstructed.

For meta model evolution the Epatch format is used to describe the differences between meta models which then builds the basis for co-adapting the meta model's instances.

4.1 Format Requirements

To render more precisely what the patch format should provide, the following sections define the requirements and state why they are useful in the context of meta model evolution.

4.1.1 Complete

For any model A and any model B with A and B being instances of the same meta model the Epatch must be able to create B from A (and A from B, which is required by bidirectionality).

4.1.2 Bidirectional

An Epatch must be bidirectional. If a model B is created by applying an Epatch to model A, bidirectionality requires that by applying the same patch to model B model A is reconstructed.

For meta model evolution this ensures that any version of the meta model can be reconstructed, no matter whether the needed version is older or newer than the provided version.

4.1.3 Declarative

By being declarative, an Epatch focuses on *how* models differ, rather than *what* steps need to be executed when applying a Epatch. As opposed to a declarative format, an *imperative* format would be a list of operations that have to be executed sequentially to change a model from one version to another version.

For meta model evolution, the imperative approach would require to co-adapt the meta model's instances for every single change operation and therefore lead to many intermediate versions of meta models and meta model instances. Using the declarative approach promises to allow the co-adaptation in one step.

Furthermore, the imperative approach does not guarantee that its change operations describe the direct way from a version A to a version B. The list of operations can contain operations that annul other ones.

4.1.4 Meta Model Agnostic

By being meta model agnostic, the Epatch format allows the to-be-patched models to be instances of any EMF-based meta model. However, they need to be instances of the same meta model, otherwise they can not have a common subset of elements.

For meta model evolution, this is not a mandatory requirement, since the to-be-patched models in meta model evolution are always instances of EMF Ecore. Being agnostic to meta models can make the Epatch format applicable even outside the scope of meta model evolution.

4.1.5 Able to Describe Moves and Copies

In the case that an element *EA* has been removed from location *LA* and an element *EB* has been added to location *LB*, the patch should describe whether *EA* and *EB* are the same elements. The same is valid for differenciating between newly instantiated elements and elements that start as a copy from another element.

At first, this capability helps to keep patches slim, since in case of an *add* and a *remove*, a bidirection patch format has to store the complete contents of both the added and the removed element.

Secondly, in meta model evolution it is valuable to know when a meta model's element has been moved to a different location, since the element might have instances that need to be migrated along. For example when moving an EAttribute from one EClass to another, it can be wanted to also move the EAttributes values from one EObject to another (subsection 5.5.3).

4.1.6 Support of Multiple Resources

In EMF, a Resource is responsible for serializing/persisting a model. By having an Epatch support multiple resources allows to have an Epatch split up a model from one file to multiple files, to merge models from multiple files to one file, or to redistribute the contents of many files between them. For meta model evolution this means not to restrict the meta model engineer to have his meta model stored in one single `.ecore` file.

4.1.7 Textual Representation

A textual representation of an Epatch provides an easy and convenient way to view, edit and persist an Epatch. Still, the usage of the textual representation should be optional to avoid dependencies on tooling needed only for the textual representation. For meta model evolution, having a textual patch format provides the opportunity to embed M2M-transformation code snippets which describe fragments of the model co-adaptation within the Epatch.

4.2 Use Cases

This section describes what can be done using Epatches and whether an implementation for this use case has been developed in the context of this thesis. For each provided implementation there is a dedicated chapter to explain the details.

4.2.1 Creating an Epatch: Comparing vs. Recording

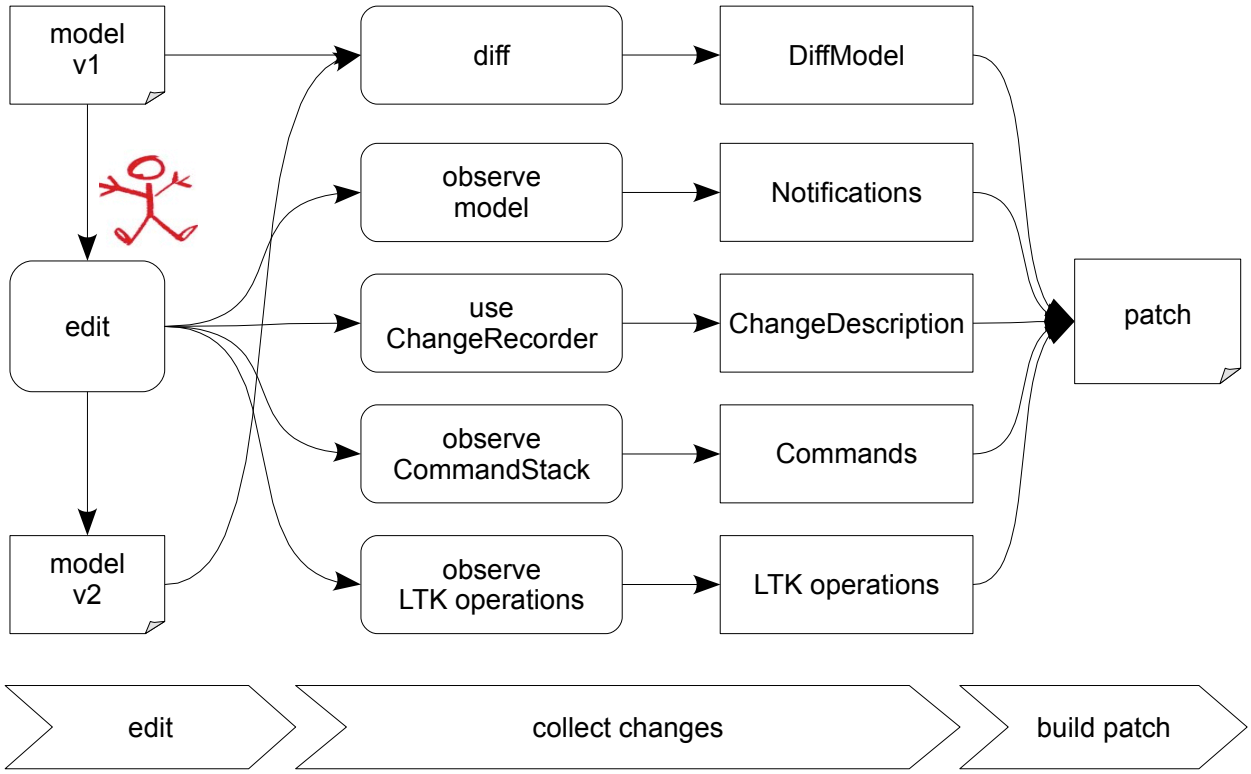

Figure 4.1: Epatch Big Picture

At the very beginning the developer must be able to create an Epatch. To do so, he can decide whether he wants to record an Epatch while editing a model or whether he wants to compare two existing versions of the models. These options are visualized in Figure 4.1, both have their advantages and disadvantages.

At first, it is easy to see that comparing two models leads to a declarative description of the model's differences while recording the changes applied to a model leads to an imperative list of change operations. Since the Epatch is supposed to be a declarative format (subsection 4.1.3), it is obvious that an implementation for creating an Epatch from model differences will be more straightforward. Additionally, with EMF Compare ([Eclh]) there is a framework available that can do the actual work of comparing the models. On the other hand, the difficult part of comparing models is to match the model elements of the left and the right side with each other. Only in the case when elements have unique IDs (for example Universally Unique Identifier: UUIDs [PL]), an implementation can verify whether two elements are identical. In all other cases, the implementation can judge about two objects equality by comparing the object's feature's values. This matching process might come to the wrong conclusions in some cases, which means that the developer might want to change the decisions of the comparison algorithm in some cases.

To record changes of a model, as shown by Figure 4.1, there are different ways to hook into the editing process of a model to get notified about the single changes. These options are explained in section 4.8. The main challenge when implementing the recorder lies in translating the imperative list of change operations into a declarative format.

Another disadvantage of the recorder approach is that as opposed to the comparison approach editor integration is needed. The details of the comparison approach are covered by section 4.7 and the details of the recording approach are covered by section 4.8.

4.2.2 Applying an Epatch while Copying the Model

One way to apply an Epatch is to copy a model and do the modification within the copying process. That way there will be two instances of the model, one in the original version and one in the modified version. This approach provides the opportunity to create a mapping from the model elements in the one model version to the model elements in the other model version. In meta model evolution, this mapping is the basis for migrating the instances (section 5.5). The details of the implementation are covered by section 4.9.

4.2.3 Applying an Epatch by Modifying the Model

Another way to apply an Epatch is to directly modify a model – without creating a copy from it. By avoiding the copying process, time of execution and the memory footprint have the potential to be lower. Since this behavior is not necessary to implement meta model evolution, there is no implementation in the context of this thesis.

4.3 Related Formats

In the Eclipse field there are two formats available to describe differences between models.

One is the *ChangeDescription* model ([Ecle]). It directly references the to-be-modified models and thereby requires the model to be available when working with the change description. The Epatch format avoids such hard references: subsection 4.4.2. Furthermore, the ChangeDescription is not bidirectional.

The other format is EMF Compare's *DiffModel* ([Eclh]), which has hard references to both compared models and thereby can not be applied to one model, since it requires both to be available when loading the DiffModel.

Thereby both formats do not qualify to be the Epatch format.

4.4 Design Decisions

To make the Epatch format usable and implementable, some design decisions are made. They are documented in this section, beginning with introducing some terminology:

4.4.1 Terminology

left/right An Epatch has a left and a right side concerning the modified resources and model elements. Typically, the elements/models/resources on the left side are the original versions and the elements/models/resources on the right side are the modified versions. For example, by applying an Epatch from *left to right*, the *right models* are created from the *left models*.

source/target When applying an Epatch from *left to right*, the left elements/models/resources become *source* elements/models/resources and the right elements/models/resources become *target* elements/models/resources. When applying an Epatch from *right to left* the roles of being *source* and *target* are swapped.

imported models There are models and model elements, that are involved in the patching process, which are not modified and which are not contained in one of the modified resources. They are *imported models*, for details see subsection 4.4.3.

4.4.2 Self Contained

EMF provides a meachnism to have (inter-model) references, which stores the URI of the referenced model and the fragmentURI[1] of the referenced model element. When loading a model containing a reference to another model, the reference first points to a so-called *proxy object*, which holds the mentioned URI and fragmentURI. This proxy can be resolved on demand to the actual referenced model element.

The Epatch has to reference elements in the left model, the right model and in imported models which might be referenced by the left and right models or which serve as meta models. When applying the Epatch, only the imported model and either the left model or the right model is available. This requires fine-grained control over the time at when proxies are resolved.

Then, before applying an Epatch the resource URIs of the left models and the right models are unknown, there is no determined value for the proxy to point to.

Furthermore, since there is a textual representation of the model, references should be human-readable.

As a conclusion the Epatch avoids the proxy mechanism to reference external elements and stores the fragmentURIs as plain strings instead, together with aliases for the containing resources.

4.4.3 Separation between Modified Resources and Referenced Resources

In the patching process one can differentiate between resources that contain model elements which are being modified and resources that contain model elements which are not modified. A clear separation of those two groups is valuable to make intuitively sure which resources are going to be affected when applying a patch. The not-to-be-modified resources are *imported resources/models*, which contain either referenced elements or which hold the types of the elements involved in the patching process and thereby play the role of a meta model.

4.4.4 Ignore Transient Values

EMF has the capability to mark EStructuralFeatures as *transient* which means that the feature's values are not serialized/persisted. Since these are usually derived values, they should be ignored by Epatch implementations.

[1] In the context of EMF, a fragmentURI is a URI's fragment which idenifies an `EObject` within a resource. Example: `file:/mydir/model.xmi#//MyEClass`.

4.4.5 No dependency on Xtext if not necessary

The textual representation of the Epatch should be optional, which means that the implementations for diffing (section 4.7), recording (section 4.8) and applying (section 4.9) should not have a dependency on the framework for textual DSLs, Xtext ([Eclf]). This helps to keep applications which integrate the named capabilities slim.

4.5 The Meta Model

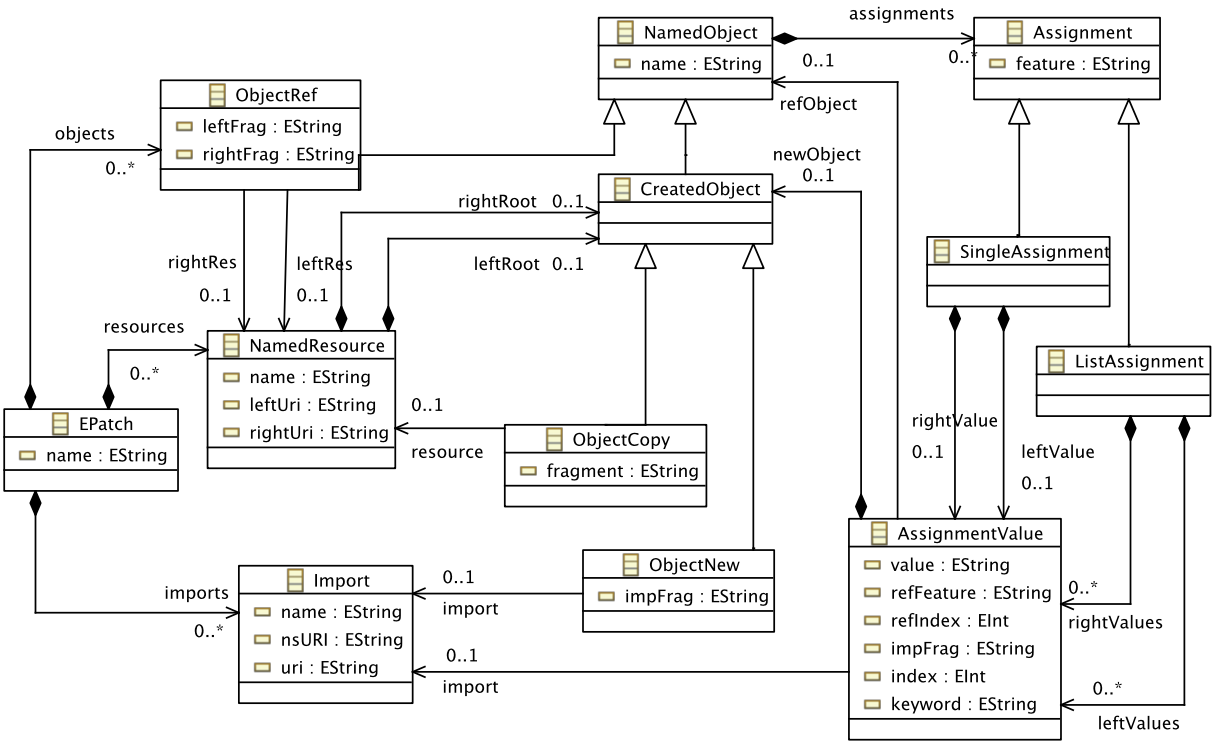

Figure 4.2: The Epatch Meta Model

An Epatch itself is a model and has a meta model, which is an instance of Ecore and has been derived from an Xtext grammar (section A.1). It is shown in Figure 4.2. This section gives an overview over the Epatch's structure. For possible instances and how they would be applied please refer to section 4.6 and section A.3.

The Epatch's root of the containment hierarchy is the EClass `Epatch`. It contains `Import`s, which declare *imported models* (subsection 4.4.1). Then there are `NamedResource`s, which declare the *left/right resources* (subsection 4.4.1) and `ObjectRef`s, which reference all model elements that are modified *and* present in both the left and the right resource.

`Epatch`es have names, but there is no semantic meaning for the name. They can be used for the user as well as for implementations to tell Epatches apart without having to depend on a file name. `Import`s, `NamedObject`s and `NamedResource`s have names as well, which are optional unless they are needed by the textual representation to reference objects of these EClasses - in case they are being referenced.

Imports can refer to an external model either via Namespace URI, which identifies an EPackage, or a URI which specifies the location of a model. Thereby URL would properly be a better name for it, but the EMF does not differentiate between URIs and URLs.

NamedResources define for both the left and the right side either a referenced model, or a newly created object. The latter becomes necessary for scenarios where the amount of left resources does not equal the amount of right resources.

There are different kind of EClasses that refer to EObjects in the to-be-patched model, all inherit from NamedObject and have a list of Assignments. There are ObjectRefs, which represent EObjects that are present both on the left and the right side of the patch and there are CreatedObjects, which are present either on the left side of the patch or on the right side of the patch. CreatedObjects are either ObjectCopys or CreatedNews. The latter references the EClass in an imported model which is to be instantiated when the object needs to be created. ObjectCopys reference another object, which is copied when the object needs to be created. It is not needed to have a special EClass to describe moves of EObjects since an EObject is implicitly moved when assigned to a containment reference.

As said, all EObject-referring EClasses have a list of Assignments whereas every Assignment refers to exactly one EStructuralFeature. It is not allowed to have multiple assignments for the same feature in the same object. Assignments are either SingleAssignments or ListAssignments, depending on whether isMany is true for the referd feature.

SingleAssignments hold one AssignmentValues for each the left and the right side, and ListAssignments hold AssignmentValues for all items on one side that differ from the other side.

AssignmentValues have different ways to describe a feature's value. If value is set, this is the string representation of an EAttribute's value. If refObject and optionally refFeature and refIndex are set, another EObject or feature's value is referenced. If import and impFrag is set, and EObject from an imported resource is described. If newObject is set, the value is a newly created EObject. index refers to the value's index within a ListAssignments list and keyword is set in case the value is null.

4.6 The Textual Representation

This section explains possible Epatch instances using the textual representation and characteristical examples. For more extensive examples see section A.3. The Epatch's grammar can be found in section A.1.

4.6.1 Hello World

The first example in Figure 4.3 shows a minimal Epatch. The Epatch has the name "ChangeString", and contains a NamedResource named "res0" as well as an ObjectRef. The NamedResource is present on the left side as "model-v1.xml" and on the right side as "model-v2.xml". This implies that the resource is neither added nor removed by the patch, but modified. The modification itself is described within the ObjectRef, which identifies an EObject in resource "res0" that has the fragmentURI "/" (which is the root object). Since resource and fragmentURI are the same for the ObjectRef on the left and right side, they are only stored once. ObjectRef holds a SingleAssignment for the feature "strval" which

contains the feature's value of the left model ("MyStringValue") and the feature's value of the right model ("Hello World"). The values are separated by a vertical line (|).

When then Epatch is applied from left to right, the feature "strval" from EObject "/" in resource "model-v1.xml" is expected to have the value "MyStringValue". This value is then modified to "Hello World".

When then Epatch is applied from right to left, the feature "strval" from EObject "/" in resource "model-v2.xml" is expected to have the value "Hello World". This value is then modified to "MyStringValue".

```
1 epatch ChangeString {
2   resource res0 {
3     left uri 'model-v1.xmi';
4     right uri 'model-v2.xmi';
5   }
6   object res0#/ {
7     strval = 'MyStringValue' | 'Hello World';
8   }
9 }
```

Figure 4.3: A minimal Epatch

4.6.2 Modify Lists

The second example, which is shown in Figure 4.4, modifies a list by adding an element, moving an element, and removing an element. The EStructuralFeature which is to be modified, and for which `isMany` is expected to be `true`, is named "intlist" in this example and is located within the left and right model in the same way as "strval" is located in the previous example.

This time, the assignment is a `ListAssignment` which can be recognized by it's content being surrounded by a left square bracket (`[`) and a right square bracket (`]`). The items left of the vertical line describe what is expected to be present in the left model and the elements right of the vertical line (`|`) do the same for the right model. Since the data structure is a list, every item can be identified by a unique index, which is the number in front of the colon (`:`). Numbers surrounded by square brackets (`[4]` and `[1]` in this example) are references to elements in the other side's model's list. The items are sorted by their indexes, the items on the left side of the vertical line descending and the items on the right side ascending. This sorting is valuable when executing the Epatch:

When the Epatch is applied from the left to the right, the EStructuralFeature "intlist" in EObject "/" in resource "model-v1" is expected to be a list which holds the value "15" at index 5 and some value (it can be determined which one it was supposed to be after the patch has been applied) at index 4. To do the actual modification the items can be processed straight forward from the left to the right, starting with `5:'15'` and ending with `4:[1]`. All values identified by the items on the left side of the vertical line have to be removed from the list and the values described by the items on the right have to be inserted into the list. The "add" operation is considered to be an "insert" at the end of the list. Before starting to modify the list it is needed to memorize the values referenced by items on the right side

of the vertical line since they might be removed from the list and modifying the list might make the index point to the wrong value. The actual modification process is:

1. The value at index 5 is removed.
2. The value at index 1 is removed.
3. The value “12” is inserted at index 2.
4. The value which was at index 1 in the unmodified list is inserted at index 4.

Forcing this order of execution ensures that all indexes point to the intended values, even though the list is modified.

When applying the patch from the right to the left, the order of execution is inverted:

1. The value at index 4 is removed.
2. The value at index 2 is removed.
3. The value which was at index 4 in the unmodified list is inserted at index 1.
4. The value “15” is inserted at index 5.

```
1  epatch ModifyList {
2    resource res0 {
3      left uri 'model-v1.xmi';
4      right uri 'model-v2.xmi';
5    }
6
7    object res0#/ {
8      intlist = [ 5:'15', 1:[4] | 2:'12', 4:[1] ];
9    }
10 }
```

Figure 4.4: An Epatch Modifying a List

4.6.3 Create and Reference Objects

The third example, which is shown in Figure 4.5, creates a new object, adds it to a list, and creates a reference pointing at it. Besides the modified resource this Epatch imports another resource, which is expected to be an EPackage, since it is imported via namespace “http://www.itemis.de/emf/epatch/testmm1”. The imported resource can be accessed from the patch via the name “mm”.

