An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models

Markus Herrmannsdoerfer[1], Sander D. Vermolen[2], and Guido Wachsmuth[2]

[1] Institut für Informatik,
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
`herrmama@in.tum.de`
[2] Software Engineering Research Group,
Delft University of Technology
The Netherlands
`{S.D.Vermolen,G.H.Wachsmuth}@tudelft.nl`

Abstract. Modeling languages and thus their metamodels are subject to change. When a metamodel is evolved, existing models may no longer conform to it. Manual migration of these models in response to metamodel evolution is tedious and error-prone. To significantly automate model migration, operator-based approaches provide reusable coupled operators that encapsulate both metamodel evolution and model migration. The success of an operator-based approach highly depends on the library of reusable coupled operators it provides. In this paper, we thus present an extensive catalog of coupled operators that is based both on a literature survey as well as real-life case studies. The catalog is organized according to a number of criteria to ease assessing the impact on models as well as selecting the right operator for a metamodel change at hand.

1 Introduction

Like software, modeling languages are subject to evolution due to changing requirements and technological progress [9]. A modeling language is adapted to the changed requirements by evolving its metamodel. Due to *metamodel evolution*, existing models may no longer conform to the evolved metamodel and thus need to be migrated to reestablish conformance to the evolved metamodel. Avoiding *model migration* by downwards-compatible metamodel changes is often a poor solution, since it reduces the quality of the metamodel and thus the modeling language [5]. Manual migration of models is tedious and error-prone, and hence model migration needs to be automated. In *coupled evolution* of metamodels and models, the association of a model migration to a metamodel evolution is managed automatically. There are two major coupled evolution approaches: difference-based and operator-based approaches.

Difference-based approaches use a declarative evolution specification, generally referred to as difference model [6,11]. The difference model is mapped onto a model migration, which may be specified declaratively as well as imperatively.

Operator-based approaches specify metamodel evolution by a sequence of operator applications [24,13]. Each operator application can be coupled to a model migration separately. Operator-based approaches generally provide a set of reusable coupled operators which work at the metamodel level as well as at the model level. At the metamodel level, a coupled operator defines a metamodel transformation capturing a common evolution. At the model level, a coupled operator defines a model transformation capturing the corresponding migration. Application of a coupled operator to a metamodel and a conforming model preserves model conformance.

In both operator-based and difference-based approaches, evolution can be specified manually [24], can be recorded [13], or can be detected automatically [6,11]. When recording, the user is restricted to a recording editor. Using automated detection, the building process can be completely automated, but may lead to an incorrect model migration.

In this paper, we follow an operator-based approach to automate building a model migration for EMOF-like metamodels [18]. The success of an operator-based approach highly depends on the library of reusable coupled operators it provides [20]. The library of an operator-based approach needs to fulfill a number of requirements. A library should seek completeness so as to be able to cover a large set of evolution scenarios. However, the higher the number of coupled operators, the more difficult it is to find a coupled operator in the library. Consequently, a library should also be organized in a way that it is easy to select the right coupled operator for the change at hand.

To provide guidance for building a library, we present an extensive catalog of coupled operators in this paper. To ensure completeness, the coupled operators in this catalog are either motivated from the literature or from case studies that we performed. However, we do not target theoretical completeness—to capture all possible migrations—but rather practical completeness—to capture migrations that likely happen in practice. To ease usability, the catalog is organized according to a number of criteria. The criteria do not only allow to select the right coupled operator from the catalog, but also to assess the impact of the coupled operator on the modeling language and its models. For difference-based approaches, the catalog serves as a set of composite changes that such an approach needs to be able to handle.

The paper is structured as follows: Section 2 presents the EMOF-like metamodeling formalism on which the coupled operators are based. Section 3 introduces the papers and case studies from which the coupled operators originate. Section 4 defines different classification criteria for coupled operators. Section 5 lists and groups the coupled operators of the catalog. Section 6 discusses the catalog, and Section 7 concludes the paper.

2   Metamodeling Formalism

Metamodels can be expressed in various metamodeling formalisms. Well-known examples are the Meta Object Facility (MOF) [18], the metamodeling standard

proposed by the Object Management Group (OMG) and Ecore [21], the meta-modeling formalism underlying the Eclipse Modeling Framework (EMF). In this paper, we focus only on the core metamodeling constructs that are interesting for coupled evolution of metamodels and models. We leave out annotations, derived features, and operations, since these cannot be instantiated in models. An operator catalog will need additional operators addressing these metamodeling constructs in order to reach full compatibility with Ecore or MOF.

Metamodel. Figure 1 gives a textual definition of the metamodeling formalism used in this paper. A metamodel is organized into *Package*s which can themselves be composed of *sub packages*. Each package defines a number of *Type*s which can be either primitive (*PrimitiveType*) or complex (*Class*). Primitive types are either *DataType*s like Boolean, Integer and String or *Enumeration*s of *literals*. Classes consist of a number of *features*. They can have *super types* to inherit features and might be *abstract*, i.e. are not allowed to have objects. The *name* of a feature needs to be unique among all features of a class, including inherited ones. A *Feature* has a multiplicity (*lower bound* and *upper bound*) and is either an *Attribute* or a *Reference*. An attribute is a feature with a primitive *type*, whereas a reference is a feature with a complex *type*. An attribute can serve as an *id*entifier for objects of a class, i.e. the values of this attribute must be unique among all objects. A reference may be *composite* and two references can be combined to form a bidirectional association by making them *opposite* of each other.

```
abstract class NamedElement {
    name        :: String (1..1)
}

class Package : NamedElement {
    subPackages <> Package (0..*)
    types       <> Type (0..*)
}

abstract class Type : NamedElement {}

abstract class PrimitiveType : Type {}

class DataType : PrimitiveType {}

class Enumeration : PrimitiveType {
    literals    <> Literal (0..*)
}

class Literal : NamedElement {}
```

```
class Class : Type {
    isAbstract  :: Boolean
    superTypes  -> Class (0..*)
    features    <> Feature (0..*)
}

abstract class Feature : NamedElement {
    lowerBound  :: Integer
    upperBound  :: Integer
    type        -> Type
}

class Attribute : Feature {
    isId        :: Boolean
}

class Reference : Feature {
    isComposite :: Boolean
    opposite    -> Reference
}
```

Fig. 1. Metamodeling formalism providing core metamodeling concepts

Model. At the model level, instances of classes are called *objects*, instances of primitive data types are called *values*, instances of features are called *slots*, and instances of references are called *links*. The set of all links of composite references forms a containment structure, which needs to be tree-shaped and span all objects in a model.

