Programming against Multi-Version Metamodels: A Model Differencing and Virtualization Approach

Robert Bill and Manuel Wimmer

Business Informatics Group
Vienna University of Technology
{lastname}@big.tuwien.ac.at

Abstract. Current model manipulation programs more and more have to cope with multiple metamodel versions. This includes tool integration scenarios and language evolution scenarios where newer metamodel versions are available while legacy metamodels are still being used. However, in current metamodeling frameworks such as the Eclipse Modeling Framework (EMF), specific model manipulation programs are needed for each version leading to code duplication, and thus, to high development and maintenance efforts.
By using virtualization techniques, interfaces for manipulating a set of models instead of a single model on the Java level can be automatically generated. This allows to avoid code duplication by introducing only a small syntactic overhead. By using Java annotations for defining the virtualization strategy, we can achieve virtual models being seamlessly integrated into Java as POJOs. Based on a running example, we demonstrate our architecture. The proposed approach is implemented as an open-source project on top of EMF and has been already successfully applied for model migration scenarios.

1 Introduction

A common strategy when handling a set of different models to perform a particular task is model integration with the aim to homogenize heterogeneous models. For example, each model may be virtually transformed to the a unified representation (a.k.a. global-as-view) or may be derived from a unified representation (a.k.a. local-as-view) [5]. Other options include linking, merging or migrating different models [1, 4] and then migrating the instance data [3].

However, migration to a unified representation which is used instead of the different models might not be appropriate if each model should stay autonomous. For instance, it may be beneficial to use specific models for model manipulations like validation or transformation to ensure that all edit operations performed on a model are correct and lead to a well-defined output model. Furthermore, different models may exist on purpose due to slightly different domains or viewpoints on one system. In that case, working with a fully unified model may not only challenge the modelers, but also makes information exchange with specific tools more complicated since these tools then would need to support a more complex model than even needed for their domain. This is, for example, the case for business documents [8], where core model components evolve and are adjusted dependent on the domain users needs.

In the following sections, we present a solution for Java-based model manipulation programming which combines the advantages of (i) using a unified model to reduce code duplication in model manipulation programs and (ii) of using separate models to process only relevant parts of the model by building virtual models handling a set of persisted models. Section 2 introduces the running example, where different model variants are used for school management domain. Section 3 discusses the general architecture of this approach. Section 4 shows a more detailed look on how the virtualized classes are constructed and used. Finally, Section 4 concludes the paper with an outlook to future work.

Due to space constraints, we do not show concrete code examples of how to use the tool VirtAPT (Virtualization with APT) for model migration scenarios and discussing the code duplication reduction, but kindly refer the interested reader to `http://cosimo.big.tuwien.ac.at/virtapt`, where examples for handling model variants and code migration are discussed.

2 Running Example

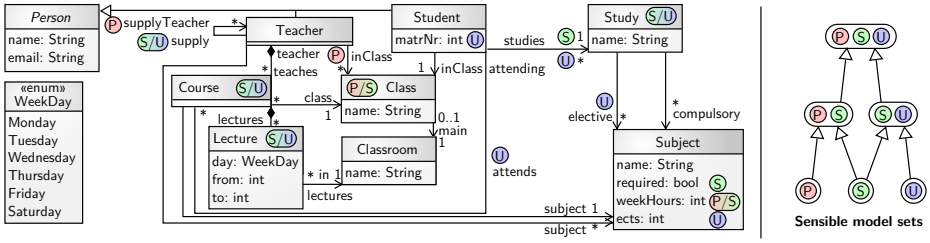

Fig. 1: Running example metamodel variants

The running example for the following sections is a simplified education management system done for three types of schools: primary schools, secondary schools and universities. Due to the slightly different nature of these school types, three different model variants were used as shown in Fig. 1. Classes and attributes are marked with the symbols (P), (S) and (U) for *primary school*, *secondary school*, and *university*, respectively. Note that corresponding class and attribute names are mostly equal on purpose for this example as EMFCompare, in its default setting, as many other model comparison frameworks, strongly relies on string similarities to find correspondences. The attribute specifying supply teachers, which is different for primary schools than for other school types, is an exception just to show that the rest of the approach can handle heterogeneous attribute names in principle.

3 Architecture

Fig. 2 shows the architecture of our approach. Java annotations like `@EcoreVM(name=<pkgname>,ecoreFiles={<model1>,<model2>,...})` can be attached to a method or a class specifying that the models stored in the Ecore-based metamodels should be handled together and all classes generated should be made available in the package

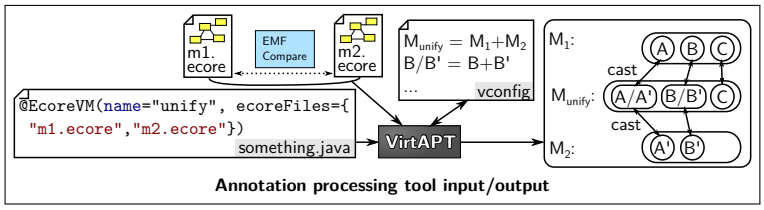

Fig. 2: Architecture of VirtAPT

with the specified name. The annotation processor reads in the specified Ecore-based metamodels and uses EMFCompare to find matching classes, attributes and references. Features which match in all pairwise comparisons are mapped to the same feature in the virtual class. Due to the nature of the annotation processor, only the annotations of a compiled Java file are available. A file named `vconfig` stores which model sets have been considered. This file is read and updated in each annotation processing step. Each line contains `<pkgname>:<foldername>` to specify which packages are available and in which folder they are found.