The Epatch modifies two EObjects, one located at “/@tree” and the other one has differing fragmentURIs in the left and the right model, thereby the Epatch stores both: “//@tree/@children.1” and “//@tree/@children.0”. The fragmentURI changes since the EObject changes its position within the containing list.

When the Epatch is applied from left to right, the list contained by EStructuralFeature "children" in EObject "//@tree" in resource "res0" gets a newly created EObject inserted at index 0. The type for the new EObject has the fragmentURI "//CompositeNode" in the imported resource "mm". Furthermore, the newly created EObject can be addressed from within the Epatch using the name "CompNode1" and its EStructuralFeature "name" is initialized with the value "CompositeNode1". The second modified EObject is being located in the left resource via fragmentURI "//@tree/@children.0". Its EStructuralFeature "friend" is expected to have the value `null` which is then changed to the newly created EObject. It is important to note that either "children" or "friend" can be a containment reference. The construction of the new EObject via the "new" keyword is not required to take place at the assignment of a containment reference, but for reasons of readability, it is recommended that way. However, every created EObject must have a container after the patch has been applied completely.

When the patch is applied from right to left, the value at index 0 is removed from the "children"-list in EObject "//@tree" in resource "res0" and "friend" is set to `null` in EObject "//@tree/@children.1" in the same resource.

```
1  epatch AddObject {
2    import mm ns 'http://www.itemis.de/emf/epatch/testmm1'
3
4    resource res0 {
5      left uri 'model-v1.xmi';
6      right uri 'model-v2.xmi';
7    }
8
9    object res0#//@tree {
10     children = [ | 0:new mm#//CompositeNode CompNode1 {
11       name = 'CompositeNode1';
12     } ];
13   }
14
15   object left res0#//@tree/@children.0 right res0#//@tree/@children.1 {
16     friend = null | CompNode1;
17   }
18 }
```

Figure 4.5: An Epatch Creating and Referencing an EObject

4.7 Diff

Besides recording (section 4.8), comparing two existing versions of the model to derive the changes is one way to create an Epatch. As stated in subsection 4.2.1, the main advantage of the diffing approach is that no editor integration is needed. The downside is that matching the model elements can be difficult.

This section describes the implementation of the comparison approach. The actual work of doing the comparison is done by EMF Compare ([Eclh]) which creates a `MatchModel` as well

as a `DiffModel`. In the context of this thesis a `DiffEpatchService` has been implemented, which takes the `MatchModel` and the `DiffModel` as parameters and translates them to the Epatch format. Since the `DiffModel` already describes the differences in a declarative format the translation process is straight forward.

There is one limitation of EMF Compare, though. It does not detect when items are moved within a list (just their index changes, but not their container). This is not a problem for the context of meta model evolution since the position within a list of data structure defining elements in Ecore has no semantical meaning. However, this can lead to incomplete (subsection 4.1.1) Epatches.

4.8 Recorder

To record an Epatch, the implementation has to hook into the editor which the developer uses to modify the model and record all changes. As opposed to the comparison approach (section 4.7), this requires editor integration, which is a disadvantage since there are many editors and not all provide the needed hooks. The main advantage on the other hand is that this approach provides the richest information about the changes since every single element can be traced and every single operation can be recorded. In general, the quality of the recorded data depends on how straight forward the developer modifies the model. For example, when he renames one element twice, the second renaming anuls the first one. When considering A as the source model and B as the target model, it is likely that the recorded change operations do not describe the direct way from A to B. To identify these overlapping operations and to merge or exclude them is the challenge of transforming the list of recorded change operations (which is an imperative format) to an Epatch (which is a declarative format). This leaves two challenges for the implementation:

4.8.1 Editor Integration and Observing Changes

For Eclipse based model editors integration can be done in a generic way by using the user's current selection. Every Eclipse plug-in can access the current selection which is an `EObject` (or a subclass of EObject) for an EMF based model. Having one EObject, all objects in the same containment hierarchy, the `Resource` and the `ResourceSet` become accessible. If there is an `EditingDomain`, it can be obtained via `AdapterFactoryEditingDomain .getEditingDomainFor(EObject)`. For observing the changes, Figure 4.1 shows the options:

Observe Model EMF provides the possibility to hook an `Adapter` into every single `EObject`, `Resource` and `ResourceSet`. The adapter is notified about every change that occurs to the observed object's values. The information about the change is delivered in `Notification` objects, which hold besides the new value the old one as well as the positions in lists. There is no direct support to detect *move* and *copy* operations, but move operations can be reconstructed by matching a *remove* operation with the afterwards occurring *add* operation which re-adds the previously removed object. To detect *copy* operations there can only be guessed whether a newly added object might be the copy of an existing one.

ChangeRecorder The EMF's ChangeRecorder is an implementation which uses the previously described `Adapter` pattern to construct a `ChangeDescription` model ([Ecle]).

There is no direct support to describe *moves* and *copies*, but the `ChangeDescription` model has a declarative format.

Observe CommandStack Models that are modified within an editor typically have their `ResourceSet` assigned to an `EditingDomain`. This `EditingDomain` holds the `CommandStack` which is the basis for undo and redo functionality of the editor. Every change operation which is applied to the model is encapsulated within a `Command` object, that is pushed on top of the `CommandStack` when the operation is executed and removed from the stack when the operation is undone. There are commands for move and copy operations, but the needed information tends to be hidden in private member variables. Furthermore, some editors implement their own commands and the change operation of a command can have an arbitrary complexity. Still, observing the command stack would be the basis of making the recorder aware of user-triggered undo and redo actions.

Observe LTK Operations The Eclipse Language Toolkit (LTK) is the commonly used framework to implement refactorings and other substantial and/or wizard-based code manipulations for Eclipse. It allows refactorings to have participants – a pattern which makes *LTK operations* extensible and allows participants to contribute code manipulations while an LTK operation is executed. A patch recorder would surely not contribute code manipulations, but the information about the executed operation, which is supplied to the participant, is valuable. However, at the time of writing the thesis, there are no LTK based EMF Model manipulations, but a prototype of the author shows the applicability of this approach. The available LTK operations focus on programming languages such as Java, C, C++, etc.

4.8.2 Converting the List of Change Operations to an Epatch

The following things must be considered by the `EpatchRecorder` for the recording process. It is assumed that the left model is the source model and the right model is the target model.

- FragmentURIs change during the editing process. The Epatch format requires that elements in the left model are addressed via fragmentURI and all elements in the right model are addressed via fragmentURI. At the beginning of the recording process, the observed is in the state of the left model. Since it can not be foreseen which elements will be modified, the fragmentURIs of all observed model elements have to be memorized at the beginning of the recording process. That way the source-fragmentURI is still available even after the model has been modified. The fragmentURIs referencing the right model must be calculated at the end of the recording process, since at this point the observed model is in the state of the right model.

- Indexes within lists change while the editing process. As it is true for fragmentURIs, indexes must describe valid positions of values for either the left or the right model. One way to solve this problem is to memorize every values index in the beginning of the recording process and to look up all indexes at the end of the recording process. However, one Java object can appear multiple times within a list and a list can contain multiple equal objects. Therefore, neither object identity nor object equality qualify for identifying the objects – which would be necessary to cache their values. Another

way is to re-calculate the indexes for every change operation concerning the list. Since a `ListAssignment` contains the indexes of all elements that have already be removed and the indexes of all elements which are added, all information needed for this algorithm is provided.

- EObjects, which are removed and re-added, must be described as moved EObjects. In case the containing EReference and the container EObject are the same, the remove and add operation cancel each other.
- Values, which are added and removed later, must be ignored.
- When setting a feature's value multiple times, only the original value and the final value have to be stored in the Epatch. All intermediate values have to be discarded. In case the feature's value is restored to its original value, all change operations for this feature have to be ignored.

4.9 Applier/Patcher

The implementation supplied in the context of this thesis applies the Epatch while creating a copy of the model. This way, the source model is not modified and a mapping between elements in the source model and elements in the target model can be created. This approach has been introduced in subsection 4.2.2. This section first describes the input-parameters, then the results provided by the implementation (output) and finally outlines the implemented algorithm.

4.9.1 Input

To apply an Epatch, the implementation needs to be supplied with the following information.

Apply Strategy The `ApplyStrategy` distinguishes whether the Epatch should be applied from left to right or from right to left.

The Epatch The Epatch which is to be applied.

Import Mapping This mapping has to map every `Import` of the Epatch to an EMF `Resource`.

Resource Mapping This mapping maps every `NamedResource` of the Epatch to an EMF `Resource`. These Resources are be copied while the patching process.

Output ResourceSet This supplied `ResourceSet` is used to construct the needed `Resources` for the patched model(s). The ResourceSet must have the needed meta models available in it's `PackageRegistry` and the needed `ResourceFactory`s available in its `ResourceFactoryRegistry`.

Alternatively the implementation can be supplied with only an `ApplyStrategy`, the `Epatch` and an *Input* `ResourceSet`. Then the implementation tries to find the needed `Resource`s for the `Import`s and `NamedResource`s within this Input ResourceSet. The Output ResourceSet is newly constructed and reuses the Input ResourceSet's `PackageRegistry` and `ResourceFactoryRegistry`.

4.9.2 Output

After applying the Epatch, this information is available:

The Target Model(s) The newly created `Resources` are available in the *Output* `ResourceSet` as well as in an `NamedResource`-to-`Resource` mapping.

A Trace Map This map contains triples consisting of the source EObject, the affecting Epatch `NamedObject` and the target EObject. If the EObject has been newly constructed, the source EObject is null. When the target EObject is an unmodified version of the source EObject (since the Epatch did not contain an `ObjectRef` for this EObject), the `NamedObject` is null. For source EObjects that have no corresponding target EObject (since the Epatch has removed them), the target EObject is null. In the context of meta model evolution this mapping is the basis for the migration algorithm.

4.9.3 Implementation

Applying the Epatch consists of the following steps:

1. All target `Resource`s are constructed using the target `ResourceSet`.

2. The *trace map* is initialized with all `NamedObject`s from the Epatch. For `ObjectRefs`, the source EObject is set based on the resolved source fragmentURI, the target EObject is a newly constructed EObject of the same EClass as the source EObject. For `ObjectNew`s, the target EObject is constructed based on the referenced EClass and for `ObjectCopy`s based on the to-be-copied EObject's EClass. All newly constructed EObejcts are not initialized, which means that no values are assigned to their features.

3. The algorithm iterates over the target resources and their root EObjects. The latter are initialized and all EObjects which are reached via a containment reference are initialized as well. With this concept, the algorithm iterates recursively over the containment hierarchy of the target resources. During this process, the patching information is available from the *trace map*. There are three different scenarios for initializing EObjects:
   - The target EObject has been added and its feature's values are taken from the corresponding `CreateObject`s list of attributes.
   - The target EObject is not modified and therefore all feature's values are copied from the source EObject. For this case, there has no entry been created for the *trace map* in the last phase, so the target EObejct has to be constructed first and added to the trace map.
   - The target EObject is modified. In this case `SingleAttribute`s override the source EObject's feature's value and `ListAttribute`s are applied as described in subsection 4.6.2.

5 Solution: Metapatch

The Metapatch is this thesis's solution to co-evolve models to modified meta models. It implements the *diff-approach*, as introduced in section 3.10. The solution consists of the Metapatch format and the Metapatch-based migrater:

- The *Metapatch format* is based on the Epatch format (chapter 4) with the intend to describe the meta model differences using an Epatch. The Metapatch format extends the Epatch format with the capability to provide code-snippets (subsection 5.2.2) to customize the model migration. Futhermore, while the Epatch format is meta model agnostic (subsection 4.1.4), the Metapatch restricts it to Ecore models.

- The *Metapatch-based migrator* is an interpreter for Metapatches which does the actual migration of models. The migration of models can be considered an exogen model to model transformation. Since the Metapatch is optimized for scenarios in which the changes between the meta models are relatively few, there are differences between the Metapatch and existing model to model transformation languages. section 5.3 lists the differences. section 5.5 explains the migrater's algorithm.

This section starts with listing the requirements for the Metapatch format and the design decisions that were made for the implementation. After comparing the Metapatch with existing model to model transformation languages, this section continues with explaining the Metapatch format's meta model, the Metapatch-based migration algorithm and how to customize this algorithm. Then, the textual representation is explained and based on that examples are provided.

5.1 Requirements

This section starts with listing the requirement for the Metapatch format, its creation and its execution process. For each requirement, the circumstances that it originated from are mentioned.

5.1.1 Capable of migrating EMF Resources and EObject

The Metapatch-based migrater should be able to migrate single EMF `Resource`s as well as `EObject`s. In both cases all contents (As defined by the containment relationship) should be migrated, too. Since in most cases, a model is stored in one EMF `Resource`, migrating the resource means to migrate the complete model. Migrating single `EObject`s allows to migrate portions or to migrate models that are not contained in a resource.

5.1.2 Semi-Automatic Creation Process

As research has shown, it is not possible to create a complete model migration algorithm just by comparing their meta models. For example, when setting `EStructuralFeature.unique` from `false` to `true`, there are multiple strategies that can be used in the migration algorithm: Duplicate items can be deleted or renamed and a renaming strategy can be chosen. This choice has to be made by the developer. Therefore, the process of creating a Metapatch should be automated as much as possible, but must leave the possibility for the developer to intervene and customize.

5.1.3 Complexity Proportional to Changes, but not to Meta Model

In contrast to regular model to model transformations, the complexity of a Metapatches (simplified represented by the amount of model elements it consists of) should be proportional to the changes in the meta model. For example, when having a relatively large meta model with around 1000 concepts and a change modifying one of them, a regular model to model transformation would have to describe the identity transformation for 999 concepts and the migration transformation for one concept. For this example, the Metapatch format should only need to cover the migration of the one format and implicitly assume that an identity transformation is needed for the rest.

5.1.4 Completeness

The Metapatch must be able to define a migration for models for any kind of changes applied to the meta model.

5.1.5 Bidirectional

The Metapatch format should be bidirectional, i.e. one Metapatch should allow to upgrade models as well as to downgrade models. Since the developer's need to implement some parts of the migration manually, these parts may not be automatically invertible (as required for bidirectionality). The developer will have to implement these parts for both directions separately. However, the format should not force the developer to provide these implementations – therefore, it should be the developers choice whether he wants to build an unidirectional or bidirectional Metapatch.

5.1.6 Recreate Meta Model

The standard scenario of meta model evolution is to migrate old models to the current version of the meta model. To load these models, it is required to have the old meta models available. Since the Metapatch also describes the differences between meta models, it should be able to recreate old versions of the meta model by applying theses changes to the current meta model.

5.2 Design Decisions

When trying to develop an implementation which fulfills the listed requirements and when looking at the existing technology, choices emerge and decisions have to be made. This section lists the options an explains the decisions.

5.2.1 Textual Representation for Easy Customization

Since it is a requirement for a Metapatch to be easily customizable, an editor is needed. One choice would be to implement a diagram-based or a tree/table based editor. Both would be much effort and reach its limits when the integration of an expression language is needed, since such a language is typically a textual language and having a graphical editor with text fields for textual expressions does not provide a good editing experience.

Therefore, the choice is to have an Xtext based textual representation of the Metapatch format. The editor generated by Xtext provides a good editing experience and the textual representation allows a seamless integration of an expression language.

5.2.2 Java or Xtend Expression for Customization

It is a fundamental requirement to make the migration algorithm defined by a Metapatch customizable. Since the level of needed customization can reach arbitrary complexity and only depends on the needs of the meta model engineer, it is reasonable to allow a turing-complete programming language for customization. The requirements for such a language are:

- It should be able to operate on dynamic EMF, since in the scenario of meta model evolution there are typically no Java files (static EMF) generated for old meta models.
- It should integrate well with Xtext.

The language of choice is the *Xtend Expression* language, which is part of the Eclipse M2T Xpand project and a successor of the openArchitectureWare Expression language ([oAWb]). Due to its abstraction layer for meta models it is proven to work for Dynamic EMF. Since it has an Antlr-based parser, its antlr-grammar could be translated to Xtext and therefore integrates with the Metapatch format.

The second choice for the language is the Java programming language, to keep the dependency on Xpand optional and since Java's execution performance is superior. However, to work with Dynamic EMF in Java the developer has to program against EMF's reflection layer, which tends to require verbose statements for simple tasks. Furthermore, while Expression statements can be in-lined within the Metapatch format, this is not possible with Java code. Instead, static Java methods have to be implemented separately which the Metapatch is then capable to call.

The Metapatch format supports the Xtend Expression language as well as calling Java methods. It's the meta model engineer's choice which one to use.

5.2.3 Optional dependency on M2T Xpand and TMF Xtext

Since the Metapatch-based migrater will be needed to be integrateable into existing applications, it is valuable to keep it lightweight. Therefore, the dependency on M2T Xpand should be optional and only needed in cases where the Expression language has been used within a Metapatch. Since the Metapatch format is an EMF model, it can be stored as XMI as well as in its Xtext-based textual representation. When supplying Metapatches serialized as XMI, the Metapatch-based migrater does not need to depend on Xtext.

5.2.4 Interprete Metapatches

Since Metapatches are models, there are the options to interpret them or to generate code from them. An interpreter promises to be easier to implement and easier to test via JUnit, have a minor inferior runtime-performance compared to generated code and will prove the feasibility of the proposed concepts as well. Therefore, an interpreter should be sufficient for the context of this thesis.

5.2.5 Semi-Automatical Creation Process

The part that can be created automatically is the description of the differences between the two meta model (versions). Furthermore, the migrater can make assumptions for migration strategies based on theses differences, which is further described in section 5.5. For the Metapatch creation process this means that part that can be automated is already covered by the Epatch's `PatchRecorder` and `DiffEpatchService`.

5.3 Related formats

The Metapatch format is not restricted to migrate models due to changes in the meta model. Instead, it is a specialized model to model (m2m) transformation language. This brings up the question how it differentiates from existing m2m transformation languages such as Xtend, QVT and ATL, as they are introduced in subsection 2.1.2.

- The Metapatch format is optimized for scenarios of exogen model to model transformations. Since in exogen transformation, every model element needs to be copied to ensure the new instance is an instance of the other meta model's concept, common model to model transformation languages need to define an identity transformation for every meta model concept. In contrast, the Metapatch only needs to specify a transformation for the meta model concepts that have actually been modified (subsection 5.1.3).

- Based on the description of differences between two meta models, the Metapatch-based migrater can make assumption about how to correctly migrate the models. Therefore, no explicit transformation needs to be specified for these cases. The list of these cases is further described in section 5.5.

- Since the Metapatch is based on the Epatch, it can recreate one version of the meta model if the other version is available (subsection 5.1.6).

5.4 The Meta Model

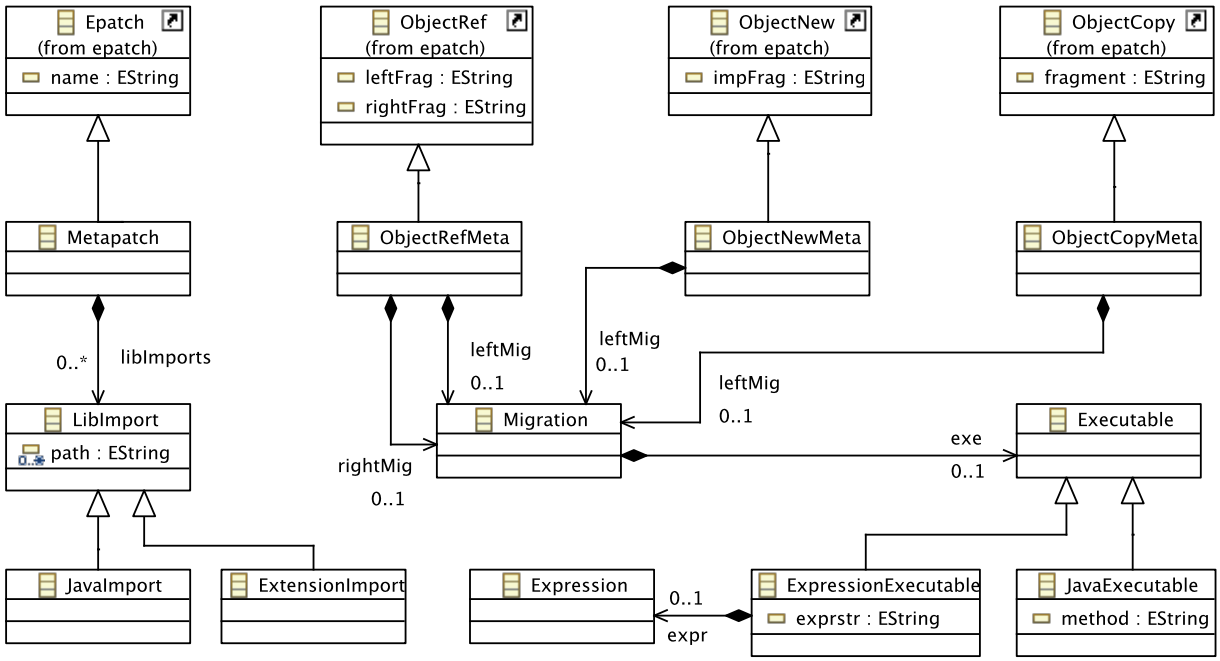

Figure 5.1: The Metapatch Meta Model

A Metapatch is, as the Epatch, a model and has a meta model, which is an instance of Ecore and has been derived from an Xtext grammar (section A.2). It is shown in Figure 5.1 and gives an overview over the Metapatch's structure. For possible instances please refer to section 5.7 and section A.4.

The Metapatch format extends the Epatch format. This is valid for the meta model and can be recognized in Figure 5.1 by EClasses such as `Metapatch` which have a generalization relationship to EClasses from the Epatch EPackage. On the grammar level, this is valid as well and realized using Xtext's support for grammar inheritance.