Notational Conventions. Throughout the paper, we use the textual notation from Figure 1 for metamodels. In this notation, features are represented by their name followed by a separator, their type, and an optional multiplicity. The separator indicates the kind of a feature. We use `::` for attributes, `->` for ordinary references, and `<>` for composite references.

3   Origins of Coupled Operators

The coupled operators are either motivated from the literature or from case studies that we performed.

Literature. First, coupled operators originate from the literature on the evolution of metamodels as well as object-oriented database schemas and code.

Wachsmuth first proposes an operator-based approach for *metamodel* evolution and classifies a set of operators according to the preservation of metamodel expressiveness and existing models [24]. Gruschko et al. envision a difference-based approach and therefore classify all primitive changes according to their impact on existing models [2,4]. Cicchetti et al. list a set of composite changes which they are able to detect using their difference-based approach [6].

Banerjee et al. present a complete and sound set of primitives for schema evolution in the *object-oriented database* system ORION and characterize the primitives according to their impact on existing databases [1]. Brèche introduces a set of high-level operators for schema evolution in the object-oriented system O_2 and shows how to implement them in terms of primitive operators [3]. Pons and Keller propose a three-level catalog of operators for object-oriented schema evolution which groups operators according to their complexity [19]. Claypool et al. list a number of primitives for the adaptation of relationships in object-oriented systems [7].

Fowler presents a catalog of operators for the refactoring of *object-oriented code* [10]. Dig and Johnson show—by performing a case study—that most changes on object-oriented code can be captured by a rather small set of refactoring operators [8].

Case Studies. Second, coupled operators originate from a number of case studies that we have performed. Table 1 gives an overview over these case studies. It mentions the tool that was used in a case study, the name of the evolving metamodel, an abbreviation for the case study which we use in other tables throughout the paper, and whether the evolution was obtained in a forward or reverse engineering process. In forward engineering, the tool is used to aid and possibly record evolution as it happens, whereas in reverse engineering, the tool is used to reconstruct evolution after it occurred. To provide evidence that the case studies are considerable in size, the table also shows the number of different kinds of metamodel elements at the end of the evolution as well as the number of operator applications needed to perform the evolution.

Table 1. Statistics for case studies

Tool	Abbreviation	Name	Kind	Packages	Classes	Attributes	References	Data Types	Enumerations	Literals	Operator Applications
[12]	F	FLUID	reverse	8	155	95	155	0	1	10	223
	T	TAF-Gen		15	97	81	114	1	13	76	134
COPE	[13]	PCM	reverse	19	99	18	135	0	4	19	101
	[14]	GMF		4	252	379	278	0	27	166	737
	U	Unicase	forward	17	77	88	161	0	11	49	58
	Q	Quamoco		1	22	14	35	0	1	2	423
Acoda	B	BugZilla	reverse	–	51	208	64	–	–	–	237
	R	Researchr	forward	–	125	380	278	–	6	31	64
	Y	YellowGrass		–	12	33	21	–	0	0	30

Herrmannsdoerfer et al. performed a case study on the evolution of two industrial metamodels to show that most of the changes can be captured by reusable coupled operators [12]: Flexible User Interface Development (FLUID) for the specification of automotive user interfaces and Test Automation Framework - Generator (TAF-Gen) for the generation of test cases for these user interfaces.

Based on the requirements derived from this study, Herrmannsdoerfer implemented the operator-based tool *COPE*[1] [13] which records operator histories on metamodels of the Eclipse Modeling Framework (EMF). To demonstrate its applicability, COPE has been used to reverse engineer the operator history of a number of metamodels: Palladio Component Model (PCM) for the specification of software architectures [13] and Graphical Modeling Framework (GMF) for the model-based development of diagram editors [14]. Currently, COPE is applied to forward engineer the operator history of a number of metamodels: Unicase[2] for UML modeling and project management and Quamoco[3] for modeling the quality of software products.

Vermolen implemented the operator-based tool *Acoda*[4] [22] which detects operator histories on object-oriented data models. To demonstrate its applicability, Acoda has been used to reverse engineer the operator history of the data model behind BugZilla which is a well-known tool for bug tracking. Currently, Acoda is applied to forward engineer the operator-based evolution of a number of data models: Researchr[5] for maintaining scientific publications and Yellow-Grass[6] for tag-based issue tracking. The crossed-out cells in Table 1 indicate

---

[1] COPE web site, `http://cope.in.tum.de`

[2] Unicase web site, `http://unicase.org`

[3] Quamoco web site, `http://www.quamoco.de`

[4] Acoda web site, `http://swerl.tudelft.nl/bin/view/Acoda`

[5] Researchr web site, `http://researchr.org`

[6] YellowGrass web site, `http://yellowgrass.org`

that the metamodeling constructs are currently not supported by the used data modeling formalism.

4   Classification of Coupled Operators

Coupled operators can be classified according to several properties. We are interested in language preservation, model preservation, and bidirectionality. Therefore, we stick to a simplified version of the terminology from [24].

Language Preservation. A metamodel is an intensional definition of a language. Its extension is a set of conforming models. When an operator is applied to a metamodel, this has an impact on its extension and thus on the expressiveness of the language. We distinguish different classes of operators according to this impact [15,24]: An operator is a *refactoring* if there exists always a bijective mapping between extensions of the original and the evolved metamodel. An operator is a *constructor* if there exists always an injective mapping from the extension of the original metamodel to the extension of the evolved metamodel. An operator is a *destructor* if there exists always a surjective mapping from the extension of the original metamodel to the extension of the evolved metamodel.

