

Coevolution of Models, Metamodels and Transformations

Joachim Höbner, Michael Soden, Hajo Eichler
ikv⁺⁺ technologies ag
{hoessler, soden, eichler}@ikv.de

Abstract

OMG's MOF standard defines a four layer modelling architecture that enables software developers to quickly design their own domain specific modelling language. Using such language definitions (so called metamodels) it is possible to automatically derive model repositories where models conforming to that metamodel can be stored and accessed through standardized interfaces. Although this approach offers great flexibility in choosing an adequate modelling language, once a project has been started it is difficult to change the language itself without invalidating the already designed models. But as models change and evolve in a project, so do metamodels. In this paper we discuss what can be done to transfer models between different versions of a metamodel, thus allowing a simultaneous evolution of models and metamodels. For this purpose, we introduce a unified model for the presentation of models at all meta layers and identify common metamodel refactoring patterns, where model transformations can be derived automatically.

1 Motivation

As models gain importance in the software development process, the decision on the right modelling language becomes more important as well. For many companies and projects, standardized languages such as the *Unified modelling Language* (UML) are well fitted and many tool vendors provide tools supporting that language. But if there is no standard language which offers the artifacts one needs to model a software system as required, one can develop an own modelling language using for example the *Meta Object Facility* (MOF) [6]. MOF is not only standardized by the *Object Management Group* (OMG), but also used by the OMG itself to define visual (e.g. UML) as well as textual languages (e.g. OCL [7].)

In addition to the MOF metamodeling language itself, the MOF standard series [6] defines interfaces and the semantic of databases, referred to as "*repositories*," for storage of metadata described by metamodels. This framework standardizes access interfaces to enable interoperable and consistent usage of models independent of the implementation from various vendors.

But plain storage of models does not suffice, since in a software development process models are not created once and then never changed again, but are in rather steady development. Different versions of the same model are created and must be managed. Versioning of programming artifacts is common practice throughout the software industry and several tools for that purpose exist. Most of these versioning system implementations are file based (CVS, Clearcase, SourceSafe) and, therefore, can be used to store any artifact which have a file representation, e.g. a (java, c++) source code file, a documentation html file, a build file or a distributable binary. These systems can also be used to store MOF compliant models using its file representation (see XMI [4]).

A versioning concept inside MOF repositories is also being developed by the OMG[1]. One advantage of MOF repositories when compared to classical, file based versioning systems is the fact that MOF repositories "know" about the metamodel of the data they store (since repositories are derived from a metamodel). Therefore, only syntactically correct models can be stored, and this syntactic information of stored models can be used for further repository related activities, such as model comparison, model merge, etc.

But software development takes place at "multiple dimensions," as argued in [2], e.g. not only at the model "dimension," but also at the metamodel dimension. Metamodels can contain bugs that must be fixed (e.g. a wrongly typed attribute), lack a concept that needs to be supported or contain redundant concepts. As defined in the MOF specifications, MOF repositories are derived from a fixed MOF metamodel; once the repository is populated with models conforming to that metamodel, each change to the metamodel itself requires the generation of a new repository that then can store new models conforming to that new metamodel. Thus, models are tied

to their metamodels. If one wants to transfer a model to another metamodel, a special transformation has to be developed.

In this paper, we discuss a set of common metamodel changes (or "refactoring" steps,) and present a technique for an easy transfer (or reinterpretation) of corresponding models to the new version of their metamodel. But first, the basic concepts of MOF are introduced in Section 2, followed by a description of an instance concept for the four level modelling world in Section 3. Section 4 discusses model/metamodel coevolutionary transformations.

2 MOF

The *Meta Object Facility* (MOF) has been standardized by the OMG since 1999 and its version 2 is currently in its finalization phase (along with UML 2.) MOF 2 comes in two different flavors, the Essential MOF (EMOF) and the Complete MOF (CMOF). While CMOF offers a richer set of modelling concepts, we focus on EMOF which already includes all basic concepts.

Using the object-oriented paradigm, MOF is organized in a 4-layer meta-data architecture where a model at one layer is described as an instance of a metamodel on the layer directly above (strict metamodeling). On top, the MOF Model is defined recursively by itself, avoiding an endless continuation of meta-layers. The meta-layers - counted from M0 to M4 - provide a terminological and conceptual basis to rank a model's "meta-ness." Most important is the use of M2-level metamodels to define the abstract syntax of languages (i.e. their static semantic) or domains at M1, which is a similar approach to the concept of abstract syntax trees (AST). The EMOF-model consists of the following concepts for the definition of meta-models:

Classes Classes are first-class modelling constructs. Instances of classes have identity, state and behavior. Features of classes are properties and operations. Classes can be organized in a specialization/generalization hierarchy where conform redefinitions of properties are allowed.

Properties A property describes a typed attribute of a class. Since EMOF does not have an explicit association concept, binary association can be defined using "entangled" properties.

Data types Data types are used to specify types whose values have no identity.

Packages The purpose of packages is to organize (modularize, partition and package) metamodels.

The MOF specification defines these concepts as well as supporting concepts in detail. For the visualization of MOF, the UML notation has been deliberately used to visualize selected concepts (see figure 1), as defined in [3].

The advantage of the MOF-approach is that the definition of (meta-) models is made independent of the concrete application domain of the models and that it provides a concise and unique set of concepts for the definition of meta-models. Moreover, it has already been shown how relations between meta-models can be utilized as a basis for model transformations based on these meta-models (cp. Query/View/Transformations process of the OMG [5]).

Even though MOF defines a four layer modelling architecture, the MOF language itself does not define semantics for all of those layers, instead, only for the upper three levels: for the metamodel layer (M2), and, due to its self describing nature, for the meta metamodel layer (M3), and, to some degree, for the model layer (M1). The semantics of objects at the M0 layer must be defined somewhere else (e.g. in the metamodel), using another formalism, if any.

Some of EMOF's concepts do not have any impact on the M1 level, for example the class *Package* or the property *Name* are only used to structure M2 metamodels, but cannot be used as modeling concepts in M1 models. On the other hand, the following EMOF classes define some semantic spanning two layers into the M1 layer:

- EMOF's **Class** is used on a metamodel level (eg M2) to describe "things" that then can be instantiated on the model level as *Objects*.
- Features of objects are described in a metamodel using EMOF's **Property**. At the model level, these properties are instantiated to hold the actual values of a property (eg. using *Slots*.)

As shown, Class and Property can be instantiated twice, across two modelling layers. Furthermore, since all objects in the EMOF model itself are (direct) instances of Class or Property, in the following chapter we present an instance model of EMOF which is centered around these both concepts.