By inheriting from one grammar, the sub-grammar inherits all the rules of the super grammar. While rules can be overridden in the sub-grammar, the inherited meta model is read-only and can only be extended.

The Metapatch format extends the Epatch format at two positions:

At first, the `Metapatch`-EClass extends the EClass `Epatch` and while the Epatch already allows to import models (Figure 4.2) the Epatch can additionally import "libraries", which can be Java-Classes (`JavaImport`) or Xtend-files (`ExtensionImport`). The functionality of the imported libraries[1] is made available to be used in the `Executable`s, which will be covered later in this section.

At second, the sub-types of `NamedObject` (`ObjectRef`, `ObjectNew` and `ObjectCopy`) are extended to hold `Migration`-information. While each `NamedObject` of the Epatch describes the modification, addition or removal of a model element, this model element must be a meta model element, since the Metapatch restricts an Epatch to instances of Ecore. So while

[1]The Java-Methods and Xtend-Extensions

the NamedObject describes changes to a meta model element, the Migration-information describes how to migrate the instances of this meta model element. The details about this algorithm are explained in section 5.5 and section 5.6. For this section, it is sufficient to know that Migration-information is needed whenever an instance of the related meta model element has to be created or initialized by the migration algorithm.

Therefore, since an ObjectRefMeta (extending ObjectRef) refers to a meta model element which is present on the left and the right side, it must be able to hold distinct Migration-information for migration form left to right and from right to left. The elements described by ObjectNewMeta (extending Objectnew) and ObjectCopyMeta (extending ObjectCopy) are only present on either the left or the right side and therefore only one field for Migration-information is needed.

The Migration-information itself consists of an Executable, which is either a JavaExecutable or a ExpressionExecutable. The JavaExecutable refers to a Java method which is declared in one of the Java classes that have been imported via JavaImport. The ExpressionExecutable holds an Xtend-Expression as a string and as Abstract Syntax Tree[2].

5.5 The Algorithm

After having explained the Metapatch format, this section describes how a Metapatch can be executed. This execution does the actual migration of a model from one meta model to another meta model. The central class implementing this is named MetapatchMigrater. When referring to a Metapatch as a model to model transformation, the MetapatchMigrater represents the runtime environment. When seeing the MetapatchMigrater as an algorithm, a Metapatch is the configuration for this algorithm, since the MetapatchMigrater executes the Metapatch by interpreting it.

To understand how the MetapatchMigrater works, it is a good start to see it as a copying tool which copies models and applies the instructions given by a Metapatch during the copying process. Doing an in-place-migration instead would lead to the scenario that instances of concepts from distinct meta models would be mixed in one model, which would be an inconsistent state for the model.

An inspiration for the MetapatchMigrater has been EcoreUtil.Copier which is part of EMF's build-in set of tools. As the name says, this tool copies models, but it is not capable of modifying them as the MetapatchMigrater does.

During the copying process, one challenge is to copy cross references. A cross reference in EMF is represented by a non-containment EReference. When such a cross reference is

[2]This is a trick to make Xtend-Expression usable within Xtext-based DSLs without having to embed them into strings. At the time of writing this, Xtend has an Antrl-based parser (instead of being implemented via Xtext) and Xtext allows inheritance of single grammars – multiple inheritance is still in the planing phase. Ideally, Xtend would be based on Xtext and the Metapatch grammar would inherit from the Xtend grammar and the Epatch grammar. To implement this feature anyway, the author has translated the Xtend grammar to Xtext and included it in the Metapatch grammar. This works fine for parsing and avoids multiple inheritance. However, for executing the parsed expressions, it would be necessary to port the Xtend runtime to operate on the meta model which has been generated by Xtext (and which is equivalent to the Xtend AST in in this case). Since this task exceeded the available resource, the ExpressionExecutable holds the Expression also as String. This value is being set right after the parsing process and is created by serializing the attached Expression-element. When executing the Metapatch to migrate a model, the Expression string can easily be passed to the Xtend-Runtime, which parses it again using Antlr and executes it.

being initialized in the copying process, it has to point to the copied version of it's original target. The question is when to create the copy of the target object. One option is to create it the first time it is needed and store it in a cache to ensure it will not be created multiple times. This approach works for any graph-like structured models and does require a tree-like structure which could be traversed recursively. Xtend emphasizes this approach by implicitly caching the return values of it's Extensions. However, when applying this to EMF, this means to create `EObject`s which do not reside in a containment-reference, which is an inconsistent state for the model while the copying process. The other option is the approach implemented by `EcoreUtil.Copier`. It traverses the containment-hierarchy of an EMF model, and it does it twice. While the first traversion it copies all containment-references and thereby all `EObject`s and `EAttribute`s. While this procedure it creates a map which associates each source-`EObject` with it's copied version, the target-`EObject`. While the second traversal of the model's containment-hierarchy this mapping is used to let all cross-references point to the corresponding target-`EObjects`. This approach also allows to copy multiple models which reference each other. First, the containment-hierarchy is copied for each model separately. Second, when having the mapping, the cross references of all models can be initialized. For copying multiple models this has the advantage that the models can be process in an arbitrary order – and independently of how they depend on each other. The `MetapatchMigrater` implements this approach and thereby has a similar interface as `EcoreUtil.Copier`.

To explain how the `MetapatchMigrater` works in detail, this section first looks at it as a black box and describes the expected inputs and outputs and then continues with "opening the box" and describing how the implementation works internally.

5.5.1 Input

To start with a look from a high level of abstraction, this section describes the needed inputs for the `MetapatchMigrater` and which methods need to be called to migrate models. The following values are needed to be supplied to instantiate a `MetapatchMigrater`:

ApplyStrategy The `ApplyStrategy` defines whether models should be migrated from the left to the right or from the right to the left. This is equivalent to the definition in subsection 4.4.1. While an Epatch has left and right models, this means for the Metapatch that since these models are meta model, they have instances, which are the models the `MetapatchMigrater` migrates.

Trace Map The *TraceMap* results from applying an Epatch: section 4.9. It holds a mapping between the left model elements and the right models elements. For the Metapatch these models elements are meta models concepts, such as `EClasse`s, `EDataTypes`, and `EAttributes`.

Metapatch The `Metapatch` allows to customize the algorithm, as described in section 5.6. For plain copying, the mapping supplied via the *trace map* is already sufficient.

After the `MetapatchMigrater` has been instantiated, it provides two methods that have to be called to migrate one or multiple models:

- `EObject migrate(EObject model)`: As the name says, this migrates the supplied `model` and returns the migrated model. The supplied model will not be modified. However, this migrates only containment `EReference`s and `EAttribute`s. For migrating the remaining parts of the model, the non-containment `EReference`s, `migrateReferences()` has to be called.

- `void migrateReferences()`: This method has to be called after `migrate()` has been called for every model. Then, it migrates the non-containment `EReference`s. It does not need the model as a parameter since it operates on the `EObject`s which have been stored in the *trace map*, which are all `EObject`s that have been migrated by previous calls of `migrate()`.

5.5.2 Output

After the migration has finished, the following information is available:

Migrated Models The migrated models, which are the target-models, have been returned when calling the method `EObject migrate(EObject model)`.

Migration Mapping The migration mapping maps each source-`EObject` to the corresponding target-`EObjects`, which have been created while the migration process. Usually, there is exactly one target-`EObject` for each source-`EObject`, but the customization of the algorithm (section 5.6) allows to split up information stored in one source-`EObject` to multiple target-`EObjects`. Furthermore, the mapping maps each target-`EObject` to the originating source-`EObject`. The same principle applies here: Whithout customization, there is exactly one source-`EObject` for each target-`EObject`, but with customization, a target-`EObject` can originate from multiple source-`EObjects`.

5.5.3 Implementation

To do the actual migration, the following algorithm is used. This algorithm does not allow any customization and can be considered as a copying algorithm that uses a mapping for types and properties. In section 5.6 this algorithm is extended to allow customization. The description of the algorithm is a simplification of the real source code, written in spoken language and leaves out some capabilities, such as error handling and the processing of lists[3] to keep the complexity to a level that fits in a section of this thesis. The implementation is strongly based on EMF's reflectioning layer.

The method `EObject migrate(EObject model)` (subsection 5.5.1) is implemented as follows. The parameter `model` is considered to be the source-`EObject` and the *trace map* described in subsection 5.5.1 is expected to be available.

1. Look up the target-`EClass` for the `EClass` of the source-`EObject` using the *trace map*.

2. Instantiate an `EObject` of type target-`EClass`. This is the new target-`EObject`.

[3] `EStructuralFeatures` with `isMany()` == `true`.

3. Iterate over all EStructuralFeatures[4] of the target-EClass. Therefore, for each target-EStructuralFeature do:
   a) If the target-EStructuralFeature is transient or a non-containment EReference, ignore it and continue with the next EStructuralFeature.
   b) Look up the source-EStructuralFeature for the target-EStructuralFeature using the *trace map*.
   c) If the target-EStructuralFeature is an EAttribute, expect the source-EStructuralFeatur to be an EAttribute as well. Then, read the value for the source-EAttribute from the source-EObject and assign it to the target-EObject using the target-EAttribute. For the case that source- and target-EAttribute have incompatible EDataTypes, use the EFactory to convert the value to a string and try to convert the string back to an instance of the target-EDataType.
   d) If the target-EStructuralFeature is an EReference, expect the source-EStructuralFeatur to be an EReference, too. Then, read the EObject for the source-EReference from the source-EObject and call migrate(...) for this object. Assign the return value to the target-EObject for the target-EReference. The calling of migrate() lets this method recursively iterate over the model's containment hierarchy.
4. Store the source-EObject and the target-EObject in the *migration map*.

Now that the containment-tree has been migrated, the non-containment EReferences have to be migrated. This is implemented in the method void migrateReferences():

1. For each target-EObject that is stored in the *migration map*, do:
   a) For each EStructuralFeature of the target-EObject, do:
      i. If the target-EStructuralFeature is not a non-containment EReference, ignore it and continue with the next target-EStructuralFeature.
      ii. Look up the source-EReference from the *trace map* for the target-EReference.
      iii. Look up the source-EObject from the *migration map* for the target-EObject.
      iv. Read the EObject for the source-EReference from the source-EObject and call it source-value.
      v. Look up the target-value for the source-value from the *migration map*. This is possible since the source-value is an EObject and all target-EObjects have already been instantiated by migrate().
      vi. Assign the target-value to the target-EObject for the target-EReference.

When looking at this implementation, some first qualities become obvious: For example: The algorithm is agnostic to changes of EClass-names and EStructuralFeature-names, since the mapping is already provided by the *trace map*. Furthermore, it can already cover simple transformations of EDataType-values as long as their string representations are compatible: Converting an EInt to an EString works flawlessly, but converting an EString to an EInt will fail as soon as the EString contains anything else than digits.

[4] In EMF, a EStructuralFeature describes a property of an EObject. EAttribute and EReference are sub-classes of EStructuralFeature, while EAttributes describe properties which hold an EDataType and EReferences describe properties which reference other EObjects.

5.6 Customizing the Algorithm

The algorithm described in section 5.5 is capable of migrating models from one meta model to another while tolerating minimal differences between the meta models. For more sophisticated differences, however, a customization for the algorithm becomes necessary. As a motivation, this section starts with listing meta model differences that have to be covered with this customization. Then, it explains at which points in the algorithm this customization can be applied and how to implement it.

The following scenarios are cases in which customization is needed. The list is intended to serve as motivation and does not intend to be complete.

- *Compose EStructuralFeatures*: Merge the contents of multiple `EStructuralFeature`s into one `EStructuralFeature`.
- *Decompose EStructuralFeatures*: Split up the contents of one `EStructuralFeature`s and store it into multiple `EStructuralFeature`s.
- *"vast" move of EStructuralFeatures*: Moving `EStructuralFeatures` up or down the inheritance hierarchy is covered by the non-customized algorithm[5], since the `EObject` holding the `EStructuralFeature`'s value stays the same in this scenario. However, when a `EStructuralFeature` is moved to a different `EClass`, the relation between the source-`EObject` and the target-`EObject` determines how the `EStructuralFeature`'s value can be migrated. The relation is defined as how the two `EObject`s reference each other, including the cardinalities.
- *Convert Values*: To convert the values between different `EDataTypes` or to manipulate the value while keeping the `EDataType`.
- *Instantiate new Classes*: To instantiate `EClasses` which are present in the target-meta model but have no corresponding `EClass` in the source-meta model.

5.6.1 Integration Points

There are two points at which the `MetapatchMigrater` allows customization. The customization for an integrations point (hook) is implemented using an Xtend Expression or a Java method. In both cases, there is a context available for the implementation, which consists of a set of variables, which are explained in subsection 5.6.2. Furthermore, a return value is expected.

Feature Level This hook allows to assign custom values to `EObject`s for a certain `EStructuralFeature`. If there is a mapping (defined via the *trace map*) for this feature, this mapping can be utilized or the value for the feature can be determined independently of the mapping.

- For `EAttributes`, the hook allows to introduce a custom implementation for determining the value for each `EObject` for this `EAttribute`. The complete source-model is accessible for this calculation.

[5]However, there can be customization required for moving features down the inheritance hierarchy, since instances of the super-class will no longer have the `EStructuralFeature`. This is a case of decreased information capacity.

- For `EReferences`, the hook has to determine the information needed to create the target-`EObject` it will be holding. For this, at first, the source-`EObject`, which should be migrated, needs to be selected. However, selecting the source-`EObject` is not required and there can be multiple source-`EObject`s for one target-`EObject` as well. The source-`EObject` is only needed for an automatic migration of its feature's values. At second, the `EClass` to instantiate the target-`EObject` is needed. In case there is exactly one source-`EObject`, the target-`EClass` can automatically be determined by looking up the corresponding `EClass` of the source-`EObject`'s `EClass` in the *trace map*. However, when instantiating a new `EObject`, it is discouraged to initialized its features in the same hook where the object is instantiated. Doing this in the corresponding feature's hook keeps the implementation of a single hook slim and avoids duplicate implementations.

Class Level This hook allows to implement manipulations to `EObject`s based on their `EClasse`s. Everything that can be archived within a class-level-hook can be archived in a feature-level-hook as well. However, if there are multiple `EReference`s that hold `EObject`s of the same type, implementing their manipulations in a class-level hook once instead of implementing it for each `EReference` separately can reduce redundancy. Furthermore, for `EReference`s that hold lists, the feature-level hook is executed once expecting a list as return value. This allows to extend, reduce, or re-order the list. In contrast, the class level hook is executed once for every `EObject`. Furthermore, the class-level-hook is only executed if the hook for the current `EReference` is not implemented or if the feature-level-hook requests an automatic migration of an source-`EObject` by calling `MigrationHelper.migrate(srcObj)`. Furthermore, the class-level-hook can only be executed if a source-`EObject` is available. Then, within the implementation of the class-level-hook, further source-`EObject`s can be chosen and the target-`EObject` has to be instantiated.

5.6.2 Implementing the Hook

Within the implementation of a hook there is the same environment available for the feature-level-hook and for the class-level-hook. This environment consists of a set of functions (Figure 5.2) and a set of read-only-variables (Figure 5.3).

When using *Java*, the functions and variables are available as methods and getters as shown in Figure 5.2 and Figure 5.3.

When using *Xtend Expressions*, they are available as Extensions and local variables. A variables name can be derived from a getter by removing the prefix "get" and converting the first character to lowercase.

```
1 public interface MigrationHelper {
2         public EObject migrate(EObject src);
3         public EObject migrateFrom(EObject src, EObject dst);
4 }
```

Figure 5.2: The MigrationHelper Java Interface

The methods shown in Figure 5.2 are stateless, i.e. they are independent of the context.

- `EObject migrate(EObject src)` expects a source-`EObject` as parameter, and creates and returns a target-`EObject` for it. The mapping is registered in the *migration map*. The target-`EObject` is instantiated based on the source-`EObject`'s `EClass`. If a class-level-hook exists for it, it is executed to create the target-`EObject`. Otherwise, the *trace map* is consulted for a mapping of the source-`EClass` to a target-`EClass`.

- `EObject migrateFrom(EObject src, EObject dst)` registers a source-`EObject` for an already existing target-`EObject`. The target-`EObject` is returned to allow a fluent API [FB]. The method is used when either a custom `EClass` has been instantiated or when a target-`EObject` should have multiple source-`EObject`s.

```
1 public interface MigrationContext extends MigrationHelper {
2         public EObject getDstEObject();
3         public EStructuralFeature getDstFeature();
4         public EObject getSrcEObject();
5         public EStructuralFeature getSrcFeature();
6         public Object getSrcValue();
7 }
```

Figure 5.3: The MigrationContext Java Interface

The following variables are available within a hook's implementation. They are the context from which the implementation is supposed to derive it's return value.

dstEObject is the containing target-`EObject` for the to-be-returned value.

dstFeature is the containing target-`EStructuralFeature` for the to-be-returned value. After the hook's implementation has been executed, the returned value will be assigned to *dstEObject* for the `EStructuralFeature` *dstFeature*.

srcEObject is the corresponding source-`EObject` for *dstEObject*. The *srcEObject* holds the *srcValue* in the *srcFeature*. If there are multiple source-`EObject`s assigned with the target-`EObject`, *srcEObject* is first `EObject` in the list which holds a value for *srcFeature*. *srcEObject* may be `null` is no source-!EObject!s have been defined for the target-`EObject`.

srcFeature is the source-`EStructuralFeature` which corresponds to *dstFeature* according to the *trace map*. *srcFeature* may be `null`, if the *trace map* contains no mapping for the `EStructuralFeature`.

srcValue is the value contained by *srcEObject* for the `EStructuralFeature` *srcFeature*. *srcValue* is `null` if *srcEObject* or *srcFeature* are `null`.

5.7 The Textual Representation

This section explains the textual representation of the Metapatch format by example. For more examples, see section A.4. The Metapatch's grammar can be found in section A.2.

Since Metapatch's textual representation is based on the Epatch's textual representation, this section is strongly based on the corresponding section for the Epatch (section 4.6). In fact, the grammar defining the Metapatch's concrete syntax extends the Epatch's grammar via grammar inheritance.

This section follows a to-down approach to explain the details of the Metapatch. When looking at the example in Figure 5.5, there are two fundamental differences compared to an Epatch. At first, the code starts with the keyword `metapatch` instead of the keyword `epatch` and at second, the keyword `instance`[6] introduces code snippets which implement the *hooks* explained in section 5.6.

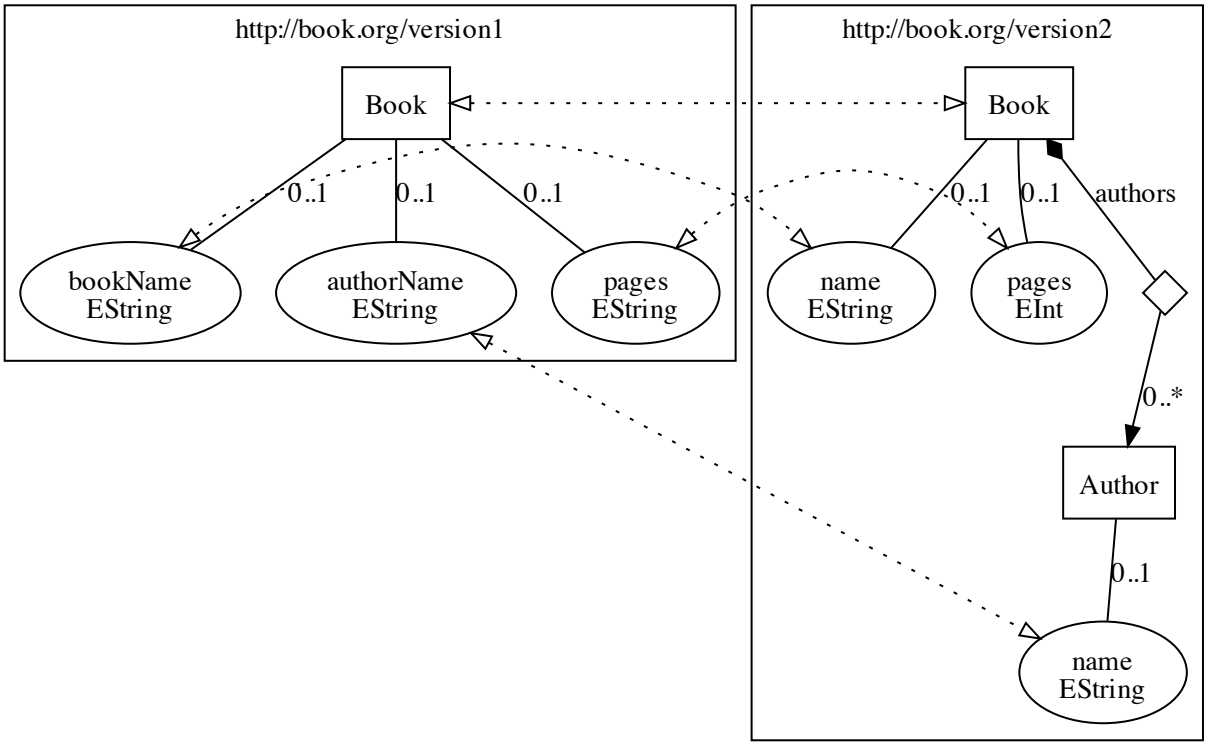

Figure 5.4: Metapatch Example Visualization

[6]The plural (`instances`) is also allowed for features that hold lists.

```
metapatch EXTRACT_AND_REFERENCE_AUTHOR_EXP {
  import ecore ns 'http://www.eclipse.org/emf/2002/Ecore'

  resource res0 {
    left uri 'book-v1.ecore';
    right uri 'book-v2.ecore';
  }

  object res0#/ {
    eClassifiers = [ | 1:new ecore#//EClass Author {
      eStructuralFeatures = [ name ];
      name = 'Author';
    } ];
    nsURI = "http://book.org/version1" | "http://book.org/version2";
  }

  object res0#//Book {
    eStructuralFeatures = [ 1:name | 2:new ecore#//EReference {
      containment = 'true';
      eType = Author;
      name = 'authors';
      upperBound = '-1';
      instances { new rightBook::Author.migrateFrom(srcEObject) };
    } ];
    left instance new leftBook::Book.migrateFrom(srcValue).
                  migrateFrom(srcValue.authors.first());
  }

  object name left res0#//Book/authorName right res0#//Author/name {
    name = 'authorName' | 'name';
  }

  object left res0#//Book/bookName right res0#//Book/name {
    name = 'bookName' | 'name';
  }

  object res0#//Book/pages {
    eType = ecore#//EString | ecore#//EInt;
    left instance srcValue > -1 ? srcValue.toString() : "unknown";
    right instance srcValue != null && srcValue.matches("[0-9]+") ?
       srcValue.asInteger() : -1;
  }
}
```

Figure 5.5: Metapatch Example

Before going into the details, the techniques of code visualization section 2.2 can help to provide an overview over the functionality of the Metapatch: Figure 5.4 visualizes the example shown in Figure 5.5. Rectangles represent `EClasse`s, ellipses represent `EAttribute`s and diamonds represent `EReferences`. Thereby, the diagram has a similar semantic to a UML Class Diagram, but reuses the concrete syntax of an Entity Relationship Diagram[7]. Furthermore, and this is the main purpose of this diagram, the dotted lines represent the mappings from the *trace map* (section 4.9 and subsection 5.5.3).