For each model set, a virtual model is generated which consists of the following files in each package: (a) an interface package containing interfaces for the signatures for all virtual classes and for each model in the model set, a package containing model-specific implementations thereof and (b) a manager class allowing to load instances from any model of the model set and (c) a `config` file containing information about the the model sets used to generate the virtual model. All implementing classes implement their behavior by proxying to the original EObject loaded from the model. Thus, no explicit synchronization is necessary to propagate changes from an object for a model set to an object for another model set based on the same original object. Virtual models are arranged in a model lattice where the topmost virtual model is generated from the largest subset of models. There are explicit cast operators to cast from one virtual model to another if the model sets for both virtual models are in a subset or superset relation. These operators return `null` if no downcasting is possible, i.e., the model is not in the specified model subset. Special downcast operators can be called to cast to a single model even if it is not defined as virtual model.

The `config` file for `SecUni` contains the file names of the merged models `second.ecore` and `univ.ecore` and lines such as `:SecUni.mvirtual1.Classroom:model/second.ecore.//Classroom` to specify that the `Classroom` class from the first virtual model originated from the `Classroom` EClass from the secondary school model.

In the following sections, we will see how virtual models are generated for these model sets in more detail.

4 The Virtual Model Manager

For each virtual model, a dedicated manager is used for loading and storing models. Fig. 4 shows how the manager uses metamodel-specific factories for loading instances. Initially, the manager selects the correct factory based on the metamodel of the model to be loaded. Each factory knows how to construct the correct proxy for a single EObject. Proxy objects resolve references on demand by letting the factory generate or retrieve proxy objects when an attribute is accessed.

The manager is able to create a *CloneState* for any factory it uses. The CloneState is able to recreate objects using a factory to create a skeleton and then using an in-

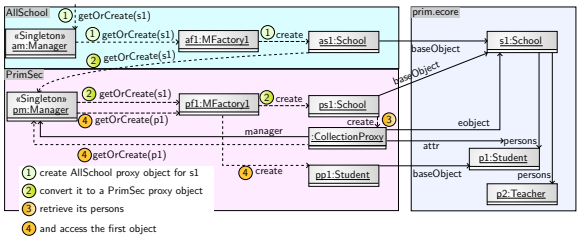

Fig. 3: Proxy object generation example

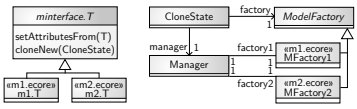

Fig. 4: Virtual model managing

place clone method generated for each class on the interface level which sets all setable attributes values to the values of the attribute for the cloned object.

Let us consider the example of reading in a primary `School` EObject as virtual instance of `AnySchool.School`, then specializing it into `PrimSec.School` and then accessing the first object returned by the `getPersons()`-method on that as shown in Fig. 3. The manager controls the generation of proxy objects. When reading in the `s1` object, it recognizes that this is an object from `prim.ecore` and thus the first factory is needed for object generation. Thus, the task of object creation is delegated to the first factory. It first checks whether it already has generated a proxy object for this EObject, but since it has not, it creates a new proxy object which is returned. Then, this object should be specialized to to an instance of `PrimSec.School`. Thus, the singleton instance of the Manager for the virtual model `PrimSec` is accessed and triggered to create the proxy object `ps1` for the base object of the current proxy object, `s1`. Like in the previous case, it delegates the task to the correct factory. Then, the `getPersons()` method is called. The `ps1` object returns a proxy collection which knows that it should proxy the `person` attribute of `s1` for the manager `pm`. When elements of this collection are accessed, the proxy object `pp1` is generated by the proxy collection by delegating to the manager for the proxy collection like in the previous cases. Accessing the next object of the proxy collection would create a proxy object for `p2`.