Model Preservation. Model preservation properties indicate when migration is needed. An operator is *model-preserving* if all models conforming to an original metamodel also conform to the evolved metamodel. Thus, model-preserving operators do not require migration. An operator is *model-migrating* if models conforming to an original metamodel might need to be migrated in order to conform to the evolved metamodel. It is *safely model-migrating* if the migration preserves distinguishability, i.e. different models (conforming to the original metamodel) are migrated to different models (conforming to the evolved metamodel). In contrast, an *unsafely model-migrating* operator might yield the same model when migrating two different models.

Classification of operators w.r.t. model preservation is related to the classification w.r.t. language preservation: Refactorings and constructors are either model-preserving or safely model-migrating operators. Destructors are unsafely model-migrating operators. Furthermore, the classification is related to a classification of changes known from difference-based approaches [2,4]: model-preserving operators perform *non-breaking changes*, whereas model-migrating operators perform *breaking, resolvable changes*. However, there is no correspondence for *breaking, non-resolvable changes*, since coupled operators always provide a migration to resolve the breaking change.

Bidirectionality. Another property we are interested in is the reversibility of evolution. Bidirectionality properties indicate that an operator can be safely

undone on the language or model level. An operator is *self-inverse* iff a second application of the operator—possibly with other parameters—always yields the original metamodel. An operator is the *inverse* of another operator iff there is always a sequential composition of both operators which does not change the metamodel. Finally, an operator is a *safe inverse* of another operator iff it is an inverse and there is always a sequential composition of both operators which is model-preserving.

5   Catalog of Coupled Operators

In this section, we present a catalog of 61 coupled operators that we consider complete for practical application. As discussed in Section 3, we included all coupled operators found in nine related papers as well as all coupled operators identified by performing nine real-life case studies. In the following, we explain the coupled operators in groups which help users to navigate the catalog. We start with *primitive* operators which perform an atomic metamodel evolution step that can not be further subdivided. Here, we distinguish *structural* primitives which create and delete metamodel elements and *non-structural* primitives which modify existing metamodel elements. Afterwards, we continue with *complex* operators. These can be decomposed into a sequence of primitive operators which has the same effect at the metamodel level but not neccessarily at the model level. We group complex operators according to the metamodeling techniques they address—distinguishing specialization and generalization, delegation, and inheritance operators—as well as their semantics—distinguishing replacement, and merge and split operators.

Each group is discussed separately in the subsequent sections. For each group, a table provides an overview over all operators in the group. Using the classifications from Section 4, the table classifies each coupled operator according to language preservation (L) into refactoring (r), constructor (c) and destructor (d) as well as according to model preservation (M) into model-preserving (p), safely (s) and unsafely (u) model-migrating. The table further indicates the safe (s) and unsafe (u) inverse (I) of each operator by referring to its number. Finally, each paper and case study has a column in each table. An x in such a column denotes occurrence of the operator in the corresponding paper or case study. Papers are referred to by citation, while case studies are referred to by the abbreviation given in Table 1. For each coupled operator, we discuss its semantics in terms of metamodel evolution and model migration.

5.1   Structural Primitives

Structural primitive operators modify the structure of a metamodel, i.e. create or delete metamodel elements. Creation operators are parameterized by the specification of a new metamodel element, and deletion operators by an existing metamodel element.

| | | Class. | | | MM | | | OODB | | | | OOC | | [12] | | COPE | | | | Acoda | | |
#	Operator Name	L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Create Package	r	p	2s		x									x	x						
2	Delete Package	r	p	1s		x									x							
3	Create Class	c	p	4s	x	x	x	x						x	x	x	x	x	x	x	x	x
4	Delete Class	d	u	3u	x	x	x	x	x				x	x		x		x	x	x	x	
5	Create Attribute	c	s	7s	x	x	x	x						x	x	x	x	x	x	x	x	x
6	Create Reference	c	s	7s	x	x	x	x						x	x	x	x	x	x	x	x	x
7	Delete Feature	d	u	5/6u	x	x	x	x						x	x	x	x	x	x	x	x	
8	Create Opposite Ref.	d	u	9u		x				x	x						x	x	x		x	x
9	Delete Opposite Ref.	c	p	8s		x				x	x			x								
10	Create Data Type	r	p	11s		x																
11	Delete Data Type	r	p	10s		x													x			
12	Create Enum	r	p	13s		x								x	x	x	x	x	x			
13	Delete Enum	r	p	11s		x												x				
14	Create Literal	c	p	15s		x												x				
15	Merge Literal	d	u	14u		x											x					

Creation of non-mandatory metamodel elements (packages, classes, optional features, enumerations, literals and data types) is model-preserving. Creation of mandatory features is safely model-migrating. It requires initialization of slots using default values or default value computations.

Deleting metamodel elements, such as classes and references, requires deleting instantiating model elements, such as objects and links, by the migration. However, deletion of model elements poses the risk of migration to inconsistent models: For example, deletion of objects may cause links to non-existent objects and deletion of references may break object containment. Therefore, deletion operators are bound to metamodel level restrictions: Packages may only be deleted when they are empty. Classes may only be deleted when they are outside inheritance hierarchies and are targeted neither by non-composite references nor by mandatory composite references. Several complex operators discussed in subsequent sections can deal with classes not meeting these requirements. References may only be deleted when they are neither composite, nor have an opposite. Enumerations and data types may only be deleted when they are not used in the metamodel and thus obsolete.

Deletion operators which may have been instantiated in the model (with the exception of *Delete Opposite Reference*) are unsafely model-migrating due to loss of information. Deletion provides a safe inverse to its associated creation operator. Since deletion of metamodel elements which may have been instantiated in a model is unsafely model-migrating, creation of such elements provides an unsafe inverse to deletion: Lost information cannot be restored.

Creating and deleting references which have an opposite are different from other creation and deletion operators. *Create Opposite Reference* restricts the set of valid links and is thus an unsafely model-migrating destructor, whereas *Delete Opposite Reference* removes a constraint from the model and is thus a model-preserving constructor.