With the visualization at hand (Figure 5.4), the reader is invited to see what the Epatch-part of the Metapatch does in this example (Figure 5.5): The model on the left side consists of one `EClass` named "book" and having the `EAttribute`s "bookName", "authorName" and "pages". When the patch is applied to the left model, it introduces the following changes:

- It changes the `EPackage`'s namespaceURI from "http://book.org/verion1" to "http://book.org/verion2".
- It creates a new `EClass` named "Author" and adds it to the list of `eClassifiers` of the `EPackage`
- It creates a new containment-`EReference` named "authors" and adds it to the list of `eStructuralFeatures` of the `EClass` "Book".
- It moves the `EAttribute` "authorName" from `EClass` "Book" to `EClass` "Author" and renames it to "name".
- It renames the `EAttribute` "bookName" to "name".
- It changes the `EDataType` of `EAttribute` "pages" from `EString` to `EInt`.

Now that it has become clear how the Metapatch (Figure 5.5) modifies the meta model, this section continues with looking at how the Metpatch is capable of migrating the models. In fact, the actual migration is done by the `MetapatchMigrater` introduced in section 5.5 which executes the Metapatch by interpreting it. The Metapatch in this example is applicable bidirectionally: Its Epatch-part is applicable bidirectionally and it contains all instructions that are necessary to upgrade as well as to downgrade models.

In the scenario where the `MetapatchMigrater` has the job to upgrade a model from meta model "http://book.org/version1" to meta model "http://book.org/version2", it would execute the following steps, as defined generally in subsection 5.5.3. For clarity, elements from meta model "http://book.org/version1" will be referred to with the prefix `v1:` and elements from "http://book.org/version2" with the prefix `v2:` in the following.

[7] This has technical reasons: The main purpose of this diagram is to visualize the *trance map* (section 4.9 and subsection 5.5.3) which requires edges connection `EReference`s, `EAttribute`s and `EClasse`s. In a classical UML Class diagram, properties (as equivalent to `EAttributes`) are nested within the boxes that represent classes and associations (as equivalent to `EReference`s) are edges themselves. This would require to have edges (`EReferences`) as endpoints for other edges (mappings form the `trace map`). On the one hand this exceeds the capabilities `graphviz`, the tool that has been used to automatically render the visualizations in this thesis and on the other hand it is questionable whether it would result in conveniently readable diagrams. By reusing the concrete syntax of an entity relationship diagram, `EAttribute` and `EReferncs` become nodes and can thereby serve as endpoints for further edges. The experience collected while writing this thesis shows that mapping to-be-visualized structures to a plain edge-and-node concept is the most reliable way to take advantage of `graphviz`' auto-layout mechanisms.

1. Being supplied with an instance of `v1:Book`, which is mapped to `v2:Book` by the *trace map*, the `MetapatchMigrater` creates a new instance of `v2:Book`. For the instance of `v1:Book` and `v2:Book` a mapping the *migration map* is created.

2. `v2:Book` has the `EStructuralFeatures` `v2:name`, `v2:pages` and `v2:authors` and the next step is to obtain values for these `EStructuralFeatures`.

3. For the `EAttribute` `v2:name`, the *trace map* holds the mapping to `v1:bookName`. Therefore, the `MetapatchMigrater` can simply migrate the value of `v1:bookName` and no special instructions for the migration are necessary.

4. `EAttribute` `v2:pages` is mapped to `v1:pages` by the *trace map*. Therefore, the `MetapatchMigrater` considers this as the *srcFeature* for the migration. However, the migration of this feature requires some extra care since the `EDataType` of the `EAttribute` has been changed from `EString` to `EInt`. Instructions for handling strings that do not represent valid integer values become necessary. For the instance of `v2:Book`, the *migration map* contains the corresponding instance of `v1:Book`, which is the *srcEObject*. By having a *secEObject* and a *srcFeature*, the *srcValue* can be obtained. These variables are made available within the `MigrationContext` (subsection 5.6.2). The Metapatch contains an implementation for the *feature level hook* for the `EAttribute` `pages` in line 40. The hook is implemented twice: Once for executing the Metapatch from left to right (upgrading models) and once for executing the Metapatch from right to left (downgrading models). The implementation can be identified by the keyword `instance`, while in this case, the implementation `right instance` is executed since the to-be-initialized `EStructuralFeature` is part of the *right* meta model. The language of the implementation is Xtend Expression. This implementation of the hook checks whether *srcValue* is `null` or if it matches the regular expression `[0-9]+`. Therefore, the implementation returns the integer value, if *srcValue* represents a valid positive natural number and `-1` in other cases. The `MetapatchMigrater` assigns this return value to `v2:pages`.

5. For the `EReference` `v2:authors`, there is no mapping in the *trace map*, since it is only present in the meta model on the right side. Therefore, the implementation of the *feature-level-hook* (see line 23 in Figure 5.5) is required to initialize `v2:authors` with one or more values. The hook does not need to be introduced with the keywords `left` or `right`, since it only can be implemented for one side. Furthermore, since `v2:authors` holds a list[8], the hook is required to return a list and uses the keyword `instances` (plural). Since there is no mapping for `v2:authors` in the *trace map*, *srcFeature* and *srcValue* are undefined, but *srcEObject* is the instance of `v1:Book`. The implementation of the hook creates a new instance of `v2:Author` and calls `migrateFrom(srcEObject)`[9] on it, which creates a mapping in the *migration map* for the newly created `v2:Author` and the instance of `v1:Book`. This mapping will be helpful to migrate the `EStructuralFeatures` of `v2:Author`. The surrounding curly brackets of the

[8] Since `upperBound` is set to "-1".

[9] This function is part of the `MigrationHelper` and explained in subsection 5.6.2. The reader may have noticed that this function requires two parameters: It is a feature of Xtend Expression to automatically pass the object the Extension is called on as the first parameter.

hook's implementation create a list filled with one entry: the newly created instance of `v2:Author`.

6. The `v2:Author`'s `EAttribute v2:name` can now be initialized since an instance for `v2:Author` has been created. Furthermore, since the mapping for `v2:name` in the *trace map* points to `v1:authorName` and the mapping for the instance of `v2Author` in the *migration map* points to the instance of `v1:Book`, which holds a value for the `EStructuralFeature authorName`, this values can be migrated automatically.

These are the steps to migrate an instance of `v1:Book` to `v2:Book` including the values of all `EAttribute`s and contained `EObject`s. However, the example Metapatch shown in Figure 5.5 is equipped with instructions to migrate instances of `v2:Book` back to instances of `v1:Book` as well. The following list describes the implementations of the hooks rather than explaining every single step the `MetapatchMigrater` makes.

- Line 25: This is a *class level hook* which is executed when an instance of `v2:Book` need to be migrated. This implementation creates a new instance of `v1:Book` and calls `migrateFrom(...)` twice on it: Once for the instance of `v2:Book`, which is the *srcValue* in this context, and once for the first instance of `v2:Author`. All other authors are lost in the migration process since the information capacity of meta model `v1` is insufficient to hold them all. If this is not acceptable, the meta model engineer can choose to implement a different strategy to handle this situation. The additional mapping of the instance of `v2:Author` as a source-`EObject` allows the `MetapatchMigrater` to automatically migrate the values of `EStructuralFeature`s, which have been moved from `v2:Author` to `v1:Book`, which is `v1:authorName` in this example.

- Line 39: This is a *feature level hook* that converts the values of `v2:pages` which are of type `EInt` to `EString` to assign them to `v1:pages`. For this conversion scenario, the implementation of this hook is not necessary, since the `MetapatchMigrater` can automatically migrate values with compatible string-representations. However, this implementation converts values smaller than zero to the string "unknown" and thereby differs from the default behavior.

For further examples please see section A.4. The following section continues with explaining how a Metapatch can be invoked from within Java code.

5.8 Applied Meta Model Evolution

After having introduced all necessary concepts and tools for meta model evolution, this section applies them in a minimal example. Besides demonstrating that the introduced tools perform as intended, this summarizes chapter 4 and chapter 5 by drawing the big picture which shows how the tools can be invoked.

The example demonstrated in this section is an Eclipse plug-in projects as shown in Figure 5.6. The file `BookDemo.java` contains the source code listed in Figure 5.7. `book-v2.ecore` is the meta model in the "new" version, which represents the current version, which a fictional application would work with. `book.metapatch` is the Metapatch as shown in Figure 5.5. It describes the differences between the "new" version of the meta model and

an "old" version of the meta model. The file `book-v1-i1.ecore` contains a model which is an instance of this "old" meta model. All other files shown in Figure 5.6 are required for every Eclipse plug-in project to run, and are not specific for this example.

When executed, the source listed in Figure 5.5 loads the "new" meta model, recreates the "old" meta model by applying the patch. Then, it loads the model using the "old" meta model and migrates it to be an instance of the "new" meta model by executing the Metapatch. Finally, the code prints the migrated model to the console. Please note that while applying the patch is a downgrade, the migration of the model is an upgrade. Both works well with the same Metapatch.

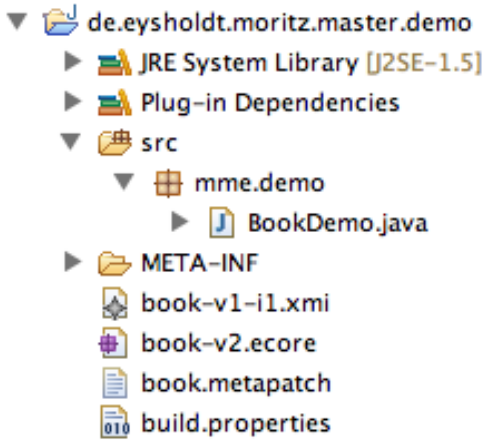

Figure 5.6: Epatch and Metapatch Example Application Files

In detail, the execution consists of the following steps: At first, calling `doSetup()` registers the Metapatch's static `EPackage` in EMF's global package registry and furthermore registers an `XtextResourceFactory` for the file extension "metapatch". The class `MetapatchStandaloneSetup` is a helper call which has been automatically generated by Xtext. Then a `ResourceSet` is created. `ResourceSet`s are a concept of EMF which contain `Resource`s and handle their life-cycles (e.g. construction and destruction). A `Resource` represents a model (or a that part of a model) which is stored in one file. In line 6, `rs.getResource(...)` loads the file `book-v2.ecore` into a `Resource`. This resource represents the meta model in the "new" version. Calling `MetapatchDSLUtil.loadMpatch(...)` loads the Metapatch from its file. This is just a helper function that wraps the EMF API and provides the convenience that it returns an Object of type `Metapatch` (section 5.4) instead of a `Resource`.

After having the initialization done, the code continues in line 10 with preparing the patching. Every Metapatch is an Epatch as well, and every Epatch contains instructions to modify one or more resources. These resources are declared using the `resource` keyword in the textual representation (section 4.6) and are represented by the `EClass NamedResource` (section 4.5). To apply a patch, each of these resource-declarations needs to be mapped to an EMF `Resource`, which is done in line 12. Since there is only one resource-declaration in this example, it is safe access it via `patch.getResources().get(0)` in line 10. Right afterwards the `CopyingEpatchApplier` is instantiated (section 4.9) and calling `apply()` applies the patch to the "new" meta model and thereby recreates the "old" meta model. Since this is a "copying" applier, the "new" meta model is not modified. The patch in this example has been implemented the way that it performs an upgrade when executed from the left to the right, but since recreating the "old" meta model is a downgrade, the `CopyingEpatchApplier` has to be initialized with the parameter `RIGHT_TO_LEFT`.

Now that the "old" meta model is available, it has to be registered in the `ResourceSet`'s package registry (line 17 to 19). This makes the meta model accessible for the `ResourceSet` when loading new resources. The package registry maps namespaceURIs to `EPackage`s and `EPackage`s are usually the root objects within resources.

In line 22 the "old" model is loaded (the model is an instance of the "old" meta model"). This is possible now since the "old" meta model has been registered in the `ResourceSet`s package registry.

To migrate the "old" model to become an instance of the "new" meta model, the `MetapatchMigrater` (section 5.5) is instantiated in line 25. It is instructed to do the migration from `LEFT_TO_RIGHT`, which is an upgrade. Besides the Metapatch, it furthermore gets the *trace map* as a parameter, which is available from the `CopyingMetapatchApplier` via `getMap()` (section 4.9). The *trace map* maps each element from the "old" meta model to the corresponding element in the "new" meta model and vice versa. Then, calling `migrate(...)` triggers the migration for the root-`EObject` of the "old" model. However, this only migrates containment-`EReference`s and `EAttributes`. The cross-`EReference`s are migrated in the next line by calling `migrateReferences()`.

Finally, the "new" model is printed to the console with the help of a utility class named `EmfFormatter`.

```
1  public static void main(String[] args) {
2    MetapatchStandaloneSetup.doSetup();
3    ResourceSet rs = new ResourceSetImpl();
4
5    // load current meta model and the metapatch
6    Resource new_mm =rs.getResource(URI.createURI("book-v2.ecore"),true);
7    Metapatch patch = MetapatchDSLUtil.loadMpatch("book.metapatch");
8
9    // recreate old meta model by patching the current
10   NamedResource patchres = patch.getResources().get(0);
11   Map<NamedResource, Resource> res_mapping = new HashMap<NamedResource,
         Resource>();
12   res_mapping.put(patchres, new_mm);
13   CopyingEpatchApplier app = new CopyingEpatchApplier(RIGHT_TO_LEFT,
        patch, res_mapping, rs);
14   app.apply();
15
16   // register the old meta model
17   Resource old_mm = app.getMap().getDstResources().get(patchres);
18   EPackage old_pkg = (EPackage) old_mm.getContents().get(0);
19   rs.getPackageRegistry().put(old_pkg.getNsURI(), old_pkg);
20
21   // load the old model
22   Resource old_m =rs.getResource(URI.createURI("book-v1-i1.xmi"),true);
23
24   // migrate the model
25   MetapatchMigrater mm = new MetapatchMigrater(LEFT_TO_RIGHT, patch,
        app.getMap());
26   EObject new_m = mm.migrate(old_m.getContents().get(0));
27   mm.migrateReferences();
28
29   // print the new model
30   System.out.println(EmfFormatter.objToStr(new_m));
31 }
```

Figure 5.7: Epatch and Metapatch Example Application

6 Evaluation

This section evaluates the quality of the concepts implemented in the context of this thesis and the quality of the implementations itself. Furthermore, it discusses how the meta model engineer can ensure the quality of the model migrations he implements. Quality is looked at with regards to performance of the migration, complexity of the developer's task, completeness of the migration, automatability of developing the migration algorithm and the correctness of the algorithm.

6.1 Performance

This section evaluates the performance of the migration and its observations are based on the properties of the algorithm (section 5.5), the implementation (the `MetapatchMigrater`, subsection 5.5.3) and experience with the utilized frameworks.

First, the algorithm is a copying algorithm which creates no intermediate states of the model or the meta model. This saves the time for creating the intermediate version as well as it saves the memory that would be needed for it. Second, the copying algorithm operates on *Dynamic EMF*, which is a reflectioning layer for EMF. In contrast to the Java reflectioning layer, Dynamic EMF is encouraged to be used and has been highly optimized with regards to performance. Statistics ([EMF]) show that Dynamic EMF is roughly two to three times slower than Static EMF, while the latter is an automatically generated Java-based implementation of an EMF model, which holds Java-Membervariables for the `EStructuralFeature`s and provides getters and setters to access them.

With regards to runtime complexity it has to be stated that the mapping for the meta model elements (the *trace map*) can be created statically: It only has to be created once per pair of meta models and can be reused afterwards. The `MetapatchMigrater` itself, and therefore the migration of a model, has a linear runtime complexity which is proportional to the amount of elements hold by the migrated model. However, this is only true as long as the implementation of the hooks (section 5.6) dos not introduce more complex calculations.

Furthermore, implementing the hooks in Xtend leads to a slower runtime performance than implementing them in Java. This is due to the fact that Xtend is executed by an interpreter which itself is written in Java and due to advanced concepts of Xtend such as multiple dispatch. This leads to the trade-off that while Java offers the better runtime performance, Xtend offers more convenience for the developer and thereby speeds up the development process.

6.2 Complexity

An essential aspect of meta model evolution is to look at the complexity that the meta model engineer has to handle when crafting the model migration algorithm (for runtime

complexity, see section 6.1). How well he can master the complexity has a strong impact on the development times and the potential number of errors in the resulting migration algorithm.

First, the diff-approach reduces complexity in comparison to the operation-approach by automatically eliminating unnecessary operations. While the operation-approach records all editing steps the meta model engineer does when modifying the meta model, this sequence of operations may contain operations that cancel each other out or that could be merged to one operation. This depends on how straightforward the meta model engineer does the modifications. In the diff-approach, however, the unneeded intermediate states caused by the mentioned operations are automatically ignored since only the differences between the original and the final version of the meta model are taken into account.

Second, the Metapatch format is concise and strives to avoid redundant information or information which could be derived. It combines the differences of the meta models with the instructions how to migrate the models and thereby intends to give all the information to the meta model engineer that he needs to implement the customization hooks.

However, eliminating unneeded information is just half the way to handle complexity. It offers a clear focus to the developer on what is relevant, but for larger scenarios he furthermore needs a way to split up the task into sub-tasks (decomposition of the problem).

For the `Metapatch` it is the following scenario: When implementing a *feature level hook* which determines the value for an `EStructualFeature` in the target meta model, the meta model engineer needs an overview over all relevant meta model elements from the source meta model. Relevant are all elements that can have an impact on the to-be-determined value. Currently, the engineer has limited assistance in finding the relevant elements. He can use the Metapatch visualization introduced in subsection 5.2.1 or he can use 3rd party tools for visualizing the meta model. However, the amount of relevant source-elements stays relatively small for each target-`EStructuralFeature` while there are no limits to the size of the meta model. Experience from this thesis shows that visualizing large meta models using the format introduced in subsection 5.2.1 can quickly exceed screen and printer capacity. Therefore, it would be interesting to develop an approach to provide a view on the source meta model which only shows the elements that are relevant to a certain target-`EStructuralFeature`. This view would have to include the source-`EClass`, its `EStructualFeatures` and the referenced `EClasses`. Furthermore, the inheritance hierarchies of the source-`EClass` and source-`EReference`s type (if the source-`EStructuralFeature` is a `EReference`) would be interesting.

For the scenario of splitting up the task of model-migration into several sub-tasks, the operation-approach has an advantage: Since every operation modifies the meta model and migrates the model, every operation can be seen as a sub-task. Since every operation's execution is required to result in a valid state of the model, the meta model engineer can look at each sub-task independently.

6.3 Completeness

This section answers the question whether the `MetapatchMigrater` and the Metapatch format are complete in the sense that for every two possible meta models for which a migration strategy exists, a Metapatch could be implemented and the `MetapatchMigrater` could migrate the models.

First, an inspection of the algorithm (section 5.5) reveals its completeness: The algorithm iterates over all *EStructuralFeature*s in the target model and the Metapatch format allows to implement a custom hook for each feature in a turing-complete language (Java or Xtend Expressions). From within the hook's implementation, the complete source model is accessible, since EMF offers the possibility to navigate to from each `EObject` to all other `EObject`s[1]. As a consequence, an arbitrary value can be determined for each `EStructualFeature` while taking the complete source meta model into account. This makes the algorithm and implementation complete in the defined sense.

However, EMF offers concepts that go beyond simple `EClass`es, `EDataType`s, `EAttribute`s and `EReference`s. Support for them has not been tested in the context of this thesis:

Generics As of EMF 2.3, EMF supports generics as they are known from Java 5.0 and later: Generic implementations (Example: lists, maps) can now be parameterized with types for the sake of static evaluateability of type safety.

FeatureMaps A FeatureMap contains tuples of `EStructualFeature`s and their values. The implementation inherits from `EList` and thereby the map is ordered and may contain duplicates. For example, EMF uses FeatureMaps to handle mixed content in XML: When text is allowed to contain XML-elements (which are mapped to `EStructuralFeature`s), it is important that the text as well as the elements are stored in one sequence.