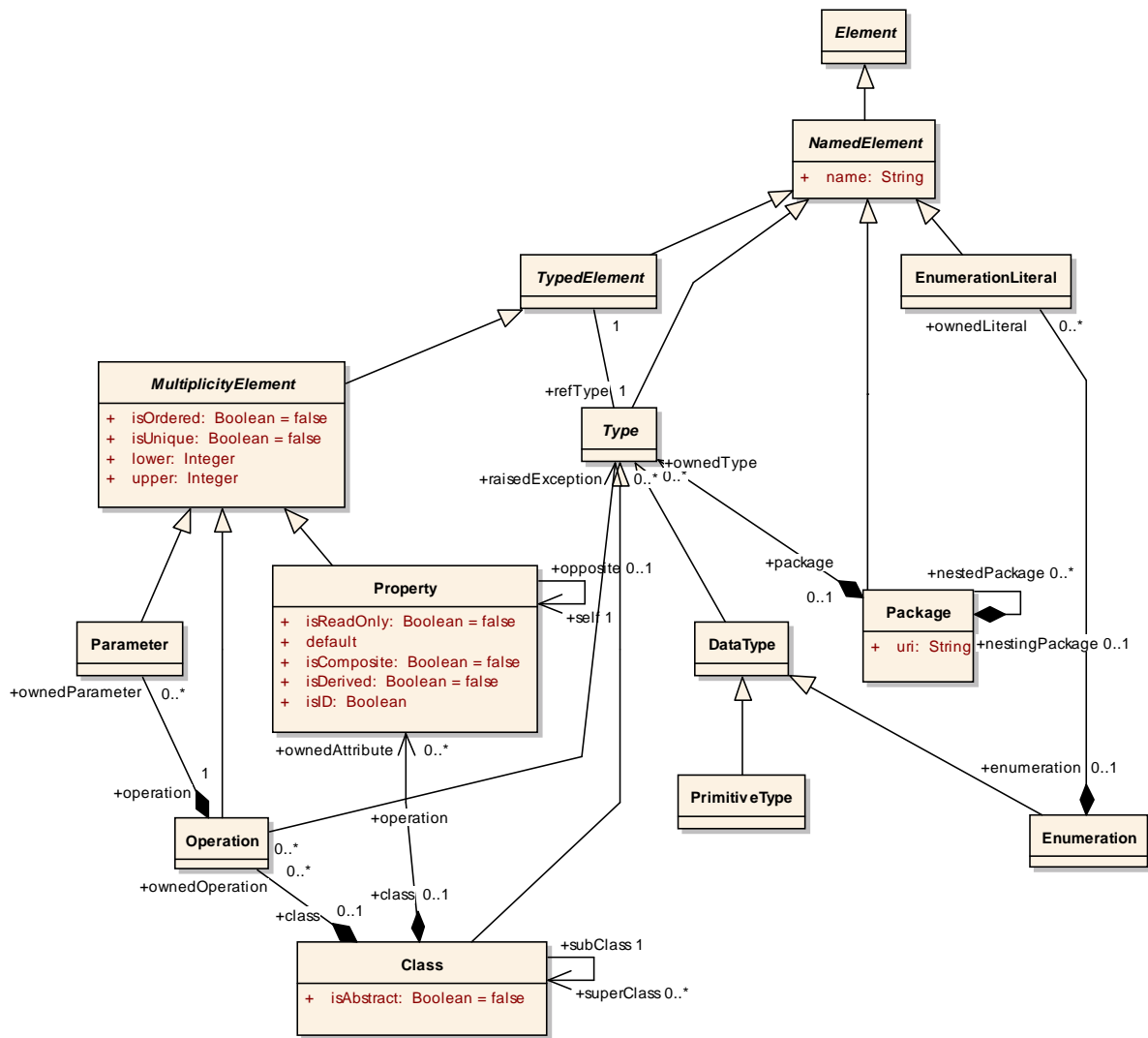


Figure 1: EMOF Metamodel

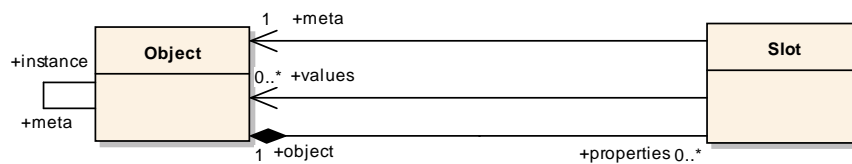


Figure 2: Instance Model

3 Instance Model

Although models, metamodels and the meta metamodel (here: EMOF), are distinct entities that are produced at different stages in a development process (and, probably, by different people), they are connected in a very specific way: the metamodel is an instance of the EMOF metamodel and the models are instances of the metamodel. This "instance of" relation can be specified formally (see CMOF specs), but to do so, we first have to develop a definition of how elements at all layers of the MOF architecture can be represented.

When trying to develop a uniform representation of models at different MOF levels, it is difficult to develop a consistent mapping to other languages such as UML or Java, because they already offer an instance concept, however, this instance concept always has only two layers: the class and its instances. Therefore, we develop an instantiation model using UML which is not aligned at some level of the MOF hierarchy, but is orthogonal to that level concept. This model describes classes from which objects can be instantiated that can "live" at any MOF level and where the object-class-metaobject relation is constructed using links between these objects. The core of this model consists of only two classes (see Figure 2):

- The **Object** represents any model element¹. If it has any features, they are stored in accompanying Slots.
- These **slots** store objects for all values an object can have.

Using this simple object model, one can start to connect models at different meta levels: as described in Section 2, for each object there is always a meta object describing its structure. Therefore, objects can be connected via an "instanceOf" relation, as well as slots since they are also described via meta properties.

Note that in this simple architecture, there is no explicit concept of a "model." A "model" is rather a set of objects and slots. Although this simplified approach suffices in the context of this paper, in other contexts an explicit formalization in the model may be desirable, for example another class "Model" that contains objects and an additional constraint that restricts all object in a model to be instances of the same metamodel.

Using this instance model, we can now develop a representation of EMOF. Since EMOF is its own metamodel, all meta references refer back to the model itself. Figure 3 demonstrates how some parts of the EMOF model can be represented:

- EMOF's class "Class" is represented as instance of object. It is its own meta object.
- EMOF's class "Property" is also an instance of object. Its meta object is "Class."
- An instance of "Property" is the object "Name" which represents the ability to give classes names.
- "Class" has a set of slots, e.g. "Name", which is itself an instantiation of the Property "Name". This Property now can be used to specify the name of the class (here: "Class").

This instance model together with the EMOF model representation can be used to construct any EMOF compliant metamodel (as instances of EMOF) together with its instances, and at the same to describe the meta relationship of these models across the three upper layers of the MOF architecture.

¹There is no special entity for primitive objects such as String, Integer, etc. To keep the instance model as simple as possible, even primitive objects are considered "normal" objects with object identity, etc.