Create / Delete Data Type and *Create / Delete Enumeration* are refactorings, as restrictions on these operators prevent usage of created or deleted elements. Deleting enumerations and data types is thus model-preserving. *Merge Literal*

deletes a literal and replaces its occurrences in a model by another literal. Merging a literal provides a safe inverse to *Create Literal*.

5.2   Non-structural Primitives

Non-structural primitive operators modify a single, existing metamodel element, i.e. change properties of a metamodel element. All non-structural operators take the affected metamodel element, their subject, as parameter.

#	Operator Name	Class.			MM			OODB				OOC		[12]		COPE				Acoda		
		L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Rename	r	s	1s	x	x	x	x				x	x	x	x	x	x	x	x	x	x	x
2	Change Package	r	s	2s		x								x	x	x			x			
3	Make Class Abstract	d	u	4u		x								x		x			x			
4	Drop Class Abstract	c	p	3s		x											x		x			
5	Add Super Type	c	p	6s		x		x							x	x	x	x	x		x	x
6	Remove Super Type	d	u	5u		x		x							x	x	x	x	x			
7	Make Attr. Identifier	d	u	8u		x								x							x	
8	Drop Attr. Identifier	c	p	7s		x								x		x						
9	Make Ref. Composite	d	u	10u		x						x		x	x	x			x			
10	Switch Ref. Composite	c	s	9s		x		x				x		x	x				x			
11	Make Ref. Opposite	d	u	12u		x					x			x							x	x
12	Drop Ref. Opposite	c	p	11s		x					x					x		x				

Change Package can be applied to both package and type. Additionally, the value-changing operators *Rename*, *Change Package* and *Change Attribute Type* are parameterized by a new value. *Make Class Abstract* requires a subclass parameter indicating to which class objects need to be migrated. *Switch Reference Composite* requires an existing composite reference as target.

Packages, types, features and literals can be renamed. *Rename* is safely model-migrating and finds a self-inverse in giving a subject its original name back. *Change Package* changes the parent package of a package or type. Like renaming, it is safely model-migrating and a safe self-inverse.

Classes can be made abstract, requiring migration of objects to a subclass, because otherwise, links targeting the objects may have to be removed. Consequently, mandatory features that are not available in the super class have to be initialized to default values. *Make Class Abstract* is unsafely model-migrating, due to loss of type information and has an unsafe inverse in *Drop Class Abstract*. Super type declarations may become obsolete and may need to be removed. *Remove Super Type* s from a class c implies removing slots of features inherited from s. Additionally, references targeting type s, referring to objects of type c, need to be removed. To prevent breaking multiplicity restrictions, *Remove Super Type* is restricted to types s which are not targeted by mandatory references—neither directly, nor through inheritance. The operator is unsafely model-migrating and can be unsafely inverted by *Add Super Type*.

Attributes defined as identifier need to be unique. *Make Attribute Identifier* requires a migration which ensures uniqueness of the attribute's values and is thus unsafely model-migrating. *Drop Attribute Identifier* is model-preserving and does not require migration.

References can have an opposite and can be composite. An opposite reference declaration defines the inverse of the declaring reference. References combined with a multiplicity restriction on the opposite reference restrict the set of valid links. *Make Reference Opposite* needs a migration to make the link set satisfy the added multiplicity restriction. The operator is thereby unsafely model-migrating. *Drop Reference Opposite* removes cardinality constraints from the link set and does not require migration, thus being model-preserving.

Make Reference Composite ensures containment of referred objects. Since all referred objects were already contained by another composite reference, all objects must be copied. To ensure the containment restriction, copying has to be recursive across composite references (deep copy). Furthermore, to prevent cardinality failures on opposite references, there may be no opposite references to any of the types of which objects are subject to deep copying. *Switch Reference Composite* changes the containment of objects to an existing composite reference. If objects of a class A were originally contained in class B through composite reference b, *Switch Reference Composite* changes containment of A objects to class C, when it is parameterized by reference b and a composite reference c in class C. After applying the operator, reference b is no longer composite. *Switch Reference Composite* provides an unsafe inverse to *Make Reference Composite*.

5.3   Specialization / Generalization Operators

Specializing a metamodel element reduces the set of possible models, whereas generalizing expands the set of possible models. Generalization and specialization can be applied to features and super type declarations. All specialization and generalization operators take two parameters: a subject and a generalization or specialization target. The first is a metamodel element and the latter is a class or a multiplicity (lower and upper bound).

#	Operator Name	Class.			MM			OODB				OOC		[12]		COPE				Acoda		
		L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Generalize Attribute	c	p	2s	x	x	x							x	x	x	x		x	x	x	x
2	Specialize Attribute	d	u	1u	x	x	x							x		x	x		x			
3	Generalize Reference	c	p	4s	x	x	x							x	x		x		x		x	x
4	Specialize Reference	d	u	3u	x	x	x							x	x	x	x		x			
5	Specialize Composite Ref.	d	u	3u										x		x			x			
6	Generalize Super Type	d	u	7u		x										x					x	
7	Specialize Super Type	c	s	6s		x				x				x	x	x	x		x			

Generalization of features does not only generalize the feature itself, but also generalizes the metamodel as a whole. Feature generalizations are thus model-preserving constructors. Generalizing a super type declaration may require removal of feature slots and is only unsafely model-migrating. Feature specialization is a safe inverse of feature generalization. Due to the unsafe nature of the migration resulting from feature specialization, generalization provides an unsafe inverse to specialization. Super type generalization is an safe inverse of super type specialization which is a unsafe inverse vice versa.

Specialize Attribute either reduces the attribute's multiplicity or specializes the attribute's type. When reducing multiplicity, either the lower bound is increased or the upper bound is decreased. When specializing the type, a type conversion maps the original set of values onto a new set of values conforming the new attribute type. Specializing type conversions are surjective. *Generalize Attribute* extends the attribute's multiplicity or generalizes the attribute's type. Generalizing an attribute's type involves an injective type conversion. Type conversions are generally either implemented by transformations for each type to an intermediate format (e.g. by serialization) or by transformations for each combination of types. The latter is more elaborate to implement, yet less fragile. Most generalizing type conversions from type x to y have a specializing type conversion from type y to x as safe inverse. Applying the composition vice versa yields an unsafe inverse.

