# EMF Ecore Based Meta Model Evolution and Model Co-Evolution

Moritz Eysholdt
itemis AG
Schauenburgerstr. 116, 24118 Kiel
moritz.eysholdt@itemis.de

Sören Frey and Wilhelm Hasselbring
Software Engineering Group
University of Kiel, 24118 Kiel
{sfr|wha}@informatik.uni-kiel.de

## Abstract

The description of data used for e.g. persisting or transmitting information should be defined in a structured way. The structure itself can therefore be seen as a meta model specifying the data model. Over time a software system evolves and the inherent meta models tend to be unstable. Nevertheless, older formats often have to be supported during transition periods. Previous data models, as being instances of the old meta models, inevitably need to get converted to be valid against the new meta model versions. In this paper we present our approach of restructuring EMF Ecore based meta models together with co-evolving their instances which incorporates constructing a meta model patch format.

## 1 Introduction

The motivation for meta model evolution and model co-evolution is twofold: Using models to handle structured data and the requirement to continuously adapt and improve an application. It is characteristic for models that their structure definition is a model as well, which makes it the meta model. Therefore, it can be said that the model is an instance of the meta model. Since models depend on their meta models, they need to be co-evolved (migrated) when the meta models evolve.

We take meta models into account, which themselves are instances of the popular EMF[1] Ecore, acting as a meta meta model. The changes between different versions of a model are described by our *Epatch* format. An Epatch is derived from two available versions of a meta model. Applying an Epatch with our Patcher tool allows automatically upgrading an old version of a meta model to a new version and vice versa.

The *Metapatch* format is based on the Epatch format and represents a solution to co-evolve models towards their modified EMF Ecore based meta models. In contrast to regular model-to-model transformations, the Metapatch format's complexity is proportional to the number of changes in the meta model, not the overall size of the meta model. For simple modifications the co-evolution of the corresponding models can be performed automatically. More complex changes require the meta-model engineer to define transformation instructions. Model-driven software development [1] is an applicable field for meta model evolution and furthermore an approach which is applied itself to develop the Epatch and the Metapatch tools [2].

The remainder of the paper is structured as follows: Section 2 describes the process of how to create model migration algorithms and how to execute them. The Epatch and Metapatch formats are presented in Section 3, before Section 4 concludes.

## 2 The Process

Figure 1 illustrates the Meta Model Evolution and Model Co-Evolution process. It is an extended version of the process suggested by [3]. On the left hand side the process of the meta-model engineer can be seen, who edits the meta model and creates a model migration algorithm. Further on, [3] defines changes to the meta model which preserve the models as valid instances as *non-breaking* changes. Changes leading to invalid instances which cannot be automatically resolved are called *breaking* changes, otherwise *resolvable*. The meta model engineer will have to specify the migration algorithm's behavior manually for breaking changes.

On the right hand side of Figure 1 the process of the model engineer is illustrated. This 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 to-be-migrated model has to be detected. 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.

[1] http://www.eclipse.org/modeling/emf/

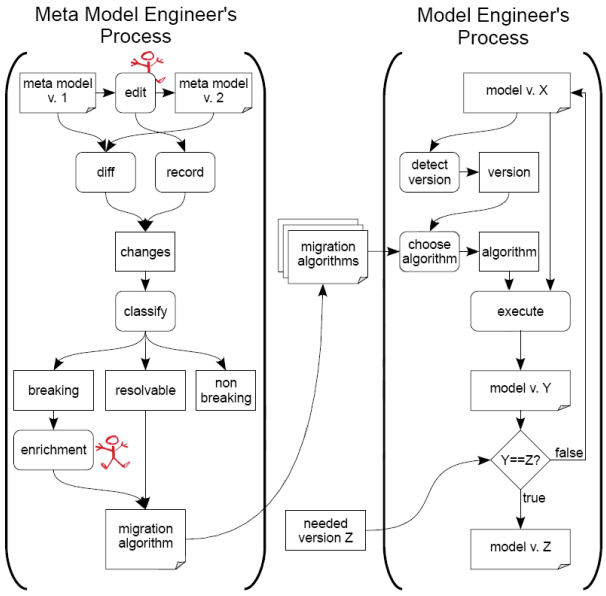


Figure 1: The Meta Model Evolution and Model Co-Evolution process

## 3 The Epatch and Metapatch Format

Both formats constitute textual Domain Specific Languages (DSLs) [4], each one defined by its own Xtext[2] grammar. They therefore provide a convenient IDE, including e.g. code completion. Epatch is declarative, self-contained, meta model agnostic, and not tied to scenarios of meta model evolution. By being meta model agnostic, the Epatch format does not require the to-be-patched models to be instances of a certain meta model. The differences between models are extracted by accomplishing a comparison. This comparison is build on top of EMF Compare[3], whose own *DiffModel* has the disadvantage of having hard references to both compared models and thereby cannot be applied to just one model, since it requires both to be available when loading the DiffModel. For the dimension of a meta model, Epatch is able to specify changes resulting from constructing, refactoring, or destructing meta model elements. The Patcher tool implemented in the context of our work 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.

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[4] to customize the model migration algorithm and it restricts the Epatch to meta models (EMF Ecore models). Second, the mapping of (meta) model elements, that is created when an Epatch is applied, is an essential input for the migration algorithm. The implementation of the migration algorithm is the *MetapatchMigrater*, which is an interpreter for Metapatches. With the mapping of meta model elements as input, it 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 migration of the corresponding model elements. This concept allows the MetapatchMigrater to automatically migrate the parts of a model that conform to meta model 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.

[2]http://www.xtext.org/
[3]http://wiki.eclipse.org/index.php/EMF_Compare/
[4]http://www.openarchitectureware.org/

## 4 Conclusion

We presented our approach for managing changes between different versions of EMF Ecore based meta models and co-evolving corresponding model instances. Therefore, the Epatch and Metapatch formats were introduced which are capable of describing the modifications and routines supporting the migration. The Epatch format is now part of EMF Compare. The Metapatch format will be part of the Eclipse Edapt project [5]. Furthermore, the overall process for conducting the migration was described.

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.

## References

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

[2] Moritz Eysholdt. EMF Ecore Based Meta Model Evolution and Model Co-Evolution. Master's thesis, University of Oldenburg, April 2009.

[3] Steffen Becker, Thomas Goldschmidt, et al. 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.

[4] M. Mernik, J. Heering, and A. M. Sloane. When And How To Develop Domain-Specific Languages. *ACM Computing Surveys, Vol. 37, No. 4*, pages 316–344, 2005.

[5] M. Herrmannsdörfer and M. Eysholdt. Eclipse Edapt - Project Proposal. `http://www.eclipse.org/proposals/edapt/`.