EAnnotations EAnnotations allow EMF-based meta models to be extended with additional information (Example: documentation), which may also have a semantic meaning (Example: restrictions). For example, this is used, when transforming XML Schema to Ecore, to cover the XSD's semantics which is not supported by Ecore natively. Since the semantics of EAnnotation is therefore not specific to Ecore but varies for different use cases, it has to be evaluated for each case whether the semantics has an impact on meta model evolution. If it does, a specific implementation is necessary.

6.4 Automatability

The `MetapatchMigrater` can migrate the models automatically for certain meta model changes. This section tries to identify these changes. Changes, for which the models can not be automatically migrated, require implementing the *class level hook* or the *feature level hook*.

Thereby, this section connects to section 3.9, where changes have been classified into *non-breaking*, *breaking but resolvable*, and *breaking* changes. For *non-breaking* and *breaking but resolvable* changes the models can be migrated automatically. *Breaking* changes require the implementation of the *class level hook* or the *feature level hook*.

Before an answer can be found, the question has to be rendered more precisely: First, it has to be defined what "a change" is. There are two kind of changes:

Simple Change A simple change is the modification of a single `EStructuralFeature`'s value. This can be a *set*, *unset*, *add* or *remove*. Furthermore it is necessary to distinguish between different values: For example `ETypedElement.upperBound`, which,

[1] `eContainer` reveals an `EObject`'s container (parent) and `eContents` reveals all children.

among others[2], defines the cardinality of `EStructuralFeature`. For the following list, `s` is the original (source) value and `t` the new (target) value:

- `s == 1 && t == -1`: Changes a single-value `EStructuralFeature` to a list - `EStructuralFeature`. This increases the information capacity and can be handled automatically.
- `s > 1 && t == -1`: Changes the upper bound of a list - `EStructuralFeature` to *unlimited*. This increases the information capacity and can be handled automatically.
- `s == -1 && t == 1`: Changes a list - `EStructuralFeature` to a single-value `EStructuralFeature`. This decreases information capacity and can only by handled automatically for source meta models that contain only one value for this `EStructuralFeature`.
- `s == -1 && t > 1`: Changes the upper bound of a list - `EStructuralFeature` from *unlimited* to some defined value. This decreases information capacity and can only by handled automatically for source meta models that contain `t` or less values for this `EStructuralFeature`.
- `s > 1 && t == 1`: Changes a list - `EStructuralFeature` to a single-value `EStructuralFeature`. This decreases information capacity and can only by handled automatically for source meta models that contain only one value for this `EStructuralFeature`.
- `s > 1 && s > t && t > 1`: Decreases the upper bound of a list - `EStructuralFeature`. This also decreases information capacity and can only by handled automatically for source meta models that contain `t` or less values for this `EStructuralFeature`.
- `s >= 1 && s < t`: Increases the upper bound of a list - `EStructuralFeature`. This also increases information capacity and can by handled automatically.
- `s == 0 || t == 0`: Undefined state.
- `s == t`: No change.

Complex Change Complex Changes compose multiple *Simple Changes*. These groupings are not explicitly represented by the Metapatch. Instead, they can be seen as patterns that appear when looking at multiple *Simple Changes* in combination. *Complex Changes* change the semantic of the contained *Simple Changes* and may help the meta model engineer to reduce complexity (section 6.2). Examples are:

Move EStructuralFeature This combines the removal of an `EStructuralFeature` from `EClass.eStructuralFeatures` with the adding of the `EStructuralFeature` to a different `EClass`. To understand the consequences, the two `EClass`'s instances (the source-`EObject` and the target-`EObject`) have to be looked upon. Their relation to each other determines how the `EStructualFeature`s values can be migrated. If they are in the same inheritance hierarchy, the super class is not abstract, and the `EStructuralFeature` is being moved to the super class, information capacity increases and the migration can be handled automatically.

[2]The other two sub classes of `ETypedElement` are `EOperation` and `EParameter`. Since both are meta model concepts that can not be instantiated, changes to them are *non-breaking* and therefore they do not have to be taken into account any further.

If the `EStructuralFeature` is being moved to the sub-class, information capacity decreases and only models that do not contain instances of the super class but only of the sub-class can be migrated automatically. If the source-`EObject` and the target-`EObject` do not belong to the same inheritance hierarchy, the source-`EObject` can not be identified automatically for a target-`EObject`. For this, a *class level hook* or a *feature level hook* has to be implemented. After the source-`EObject` has been idetified, the `EStructuralFeature`'s value can be migrated automatically.

Compose EStructuralFeatures Composing multiple `EStructuralFeature`s combines the values of multiple `EStructuralFeature`s and stores them to a single `EStructuralFeature`. Accordingly, the removals and the addition of `EStructuralFeature`s to `EClass.eStructuralFeature`s can be involved. The Metapatch has no explicit semantic for this scenario and a *feature levl hook* has to be implemented to handle it appropriately.

Instantiation and destruction of `EClass`es does not need to be taken into account separately, since there is always an `EStructuralFeature` involved, that the new instance is assigned to, and which can be described by a *Simple Change*.

After having explored the possible kinds of changes, the question emerges how to obtain a list of all possible changes. There are two options:

Ecore Listing all `EStructuralFeature`s from EMF Ecore and examining the semantics that are cause by changing their values leads to a cumulative list of simple changes. Such a list has been composed in [BGGK07]. However, it must be noted that Ecore also contains meta model concepts which can not be instantiated (`EOperation`, `EParameter`, etc.) and concepts which have an explicitly undefined semantic with regards to their instances (*EAnnotation*).

Empiric Data Real-world's meta models can be analyzed for changes. Since they are usually stored in SCMs (such as CVS, SVN, or Git), their old versions are still available. In contrast to analyzing Ecore, this would allow to detect complex changes. Furthermore, since such an evaluation would provide quantities for each kind of change, the importance of changes could judged, which could be used to optimize the tooling. On the other hand, empirical studies will not be able to lead to a complete list of possible changes. Another danger is that existing meta models have been evolved very conservatively due to the lack of good tooling. However, collecting exhaustive empirical data exceeded the timeframe of this thesis.

At the current state of the `MetapatchMigrater` and the Metapatch format, there is explicit support for:

Move EStructuralFeatures As described in the example for *Complex Changes* in this section.

Renaming ENamedElements Since `EClass`, `EDataType`, `EAttribute`, `EReference` etc. extend `ENamedElement`, their renaming is supported as well. Instances of renamed meta model elements can be migrated completely automatically since the *trace map* provides a mapping of the meta model elements (section 5.5). Thereby, the meta model element's names become irrelevant for the `MetapatchMigrater`.

Changing upperBound/lowerBound As described in the example for *Simple Changes* in this section, values of `EStructuralFeature`s can be automatically migrated when `ETypedElement.upperBound` and `ETypedElement.lowerBound` change, as long as meta model's information capacity is preserved or increases. If the information capacity decreases, only models can be migrated that do not exceed the information capacity of the target meta model.

Changing DataTypes Values of `EAttribute`s can be automatically migrated even though the `EAttribute`'s `DataType` changes, if the string representations of the values are compatible.

6.5 Correctness

It is the meta model engineer's responsibility to develop a correct model migration algorithm. A correct model migration algorithm is defined as a transformation that transforms a model, which is an instance of a first meta model, to a semantically equivalent model which is an instance of a second meta model. Good tooling may assist the meta model developer in ensuring the correctness of the model migration algorithm.

Correctness is related to complexity (section 6.2) in the sense that an increased complexity increases the difficulty for the meta model engineer to grasp the model migration algorithm. However, a good understanding of the model migration algorithm is essential to develop a correct implementation.

First, the meta model developer has to decide which models are targeted by the model migration algorithm:

All Possible Models All possible instances of the source meta model must be migrated correctly by the model migration algorithm.

Some Known Models There is a set of known models which have to be migrated. This has the advantage that the migration algorithm does not need to cover all possible instances – but a smaller and defined subset. Thereby, the complexity of the migration algorithm can be kept at a lower level.

The set of possible models is reduced further if constraints for the models are defined. This can be constraints as used for model validation (subsection 2.1.2) or a grammar[3].

Second, there are two approaches available for the meta model engineer to ensure correctness of the model migration algorithm.

Formal A formal approach would be to decompose the model migration algorithm into subtasks until the correctness for each task's implementation becomes provable. Then, it has to be ensured that the task's implementations can be combined without causing side-effects. However, decomposing a migration algorithm that has been developed using the diff-approach is not a trivial task as discussed in section 6.2.

Testing The second approach is to run automated tests, which can be easily implemented using JUnit ([GB99]) and which can be extended to Test Driven Development (TDD,

[3]Example: For an Xtext grammar, the rule *MyRule: (ele1+=STRING ele2+=INT)*;* implies, that there have to be as many values for *ele1* as there are for *ele2*.

[Bec03]). All implementations introduced in this thesis have been tested using JUnit tests and the examples listed in section A.3 and section A.4 originate from JUnit tests. Therefore, it can be said that model migration algorithms that have been implemented using the MetapatchFormat and the `MetapatchMigrater` can be easily tested using JUnit tests. The challenges of unit testing of a model migration algorithm are the same as for unit testing in general: The developer (the meta model engineer) has to identify the right cases that have to be tested and he has to implement the tests. For meta model evolution, there are two approaches to test the migration algorithm.

- Verify if a source model is migrated to the expected target model. The meta model engineer manually creates a source model and a corresponding target model. Then, he applies the migration algorithm to the source model and verifies that the result equals the manually modeled target model. This approach is proven to work since the `MetapatchMigrater` and model-comparison with EMF Compare ([Eclh]) work well from within JUnit tests.
- Verify if all model elements from the source model are migrated. Instead of focusing on the correctness of the target mode, this approach ensures that all model elements and values from the source model are included in the migration. This approach becomes useful when there are many existing source models which are to be involved in the testing process, but there is not the time to manually create a target model for each of them. The `MetapatchMigrater` supports this scenario by creating the *migration map* (subsection 5.5.2), which maps each element from the source model to the corresponding model element in the target model. However, the *migration map* only maps `EObject`s, but not the values of `EAttribute`s.

7 Conclusion

It can be said that the suggested approach, to use Metapatches to migrate models when the meta model changes, works as expected and is capable of covering a large amount of meta model evolution scenarios. This section summarises the achievements of this work and points out areas for future research.

Treating structured data as models with the structure definition being the meta model has helped to unify various perceptions of models. The observation that there is a common subset of operations that can be applied on structured data, such as (de)serialization, transformation, validation, comparison, etc., allows to furthermore sharpen the definition of models. The concept of meta models allows generic implementations for these functionalities, which can be reused in the form of frameworks and libraries. The result of this thesis extends this family of frameworks and libraries with one member: Support for co-evolving models to evolved meta models.

One field of application for meta model evolution is the field of MDSD, where models are used to describe structural and behavioral properties of software. Furthermore, the techniques of MDSD have been applied in this thesis and parts of the developed implementation have automatically been generated from models.

On the way to the concepts of meta model evolution, evolution in biology and computer science has been compared. It turns out that there are two meanings of evolution: While the original definition by Charles Darwin describes the repeated process of mutation and natural selection, there are other fields which use the word evolution as equivalent to “change”. This is the case for the field of Software Evolution and its sub-field Meta Model Evolution. The latter becomes a sub-filed of the former due to the fact that meta models are an artifact of the software which is processing the models. Meta Model Evolution is not to be confused with Evolutionary Algorithms, which simulate the process of mutation and selection.

Furthermore, it has been noticed that there are also cases of co-evolution in biology: The predator needs to adapt its hunting skills with regards to the prey and the prey is interested in optimizing its capabilities for hiding and defending with regards to the predator. In Meta Model Evolution, however, the relationship is unidirectional since the model needs to be migrated with regards to the changes of the meta model.

For the implementation suggested by this thesis, some inspiration has been taken from the related fields of research, such as Database Schema Evolution and XML Schema Evolution. However, the suggestions could only be transferred partially due to the difference in the power of expression of the languages which are used to define the data structures.

The requirement to co-evolve data to the changes of its structure definition, which can be implemented using the solution provided by this thesis, can be seen in two major fields: First, when data is persisted and the application, which is processing the data, is updated. Second, it can be seen when two applications communicate with each other by sending data over a network, for example in the form of Remote Procedure Calls (RPC). In case these applications require different versions of the protocol/interface (which defines the structure of the exchanged data), a migration of the data becomes necessary. These use cases demon-

strate that at the time when models are migrated, the developer who has implemented the application is likely not to be present. This creates the need for a migration algorithm that can run autonomously and it allows to distinguish between a model engineer and a meta model engineer.

The classification for meta model changes introduced by this thesis helps the meta model engineer to fully understand their impact on the model, which is crucial to implement a model migration algorithm: (a) For the dimension of the meta model, changes can construct, refactor or destruct meta model elements; (b) For the dimension of the model, this leads to an expansion, a preservation or a reduction of the information capacity. It is shown that a classification of a change according to (a) does not imply the classification according to (b). For example, constructing a meta model element does not imply that the information capacity is increased. (c) For the dimension of the algorithm, it can be distinguished whether a change is breaking, breaking but resolvable or non-breaking. This relates to whether a change breaks the compliance of a model to a meta model and whether an automatically derived model migration algorithm could re-establish this compliance.

Experiments with early EMF-based prototypes revealed that the meta model can not be modified as long as it has in-memory models. Therefore, EMF models would have to be converted into a more flexible data structure to be co-evolved with their meta models using a sequence of operations (operation-approach). Based on this, the decision has been made to migrate models in one step (without creating intermediate versions), which led to the diff-approach. Instead of recording all operations, which the meta model engineer uses to edit the model, the source meta model and the target meta model are compared and based on the obtained differences a model migration algorithm is created.

Within the context of this thesis, an implementation of the diff-approach has been developed. The implementation consists of the Epatch, which describes differences between two (meta) models and the Metapatch, which specializes and extends the Epatch to describe the model migration. The Epatch format and the Metapatch format are textual DSLs that have been implemented using Xtext ([Eclf]).

The Epatch format declaratively describes the differences between two models. An Epatch can be recorded while a model is edited or result from the comparison of two models. The latter has been implemented using EMF Compare ([Eclh]). Furthermore, an Epatch can be applied to a model. Since an Epatch is bidirectional, it can create model A from model B as well as it can recreate model B from model A. The Epatch format and implementation is meta model agnostic and not tied to scenarios of meta model evolution. It has been contributed to EMF Compare ([Eclh]) and will be part of the Eclipse Galileo release in June 2009.

The Metapatch uses the Epatch in two ways: First, the Metapatch format extends the Epatch format via grammar inheritance: It additionally allows to include instructions in Java or Xtend ([oAWd]) to customize the model migration algorithm and it restricts the Epatch to meta models (Ecore models). Second, the mapping of (meta) models elements, that is created when an Epatch is applied, is an essential input for the migration algorithm.

The implementation of the migration algorithm, the `MetapatchMigrater`, has been implemented as interpreter for Metapatches. For future research it might be interesting to evaluate the advantages a compiler for Metapatches could offer. With the mapping of meta model elements as input, the `MetapatchMigrater` is capable of migrating a model from one meta model to another. In cases where the mapping is not present for certain types, or does not lead to the expected results, the instructions stored in the Metapatch specify the mi-

gration of the corresponding model elements. This concept allows the `MetapatchMigrater` to automatically migrate the parts of a model that conform to meta models elements which have been modified by non-breaking or breaking but resolvable changes or which have not been modified at all. Breaking changes have to be covered with instructions stored in the Metapatch. This behavior categorizes the Metapatch format as a model to model transformation language, which is optimized for transformations between similar meta models: Since the Metapatch implicitly assumes the identity transformation for models elements, which are related to unmodified meta model elements, the size of a Metapatch is proportional to the amount of changes between source and the target meta model. This is an advantage especially for large meta models with relatively few changes. The Metapatch format will be part of the Eclipse Edapt project ([EH]), which is in the proposal phase at the time of writing this thesis.

While evaluating the quality of the approach and the implementation it has been observed that the `MetapatchMigrater` has a linear runtime complexity. The needed time is proportional to the number of model elements. Furthermore, it does not create intermediate versions of the model or the meta model. The Metapatch format is complete in the sense that migration algorithms for all possible EMF based model migration scenarios can be implemented. However, support for generics and feature maps has not been tested. Furthermore, there is no explicit support for `EAnnotations` that extend Ecore's semantics. Support for generics, feature maps and `EAnnotations` (for example to fully support EMF's XSD implementation) can be considered as fields for future research. With regards to automatability, it is interesting to identify the breaking, but resolvable meta model changes. For changes which solely affect a single Ecore `EStructuralFeature` (simple changes), a list has been composed in [BGGK07]. To identify changes which compose multiple simple changes and thereby allow improved migration strategies, empirical observations would be needed to reduce the amount of possible combinations to the reasonable ones, which can as well be seen as a filed for future research. The `MetapatchMigrater` provides explicit support for automatically resolving the renaming of `ENamedElements` (and thereby its subclasses: `EClass`es, `EDataType`s, `EAttribute`s, `EReference`s, etc.) and type changes of `EAttribute` for types which have compatible string representations for their values. Furthermore, the migration of values from `EAttribute`s and `EReference`s which have been moved to a different `EClass` can be done automatically if the `MetapatchMigrater` can identify the source-`EObject`. To reduce the complexity, the meta model engineer has to handle when implementing a migration algorithm, the Metapatch format has two merits: Compared to the operation-approach, the complexity is reduced by automatically excluding operations that would effectively cancel each other out. Furthermore, the Metapatch format is a concise DSL which purely describes the meta model differences and the model migration. Thereby, the Metapatch aims at describing the model migration algorithm at the highest possible level of abstraction. However, for migration scenarios of large meta models with many changes, a concept to reduce complexity by decomposing the task into sub-tasks is desirable. This could be an interesting field for future research. A first start would be to develop tooling which assists the meta model engineer in understanding the impact of the meta model changes by providing a specialized view on the Metapatch: It might resemble the graphical visualization format for Metapatches that has been introduced in this thesis, but since the visualization tends to lead to oversized diagrams, it could dynamically filter out all elements that have no impact on a selected target meta model element. With regards to correctness it can be said that all examples and implementations shown in this thesis are covered with JUnit tests and that

it is recommended for the meta model engineer to implement JUnit tests for his own model migration algorithms.

The implementation of the Metapatch and the Epatch can easily be integrated into existing applications. However, there are points for future improvements: First, when there are multiple versions of a meta model and multiple Metapatches, the version of the model could be identified automatically based on the namespace URI and the appropriate Metapatches could be executed sequentially to migrate the model to the most recent meta model version. Second, old versions of meta models could be recreated automatically and on-demand by applying Metapatches sequentially to the current meta model.