Similar to attributes, reference multiplicity can be specialized and generalized. *Specialize / Generalize Reference* can additionally specialize or generalize the type of a reference by choosing a sub type or super type of the original type, respectively. Model migration of reference specialization requires deletion of links not conforming the new reference type. *Specialize Composite Reference* is a special case of reference specialization at the metamodel level, which requires contained objects to be migrated to the targeted subclass at the model level, to ensure composition restrictions. *Specialize Composite Reference* is unsafely model-migrating.

Super type declarations are commonly adapted, while refining a metamodel. Consider the following example, in which classes A, B and C are part of a linear inheritance structure and remain unadapted:

```
class A     { }                      class A     { }
class B : A { f :: Integer (1..1) }  class B : A { f :: Integer (1..1) }
class C : A { }                      class C : B { }
```

From left to right, *Specialize Super Type* changes a declaration of super type A on class C to B, a sub type of A. Consequently, a mandatory feature f is inherited, which needs the creation of slots by the migration. In general, super type specialization requires addition of feature slots which are declared mandatory by the new super type. From right to left, *Generalize Super Type* changes a declaration of super type B on class C to A, a super type of B. In the new metamodel, feature f is no longer inherited in C. Slots of features which are no longer inherited need to be removed by the migration. Furthermore, links to objects of A that target class B, are no longer valid, since A is no longer a sub type of B. Therefore, these links need to be removed, if multiplicity restrictions allow, or adapted otherwise.

5.4 Inheritance Operators

Inheritance operators move features along the inheritance hierarchy. Most of them are well-known from refactoring object-oriented code [10]. There is always a pair of a constructor and destructor, where the destructor is the safe inverse of the constructor, and the constructor is the unsafe inverse of the destructor.

#	Operator Name	Class.			MM			OODB				OOC		[12]		COPE				Acoda		
		L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Pull up Feature	c	p	2s	x		x					x	x	x	x		x		x			
2	Push down Feature	d	u	1u	x		x					x	x		x		x					
3	Extract Super Class	c	p	4s	x		x	x	x			x	x	x	x		x		x		x	x
4	Inline Super Class	d	u	3u	x		x		x			x	x		x		x	x	x			
5	Fold Super Class	c	s	6s										x					x			
6	Unfold Super Class	d	u	5u													x		x			
7	Extract Sub Class	c	s	8s				x	x			x					x		x			
8	Inline Sub Class	d	u	7u					x			x			x				x			

Pull up Feature is a constructor which moves a feature that occurs in all sub-classes of a class to the class itself. For migration, slots for the pulled up feature are added to objects of the class and filled with default values. The corresponding destructor *Push down Feature* moves a feature from a class to all its subclasses. While objects of the subclasses stay unaltered, slots for the original feature must be removed from objects of the class itself.

Extract Super Class is a constructor which introduces a new class, makes it the super class of a set of classes, and pulls up one or more features from these classes. The corresponding destructor *Inline Super Class* pushes all features of a class into its subclasses and deletes the class afterwards. References to the class are not allowed but can be generalized to a super class in a previous step. Objects of the class need to be migrated to objects of the subclasses. This might require the addition of slots for features of the subclasses.

The constructor *Fold Super Class* is related to *Extract Super Class*. Here, the new super class is not created but exists already. This existing class has a set of (possibly inherited) features. In another class, these features are defined as well. The operator then removes these features and adds instead an inheritance relation to the intended super class. In the same way, the destructor *Unfold Super Class* is related to *Inline Super Class*. This operator copies all features of a super class into a subclass and removes the inheritance relation between both classes. Here is an example for both operators:

```
class A     { f1 :: Integer }
class B : A { f2 :: Integer }
class C     { f1 :: Integer
              f2 :: Integer
              f3 :: Integer }
```

```
class A     { f1 :: Integer }
class B : A { f2 :: Integer }
class C : B { f3 :: Integer }
```

From left to right, the super class B is folded from class C which includes all the features of B. These features are removed from C, and B becomes a super class of C. From right to left, the super class B is unfolded into class C by copying features A.f1 and B.f2 to C. B is not longer a super class of C.

The constructor *Extract Subclass* introduces a new class, makes it the subclass of another, and pushes down one or more features from this class. Objects of the original class must be converted to objects of the new class. The corresponding destructor *Inline Subclass* pulls up all features from a subclass into its non-abstract super class and deletes the subclass afterwards. References to the class are not allowed but can be generalized to a super class in a previous step. Objects of the subclass need to be migrated to objects of the super class.

5.5 Delegation Operators

Delegation operators move metamodel elements along compositions or ordinary references. Most of the time, they come as pairs of corresponding refactorings being safely inverse to each other.

#	Operator Name	Class.			MM			OODB				OOC		[12]		COPE			Q	Acoda		
		L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U		B	R	Y
1	Extract Class	r	s	2s	x		x	x	x			x	x	x	x		x		x	x		x
2	Inline Class	r	s	1s	x		x		x			x	x		x		x			x		x
3	Fold Class	r	s	4s												x	x	x				
4	Unfold Class	r	s	3s																		
5	Move Feature over Ref.	c	s	6s	x		x		x			x	x		x		x			x		x
6	Collect Feature over Ref.	d	u	5u													x		x			

Extract Class moves features to a new delegate class and adds a composite reference to the new class together with an opposite reference. During migration, an object of the delegate class is created for each object of the original class, slots for the moved features are moved to the new delegate object, and a link to the delegate object is created. The corresponding *Inline Class* removes a delegate class and adds its features to the referring class. There must be no other references to the delegate class. On the model level, slots of objects of the delegate class are moved to objects of the referring class. Objects of the delegate class and links to them are deleted. The operators become a pair of constructor and destructor, if the composite reference has no opposite.