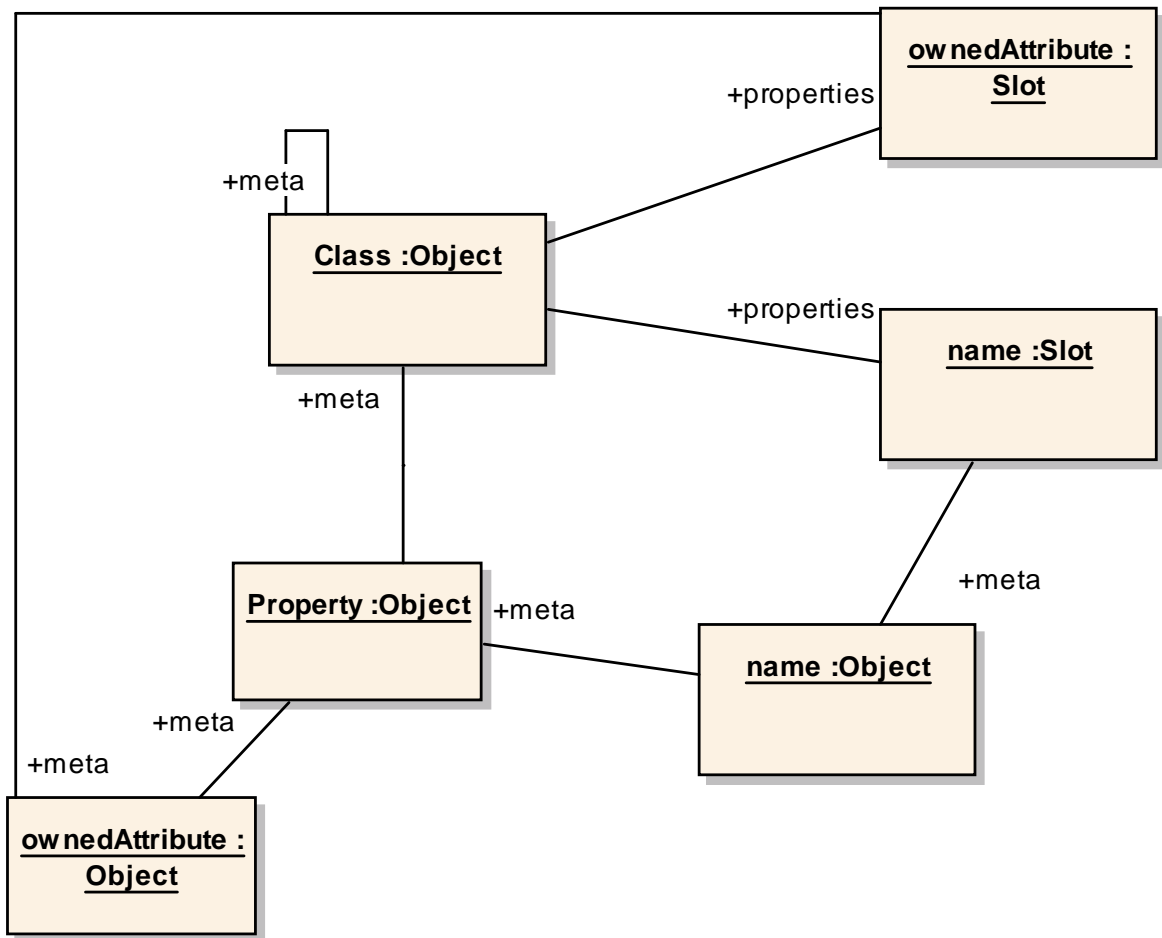


Figure 3: Small Part of EMOF Model Representation

3.1 Constraints

To assure that objects linked by the "instanceOf" relation really have the wanted object - meta object relation some constraints are required. The class diagram introduced with the instance model does not suffice. In order to "implement" an instance behavior as specified in EMOF, the instance model is refined through the addition of a set of constraints. In this paper, the *Object Constraint Language* (OCL) [7] is used, a formal language for the description of expressions on UML models. It enables to state invariants: expressions that always must evaluate to true.

As these constraints formalize semantics derived from the EMOF specification, they need to reference objects in the EMOF model representation developed above. This EMOF representation, therefore, must be present in all instances of our instance model.

For the class `Object`, the following two constraints are required:

```
context Object
-- C01
-- only EMOF's class can be instantiated across two meta layers
inv C01: self.meta.meta = EMOF.Class

-- C02
-- for each attributes of the meta class and its superclasses
-- there is one corresponding slot in the instance
inv C02:
let
  metaAttributes = self.meta.collectSupertypes->collect
    ( properties->select ( prop.meta =
                        EMOF.OwnedAttribute).values)
in
  metaAttributes->forAll
    ( attr | self.properties.exists(s|s.meta = attr))
  and
  metaAttributes->size() = properties.size()
```

For the class `Slot`, we define these constraints:

```
context Slot

-- C03
-- the meta object must be an instance of EMOF.Property
inv C03:
meta.meta = EMOF.Property

-- C04
-- type of values must correspond to their specification
-- in the meta property
inv C04:
let
  type = meta.properties.select(meta = EMOF.refType).values
in
  type.notEmpty() implies values->forAll( meta = type)

-- C05
-- Cardinality of values must correspond to specification in meta
-- property
inv C05: let
  lower = meta.properties.select(meta = EMOF.lower).values
  upper = meta.properties.select(meta = EMOF.upper).values
```

```

in
    values->size() > lower and values->size() < upper

-- C06
-- if isOrdered or isUnique is set in meta property, this
-- must be enforced in slot
inv C06:
let
    isUnique = meta.properties.select(meta = EMOF.isUnique)
in
    isUnique implies values.isUnique()

-- helper operation to collect all supertypes
def collectSupertypes:Set(Object) = let
    supertypes = self.properties.select(meta=EMOF.SuperClass)
in
    supertypes.collect(collectSupertypes) .
    union(supertypes).union(self)

```

Using this combination of our orthogonal instance model, its constraints and the EMOF model represented as instance of the instance model, we are now able to develop metamodels and models that are correct regarding their metamodels. If no constraint is violated, a model **conforms to** its metamodel. According to satisfaction in logic, we write $M \models MM$ if a model M conforms to a metamodel MM .

3.2 Example

A rather extensive example on how models at different modelling layers are represented and connected using this instance model can be found in Figure 4. On the left side, you find a conventional UML style depiction of a small part of the EMOF meta metamodel at M3 level, a simple state machine metamodel at M2 level, and, at the M1 level, a simple model of the possible states of a door. On the right part of the diagram, these models are represented in an UML object diagram as instance of our instance model. Please note that, although, all objects have names, this naming concept is not accurately modeled here.

3.3 Versioning

As stated above, models and metamodels evolve in a development process, and so do metamodels. Having developed a uniform representation for models at different metalevels, we can extend this representation to include support for versioning. For the purpose of this paper, it suffices to add one association to the instance model introduced above. This association (as depicted in Figure 5) connects every object with its predecessor in an earlier version of its model, and with its successor in the next version of its model. Of course, in a "real world" versioning system, objects must be able to have more than one successor and predecessor to allow simultaneous work on the same model in different branches. But this limitation has been added here for simplification and since such scenarios are out of the scope of this work.

Using this versioning concept, we can connect different versions of the same model (or metamodel) and follow along the evolution of single elements. Note that, although, we now can version models and metamodels, the M3 layer EMOF model must not be changed and versioned since it has to be fixed to be able to serve as "fixed point" in our instance concept; all constraint expressions make use of the EMOF model.

4 Coevolution through Transformations

Along with the evolution of individual domain models and their well-defined meta-data, there is an increasing need for highly automated and provenly correct model transformations. Thereby, transformations provide the basis to express and define changes and relations of models in a formal way.