A Appendix

A.1 Epatch Grammar

This is the Xtext ([Eclf]) grammar, which defines the textual representation of the Epatch format (section 4.6) and from which the Epatch's meta model (section 4.5) is derived.

```
grammar org.eclipse.emf.compare.epatch.dsl.Epatch with org.eclipse.xtext.common.Terminals

generate epatch "http://www.eclipse.org/emf/compare/epatch/0.1"


Epatch:
        "epatch" name=ID "{" imports+=Import* resources+=NamedResource* objects+=ObjectRef* "}";

Import:
        "import" name=ID ("uri" uri=STRING | "ns" nsURI=STRING);

NamedResource:
        "resource" name=ID "{"
                "left" ("uri" leftUri=STRING | leftRoot=CreatedObject) ";"
                "right" ("uri" rightUri=STRING | rightRoot=CreatedObject) ";"
        "}";

NamedObject: ObjectRef | CreatedObject;

ObjectRef:
        "object" name=ID? (
                (leftRes=[NamedResource] leftFrag=FRAGMENT) |
                ("left" leftRes=[NamedResource] leftFrag=FRAGMENT "right" rightRes=[NamedResource] rightFrag=FRAGMENT)
        )
        ( "{"
                (assignments+=BiSingleAssignment|assignments+=BiListAssignment)*
        "}" )?;

CreatedObject: ObjectNew | ObjectCopy;

Assignment returns Assignment:
        BiSingleAssignment | BiListAssignment | MonoSingleAssignment | MonoListAssignment;

BiSingleAssignment returns SingleAssignment:
        feature=ID "=" leftValue=SingleAssignmentValue "|" rightValue=SingleAssignmentValue ";";

BiListAssignment returns ListAssignment:
        feature=ID "=" "[" (leftValues+=ListAssignmentValue ("," leftValues+=ListAssignmentValue)*)? "|" (rightValues+=
             ListAssignmentValue ("," rightValues+=ListAssignmentValue)*)? "]" ";";

MonoSingleAssignment returns SingleAssignment:
        feature=ID "=" leftValue=SingleAssignmentValue ";";

MonoListAssignment returns ListAssignment:
        feature=ID "=" "[" (leftValues+=AssignmentValue ("," leftValues+=AssignmentValue)*)? "]" ";";

AssignmentValue returns AssignmentValue:
        value=STRING | (refObject=[NamedObject] ("." refFeature=ID ("[" refIndex=INT "]")?)?) | newObject=CreatedObject |
             (^import=[Import] impFrag=FRAGMENT);

ListAssignmentValue returns AssignmentValue:
         index=INT ":" (("[" refIndex=INT "]") | value=STRING | (refObject=[NamedObject] ("." refFeature=ID ("[" refIndex=
              INT "]")?)?) | newObject=CreatedObject | (^import=[Import] impFrag=FRAGMENT));

SingleAssignmentValue returns AssignmentValue:
          keyword="null" | value=STRING | (refObject=[NamedObject] ("." refFeature=ID ("[" refIndex=INT "]")?)?) |
               newObject=CreatedObject | (^import=[Import] impFrag=FRAGMENT);

ObjectNew:
        "new" ^import=[Import] impFrag=FRAGMENT  name=ID? ("{"
                (assignments+=MonoSingleAssignment|assignments+=MonoListAssignment)*
        "}" )?;

ObjectCopy:
        "copy" resource=[NamedResource] ^fragment=FRAGMENT name=ID? ("{"
                (assignments+=MonoSingleAssignment|assignments+=MonoListAssignment)*
        "}" )?;

terminal FRAGMENT:
        '#' ('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'/'|'['|']'|'{'|'}'|'.'|'@'|'%'|':')+;
```

A.2 Metapatch Grammar

This is the Xtext ([Eclf]) grammar, which defines the textual representation of the Metapatch format (section 5.7) and from which the Metapatch's meta model (section 5.4) is derived. It uses grammar inheritance to extend the Epatch grammar (section A.1) and overrides several grammar rules of the Epatch grammar.

```
1  grammar org.eclipse.emf.edapt.metapatch.dsl.Metapatch with org.eclipse.emf.compare.epatch.dsl.Epatch
2
3  generate metapatch "http://www.eclipse.org/emf/edapt/metapatch/0.1"
4
5  import "platform:/resource/org.eclipse.emf.compare.epatch/model-gen/Epatch.ecore"
6  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
7
8
9  Epatch returns Metapatch:
10         "metapatch" name=ID "{"
11                 (imports+=ModelImport | libImports+=LibImport)*
12                 resources+=NamedResource*
13                 objects+=ObjectRef*
14         "}";
15
16 LibImport: JavaImport | ExtensionImport;
17
18 JavaImport:
19         "import" "java" path+=ID ("." path+=ID)*;
20
21 ExtensionImport:
22         "import" "extension" path+=ID ("::" path+=ID)*;
23
24 ObjectRef returns ObjectRefMeta:
25         "object" name=ID? (
26                 (leftRes=[NamedResource] leftFrag=FRAGMENT) |
27                 ("left" leftRes=[NamedResource] leftFrag=FRAGMENT "right" rightRes=[NamedResource] rightFrag=FRAGMENT)
28         )
29         ( "{"
30                 (assignments+=BiSingleAssignment|assignments+=BiListAssignment)*
31                 ("left" leftMig=Migration)?
32                 ("right" rightMig=Migration)?
33         "}" )?;
34
35 ObjectNew returns ObjectNewMeta:
36         "new" ^import=[ModelImport] impFrag=FRAGMENT  name=ID? ("{"
37                 (assignments+=MonoSingleAssignment|assignments+=MonoListAssignment)*
38                 leftMig=Migration?
39         "}" )?;
40
41 ObjectCopy returns ObjectCopyMeta:
42         "copy" resource=[NamedResource] ^fragment=FRAGMENT name=ID? ("{"
43                 (assignments+=MonoSingleAssignment|assignments+=MonoListAssignment)*
44                 leftMig=Migration?
45         "}" )?;
46
47 Migration:
48         ("instance"|"instances") exe=Executable ";";
49
50 Executable:
51         JavaExecutable | ExpressionExecutable;
52
53 JavaExecutable:
54         "java" method=ID "(" ")";
55
56 ExpressionExecutable:
57         expr=Expression | "exp" exprstr=STRING;
58
59 // Expression references the grammar for Xtend Expressions, which is avaialable at:
60 // https://bugs.eclipse.org/bugs/show_bug.cgi?id=264188
```

A.3 Epatch Examples/Tests

A.3.1 SingleChanges

Change Int

See Figure A.1 and Figure A.2
From left to right: The value of attribute `intval` is set to `1`
From right to left: The value of attribute `intval` is set to `10`

```
epatch CHANGE_INT {
  resource res0 {
    left uri "SimpleMM1Instance1.xmi";
    right uri "SimpleMM1Instance11.xmi";
  }

  object res0#/ {
    intval = "10" | "1";
  }

}
```

Figure A.1: Change Int Textual Epatch

```
Epatch {
    attr EString name 'CHANGE_INT'
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: SingleAssignment {
                    attr EString feature 'intval'
                    cref AssignmentValue leftValue AssignmentValue {
                        attr EString value '10'
                    }
                    cref AssignmentValue rightValue AssignmentValue {
                        attr EString value '1'
                    }
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '/'
        }
    ]
}
```

Figure A.2: Change Int Epatch Model

Change String

See Figure A.3 and Figure A.4
From left to right: The value of attribute `strval` is set to `MyNewStringValue`
From right to left: The value of attribute `strval` is set to `MyStringValue`

```
epatch CHANGE_STRING {
  resource res0 {
    left uri "SimpleMM1Instance1.xmi";
    right uri "SimpleMM1Instance11.xmi";
  }

  object res0#/ {
    strval = "MyStringValue" | "MyNewStringValue";
  }

}
```

Figure A.3: Change String Textual Epatch

```
Epatch {
    attr EString name 'CHANGE_STRING'
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: SingleAssignment {
                    attr EString feature 'strval'
                    cref AssignmentValue leftValue AssignmentValue {
                        attr EString value 'MyStringValue'
                    }
                    cref AssignmentValue rightValue AssignmentValue {
                        attr EString value 'MyNewStringValue'
                    }
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '/'
        }
    ]
}
```

Figure A.4: Change String Epatch Model

Set String

See Figure A.5 and Figure A.6
From left to right: The value of attribute `strvalunset` is set to `StrVal`
From right to left: The value of attribute `strval` is unset. This is the same as calling eobject.eUnset() for the attribute or as assignng the default value

```
epatch SET_STRING {
  resource res0 {
    left uri "SimpleMM1Instance1.xmi";
    right uri "SimpleMM1Instance11.xmi";
  }

  object res0#/ {
    strvalunset = null | "StrVal";
  }

}
```

Figure A.5: Set String Textual Epatch

```
Epatch {
    attr EString name 'SET_STRING'
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: SingleAssignment {
                    attr EString feature 'strvalunset'
                    cref AssignmentValue leftValue AssignmentValue {
                        attr EString keyword 'null'
                    }
                    cref AssignmentValue rightValue AssignmentValue {
                        attr EString value 'StrVal'
                    }
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '/'
        }
    ]
}
```

Figure A.6: Set String Epatch Model

A.3.2 ListChanges

Add Int

See Figure A.7 and Figure A.8
From left to right: Sets the value of attribute `intlist` to `1` at index 5. Assuming that the list only had five items before, this means adding one to the list's end.
From right to left: The value at index 5 is removed from the attribute `intlist`.

```
epatch ADD_INT {
  resource res0 {
    left uri "SimpleMM1Instance1.xmi";
    right uri "SimpleMM1Instance11.xmi";
  }

  object res0#/ {
    intlist = [ | 5:"1" ];
  }

}
```

Figure A.7: Add Int Textual Epatch

```
Epatch {
    attr EString name 'ADD_INT'
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: ListAssignment {
                    attr EString feature 'intlist'
                    cref AssignmentValue rightValues [
                        0: AssignmentValue {
                            attr EString value '1'
                            attr EInt index '5'
                        }
                    ]
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '/'
        }
    ]
}
```

Figure A.8: Add Int Epatch Model

Add Remove

See Figure A.9 and Figure A.10
From left to right: From the attribute `intlist`, first the value at index 2 is removed and then value `1` is inserted at index 4
From right to left: From the attribute `intlist`, first the value at index 4 is removed and then value `6` is inserted at index 2

```
epatch ADD_REMOVE {
  resource res0 {
    left uri "SimpleMM1Instance1.xmi";
    right uri "SimpleMM1Instance11.xmi";
  }

  object res0#/ {
    intlist = [ 2:"6" | 4:"1" ];
  }

}
```

Figure A.9: Add Remove Textual Epatch

```
Epatch {
    attr EString name 'ADD_REMOVE'
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: ListAssignment {
                    attr EString feature 'intlist'
                    cref AssignmentValue leftValues [
                        0: AssignmentValue {
                            attr EString value '6'
                            attr EInt index '2'
                        }
                    ]
                    cref AssignmentValue rightValues [
                        0: AssignmentValue {
                            attr EString value '1'
                            attr EInt index '4'
                        }
                    ]
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '/'
        }
    ]
}
```

Figure A.10: Add Remove Epatch Model

List Set

See Figure A.11 and Figure A.12
From left to right: Sets the value of attribute `intlist` to 23 at index 2
From right to left: Sets the value of attribute `intlist` to 6 at index 2

```
epatch LIST_SET {
  resource res0 {
    left uri "SimpleMM1Instance1.xmi";
    right uri "SimpleMM1Instance11.xmi";
  }

  object res0#/ {
    intlist = [ 2:"6" | 2:"23" ];
  }

}
```

Figure A.11: List Set Textual Epatch

```
Epatch {
    attr EString name 'LIST_SET'
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: ListAssignment {
                    attr EString feature 'intlist'
                    cref AssignmentValue leftValues [
                        0: AssignmentValue {
                            attr EString value '6'
                            attr EInt index '2'
                        }
                    ]
                    cref AssignmentValue rightValues [
                        0: AssignmentValue {
                            attr EString value '23'
                            attr EInt index '2'
                        }
                    ]
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '/'
        }
    ]
}
```

Figure A.12: List Set Epatch Model

Move Int

See Figure A.13 and Figure A.14
From left to right: For the attribute `intlist`, the element at index 1 is removed from the list and then inserted again at index 4. This effectively moves the element from index 1 to index 4.
From right to left: For the attribute `intlist`, the element at index 4 is removed from the list and then inserted again at index 1. This effectively moves the element from index 4 to index 1.

```
epatch MOVE_INT {
  resource res0 {
    left uri "SimpleMM1Instance1.xmi";
    right uri "SimpleMM1Instance11.xmi";
  }

  object res0#/ {
    intlist = [ 1:[4] | 4:[1] ];
  }

}
```

Figure A.13: Move Int Textual Epatch

```
Epatch {
    attr EString name 'MOVE_INT'
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: ListAssignment {
                    attr EString feature 'intlist'
                    cref AssignmentValue leftValues [
                        0: AssignmentValue {
                            attr EInt refIndex '4'
                            attr EInt index '1'
                        }
                    ]
                    cref AssignmentValue rightValues [
                        0: AssignmentValue {
                            attr EInt refIndex '1'
                            attr EInt index '4'
                        }
                    ]
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '/'
        }
    ]
}
```

Figure A.14: Move Int Epatch Model

Multiple Add

See Figure A.15 and Figure A.16
From left to right: For the attribute `intlist`, the value 11 is inserted at index 2, then value 12 is inserted at index 3, and so on. The values are inserted into the list in exactly this order to ensure that after the adding process each value is at the index desctibed for it in the Epatch
From right to left: For attribute `intlist`, the values at the indices 5, 4, 3, and 2 are removed, starting with the element at the highest index. This order ensures that the removal of one element has no sideeffefts on the indices of elements sthat sill have to be removed.

```
epatch MULTIPLE_ADD {
  resource res0 {
    left uri "SimpleMM1Instance1.xmi";
    right uri "SimpleMM1Instance11.xmi";
  }

  object res0#/ {
    intlist = [ | 2:"11", 3:"12", 4:"13", 5:"14" ];
  }

}
```

Figure A.15: Multiple Add Textual Epatch

```
Epatch {
    attr EString name 'MULTIPLE_ADD'
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: ListAssignment {
                    attr EString feature 'intlist'
                    cref AssignmentValue rightValues [
                        0: AssignmentValue {
                            attr EString value '11'
                            attr EInt index '2'
                        }
                        1: AssignmentValue {
                            attr EString value '12'
                            attr EInt index '3'
                        }
                        2: AssignmentValue {
                            attr EString value '13'
                            attr EInt index '4'
                        }
                        3: AssignmentValue {
                            attr EString value '14'
                            attr EInt index '5'
                        }
                    ]
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '/'
        }
    ]
}
```

Figure A.16: Multiple Add Epatch Model

Remove Int

See Figure A.17 and Figure A.18
From left to right: The value at index 2 from attribute `intlist` is removed.
From right to left: The value `6` is inserted into the attribute `intlist` at index 2

```
epatch REMOVE_INT {
  resource res0 {
    left uri "SimpleMM1Instance1.xmi";
    right uri "SimpleMM1Instance11.xmi";
  }

  object res0#/ {
    intlist = [ 2:"6" | ];
  }

}
```

Figure A.17: Remove Int Textual Epatch

```
Epatch {
    attr EString name 'REMOVE_INT'
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: ListAssignment {
                    attr EString feature 'intlist'
                    cref AssignmentValue leftValues [
                        0: AssignmentValue {
                            attr EString value '6'
                            attr EInt index '2'
                        }
                    ]
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '/'
        }
    ]
}
```

Figure A.18: Remove Int Epatch Model

A.3.3 ObjectChanges

Add Object

See Figure A.19 and Figure A.20
From left to right: A new object of type `//CompositeNode` is instantiated and it's EAttribute `name` is set to `MyNeCompositeNode`. Then, this object is stored at index 3 of EReference `childen` of the object with the fragmentURI `//@tree`
From right to left: The entry at index 3 from EReference `children` is removed.

```
epatch ADD_OBJECT {
  import mm ns "http://www.itemis.de/emf/epatch/testmm1"
  resource res0 {
    left uri "SimpleMM1Instance1.xmi";
    right uri "SimpleMM1Instance11.xmi";
  }

  object res0#//@tree {
    children = [ | 3:new mm#//CompositeNode {
      name = "MyNewCompositeNode";
    } ];
  }

}
```

Figure A.19: Add Object Textual Epatch

```
Epatch {
    attr EString name 'ADD_OBJECT'
    cref ModelImport modelImports [
        0: EPackageImport {
            attr EString name 'mm'
            attr EString nsURI 'http://www.itemis.de/emf/epatch/testmm1'
        }
    ]
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: ListAssignment {
                    attr EString feature 'children'
                    cref AssignmentValue rightValues [
                        0: AssignmentValue {
                            cref CreatedObject newObject ObjectNew {
                                cref Assignment assignments [
                                    0: SingleAssignment {
                                        attr EString feature 'name'
                                        cref AssignmentValue leftValue AssignmentValue {
                                            attr EString value 'MyNewCompositeNode'
                                        }
                                    }
                                ]
                                ref ModelImport import ref: //@modelImports.0
                                attr EString impFrag '//CompositeNode'
                            }
                            attr EInt index '3'
                        }
                    ]
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '//@tree'
        }
    ]
}
```

Figure A.20: Add Object Epatch Model

Add Object With List

See Figure A.21 and Figure A.22
From left to right: Two objects of type CompositeNode and two of type ChildNode are instantiated. Their `name`-EAttribute is set. The first CompositeNode EReference `children` is filled with the other three objects and the object itself gets stored at index 3 of EReference `children` of the object with the fragmentURI `//@tree`. The `children`-EReferences are containment-references.
From right to left: The entry at index 3 from EReference `children` is removed.

```
1  epatch ADD_OBJECT_WITH_LIST {
2    import mm ns "http://www.itemis.de/emf/epatch/testmm1"
3    resource res0 {
4      left uri "SimpleMM1Instance1.xmi";
5      right uri "SimpleMM1Instance11.xmi";
6    }
7
8    object res0#//@tree {
9      children = [ | 3:new mm#//CompositeNode {
10       children = [ new mm#//CompositeNode {
11         name = "MyComp1";
12       }, new mm#//ChildNode {
13         name = "MyLeaf1";
14       }, new mm#//ChildNode {
15         name = "MyLeaf2";
16       } ];
17       name = "MyRoot";
18     } ];
19   }
20
21 }
```

Figure A.21: Add Object With List Textual Epatch

```
Epatch {
    attr EString name 'ADD_OBJECT_WITH_LIST'
    cref ModelImport modelImports [
        0: EPackageImport {
            attr EString name 'mm'
            attr EString nsURI 'http://www.itemis.de/emf/epatch/testmm1'
        }
    ]
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: ListAssignment {
                    attr EString feature 'children'
                    cref AssignmentValue rightValues [
                        0: AssignmentValue {
                            cref CreatedObject newObject ObjectNew {
                                cref Assignment assignments [
                                    0: ListAssignment {
                                        attr EString feature 'children'
                                        cref AssignmentValue leftValues [
                                            0: AssignmentValue {
                                                cref CreatedObject newObject ObjectNew {
                                                    cref Assignment assignments [
                                                        0: SingleAssignment {
                                                            attr EString feature 'name'
                                                            cref AssignmentValue leftValue AssignmentValue {
                                                                attr EString value 'MyComp1'
                                                            }
                                                        }
                                                    ]
                                                    ref ModelImport import ref: //@modelImports.0
                                                    attr EString impFrag '//CompositeNode'
                                                }
                                            }
                                            1: AssignmentValue {
                                                cref CreatedObject newObject ObjectNew {
                                                    cref Assignment assignments [
                                                        0: SingleAssignment {
                                                            attr EString feature 'name'
                                                            cref AssignmentValue leftValue AssignmentValue {
                                                                attr EString value 'MyLeaf1'
                                                            }
                                                        }
                                                    ]
                                                    ref ModelImport import ref: //@modelImports.0
                                                    attr EString impFrag '//ChildNode'
                                                }
                                            }
                                            2: AssignmentValue {
                                                cref CreatedObject newObject ObjectNew {
                                                    cref Assignment assignments [
                                                        0: SingleAssignment {
                                                            attr EString feature 'name'
                                                            cref AssignmentValue leftValue AssignmentValue {
                                                                attr EString value 'MyLeaf2'
                                                            }
                                                        }
                                                    ]
                                                    ref ModelImport import ref: //@modelImports.0
                                                    attr EString impFrag '//ChildNode'
                                                }
                                            }
                                        ]
                                    }
                                    1: SingleAssignment {
                                        attr EString feature 'name'
                                        cref AssignmentValue leftValue AssignmentValue {
                                            attr EString value 'MyRoot'
                                        }
                                    }
                                ]
                                ref ModelImport import ref: //@modelImports.0
                                attr EString impFrag '//CompositeNode'
                            }
                            attr EInt index '3'
                        }
                    ]
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '//@tree'
        }
    ]
}
```

Figure A.22: Add Object With List Epatch Model

Add Reference

See Figure A.23 and Figure A.24
From left to right: The object with the alias `ChildWithRef` is added to non-containment-EReference `reflist` at index 2The alias `ChildWithRef` points to the object with fragmentURI `//@tree/@children.1`.
From right to left: The entry at index 2 from EReference `reflist` is removed.

```
epatch ADD_REFERENCE {
  resource res0 {
    left uri "SimpleMM1Instance1.xmi";
    right uri "SimpleMM1Instance11.xmi";
  }

  object res0#/ {
    reflist = [ | 2:ChildWithRef ];
  }

  object ChildWithRef res0#//@tree/@children.1 {
  }

}
```

Figure A.23: Add Reference Textual Epatch

```
Epatch {
    attr EString name 'ADD_REFERENCE'
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: ListAssignment {
                    attr EString feature 'reflist'
                    cref AssignmentValue rightValues [
                        0: AssignmentValue {
                            ref NamedObject refObject ref: //@objects.1
                            attr EInt index '2'
                        }
                    ]
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '/'
        }
        1: ObjectRef {
            attr EString name 'ChildWithRef'
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '//@tree/@children.1'
        }
    ]
}
```

Figure A.24: Add Reference Epatch Model

Move Object From List To List

See Figure A.25 and Figure A.26
From left to right: The object with alias `ChildWithRef` is removed from the `children`-EReference of object `//@tree` and added to the `children`-EREference of object `//@tree/@children.0`. Since ChildWithRef changes it's location within the Resource, it's fragmentURI changes. `//@tree/@children.1` identifies the object in the left model, and `//@tree/@children.0/@children.1` in the right model.
From right to left: The object with alias `ChildWithRef` is removed from the `children`-EReference of object `//@tree/@children.0` and added to the `children`-EREference of object `//@tree`.

```
1  epatch MOVE_OBJECT_FROM_LIST_TO_LIST {
2    resource res0 {
3      left uri "SimpleMM1Instance1.xmi";
4      right uri "SimpleMM1Instance11.xmi";
5    }
6
7    object res0#//@tree {
8      children = [ 1:ChildWithRef | ];
9    }
10
11   object res0#//@tree/@children.0 {
12     children = [ | 1:ChildWithRef ];
13   }
14
15   object ChildWithRef left res0#//@tree/@children.1 right res0#//@tree/@children.0/@children.1 {
16   }
17
18 }
```

Figure A.25: Move Object From List To List Textual Epatch

```
Epatch {
    attr EString name 'MOVE_OBJECT_FROM_LIST_TO_LIST'
    cref NamedResource resources [
        0: NamedResource {
            attr EString name 'res0'
            attr EString leftUri 'SimpleMM1Instance1.xmi'
            attr EString rightUri 'SimpleMM1Instance11.xmi'
        }
    ]
    cref ObjectRef objects [
        0: ObjectRef {
            cref Assignment assignments [
                0: ListAssignment {
                    attr EString feature 'children'
                    cref AssignmentValue leftValues [
                        0: AssignmentValue {
                            ref NamedObject refObject ref: //@objects.2
                            attr EInt index '1'
                        }
                    ]
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '//@tree'
        }
        1: ObjectRef {
            cref Assignment assignments [
                0: ListAssignment {
                    attr EString feature 'children'
                    cref AssignmentValue rightValues [
                        0: AssignmentValue {
                            ref NamedObject refObject ref: //@objects.2
                            attr EInt index '1'
                        }
                    ]
                }
            ]
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '//@tree/@children.0'
        }
        2: ObjectRef {
            attr EString name 'ChildWithRef'
            ref NamedResource leftRes ref: //@resources.0
            attr EString leftFrag '//@tree/@children.1'
            ref NamedResource rightRes ref: //@resources.0
            attr EString rightFrag '//@tree/@children.0/@children.1'
        }
    ]
}
```