Fold and *Unfold Class* are quite similar to *Extract* and *Inline Class*. The only difference is, that the delegate class exists already and thus is not created or deleted. The following example illustrates the difference:

```
class A { a1 :: Integer              class A { c <> C (1..1)
          a2 :: Boolean                        d <> D (1..1) opposite a }
          r1 -> B (1..1)             class B {  }
          r2 -> B (0..*) }           class C { a1 :: Integer
class B { }                                    r1 -> B (1..1) }
class C { a1 :: Integer              class D { a2 :: Boolean
          r1 -> B (1..1) }                     r2 -> B (0..*)
                                               a  -> A (1..1) opposite d }
```

From left to right, the features a1 and r1 of class A are folded to a composite reference A.c to class C which has exactly these two features. In contrast, the features a2 and r2 of class A are extracted into a new delegate class D. From right to left, the composite reference A.c is unfolded which keeps C intact while A.d is inlined which removes D.

Move Feature along Reference is a constructor which moves a feature over a single-valued reference to a target class. Slots of the original feature must be moved over links to objects of the target class. For objects of the target class which are not linked to an object of the source class, slots with default values must be added. The destructor *Collect Feature over Reference* is a safe inverse of the last operator. It moves a feature backwards over a reference. The multiplicity of the feature might be altered during the move depending on the multiplicity of the reference. For optional and/or multi-valued references, the feature becomes

optional respectively multi-valued, too. Slots of the feature must be moved over links from objects of the source class. If an object of the source class is not linked from objects of the target class, slots of the original feature are removed. Here is an example for both operators:

```
class A { f1 :: Integer (1..*)          class A { f2 :: Integer (0..*)
         r1 -> B (1..1)                          r1 -> B (1..1)
         r2 -> C (0..*) }                        r2 -> C (0..*) }
class B { }                             class B { f1 :: Integer (1..*) }
class C { f2 :: Integer (1..1) }        class C { }
```

From left to right, the feature A.f1 is moved along the reference A.r1 to class B. Furthermore, the feature C.f2 is collected over the reference A.r2 and ends up in class A. Since A.r2 is optional and multi-valued, A.f2 becomes optional and multi-valued, too. From right to left, the feature B.f1 is collected over the reference A.r1. Its multiplicity stays unaltered. Note that there is no single operator for moving A.f2 to class C which makes *Collect Feature over Reference* in general uninvertible. For the special case of a single-valued reference, *Move Feature along Reference* is an unsafe inverse.

5.6 Replacement Operators

Replacement operators replace one metamodeling construct by another, equivalent construct. Thus replacement operators typically are refactorings and safely model-migrating. With the exception of the last two operators, an operator to replace the first construct by a second always comes with a safe inverse to replace the second by the first, and vice versa.

		Class.			MM			OODB				OOC		[12]		COPE				Acoda		
#	Operator Name	L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Subclasses to Enum.	r	s	2s								x										
2	Enum. to Subclasses	r	s	1s								x						x				
3	Reference to Class	r	s	4s	x									x					x			
4	Class to Reference	r	s	3s	x								x									
5	Inheritance to Delegation	r	s	6s				x				x		x	x		x					
6	Delegation to Inheritance	r	s	5s				x				x										
7	Reference to Identifier	c	s	8s								x										
8	Identifier to Reference	d	u	7u								x				x		x		x		x

To be more flexible, empty subclasses of a class can be replaced by an attribute which has an enumeration as type, and vice versa. *Subclasses to Enumeration* deletes all subclasses of the class and creates the attribute in the class as well as the enumeration with a literal for each subclass. In a model, objects of a certain subclass are migrated to the super class, setting the attribute to the corresponding literal. Thus, the class is required to be non-abstract and to have only empty subclasses without further subclasses. *Enumeration to Subclasses* does the inverse and replaces an enumeration attribute of a class by subclasses for each literal. The following example demonstrates both directions:

```
class C { ... }                         class C { e :: E ... }
class S1 : C {}
class S2 : C {}                         enum E { s1, s2 }
```

From left to right, *Subclasses to Enumeration* replaces the subclasses S1 and S2 of class C by the new attribute C.e which has the enumeration E with literals s1 and s2 as type. In a model, objects of a subclass S1 are migrated to class C, setting the attribute e to the appropriate literal s1. From right to left, *Enumeration to Subclasses* introduces a subclass to C for each literal of E. Next, it deletes the attribute C.e as well as the enumeration E. In a model, objects of class C are migrated to a subclass according to the value of attribute e.

To be able to extend a reference with features, it can be replaced by a class, and vice versa. *Reference to Class* makes the reference composite and creates the reference class as its new type. Single-valued references are created in the reference class to target the source and target class of the original reference. In a model, links conforming to the reference are replaced by objects of the reference class, setting source and target reference appropriately. *Class to Reference* does the inverse and replaces the class by a reference. To not lose expressiveness, the reference class is required to define no features other than the source and target references. The following example demonstrates both directions:

```
class S {
  r –> T (1..*) ...
}
```

```
class S { r <> R (1..*) ... }
class R { s –> S (1..1) opposite r
          t –> T (1..1) }
```

From left to right, *Reference to Class* retargets the reference S.r to a new reference class R. Source and target of the original reference can be accessed via references R.s and R.t. In a model, links conforming to the reference r are replaced by objects of the reference class R. From right to left, *Class to Reference* removes the reference class R and retargets the reference S.r directly to the target class T.

Inheriting features from a superclass can be replaced by delegating them to the superclass, and vice versa. *Inheritance to Delegation* removes the inheritance relationship to the superclass and creates a composite, mandatory single-valued reference to the superclass. In a model, the slots of the features inherited from the superclass are extracted to a separate object of the super class. By removing the super type relationship, links of references to the superclass are no longer allowed to target the original object, and thus have to be retargeted to the extracted object. *Delegation to Inheritance* does the inverse and replaces the delegation to a class by an inheritance link to that class. The following example demonstrates both directions:

```
class C : S { ... }
```

```
class C { s <> S (1..1), ... }
```

From left to right, *Inheritance to Delegation* replaces the inheritance link of class C to its superclass S by a composite, single-valued reference from C to S. In a model, the slots of the features inherited from the super class S are extracted to a separate object of the super class. From right to left, *Delegation to Inheritance* removes the reference C.s and makes S a super class of C.