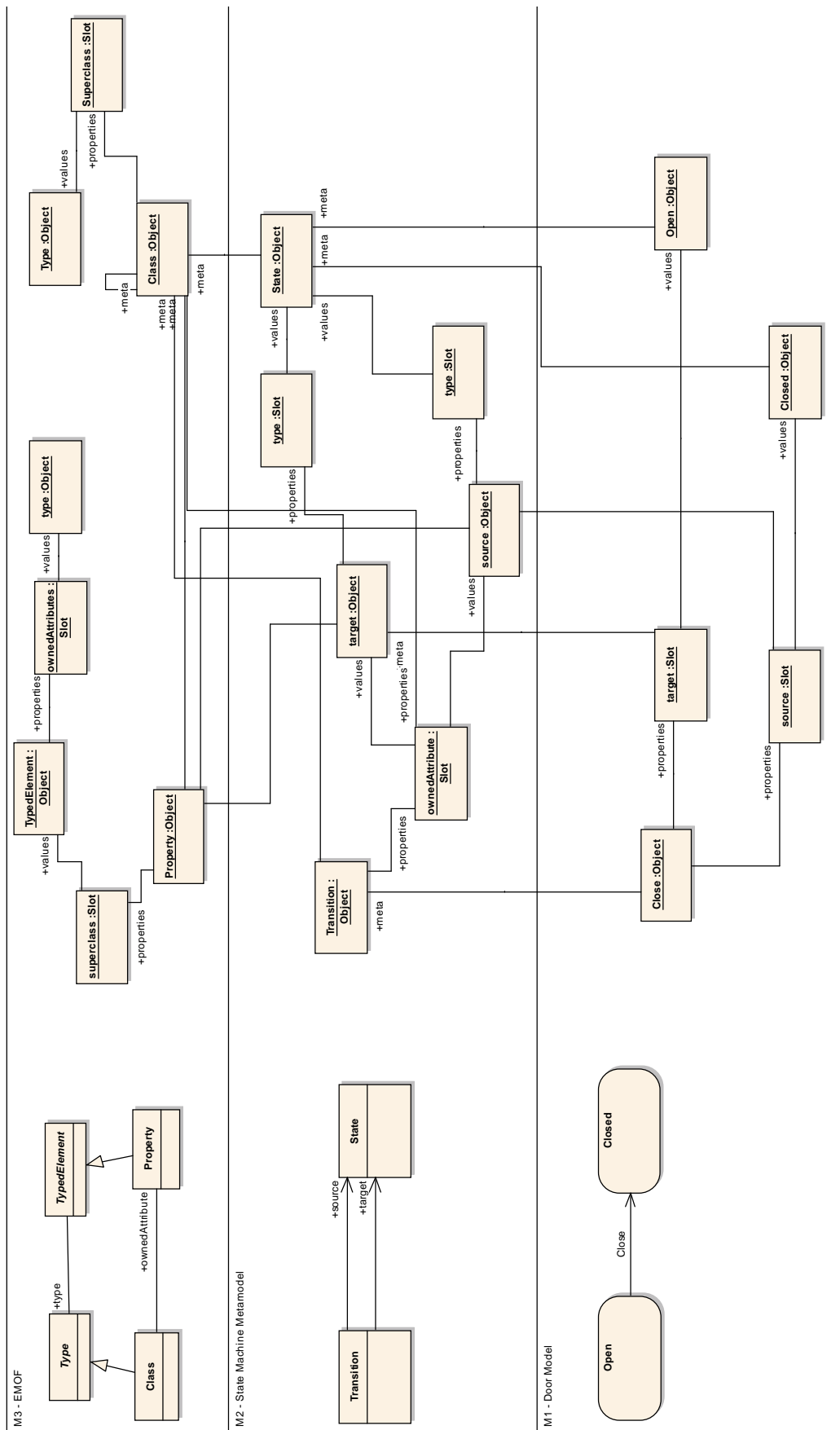


Figure 4: Example: Models at Different modelling Levels

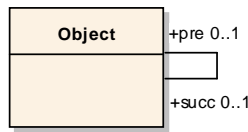


Figure 5: Instance Model with Versioning

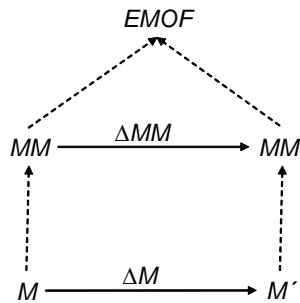


Figure 6: Transformation Model

While most application cases of transformations include the integration of models, serving as views on a system — at possibly different levels of abstraction — into a globally consistent view, we use transformations in the following sections to define the transitions between two versions of a model and its corresponding metamodel as depicted in Figure 6. The focus lies on the automatic “alignment” of a model M' (result of a transformation of its predecessor model M) to a new version of the metamodel MM (which is MM'). For this purpose, we identify common evolutionary (meta-)model changes as transformation patterns or *metamodel refactoring patterns*. In this section, we show for some specific, but common, metamodel changes that this transition to a new model M' conforming to a new metamodel MM' can be derived automatically.

4.1 Transformation Patterns

Since model transformations are not new to the area of computer science, several languages and specification techniques exist for the definition of transformations, varying in their conceptual approach, syntax, application domain and precision [8]. In general, the approaches are built on graph manipulation and are either declarative or imperative. For MOF, the Query/View/Transformation (QVT) RFP process has produced a specification that is currently in an ongoing standardization process [5]. QVT allows the specification of transformations based on MOF 2.0 compliant metamodels, thereby providing different approaches that combine declarative transformation specifications (i.e. element relationships) as well as operational mapping definitions (executable rules).

Since the QVT standard is not finished yet, and because we have an own instance model for the representation and transition of model versions, we do not use the QVT or any other language to define our patterns, since it would add an unnecessary obstacle in understanding the intended computational semantic. Note that metamodel refactoring, which is the topic of this paper, is a special case of a general model transformation, since source and target (meta-)models are closely related. Therefore, we provide a textual and visual description for this *construction* process of M' in conjunction with expressions for all values of objects and slots in the model. Afterwards, we will argue² why all constraints of our instance concept will hold for this newly created model and, thus, this new model conforms to the changed metamodel.

4.2 Metamodel Isomorphism

A fundamental question for metamodeling is: are two metamodels equivalent? On the contrary, we can distinguish whether a change to a metamodel has an impact on the metamodel’s instances or not. For example, one

²since the instance concept is not a formal calculus, we cannot provide mathematical proofs of the correctness for the constraints.

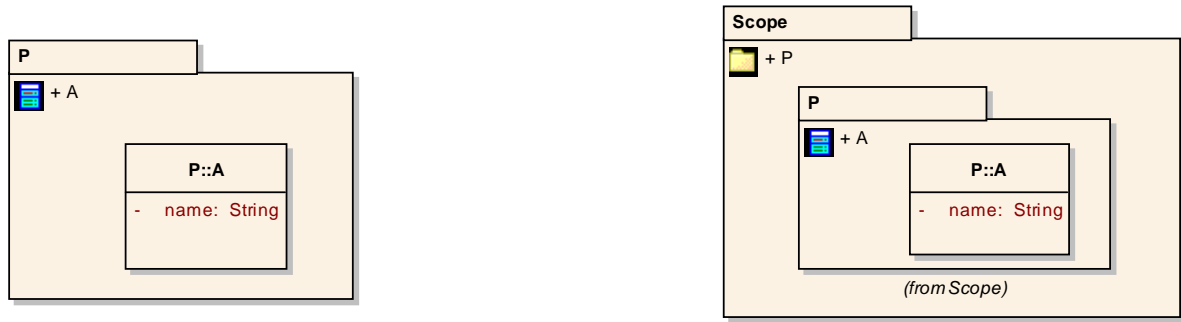


Figure 7: Example: Metamodel isomorphism

of the simplest metamodel transformations is the renaming of a meta-property. Assume a (meta-)class with the name "A". In terms of our instance model this is described by an object on the M2-layer having an `EMOF.name` property's slot set to the string "A". Intuitively, it should be clear that a change of the name, say to "B", has no effect on the instances, because the value of the name property is irrelevant to M1-layer objects of that class. Since the models, as well as the metamodels, before and after the change remain structurally equivalent with respect to the models they describe, we can answer our question to metamodel equivalence and denote this kind of equivalence as *isomorphic metamodels*.

Definition: *Metamodel Isomorphism*

Two metamodels MM , MM' are denoted as being *isomorphic*, iff the set of all instances is equivalent.

More precisely, metamodel isomorphism means that every imaginable instance that conforms to MM must also conform to MM' and vice versa. This structural equivalence of the M1-level instances is required for all EMOF concepts used in the definition of MM and MM' , taking into account isomorphic equivalence of the object graphs and values of properties.

Apart of the simple renaming example above, an isomorphism that is unlikely to be intuitive may be the addition of another package as scope (see Figure 7). As crucial as the restructuring of metamodel P may appear, it does not have an effect on its' instances, since packages are only an M2 level concept without any impact on M1 level models. To verify this, we can check the instance model's constraints $C01$ to $C06$ and their assertions on the objects and slots. We can observe that there is no constraint affected when regarding a model $M \models MM$ as instance of MM' , i.e., $M \models MM'$.

From an implementation point of view, contrary arguments exist whether the scope of a metaclass *does* matter, since namespace information is often used for managing the elements using a factory or similar. For example, existing technology mappings, such as the MOF-to-IDL mapping, take into account the package structure as scope of elements, which is reflected by IDL `modules`, derived according to package nesting in the metamodel. However, from the design of the concepts, metaclasses are objects that obtain their identity not by their name but through their properties and relations to other objects. That means, that if two classes have the same settings of owned attributes (up to an infinite depth of nested complex data-types or relations), they are virtually equal³.

Moreover, it is worth discussing which modifications to meta-properties have an effect on the models. For example, changing the value of the `EMOF.upper multiplicity` property changes the number of slots that are reserved for the property's values and, therefore, the instances structure. This impact of loosing the isomorphic property applies generally to modifications of the EMOF meta-properties `type`, `superClass`, `isAbstract` and others. We will not concentrate on a derivation of types of *isomorphism criteria* here. This is a subject for further investigations on metamodel identity.

4.3 Metamodel Extensions

One of the basic evolutionary refinements is the enhancement of available concepts by the addition of a new class or property. Metamodel additions allow the definition of new M1-level objects or values, thus, broadening the

³This approach stands in the long tradition of topography and categorical equivalence of objects that share equal properties. Note that we deliberately abandon the MOF standard.

scope of the modelling domain, i.e., the *modelling space* of a metamodel.

Apart from general metamodel extensions, we can identify more specific relationships between two metamodel versions with respect to the relationship of the models of MM to MM' . For example, if a class is added to a metamodel MM without any relations to existing classes, the resulting metamodel MM' supersedes its predecessor in the sense that all existing models M are per se instances of MM' . Hence, we denote this characteristic of MM' with the term *super-metamodel*.