Figure A.26: Move Object From List To List Epatch Model

A.4 Metapatch Examples/Tests

A.4.1 Extract And Reference Author Exp

See Figure A.28 and Figure A.27

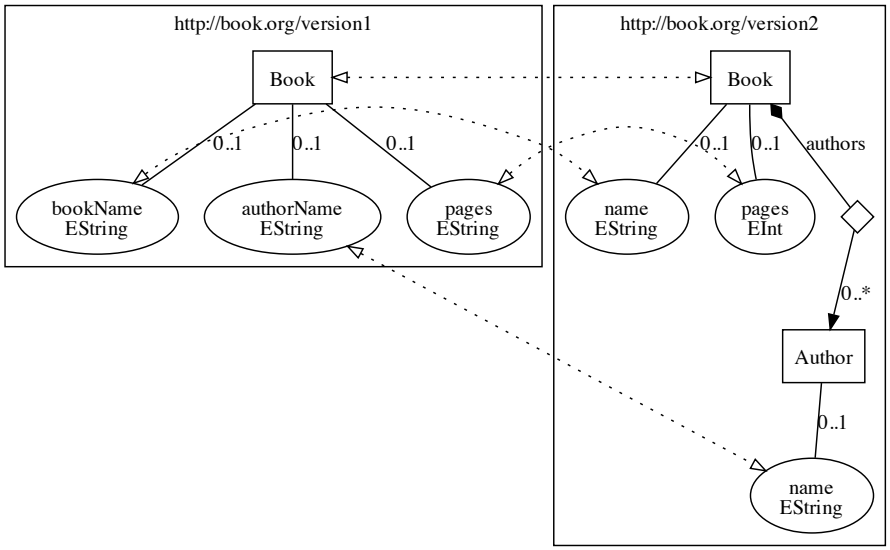

Figure A.27: Extract And Reference Author Exp Migration Graph

```
metapatch EXTRACT_AND_REFERENCE_AUTHOR_EXP {
  import ecore ns 'http://www.eclipse.org/emf/2002/Ecore'

  resource res0 {
    left uri 'book-v1.ecore';
    right uri 'book-v2.ecore';
  }

  object res0#/ {
    eClassifiers = [ | 1:new ecore#//EClass Author {
      eStructuralFeatures = [ name ];
      name = 'Author';
    } ];
    nsURI = "http://book.org/version1" | "http://book.org/version2";
  }

  object res0#//Book {
    eStructuralFeatures = [ 1:name | 2:new ecore#//EReference {
      containment = 'true';
      eType = Author;
      name = 'authors';
      upperBound = '-1';
      instances { new rightBook::Author.migrateFrom(srcEObject) };
    } ];
    left instance new leftBook::Book.migrateFrom(srcValue).
                  migrateFrom(srcValue.authors.first());
  }

  object name left res0#//Book/authorName right res0#//Author/name {
    name = 'authorName' | 'name';
  }

  object left res0#//Book/bookName right res0#//Book/name {
    name = 'bookName' | 'name';
  }

  object res0#//Book/pages {
    eType = ecore#//EString | ecore#//EInt;
    left instance srcValue > -1 ? srcValue.toString() : "unknown";
    right instance srcValue != null && srcValue.matches("[0-9]+") ? srcValue.asInteger() : -1;
  }
}
```

Figure A.28: Extract And Reference Author Exp Textual Metapatch

A.4.2 Extract And Reference Author Java

See Figure A.30 and Figure A.29

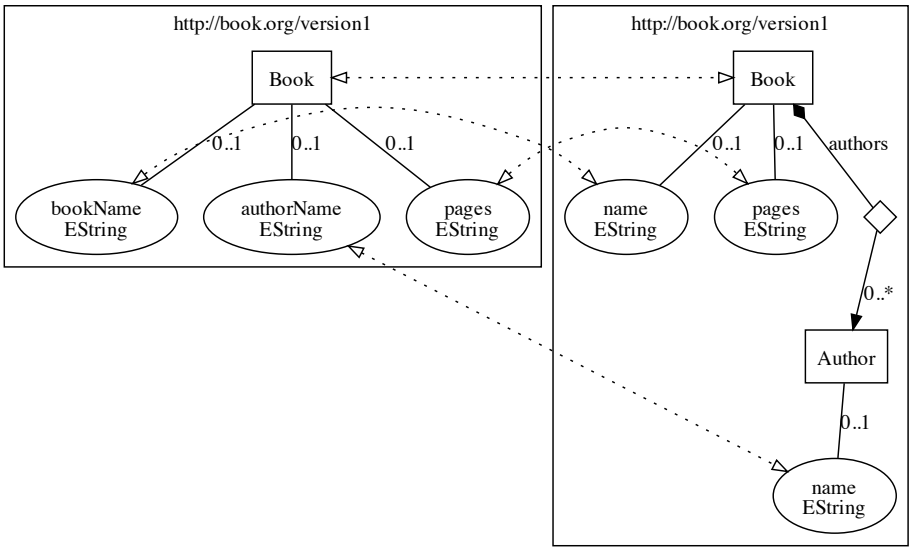

Figure A.29: Extract And Reference Author Java Migration Graph

```
metapatch EXTRACT_AND_REFERENCE_AUTHOR_JAVA {
  import ecore ns 'http://www.eclipse.org/emf/2002/Ecore'
  import java org.eclipse.emf.edapt.^metapatch.tests.book.BookChanges_EXTRACT_AND_REFERENCE_AUTHOR

  resource res0 {
    left uri 'book.ecore';
    right uri 'book1.ecore';
  }

  object res0#/ {
    eClassifiers = [ | 1:new ecore#//EClass Author {
      eStructuralFeatures = [ name ];
      name = 'Author';
    } ];
  }

  object res0#//Book {
    eStructuralFeatures = [ 1:name | 2:new ecore#//EReference {
      containment = 'true';
      eType = Author;
      name = 'authors';
      upperBound = '-1';
      instance java createAuthor();
    } ];
  }

  object name left res0#//Book/authorName right res0#//Author/name {
    name = 'authorName' | 'name';
  }

  object left res0#//Book/bookName right res0#//Book/name {
    name = 'bookName' | 'name';
  }
}
```

Figure A.30: Extract And Reference Author Java Textual Metapatch

A.4.3 Inline Class

See Figure A.32 and Figure A.31

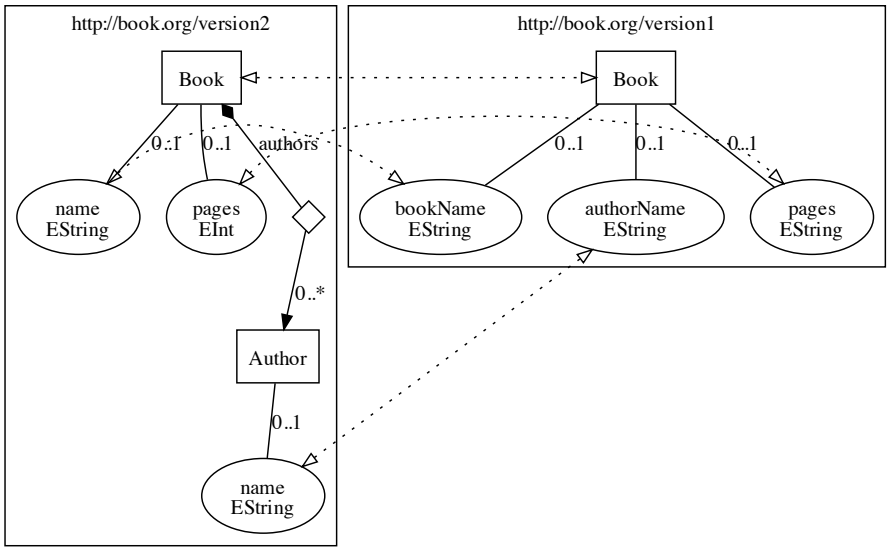

Figure A.31: Inline Class Migration Graph

```
metapatch EXTRACT_AND_REFERENCE_AUTHOR_EXP {
  import ecore ns 'http://www.eclipse.org/emf/2002/Ecore'

  resource res0 {
    left uri 'book-v1.ecore';
    right uri 'book-v2.ecore';
  }

  object res0#/ {
    eClassifiers = [ | 1:new ecore#//EClass Author {
      eStructuralFeatures = [ name ];
      name = 'Author';
    } ];
    nsURI = "http://book.org/version1" | "http://book.org/version2";
  }

  object res0#//Book {
    eStructuralFeatures = [ 1:name | 2:new ecore#//EReference {
      containment = 'true';
      eType = Author;
      name = 'authors';
      upperBound = '-1';
      instances { new rightBook::Author.migrateFrom(srcEObject) };
    } ];
    left instance new leftBook::Book.migrateFrom(srcValue).
                  migrateFrom(srcValue.authors.first());
  }

  object name left res0#//Book/authorName right res0#//Author/name {
    name = 'authorName' | 'name';
  }

  object left res0#//Book/bookName right res0#//Book/name {
    name = 'bookName' | 'name';
  }

  object res0#//Book/pages {
    eType = ecore#//EString | ecore#//EInt;
    left instance srcValue > -1 ? srcValue.toString() : "unknown";
    right instance srcValue != null && srcValue.matches("[0-9]+") ? srcValue.asInteger() : -1;
  }
}
```

Figure A.32: Inline Class Textual Metapatch

A.4.4 Rename And Retype

See Figure A.34 and Figure A.33

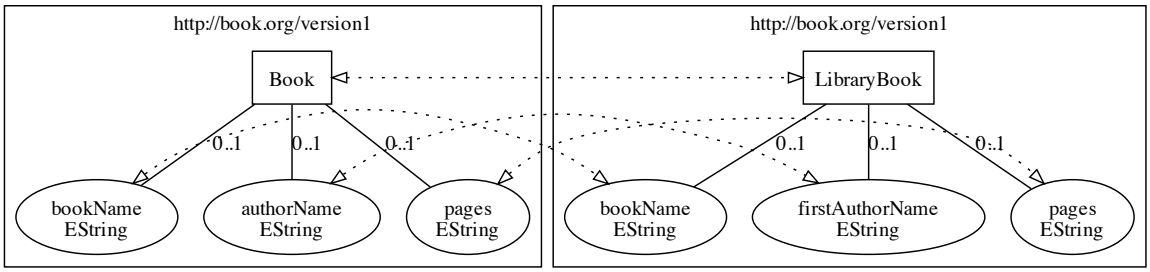

Figure A.33: Rename And Retype Migration Graph

```
1  metapatch RENAME_AND_RETYPE {
2    import ecore ns 'http://www.eclipse.org/emf/2002/Ecore'
3
4    resource res0 {
5      left uri 'book-v1.ecore';
6      right uri 'book-v2.ecore';
7    }
8
9    object left res0#//Book right res0#//LibraryBook {
10     name = 'Book' | 'LibraryBook';
11   }
12
13   object left res0#//Book/authorName right res0#//LibraryBook/firstAuthorName {
14     name = 'authorName' | 'firstAuthorName';
15   }
16
17   object left res0#//Book/pages right res0#//LibraryBook/pages {
18     eType = ecore#//EInt | ecore#//EString;
19   }
20 }
```

Figure A.34: Rename And Retype Textual Metapatch

A.4.5 Merge Features And Subclass

See Figure A.36 and Figure A.35

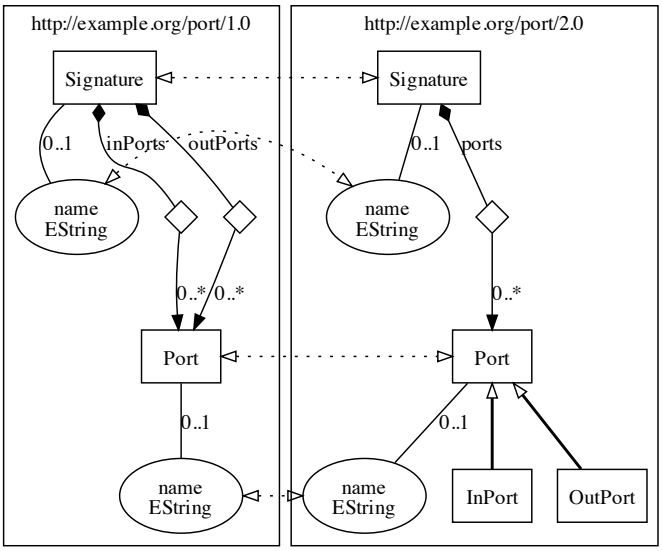

Figure A.35: Merge Features And Subclass Migration Graph

```
metapatch port_v1_v2 {
  import ecore ns "http://www.eclipse.org/emf/2002/Ecore"

  resource res0 {
    left uri "port-v1.ecore";
    right uri "port-v2.ecore";
  }

  object res0#/ {
    eClassifiers = [ | 2:new ecore#//EClass {
      eSuperTypes = [ Port ];
      name = "InPort";
    }, 3:new ecore#//EClass {
      eSuperTypes = [ Port ];
      name = "OutPort";
    } ];
    nsURI = "http://example.org/port/1.0" | "http://example.org/port/2.0";
  }

  object res0#//Signature {
    eStructuralFeatures = [ 2:new ecore#//EReference {
      containment = "true";
      eType = Port;
      name = "outPorts";
      upperBound = "-1";
    }, 1:new ecore#//EReference {
      containment = "true";
      eType = Port;
      name = "inPorts";
      upperBound = "-1";
    } | 1:new ecore#//EReference {
      containment = "true";
      eType = Port;
      name = "ports";
      upperBound = "-1";
      instance srcEObject.inPorts.collect(p|new rightPort::InPort.migrateFrom(p)).union(
               srcEObject.outPorts.collect(p|new rightPort::OutPort.migrateFrom(p)));
    } ];
  }

  object Port res0#//Port
}
```

Figure A.36: Merge Features And Subclass Textual Metapatch

A.4.6 Split Feature And Remove Subclass

See Figure A.38 and Figure A.37

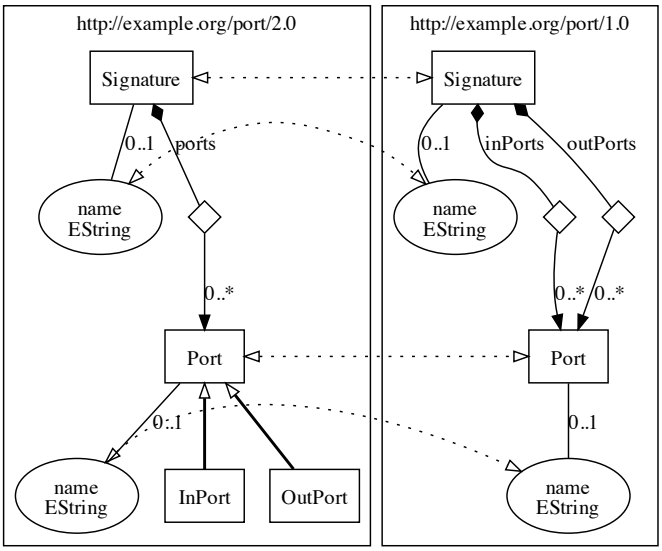

Figure A.37: Split Feature And Remove Subclass Migration Graph

```
1  metapatch port_v2_v1 {
2    import ecore ns "http://www.eclipse.org/emf/2002/Ecore"
3
4    resource res0 {
5      left uri "port-v1.ecore";
6      right uri "port-v2.ecore";
7    }
8
9    object res0#/ {
10     eClassifiers = [ | 2:new ecore#//EClass {
11       eSuperTypes = [ Port ];
12       name = "InPort";
13     }, 3:new ecore#//EClass {
14       eSuperTypes = [ Port ];
15       name = "OutPort";
16     } ];
17     nsURI = "http://example.org/port/1.0" | "http://example.org/port/2.0";
18   }
19
20   object res0#//Signature {
21     eStructuralFeatures = [ 2:new ecore#//EReference {
22       containment = "true";
23       eType = Port;
24       name = "outPorts";
25       upperBound = "-1";
26       instance srcEObject.ports.typeSelect(rightPort::OutPort).migrate();
27     }, 1:new ecore#//EReference {
28       containment = "true";
29       eType = Port;
30       name = "inPorts";
31       upperBound = "-1";
32       instance srcEObject.ports.typeSelect(rightPort::InPort).migrate();
33     } | 1:new ecore#//EReference {
34       containment = "true";
35       eType = Port;
36       name = "ports";
37       upperBound = "-1";
38     } ];
39   }
40
41   object Port res0#//Port
42 }
```

Figure A.38: Split Feature And Remove Subclass Textual Metapatch

A.4.7 Move Feature Down The Inheritace Hierarchy

See Figure A.40 and Figure A.39

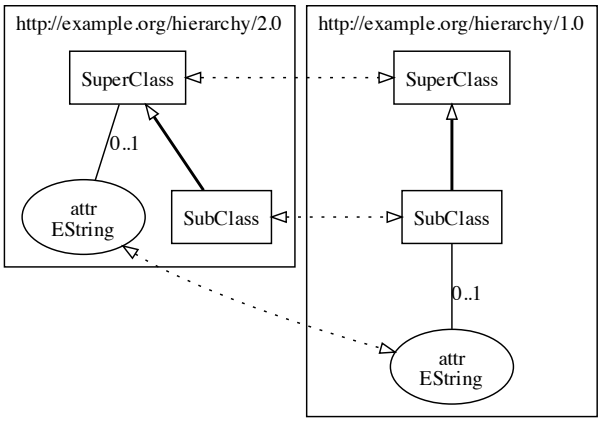

Figure A.39: Move Feature Down The Inheritace Hierarchy Migration Graph

```
1  metapatch hierarchy {
2    resource res0 {
3      left uri "hierarchy-v1.ecore";
4      right uri "hierarchy-v2.ecore";
5    }
6
7    object res0#/ {
8      nsURI = "http://example.org/hierarchy/1.0" | "http://example.org/hierarchy/2.0";
9    }
10
11   object res0#//SubClass {
12     eStructuralFeatures = [ 0:attr | ];
13   }
14
15   object res0#//SuperClass {
16     eStructuralFeatures = [ | 0:attr ];
17   }
18
19   object attr left res0#//SubClass/attr right res0#//SuperClass/attr
20 }
```

Figure A.40: Move Feature Down The Inheritace Hierarchy Textual Metapatch

A.4.8 Move Feature Up The Inheritace Hierarchy

See Figure A.42 and Figure A.41

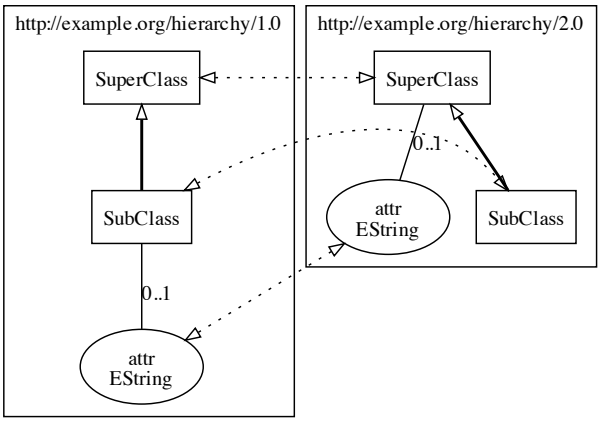

Figure A.41: Move Feature Up The Inheritace Hierarchy Migration Graph

```
1  metapatch hierarchy {
2    resource res0 {
3      left uri "hierarchy-v1.ecore";
4      right uri "hierarchy-v2.ecore";
5    }
6
7    object res0#/ {
8      nsURI = "http://example.org/hierarchy/1.0" | "http://example.org/hierarchy/2.0";
9    }
10
11   object res0#//SubClass {
12     eStructuralFeatures = [ 0:attr | ];
13   }
14
15   object res0#//SuperClass {
16     eStructuralFeatures = [ | 0:attr ];
17   }
18
19   object attr left res0#//SubClass/attr right res0#//SuperClass/attr
20 }
```