5 Model Virtualization

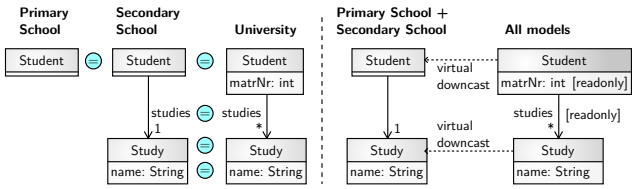

Fig. 5: Virtual classes

For generating a virtual model for a model set, we use a simple model merging algorithm. First, we pairwisely identify matching model elements using EMFCompare and iteratively greedily merge model elements with least differences, where two classes, attributes or references of the same model are never merged. The virtual model then contains all model elements occurring in any model of the model set, where the alphabetically first name is taken in case corresponding elements have different names. All

model elements which do not occur in every model are set to readonly. The type of the getter method of a merged attribute or reference is the least supertype of the corresponding types of each attribute or reference in the virtual model and the multiplicity is the most general one while the type of the setter method is a subtype of all types and none if such a type does not exist and the multiplicity is the most restrictive. Fig. 5 shows the generation virtual classes for *Student* and *Study* for the full model set. EMFCompare detects all the given classes with the same name as corresponding, thus a single class is generated for them with the same name. If the class *Student* would be called *Collegian* in the university model and *Pupil* in primary and secondary school, then it would be called *Pupil* in the virtual model for primary and secondary school, but *Collegian* in the virtual model for all models. The attributes *studies* and *matrNr* are set to readonly since they do not occur in all models. The attribute *name*, however, is not set to readonly since it occurs in every model where the class *Study* occurs.

In some cases, multiplicity conversion has to be performed. If a single features is set whereas the underlying EObject uses a collection, a new collection is created every time this feature is set. If a collection is required by the get method, but the underlying EObject uses a single value, a new proxy collection is generated for each access performing operations such as add and remove on the EObject. A proxy collection is also used for getting multi-valued features to proxy access to the EObject allowing to modify the resulting list naturally. Unfortunately, this induces an asymmetry for get and set. While changes made on a list got with the get method are reflected by the EObject, the modification of a list after calling the set method has no effect on the EObject.

6 Conclusion and Future Work

This work presented an approach to handle a family of models using model virtualization. The prototype allows Java developers to specify and use the type of virtual models they would like to while completely staying in their Java development environment. Thus, this approach may serve several needs. First, it might even further lower the entry barrier of using model-based techniques for Java developers since the inclusion of models into Java does not even take a single button push any more. Second, it might reduce code duplication when handling multiple model variants, and thus, increase code maintainability. Third, it can be used to some extent for automated model migration. The applicability of this approach is shown by an example which is available at: `http://cosimo.big.tuwien.ac.at/virtapt`.

Of course, there is room for future improvements. Advanced model merging and comparison may allow to handle more heterogenous models. To separate the task of model comparison and model virtualization, the input may be refined to use a base model and delta models [9] for the different model variants instead of the model variants themselves.

Model handling might be complicated by different contexts in model definition and use. For example, a model for schools and universities might include information about the grade of each course of the student, while for some tasks only the average grade would be important, e.g., fundings. Thus, a dedicated virtual model for each context should be built. For databases, the importance of contexts when relating elements has

already been recognized [7]. Providing a framework for dealing with model contexts might help building improved virtual models for a given program context as specified by additional Java annotation properties.

While a single integrated model may be insufficient, sophisticated model integration approaches to build each individual virtual model would certainly be beneficial. A lot of work has been done in that area in the past, cf. [2] for a survey, which may be used to improve the construction of the virtual model. Simple integration patterns mentioned by [2, 7] may be integrated in the presented prototype such as attribute-to-entity relationships, abstraction relationships, functional dependencies between model parts and so on.

Also, it might be sensible to be able to manually define or refine the model virtualization, e.g., by letting the user specify the relations between models or even directly letting them formulate the relation between virtually derived and given models.

The support for Ecore is currently not complete. EOperations are not supported and there are some issues with EEnums and custom data types. It is also currently not possible to add custom methods to the generated classes. This could be either done via a dedicated language or by using the usual `@Generated` annotation.

While initial experimental results look promising, no exhaustive evaluation of the presented approach has been conducted. There are some promising candidates for case studies in that area, e.g., the model repository used for Business Document Evolution [8]. Another promising candidate is the evolution of GMF, where both metamodel and model manipulation programs are available [6].

Acknowledgments. This work has been funded by the Vienna Business Agency (Austria) within the COSIMO project (grant number 967327).

References

1. Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009.
2. Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Comput. Surv.*, 18(4):323–364, 1986.
3. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Proc. of EDOC*, pages 222–231, 2008.
4. Cauê Clasen, Frédéric Jouault, and Jordi Cabot. Virtual Composition of EMF Models. In *Proc. of IDM*, 2011.
5. Alon Y Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.
6. Markus Herrmannsdoerfer, Daniel Ratiu, and Guido Wachsmuth. Language Evolution in Practice: The History of GMF. In *Proc. of SLE*, pages 3–22, 2012.
7. Vipul Kashyap and Amit P. Sheth. Semantic and Schematic Similarities Between Database Objects: A Context-Based Approach. *VLDB J.*, 5(4):276–304, 1996.
8. Christian Pichler and Manuel Wimmer. Model-Driven Business Document Evolution. In *Proc. of CSMR*, pages 325–328, 2011.
9. Christoph Seidl, Ina Schaefer, and Uwe Aßmann. DeltaEcore - A Model-Based Delta Language Generation Framework. In *Proc. of Modellierung*, pages 81–96, 2014.