Definition: *Super-Metamodel, Sub-Metamodel*

MM' is a *Super-Metamodel* of MM (MM is a *Sub-Metamodel* of MM'), iff all instances $M \models MM$ conform to MM' (without any changes), i.e., $M \models MM'$.

In contrast to metamodel isomorphism as defined above, the reverse direction is not required for a metamodel being a super-metamodel. Note that the concept of a super-metamodel relation is comparable to a generalization/specialization relationship of type inheritance. As sub-typing allows a substitution of instances of supertypes due to polymorphism, so do metamodels replace one another: super-metamodels provide all modelling concepts of their sub-metamodels plus additional features, hence allowing for the substitution of the latter.⁴

Non-trivial examples for this kind of relationship include the merging of two distinct metamodel packages into one or adding new classes as subtypes of existing classes. Note that all these changes do not touch the object structure at M1, since, e.g., packages and, therefore, scope of types aim at the M2 layer without any consequences for the objects in a model. This is due to the fact that all meta references remain virtually unchanged. Thus, we can deliberately define the first basic refactoring pattern:

Pattern: *Consistent Extension*

A *consistent extension* of a metamodel MM is an addition of a class or property as follows:

Class extension: Addition of a new class (without any relationships to other classes).

Type extension: Addition of a new subclass of an existing class with arbitrary new properties or a new superclass without properties.

Class enhancement: Addition of a new property in an existing class of primitive type or of classifier-type (without an opposite flag). The property must have an assigned default value or a `lowerBound` set to 0.

The intension of this definition is to easily migrate the instances M of the extended metamodel MM to the new version MM' . Using a copy algorithm, we can construct the successor version instance M' out of M as follows:

- Copy all objects in M that are not affected by the extension preserving all structural properties, i.e., slots and values.
- For all extended classes in MM that receive a new property, copy the instance in M as before but add new slots for added properties. Set the slots' values to either a defined default value or to an empty value.

□

From this simple definition, we can deduct new properties. First to notice is that if we go back to the definition of a super-metamodel, we can observe that in case of Class Extension the properties of a super-metamodel relation is fulfilled. Checking the constraints C01 to C06 (cp. section 3.1) confirms this proposition, since the extension algorithm assures preserving all slots/values consistently while at the same time adding slots for new features. In all three cases, the extensions do not interfere with existing properties and especially C04, C05 and C06 remain valid. We can conclude that $M' \models MM'$ and that MM' is a super-metamodel of MM under Class Extension.

⁴Opposite to inheritance, the terminology is reversed in accordance to mathematical terms.

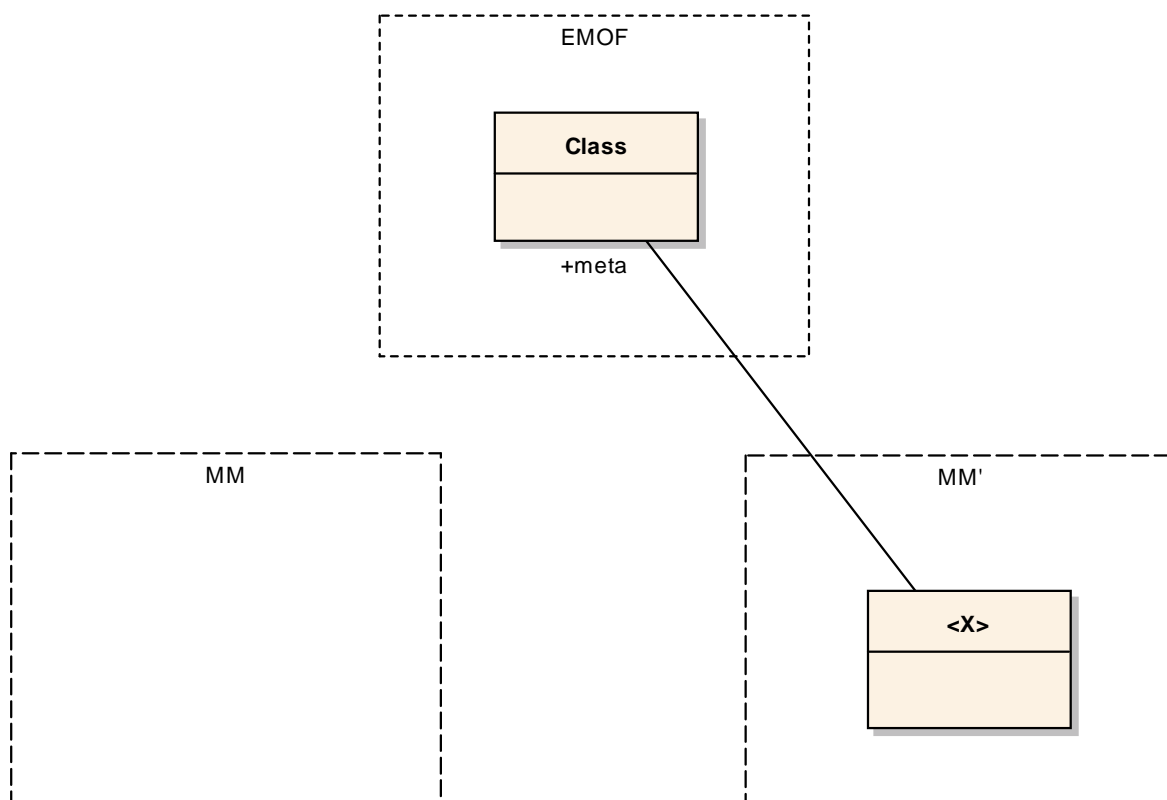


Figure 8: Class Extension Pattern

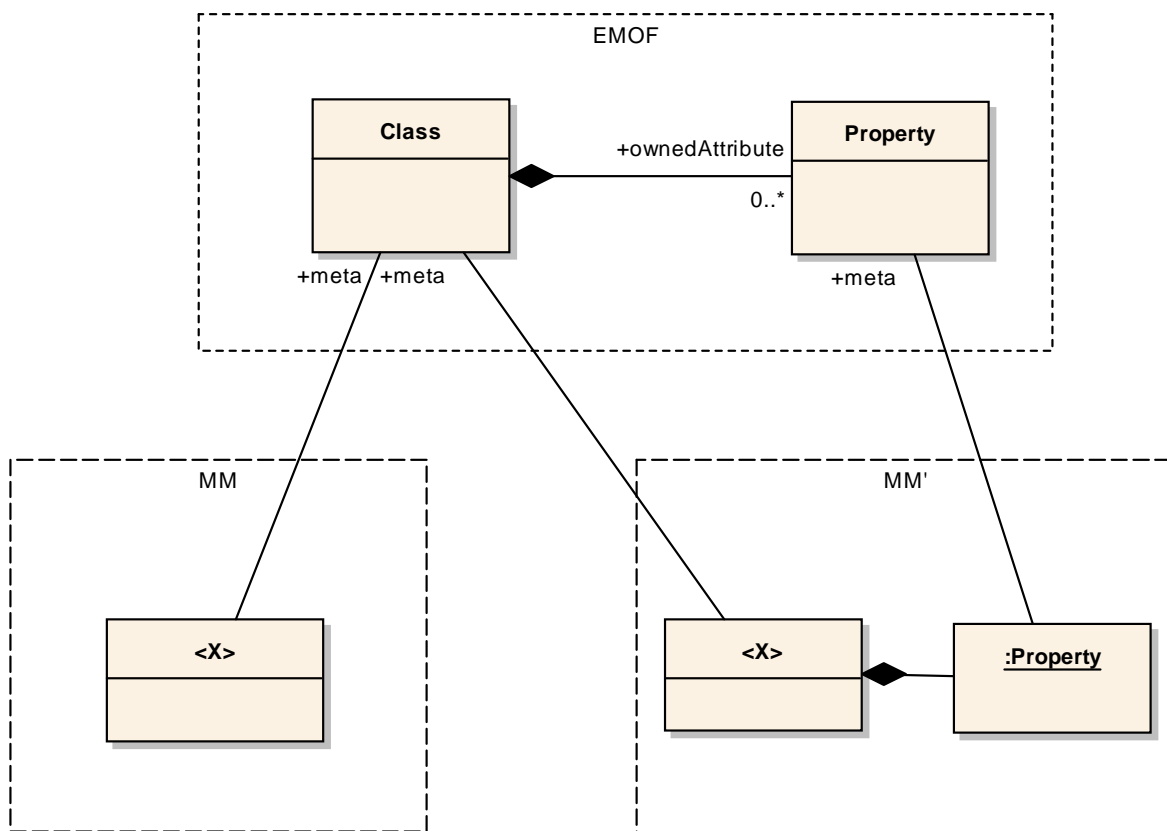


Figure 9: Property Extension Pattern

4.4 Metamodel Projections

In contrast to extensions, the other basic refactoring task is the removal of a meta class or property. Since this transformation reduces the modelling space of a metamodel, we call this refactoring pattern *projection* according to the mathematical term used, e.g., for vector spaces.