To decouple a reference, it can be replaced by an indirect reference using an identifier, and vice versa. *Reference to Identifier* deletes the reference and creates an attribute in the source class whose value refers to an id attribute in

the target class. In a model, links of the reference are replaced by setting the attribute in the source object to the identifier of the target object. *Identifier to Reference* does the inverse and replaces an indirect reference via identifier by a direct reference. Our metamodeling formalism does not provide a means to ensure that there is a target object for each identifier used by a source object. Consequently, *Reference to Identifier* is a constructor and *Identifier to Reference* a destructor, thus being an exception in the group of replacement operators.

5.7   Merge / Split Operators

Merge operators merge several metamodel elements of the same type into a single element, whereas split operators split a metamodel element into several elements of the same type. Consequently, merge operators typically are destructors and split operators constructors. In general, each merge operator has an inverse split operator. Split operators are more difficult to define, as they may require metamodel-specific information about how to split values. There are different merge and split operators for the different metamodeling constructs.

#	Operator Name	Class.			MM			OODB				OOC		[12]		COPE				Acoda		
		L	M	I	[24]	[2]	[6]	[1]	[3]	[19]	[7]	[10]	[8]	F	T	[13]	[14]	U	Q	B	R	Y
1	Merge Features	d	u							x				x					x			
2	Split Reference by Type	r	s	1s										x					x			
3	Merge Classes	d	u	4u				x	x					x		x	x		x			
4	Split Class	c	p	3s																		
5	Merge Enumerations	d	u														x					

Merge Features merges a number of features defined in the same class into a single feature. In the metamodel, the source features are deleted and the target feature is required to be general enough—through its type and multiplicity—so that the values of the other features can be fully moved to it in a model. Depending on the type of feature that is merged, a repeated application of *Create Attribute* or *Create Reference* provides an unsafe inverse. *Split Reference by Type* splits a reference into references for each subclass of the type of the original reference. In a model, each link instantiating the original reference is moved to the corresponding target reference according to its type. If we require that the type of the reference is abstract, this operator is a refactoring and has *Merge Features* as a safe inverse.

Merge Classes merges a number of sibling classes—i.e. classes sharing a common superclass—into a single class. In the metamodel, the sibling classes are deleted and their features are merged to the features of the target class according to name equality. Each of the sibling classes is required to define the same features so that this operator is a destructor. In a model, objects of the sibling classes are migrated to the new class. *Split Class* is an unsafe inverse and splits a class into a number of classes. A function that maps each object of the source class to one of the target classes needs to be provided to the migration.

Merge Enumerations merges a number of enumerations into a single enumeration. In the metamodel, the source enumerations are deleted and their literals are merged to the literals of the target enumeration according to name equality.

Each of the source enumerations is required to define the same literals so that this operator is a destructor. Additionally, attributes that have the source enumerations as type have to be retargeted to the target enumeration. In a model, the values of these attributes have to be migrated according to how literals are merged. A repeated application of *Create Enumeration* provides a safe inverse.

6   Discussion

Completeness. At the metamodel level, an operator catalog is complete if any source metamodel can be evolved to any target metamodel. This kind of completeness is achieved by the catalog presented in the paper. An extreme strategy would be the following [1]: In a first step, the original metamodel needs to be discarded. Therefore, we delete opposite references and features. Next, we delete data types and enumerations and collapse inheritance hierarchies by inlining subclasses. We can now delete the remaining classes. Finally, we delete packages. In a second step, the target metamodel is constructed from scratch by creating packages, enumerations, literals, data types, classes, attributes, and references. Inheritance hierarchies are constructed by extracting empty subclasses.

Completeness is much harder to achieve, when we take the model level into account. Here, an operator catalog is complete if any model migration corresponding to an evolution from a source metamodel to a target model can be expressed. In this sense, a complete catalog needs to provide a full-fledged model transformation language based on operators. A first useful restriction is Turing completeness. But reaching for this kind of completeness comes at the price of usability. Given an existing operator, one can always think of a slightly different operator having the same effect on the metamodel level but a slightly different migration. But the higher the number of coupled operators, the more difficult it is to find an operator in the catalog. And with many similar operators, it is hard to decide which one to apply.

We therefore do not target theoretical completeness—to capture all possible migrations—but rather practical completeness—to capture migrations that likely happen in practice. When we started our case studies, we found the set of operators from the literature rather incomplete. For each case study, we added frequently reoccurring operators to the catalog. The number of operators we added to the catalog declined with every new case study, thus approaching a stable catalog. Our latest studies revealed no new operators. Although, in most case studies, we found a few operators which were only applied once in a single case study. They were never reused in other case studies. Therefore, we decided not to include them in the catalog.

We expect similar special cases in practical applications where only a few evolution steps can not be modeled by operators from the catalog. These cases can effectively be handled by providing a means for overwriting a coupling [13]: The user can specify metamodel evolution by an operator application but overwrites the model migration for this particular application. This way, theoretical completeness can still be achieved.

Metamodeling Formalism. In this paper, we focus only on core metamodeling constructs that are interesting for coupled evolution of metamodels and models. But a metamodel defines not only the abstract syntax of a modeling language, but also an API to access models expressed in this language. For this purpose, concrete metamodeling formalisms like Ecore or MOF provide metamodeling constructs like interfaces, operations, derived features, volatile features, or annotations. An operator catalog will need additional operators addressing these metamodeling constructs to reach full compatibility with Ecore or MOF.

These additional operators are relevant for practical completeness. In the GMF case study [14], we found 25% of the applied operators to address changes in the API. Most of these operators do not require migration. The only exceptions were annotations containing constraints. An operator catalog accounting for constraints needs to deal with two kinds of migrations: First, the constraints need migration when the metamodel evolves. Operators need to provide this migration in addition to model migration. Second, evolving constraints might invalidate existing models and thus require model migration. Here, new coupled operators for the evolution of constraints are needed.