Figure A.42: Move Feature Up The Inheritace Hierarchy Textual Metapatch

A.4.9 Split Into Subclasses

See Figure A.44 and Figure A.43

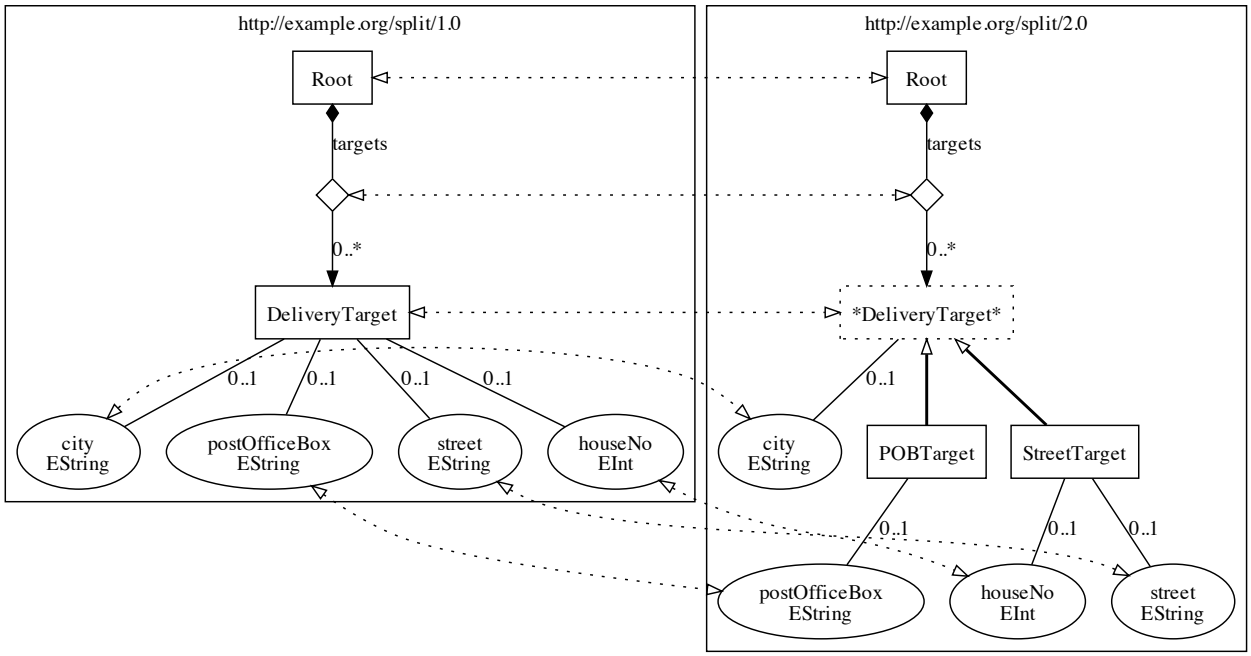

Figure A.43: Split Into Subclasses Migration Graph

```
metapatch split {
  import ecore ns "http://www.eclipse.org/emf/2002/Ecore"

  resource res0 {
    left uri "split-v1.ecore";
    right uri "split-v2.ecore";
  }

  object res0#/ {
    eClassifiers = [ | 2:new ecore#//EClass {
      eStructuralFeatures = [ postOfficeBox ];
      eSuperTypes = [ DeliveryTarget ];
      name = "POBTarget";
    }, 3:new ecore#//EClass {
      eStructuralFeatures = [ houseNo, street ];
      eSuperTypes = [ DeliveryTarget ];
      name = "StreetTarget";
    } ];
    nsURI = "http://example.org/split/1.0" | "http://example.org/split/2.0";
  }

  object DeliveryTarget res0#//DeliveryTarget {
    abstract = "false" | "true";
    eStructuralFeatures = [ 3:houseNo, 2:street, 1:postOfficeBox | ];
    right instance srcValue.postOfficeBox != null ? new rightSplit::POBTarget : new rightSplit::StreetTarget;
  }

  object houseNo left res0#//DeliveryTarget/houseNo right res0#//StreetTarget/houseNo

  object postOfficeBox left res0#//DeliveryTarget/postOfficeBox right res0#//POBTarget/postOfficeBox

  object street left res0#//DeliveryTarget/street right res0#//StreetTarget/street
}
```

Figure A.44: Split Into Subclasses Textual Metapatch

A.4.10 Remove Subclasses

See Figure A.46 and Figure A.45

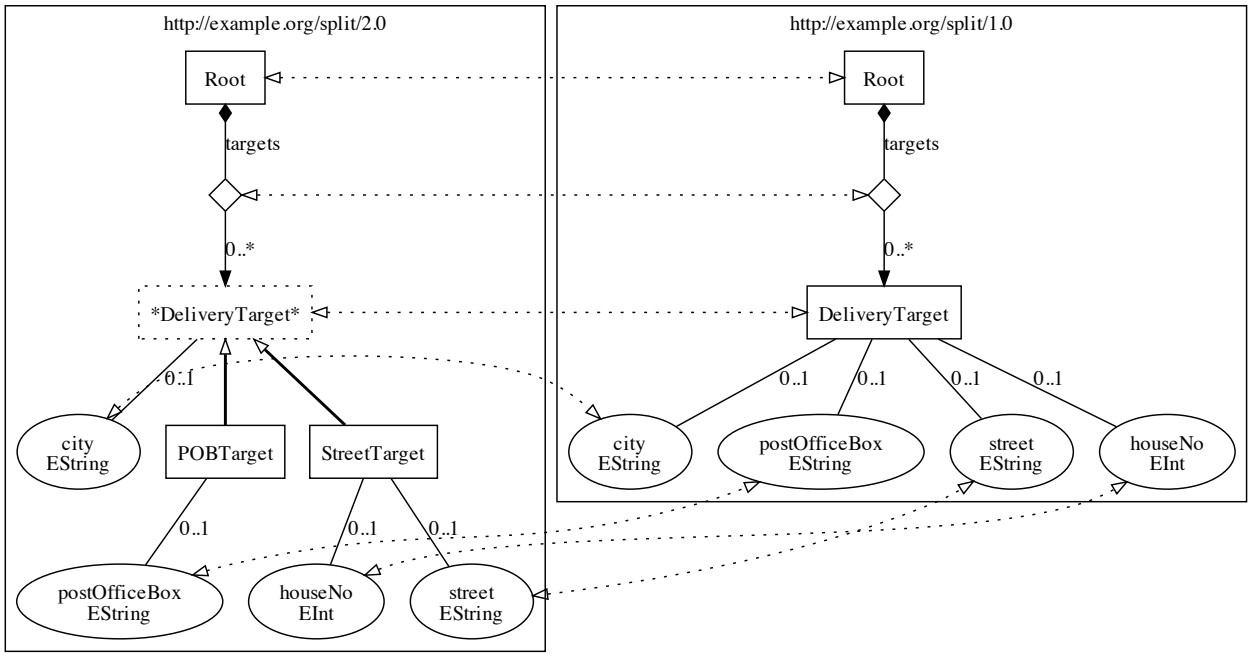

Figure A.45: Remove Subclasses Migration Graph

```
metapatch split {
  import ecore ns "http://www.eclipse.org/emf/2002/Ecore"

  resource res0 {
    left uri "split-v1.ecore";
    right uri "split-v2.ecore";
  }

  object res0#/ {
    eClassifiers = [ | 2:new ecore#//EClass {
      eStructuralFeatures = [ postOfficeBox ];
      eSuperTypes = [ DeliveryTarget ];
      name = "POBTarget";
    }, 3:new ecore#//EClass {
      eStructuralFeatures = [ houseNo, street ];
      eSuperTypes = [ DeliveryTarget ];
      name = "StreetTarget";
    } ];
    nsURI = "http://example.org/split/1.0" | "http://example.org/split/2.0";
  }

  object DeliveryTarget res0#//DeliveryTarget {
    abstract = "false" | "true";
    eStructuralFeatures = [ 3:houseNo, 2:street, 1:postOfficeBox | ];
    right instance srcValue.postOfficeBox != null ? new rightSplit::POBTarget : new rightSplit::StreetTarget;
  }

  object houseNo left res0#//DeliveryTarget/houseNo right res0#//StreetTarget/houseNo

  object postOfficeBox left res0#//DeliveryTarget/postOfficeBox right res0#//POBTarget/postOfficeBox

  object street left res0#//DeliveryTarget/street right res0#//StreetTarget/street
}
```

Figure A.46: Remove Subclasses Textual Metapatch

List of Tables

List of Figures

Bibliography

[Ash06] D. Ashlock. *Evolutionary Computation for Modeling and Optimization*. Springer-Verlag New York Inc, 2006.

[Aya07] F.J. Ayala. Colloquium Papers: Darwin's greatest discovery: Design without designer. *Proceedings of the National Academy of Sciences*, 104(suppl_1):8567, 2007.

[Bäc96] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford Univ. Pr., 1996.

[Bec03] K. Beck. *Test-driven development: By example*. Addison-Wesley Professional, 2003.

[Ber03] P.A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Conference on Innovative Data Systems Research (CIDR)*, pages 209–220, 2003.

[BGGK07] Steffen Becker, Boris Gruschko, Thomas Goldschmidt, and Heiko Koziolek. A Process Model and Classification Scheme for Semi-Automatic Meta-Model Evolution. In *Proc. 1st Workshop "MDD, SOA und IT-Management" (MSI'07)*, pages 35–46. GI, GiTO-Verlag, 2007.

[BSM+03] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and TJ Grose. *Eclipse Modeling Framework (The Eclipse Series)*. Addison-Wesley Professional, Reading, 2003.

[CB07] Antonie Cédric Brun. Presentation: EMF Comparison Framework - A quick glance . EclipseCon, 2007.

[CDRP07] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Technology*, 6(9):165–185, 2007.

[CMTZ08] C.A. Curino, H.J. Moon, L. Tanca, and C. Zaniolo. Schema Evolution In Wikipedia. In *ICEIS*, 2008.

[CS03] G. Caplat and J.L. Sourrouille. Considerations about Model Mapping. In *Proceedings of the Workshop in Software Model Engineering WiSME@ UML*, pages 1–6, 2003.

[Dar59] Charles Darwin. *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life*. London: John Murray, first edition, October 1859.

[Den08] Mariya Denysova. Refinement of Alternation Traces in Context of Model-Driven Change Management. Master's thesis, Hamburg University of Technology, 2008.

[DGF05] S. Ducasse, T. Gîrba, and J.M. Favre. Modeling Software Evolution by Treating History as a First Class Entity. *Electronic Notes in Theoretical Computer Science*, 127(3):75–86, 2005.

[dGSA07] G. de Geest, A. Savelkoul, and A. Alikoski. Building a framework to support Domain-Specific Language evolution using Microsoft DSL Tools. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modelling*, 2007.

[DW04] D. Draheim and G. Weber. Specification and generation of model 2 web interfaces. *Lecture Notes in Computer Science*, pages 101–110, 2004.

[Ecla] ATLAS Transformation Language. Available from World Wide Web: `http://www.eclipse.org/m2m/atl/` [cited April 20th, 2009].

[Eclb] Eclipse Business Process Modeling Notation (BPMN) Modeler. Available from World Wide Web: `http://www.eclipse.org/bpmn/` [cited April 20th, 2009].

[Eclc] Eclipse Graphical Modeling Framework (GMF). Available from World Wide Web: `http://www.eclipse.org/modeling/gmf/` [cited April 20th, 2009].

[Ecld] Eclipse Model Query. Available from World Wide Web: `http://www.eclipse.org/modeling/emf/?project=query` [cited April 20th, 2009].

[Ecle] Eclipse Modeling Framework Project (EMF). Available from World Wide Web: `http://www.eclipse.org/modeling/emf/` [cited April 20th, 2009].

[Eclf] Eclipse TMF Xtext. Available from World Wide Web: `http://www.xtext.org/` [cited April 20th, 2009].

[Eclg] Eclipse.org home. Available from World Wide Web: `http://www.eclipse.org/` [cited 08.08.08].

[Eclh] EMF Compare. Available from World Wide Web: `http://wiki.eclipse.org/index.php/EMF_Compare` [cited April 20th, 2009].

[EH] Moritz Eysholdt and Markus Herrmannsdörfer. Edapt - Project Proposal - Framework for Ecore model adaptation and instance migration. Available from World Wide Web: `http://www.eclipse.org/proposals/edapt/` [cited April 20th, 2009].

[EMF] Emf performance: Emf 2.0.1 vs. emf 2.1.0 rc1. Available from World Wide Web: `http://www.eclipse.org/modeling/emf/docs/performance/EMFPerformanceTestsResults.html` [cited April 20th, 2009].

[Fav05] J.M. Favre. Languages evolve too! Changing the software time scale. *Principles of Software Evolution, Eighth International Workshop on*, pages 33–42, 2005.

[FB] M. Fowler and MF Bloki. Domain Specific Language. Available from World Wide Web: `http://www.martinfowler.com/bliki/DomainSpecificLanguage.html` [cited April 20th, 2009].

[FEJ05] D.J. Futuyma, S.V. Edwards, and R. John. *Evolution*. Sinauer Associates, 2005.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[GB99] E. Gamma and K. Beck. JUnit: A cook's tour. *Java Report*, 4(5):27–38, 1999.

[GMR05] G. Guerrini, M. Mesiti, and D. Rossi. Impact of XML schema evolution on valid documents. *Proceedings of the seventh ACM international workshop on Web information and data management*, pages 39–44, 2005.

[Gru93] T.R. Gruber. A translation approach to portable ontology specifications. *KNOWLEDGE ACQUISITION*, 5:199–199, 1993.

[GT00] MW Godfrey and Q. Tu. Evolution in open source software: a case study. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 131–142, 2000.

[Har07] M. Hartung. Automatisierte Umsetzung von komplexen XML-Schemaänderungen. *BTW Workshop "Model Management und Metadaten-Verwaltung", Aachen*, 03 2007.

[HBJ08a] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Automatability of coupled evolution of metamodels and models in practice. In *11th International Conference on Model Driven Engineering Languages and Systems*. Springer, 2008.

[HBJ08b] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Cope: A language for the coupled evolution of metamodels and models. In *1st International Workshop on Model Co-Evolution and Consistency Management*, 2008.

[HBJ08c] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Cope: Coupled evolution of metamodels and models for the eclipse modeling framework. *Eclipse Modeling Symposium*, 2008.

[HD05] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. *Proceedings of the 27th international conference on Software engineering*, pages 274–283, 2005.

[HSE05] J. Hoßler, M. Soden, and H. Eichler. Coevolution of models, metamodels and transformations. *Models and Human Reasoning. Wissenschaft und Technik Verlag, Berlin*, pages 129–154, 2005.

[Jet] Jetbrains. Meta programming system. Available from World Wide Web: `http://www.jetbrains.com/mps/index.html` [cited April 20th, 2009].

[Kle07] M. Klettke. Conceptual XML Schema Evolution. *BTW Workshop "Model Management und Metadaten-Verwaltung", Aachen*, 2007.

[Kön08] P. Könemann. Model-Independent Diffs. Technical report, DTU Informatics, Building 321, Lyngby, 2008.

[KR92] J.R. Koza and J.P. Rice. *Genetic programming.* Springer, 1992.

[KR01] F. Keienburg and A. Rausch. Using XML/XMI for Tool Supported Evolution of UML Models. *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, page 10, 2001.

[LB85] MM Lehman and LA Belady. *Program evolution: processes of software change*. London Academic Press, 1985.

[LL01] Ralf Lämmel and Wolfgang Lohmann. Format Evolution. In *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155, pages 113–134. OCG, 2001.

[LR02] M.M. Lehman and J.F. Ramil. Software Evolution. *Encyclopedia of Software Engineering*, 2:1507–1513, 2002.

[LRW+97] MM Lehman, JF Ramil, PD Wernick, DE Perry, and WM Turski. Metrics and Laws of Software Evolution-The Nineties View. *4th International Software Metrics Symposium (METRICS'97)*, 1997.

[Met] Metacase. Metaedit+ modeler. Available from World Wide Web: `http://www.metacase.com/mep/` [cited April 20th, 2009].

[Mey96] Bertrand Meyer. Schema evolution: Concepts, terminology, and solutions. *Computer*, 29(10):119–121, 1996.

[MHS05] M. Mernik, JAN Heering, and A.M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[NK04] N.F. Noy and M. Klein. Ontology Evolution: Not the Same as Schema Evolution. *Knowledge and Information Systems*, 6(4):428–440, 2004.

[oAWa] openArchitectureWare Check Model Checking Language Reference. Available from World Wide Web: `http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html#Check_language` [cited April 20th, 2009].

[oAWb] openArchitectureWare Expression Language. Available from World Wide Web: `http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html#r10_expressions_language` [cited April 20th, 2009].

[oAWc] openArchitectureWare Xpand2 M2T Transformation Language Reference. Available from World Wide Web: `http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html#xpand_reference_introduction` [cited April 20th, 2009].

[oAWd] Xtend M2M Transformation Laguage Reference. Available from World Wide Web: `http://www.eclipse.org/gmt/oaw/doc/4.3/html/contents/core_reference.html` [cited 19.8.2008].

[Obj03] Object Management Group. *MDA Guide*, version 1.0.1 edition, June 2003. Available from World Wide Web: `http://www.omg.org/docs/omg/03-06-01.pdf` [cited April 20th, 2009].

[Obj07] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, final adopted specification edition, July 2007.

[OMGa] Catalog of OMG Modeling and Metadata Specifications: Object Constraint Language (OCL). Available from World Wide Web: `http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL` [cited April 20th, 2009].

[OMGb] Object Management Group (OMG). Available from World Wide Web: `http://www.omg.org/` [cited April 20th, 2009].

[OMGc] OMG Unified Modeling Language (UML). Available from World Wide Web: `http://www.uml.org/` [cited 20.8.2008].

[OMGd] OMG's MetaObject Facility. Available from World Wide Web: `http://www.omg.org/mof/` [cited April 20th, 2009].

[OMG07] *MOF Models to Text Transformation Language*. Object Management Group, beta 2 edition, August 2007. Available from World Wide Web: `http://www.omg.org/docs/ptc/07-08-16.pdf` [cited April 20th, 2009].

[OSBE01] D. Oglesby, K. Schloegel, D. Bhatt, and E. Engstrom. A Pattern-based Framework to Address Abstraction, Reuse, and Cross-domain Aspects in Domain Specific Visual Languages. In *Proc. of OOPSLA*, volume 2001, 2001.

[PJ07] M. Pizka and E. Jurgens. Tool Supported Multi Level Language Evolution. *To Appear*, 2007.

[PL] R. Salz P. Leach, M. Mealling. A universally unique identifier (uuid) urn namespace. Available from World Wide Web: `http://www.ietf.org/rfc/rfc4122.txt` [cited April 20th, 2009].

[Ray04] E.S. Raymond. *The Art of Unix Programming*. Addison-Wesley Professional, 2004.

[RB06] E. Rahm and P.A. Bernstein. An online bibliography on schema evolution. *ACM SIGMOD Record*, 35(4):30–31, 2006.

[RCN+98] C. Ryan, JJ Collins, M.O. Neill, W. Banzhaf, R. Poli, M. Schoenauer, and T.C. Fogarty. Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Proceedings of the First European Workshop on Genetic Programming*, 1391:83–95, 1998.

[Rig] Pascal Rigaux. History of Programming Languages. Available from World Wide Web: `http://merd.sourceforge.net/pixel/language-study/diagram.html` [cited 20.4.2009].

[Rod95] J.F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1995.

[SA00] A.P. Sage and J.E. Armstrong. *Introduction to systems engineering*. Wiley New York, 2000.

[SCC06] C. Simonyi, M. Christerson, and S. Clifford. Intentional Software. In *Proceedings of the 2006 OOPSLA Conference*, volume 41, pages 451–464. ACM New York, NY, USA, 2006.

[Sch98] A.W. Scheer. ARIS-House of Business Engineering. *Handbook of Life Cycle Engineering: Concepts, Models, and Technologies*, 1998.

[Sch08] M Scheidgen. *Describing Computer Languages.* PhD thesis, Humboldt-Universit"at zu Berlin, 2008.

[SHT+77] NC Shu, BC Housel, RW Taylor, SP Ghosh, and VY Lum. EXPRESS: a data EXtraction, Processing, and Restructuring System. *ACM Transactions on Database Systems (TODS)*, 2(2):134–174, 1977.

[SK04] J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15(3-4):291–307, 2004.

[SKC+01] H. Su, D. Kramer, L. Chen, K.T. Claypool, and E.A. Rundensteiner. XEM: Managing the Evolution of XML Documents. *Eleventh International Workshop on Research Issues in Data Engineering on Document Management for Data Intensive Business and Scientific Applications*, pages 103–110, 2001.

[Spi01] Diomidis Spinellis. Notable Design Patterns for Domain Specific Languages. *Journal of Systems and Software*, 56(1):91–99, February 2001. Available from World Wide Web: `http://www.spinellis.gr/pubs/jrnl/2000-JSS-DSLPatterns/html/dslpat.html` [cited April 20th, 2009].

[SV06] T. Stahl and M. Voelter. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.

[Tol08] Juha-Pekka Tolvanen. Building domain-specific modeling languages with full code generation. Tutorial Notes of the JAOO 2008 Conference in Aarhus, 2008.

[TPK07] J.P. Tolvanen, R. Pohjonen, and S. Kelly. Advanced Tooling for Domain-Specific Modeling: MetaEdit. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07)*, 2007.

[vDVW07] Arie van Deursen, Eelco Visser, and Jos Warmer. Model-Driven Software Evolution: A Research Agenda. Technical report, Tu Delft, 2007.

[W3Ca] W3C Extensible Markup Language (XML). Available from World Wide Web: `http://www.w3.org/XML/` [cited April 20th, 2009].

[W3Cb] W3C XML Schema. Available from World Wide Web: `http://www.w3.org/XML/Schema` [cited April 20th, 2009].

[W3Cc] W3C's Simple Object Access Protocol (SOAP). Available from World Wide Web: `http://www.w3.org/TR/soap/` [cited April 20th, 2009].

[W3Cd] Web ontology language (owl). Available from World Wide Web: `http://www.w3.org/2004/OWL/` [cited April 20th, 2009].

[W3Ce] Xml path language (xpath). Available from World Wide Web: `http://www.w3.org/TR/xpath` [cited April 20th, 2009].

[W3Cf] XSL Transformations (XSLT) Version 1.0. Available from World Wide Web: `http://www.w3.org/TR/xslt.html` [cited April 20th, 2009].

[Wac07] G. Wachsmuth. Metamodel Adaptation and Model Co-adaptation. *ECOOP*, 7:600–624, 2007.

[WAD07] M.J. WADE. The co-evolutionary genetics of ecological communities. *Nature reviews. Genetics(Print)*, 8(3):185–195, 2007.

[WL08] Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 921–922, New York, NY, USA, 2008. ACM.

[XSD04] *XML Schema to Ecore Mapping*, June 2004. Available from World Wide Web: `http://www.eclipse.org/modeling/emf/docs/overviews/XMLSchemaToEcoreMapping.pdf` [cited April 20th, 2009].