Pattern: *Metamodel Projection*

A removal of elements in a metamodel is called *metamodel projection*, if the removed element is either:

- A property of arbitrary type in any class.
- A class having no sub-classes, if no element exist in other classes with a `type` property set to this class.

In the second case the class must be removed together with contained properties. For all instances $M \models MM$, the following algorithm is applied to create a successor version M' :

- Copy all objects in M for which the meta-object has a successor in MM' .
- Apply the same to all slots and values, i.e., to those properties in MM that have a successor in MM' .
- For all objects with no meta-object in MM' , do either:
 1. If the object's meta-object has a supertype, create an instance of this object (using only those slots that are defined by this supertype and further supertypes recursively), "widening" the elements type.
 2. If the object's meta-object has no supertype, or the supertype is abstract, it is removed.

□

Using this algorithm, we only cut out that part of our original model M that remains valid in MM' . All instances where the meta object does not exist in MM' are removed in a transitive manner. As before, we now have to check if all constraints hold on M' . While it is rather easy so see that C01 still holds in M' , a problem may occur in C04: if the value of a slot has been removed because its meta class has been deleted, other elements might be invalidated. But not surprisingly, that cannot happen since we have demanded the classifier to be not referenced as type by any other element in the metamodel. A bit more tricky is constraint C05, since the lower bound of a slot might be violated due to removal of instances of a subtype. At this point it becomes clear for which purpose the "widening" clause must be included into the algorithm, since it prevents a violation in exactly that cases where an instance of a (removed) subtype is included in a collection of a supertype. Thereby, our shallowed object persists as an instance of the supertype in the collection and we keep our model M valid with respect to C05.

Moreover, with a closer look on the projections we can observe that a projection always produces a sub-metamodel MM' out of MM , because MM is consistently reduced to MM' .

4.5 Factoring Patterns

Based on thus far discussed basic element addition and removal patterns, this section will outline more complex patterns that describe typical actions for the refinement of a metamodel. Basically, they constitute a combination of previous patterns and can be regarded as aggregation of sequentially performed steps when manipulating a metamodel.

For example, a common modelling technique in object-orientation is the encapsulation of common properties into a separate superclass. During the evolution of a metamodel it might turn out that a property `name` should be moved into a common superclass `NamedElement`. Even if there are several single steps necessary to achieve this goal (e.g., addition of the new class with specialization, removal of properties, etc.), we can subsume them and construe the whole thing as a new pattern.

Taking this example as a starting point, we find more common changes that alter a metamodel's structure in a consistent way regarding the instances:

Pattern: *Metamodel Factoring*

A refinement of a metamodel MM is called *metamodel factoring*, if one of the following applies:

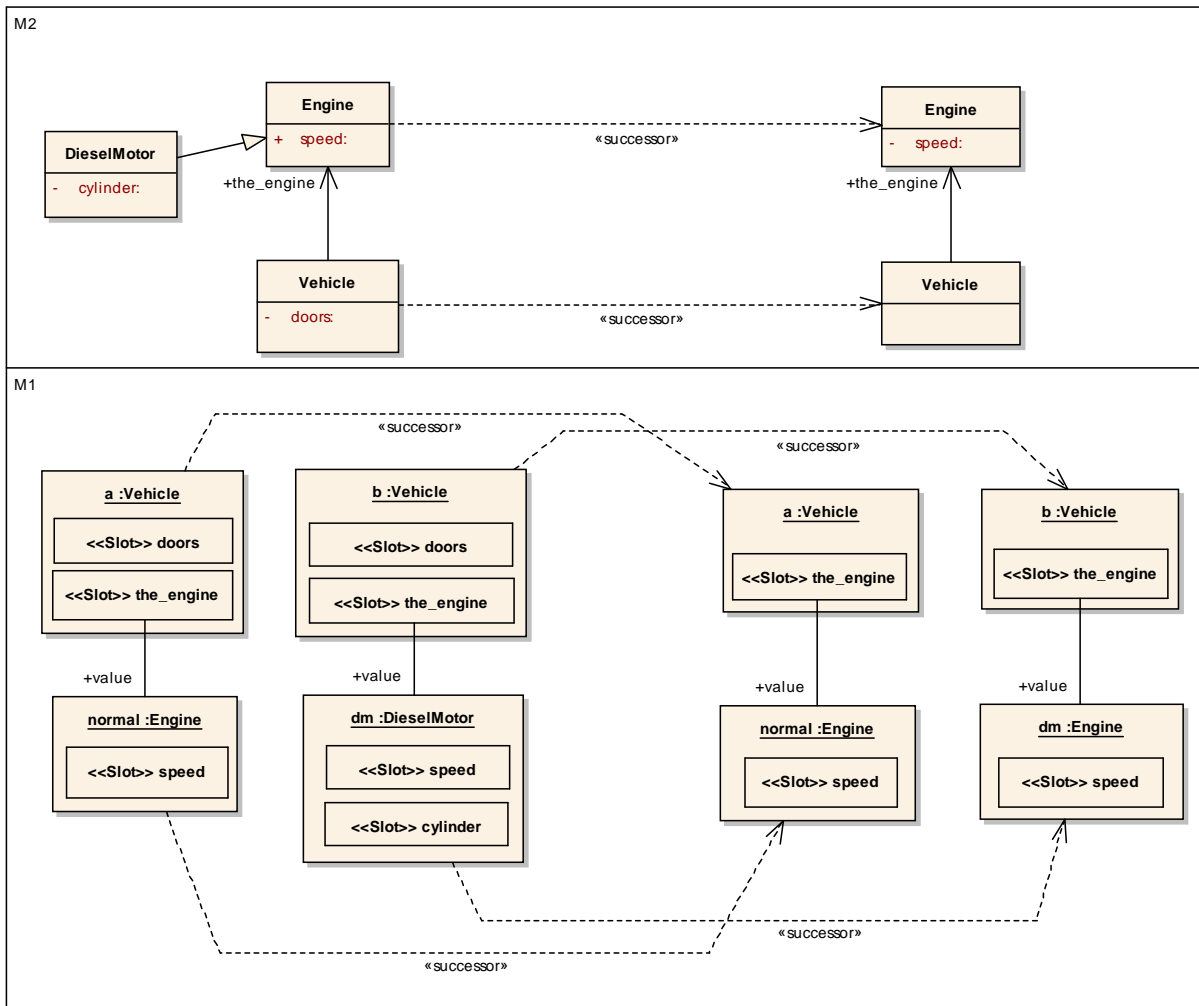


Figure 10: Example of a Projection

Property Movement A property is moved from one class into an either existing or new (direct) superclass or subclass. If the superclass/subclass exist and is not abstract, default values are required as precondition for the moved attribute.

Property Amalgamation Two or more properties with the same name, type and multiplicity are moved into an existing or new (direct) superclass or subclass.

Class Splitting A class is split into one or more classes together with a generalization relationship between them, distributing contained properties arbitrarily over these classes

Class Amalgamation Two or more classes in a generalization relation are merged into one, where in case of abstract classes the contained properties are merged in. If one of the merged classes is not abstract, the amalgamation is only allowed if default values for every merged attribute exist.