Things become more complicated when it comes to CMOF [18]. Concepts like package merge, feature subsetting, and visibility affect the semantics of operators in the paper and additional operators are needed to deal with these concepts. For example, we would need four different kinds of *Rename* due to the package merge: 1) Renaming an element which is not involved in a merge neither before nor after the renaming (*Rename Element*). 2) Renaming an element which is not involved in a merge in order to include it into a merge (*Include by Name*). 3) Renaming an element which is involved in a merge in order to exclude it from the merge (*Exclude by Name*). 4) Renaming all elements which are merged to the same element (*Rename Merged Element*).

Tool Support. In operator-based tools, operators are usually made available to the user through an operator browser [23,13]. Here, the organization of the catalog into groups can help to find an operator for a change at hand. The preservation properties can be used to reason about the impact on language expressiveness and on existing models. In grammarware, similar operators have been successfully used in [16] to reason about relationships between different versions of the Java grammar. To make the user aware of the impact on models, it can be shown by a traffic light in the browser: green for model-preserving, yellow for safely and red for unsafely model-migrating. Additionally, the operator browser may have different modes for restricting the presented operators in order to guarantee language- and/or model-preservation properties. Bidirectionality can be used to invert an evolution that has been specified erroneously earlier. Recorded operator applications can be automatically undone with different levels of safety by applying the inverse operators. Tools that support evolution detection should evade of destructors in favor of refactorings to increase the preservation of information by the detected evolution.

Difference-based tools [6,11] need to be able to specify the mappings underlying the operators from the catalog. When they allow to specify complex

mappings, they could introduce means to specify the mappings of the operators in a straightforward manner. Introducing such first class constructs reduces the effort for specifying the migration. For instance, the declarative language presented in [17] provides patterns to specify recurrent mappings.

7   Conclusion

We presented a catalog of 61 operators for the coupled evolution of metamodels and models. These so-called coupled operators evolve a metamodel and in response are able to automatically migrate existing models. The catalog covers not only well-known operators from the literature, but also operators which have proven useful in a number of case studies we performed. The catalog is based on the widely used EMOF metamodeling formalism [18] which was stripped of the constructs that cannot be instantiated in models. When a new construct is added to the metamodeling formalism, new operators have to be added to the catalog: Primitive operators to create, delete and modify the construct as well as complex operators to perform more intricate evolutions involving the construct. The catalog not only serves as a basis for operator-based tools, but also for difference-based tools. Operator-based tools need to provide an implementation of the presented operators. Difference-based tools need to be able to specify the mappings underlying the presented operators.

References

1. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. SIGMOD Rec. 16(3), 311–322 (1987)
2. Becker, S., Goldschmidt, T., Gruschko, B., Koziolek, H.: A process model and classification scheme for semi-automatic meta-model evolution. In: Proc. 1st Workshop MDD, SOA und IT-Management (MSI 2007), pp. 35–46. GiTO-Verlag (2007)
3. Brèche, P.: Advanced primitives for changing schemas of object databases. In: Constantopoulos, P., Vassiliou, Y., Mylopoulos, J. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 476–495. Springer, Heidelberg (1996)
4. Burger, E., Gruschko, B.: A Change Metamodel for the Evolution of MOF-Based Metamodels. In: Modellierung 2010. GI-LNI, vol. P-161 (2010)
5. Casais, E.: Managing class evolution in object-oriented systems. In: Object-Oriented Software Composition, pp. 201–244. Prentice Hall, Englewood Cliffs (1995)
6. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: EDOC 2008: 12th International Enterprise Distributed Object Computing Conference. IEEE, Los Alamitos (2008)

7. Claypool, K.T., Rundensteiner, E.A., Heineman, G.T.: ROVER: A framework for the evolution of relationships. In: Laender, A.H.F., Liddle, S.W., Storey, V.C. (eds.) ER 2000. LNCS, vol. 1920, pp. 893–917. Springer, Heidelberg (2000)
8. Dig, D., Johnson, R.: How do APIs evolve? a story of refactoring. J. Softw. Maint. Evol. 18(2), 83–107 (2006)
9. Favre, J.M.: Languages evolve too! changing the software time scale. In: IWPSE 2005: 8th International Workshop on Principles of Software Evolution, pp. 33–42 (2005)
10. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1999)
11. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Managing model adaptation by precise detection of metamodel changes. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 34–49. Springer, Heidelberg (2009)
12. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 645–659. Springer, Heidelberg (2008)
13. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - automating coupled evolution of metamodels and models. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 52–76. Springer, Heidelberg (2009)
14. Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G.: Language evolution in practice: The history of GMF. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 3–22. Springer, Heidelberg (2010)
15. Lämmel, R.: Grammar adaptation. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 550–570. Springer, Heidelberg (2001)
16. Lammel, R., Zaytsev, V.: Recovering grammar relationships for the Java language specification. In: SCAM 2009: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 178–186. IEEE, Los Alamitos (2009)
17. Narayanan, A., Levendovszky, T., Balasubramanian, D., Karsai, G.: Automatic domain model migration to manage metamodel evolution. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 706–711. Springer, Heidelberg (2009)
18. Object Management Group: Meta Object Facility (MOF) core specification version 2.0 (2006), http://www.omg.org/spec/MOF/2.0/
19. Pons, A., Keller, R.: Schema evolution in object databases by catalogs. In: IDEAS 1997: International Database Engineering and Applications Symposium, pp. 368–376 (1997)
20. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.: An analysis of approaches to model migration. In: Models and Evolution (MoDSE-MCCM) Workshop (2009)
21. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley, Reading (2009)
22. Vermolen, S.D., Visser, E.: Heterogeneous coupled evolution of software languages. In: Busch, C., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 630–644. Springer, Heidelberg (2008)
23. Wachsmuth, G.: An adaptation browser for MOF. In: WRT 2001: First Workshop on Refactoring Tools, pp. 65–66 (2007)
24. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)