Since the structure of the instances is preserved except its slots and meta-object reference, we can define the algorithms to construct the successor version M' as follows:

- In all cases, preserve all not affected objects, slots and values through copying as new elements into M' .
- In case of property movement, copy all instances of those classes where the properties are moved from and set their `meta` reference to the class's direct successor objects. If a property is moved into a subclass, remove its slot and possible value(s) in the instances.
- If properties are amalgamated, simply copy the objects and slots as they are. If the target class is a subclass, remove slots for the attribute in existing instances.
- In case of class splitting, copy the instances one-to-one with all properties except the `meta` link, which is set to the lowermost class of the splitting, i.e., its successor. If the object is referenced as type of another attribute in the model, this link is in turn set to the successor.
- For class amalgamation, do the same for all involved classes and set the `meta` reference and possibly existing references to the resulting (successor) class. For all instances of non abstract classes, create new slots for properties of the amalgamated class and set corresponding default values.

□

In order to point out the core idea of the factoring patterns with respect to our instance model and constraints, Figure 11, 12 and 13 show three non-trivial example transformations which will be discussed below. Thereby, we will outline what happens to the instances and how they are transferred to the new metamodel version.

The common idea of the property amalgamation and movement pattern expresses the need to consistently move, merge or change (similar) properties in the metamodel. Thereby, changes regarding the metamodel are rather drastic in comparison to the effect on their instances' structure, which is only slightly modified. Figure 11 shows an example where the instances remain completely unaffected whereas the metamodel changes through the addition of a new superclass (which obtains the previously duplicated attribute `name`). Note that this is a special case, because generally an attribute is moved up or down the inheritance-hierarchy into an existing class (Property Movement). In these cases, previously created instances need to be extended/reduced properly in order to become valid regarding our instance models' constraints (here: C02).

Similar to the movement of attributes is the splitting of a class into two or more different classes (see Figure 12). In contrast, the objects must not be altered during application of the pattern, since the newly created class generalization yields an identical slot structure for the instances. Furthermore, the example shows what happens to "clients" of a splitted class: after the pattern is applied, all attributes that refer to the splitted class (as type) remain valid (see C04).

While the splitting of a class is (surprisingly) simple, this does not apply to class amalgamation. As a consequence of shifting a property into another, already instantiated, class, these objects must obtain additional slots for the new property. As can be seen in the example depicted in Figure 13, the amalgamation may cause a logical movement of a property (here: attribute `one`) into a firstly non-related, non-abstract class (here: class `Two`). For this reason, our pattern requires all involved properties to have a default value and the update algorithm for the instances is rather difficult (Figure 13).

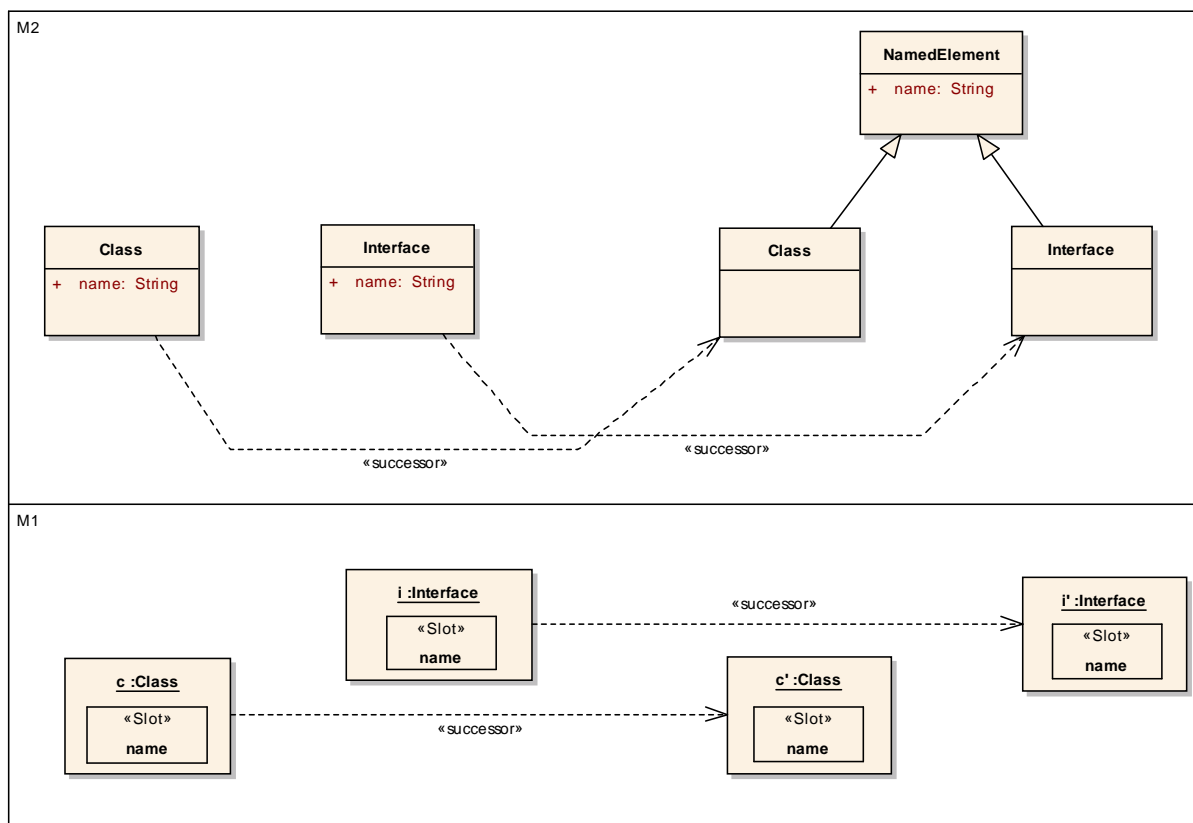


Figure 11: Example of a Property Amalgamation

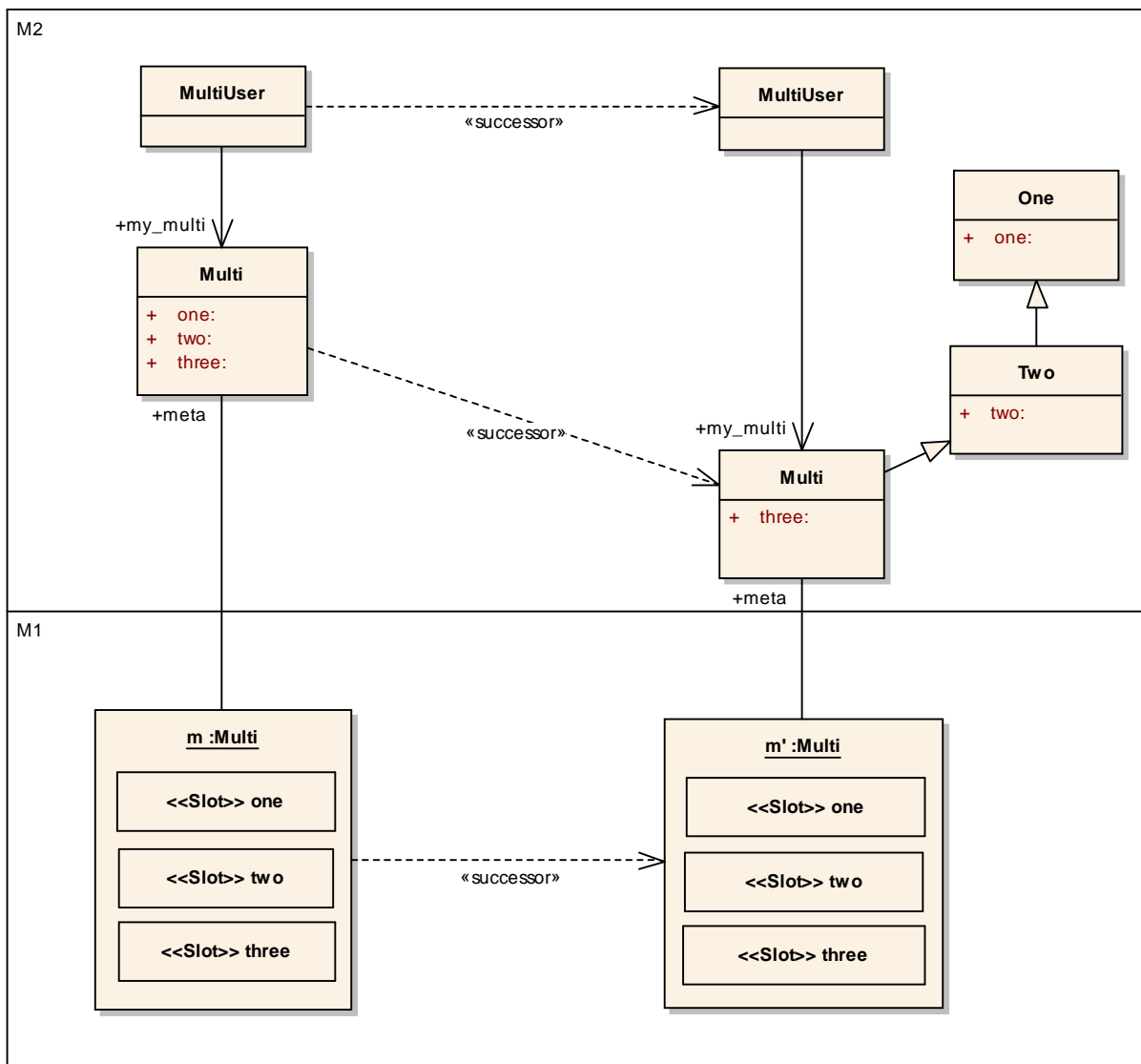


Figure 12: Class Splitting: Example

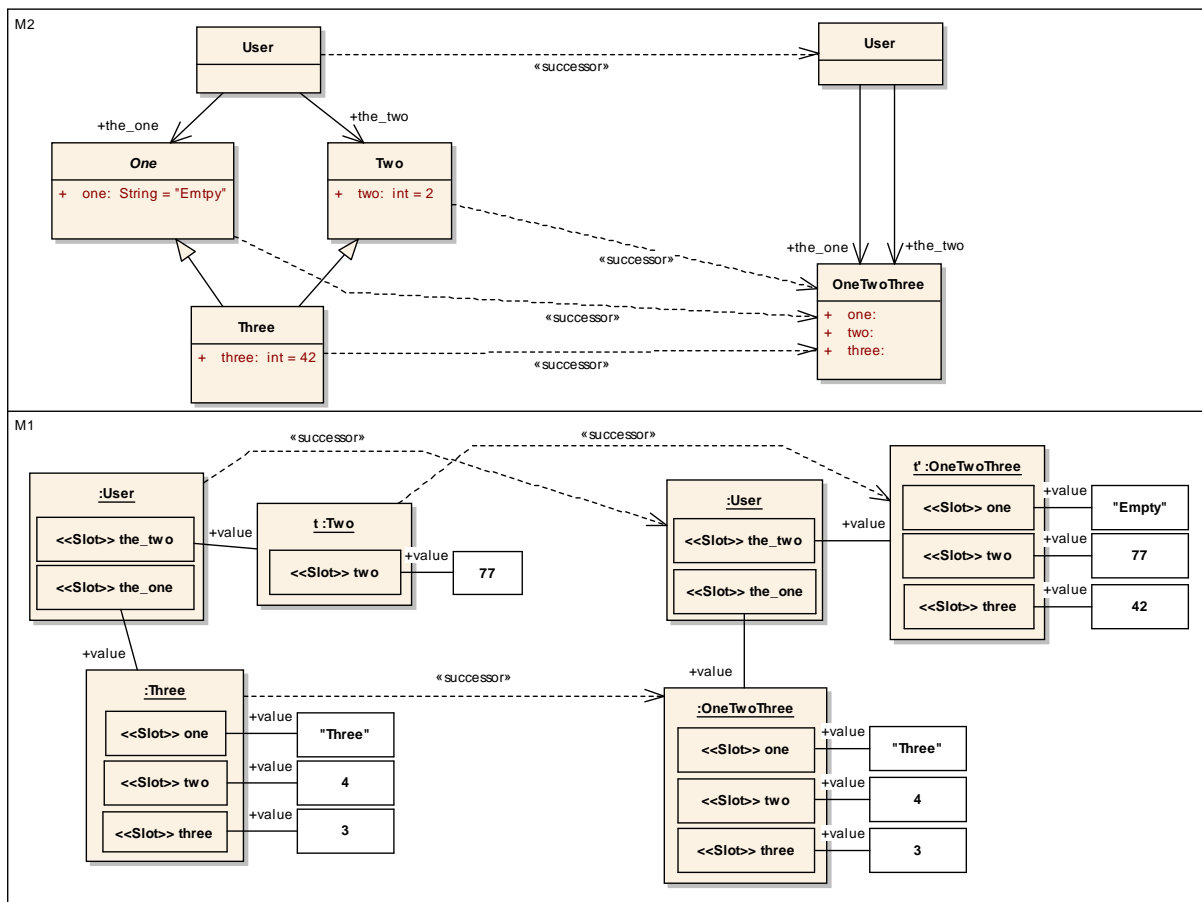


Figure 13: Amalgamation of Three Classes into One

5 Conclusion

In this paper we discussed ways to allow parallel development of models and metamodels. For this purpose we introduced a minimal model — the instance model — to describe models at all different meta-layers. For a set of standard metamodel refactorings, we presented a technique for an easy transition of the corresponding models to new versions of the metamodel, thus freeing modelers from the tedious task of transferring the models manually or developing a transformation each time the metamodel has been changed.

Unfortunately, we could only provide a small set of patterns in the extent of this publication and several topics could not be covered, these include:

Additional patterns. Some of the discussions already suggest to bring up other styles of patterns, such as re-typing of properties or including additional elements such as operations or data-types.

Variations of patterns. In most cases, additional configurations of introduced patterns build reasonable refinements or new patterns. Examples include the restriction of application cases such as types of properties or abstractness of classes.

Combining unrelated metamodels. The mechanism introduced cannot be applied only to metamodels that are originally related, but also to metamodels that have similar concepts and where, a successor — predecessor relationship can be constructed following the refactor step described above.

Metamodel types. Since our instance model makes no statement of the *type of a model*, one of the next steps for extensions is to clearly define what a model type constitutes. Our super/sub-metamodel relation provides a possible solution to define the relationship between model types.

Categorical characteristics. Further interesting questions are the categorical properties of the models and metamodels in respect to category theory. For example, do (meta-)models in conjunction with refactor patterns build a (mathematical) category, where the models are the objects and the refactor patterns form the morphisms? Which are the categorical constructs such as (Co-)Product or (Co-)Limits?

Although we provided a theoretical foundation, an implementation of the discussed concepts has not yet been developed. However, the instantiation model as described in Chapter 3 already implies a basic internal structure.

References

- [1] ADAPTIVE, AND IBM. MOF 2.0 Versioning and Development Lifecycle Specification, Initial Submission, March 2003.
- [2] FAVRE, J.-M. Meta-Model and Model Co-evolution within the 3D Software Space. ELISA 2003, 2003.
- [3] OMG. UML Profile for Meta Object Facility (MOF). <http://www.omg.org/cgi-bin/doc?formal/04-02-06.pdf>.
- [4] OMG. XML Metadata Interchange . <http://www.omg.org/technology/documents/formal/xmi.htm>.
- [5] OMG. MOF2.0 Query/Views/Transformations RFP. <http://www.omg.org/cgi-bin/doc?ad/02-04-10>, April 2002.
- [6] OMG. MOF 2.0 Specification. <http://www.omg.org/cgi-bin/doc?ptc/2003-10-04>, October 2003.
- [7] OMG. UML 2.0 OCL Final Adopted specification. <http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf>, October 2003.
- [8] VARRO, D., AND PATARICZA, A. A unifying semantic framework for multilevel metamodeling.