Metamodel Adaptation and Model Co-adaptation

Guido Wachsmuth

Humboldt-Universität zu Berlin
Unter den Linden 6
D-10099 Berlin, Germany
guwac@gk-metrik.de

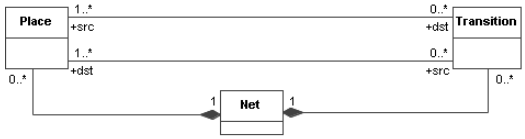
Abstract. Like other software artefacts, metamodels evolve over time. We propose a transformational approach to assist metamodel evolution by stepwise adaptation. In the first part of the paper, we adopt ideas from grammar engineering to define several semantics- and instance-preservation properties in terms of metamodel relations. This part is not restricted to any metamodel formalism. In the second part, we present a library of QVT Relations for the stepwise adaptation of MOF compliant metamodels. Transformations from this library separate preservation properties. We distinguish three kinds of adaptation according to these properties; namely refactoring, construction, and destruction. Co-adaptation of models is discussed with respect to instance-preservation. In most cases, co-adaptation is achieved automatically. Finally, we point out applications in the areas of metamodel design, implementation, refinement, maintenance, and recovery.

1 Introduction

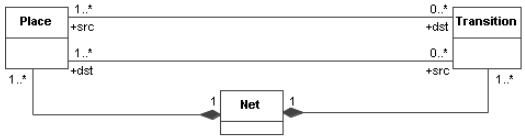
Metamodel evolution. In Model-Driven Architecture (MDA) [1], metamodels are a fundamental building block. Models occurring in a MDA process comply to metamodels, constraints are expressed at the meta-level, and model transformations are based on source and target metamodels. Like other software artefacts, metamodels evolve over time [2] due to several reasons: During design, alternative metamodel versions are developed and well-known solutions are customised for new applications. During implementation, metamodels are adapted to a concrete metamodel formalism supported by a tool. During maintenance, errors in a metamodel are corrected. Furthermore, parts of the metamodel are redesigned due to a better understanding or to facilitate reuse.

Example 1 (Petri net metamodel evolution). Fig. 1 illustrates the evolution of a metamodel for Petri nets. A Petri net consists of any number of places and transitions. Each transition has at least one input and one output place. The initial metamodel μ_0 captures these facts. Since a Petri net without any places and transitions is of no avail, we restrict `Net` to comprise at least one place and one transition. This results in a new metamodel μ_1. In a next step, we make arcs

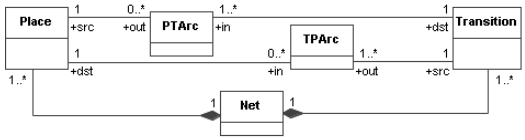
μ_0 :



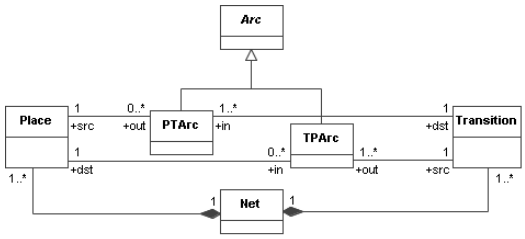
μ_1 :



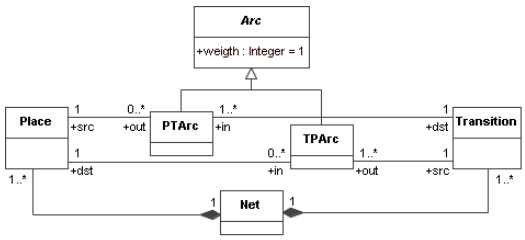
μ_2 :



μ_3 :



μ_4 :



μ_5 :

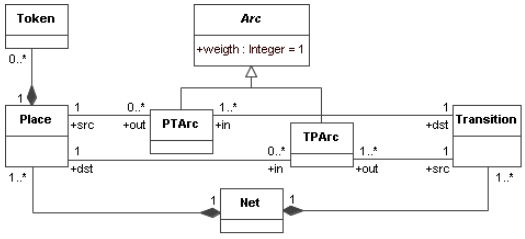


Fig. 1. Petri net metamodel evolution.

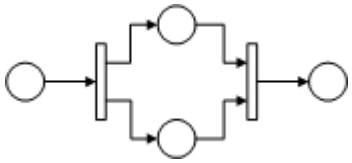


Fig. 2. Simple Petri net.

between places and transitions explicit. The extraction of `PTArc` and `TPArc` yields μ_2. This step might be useful if we want to annotate metaclasses with means for graphical or textual description in order to assist automatic tool generation. As `PTArc` and `TPArc` both represent arcs, we state this in μ_3 with a generalisation `Arc`. In an extended Petri net formalism, arcs might be annotated with weights. We can easily reproduce this extension by introducing a new attribute `weight` in μ_4. Until now, we cover only static aspects of Petri nets. To model dynamic aspects, places need to be marked with tokens as captured in μ_5.

Metamodel evolution is usually performed manually by stepwise adaptation. In this paper, we provide a theoretical basis to study the effects of metamodel evolution in terms of metamodel relations. We employ well-defined evolutionary steps for metamodels compliant to OMG's Meta Object Facility (MOF) [3]. The steps are implemented as transformations in QVT Relations, the relational part of OMG's Query-View-Transformation language [4]. Each step forms a *metamodel adaptation* and is classified according to its semantics- and instance-preservation properties. This work is mainly inspired by the ideas of object-oriented refactoring [5–7] and grammar adaptation [8, 9].

Co-evolution. Models need to co-evolve in order to remain compliant with the metamodel [2]. Without co-evolution, these artefacts become invalid.

Example 2 (Petri net model co-evolution). Fig. 2 contains a simple example of a Petri net. Models of this Petri net compliant to metamodels introduced in Example 1 are given in Figure 3. These models co-evolve with their metamodels. While the first model ι_0 is compliant to μ_0 and μ_1, new metaclasses in μ_2 enforce new instance objects in ι_1, which complies with μ_3, too. The introduction of `weight` in μ_4 necessitates the introduction of default values in the corresponding model ι_2. This model is also an instance of μ_5 because it provides an empty marking.

Like metamodel evolution, co-evolution is typically performed manually. This is an error-prone task leading to inconsistencies between the metamodel and related artefacts. From the fields of software architecture and language definition, we learnt that these inconsistencies usually lead to irremediable erosion where artefacts are not longer updated [2]. In this paper, we aim at automatic co-evolution steps deduced from well-defined evolution steps [10]. This *co-adaptation* prevents inconsistencies and metamodel erosion.

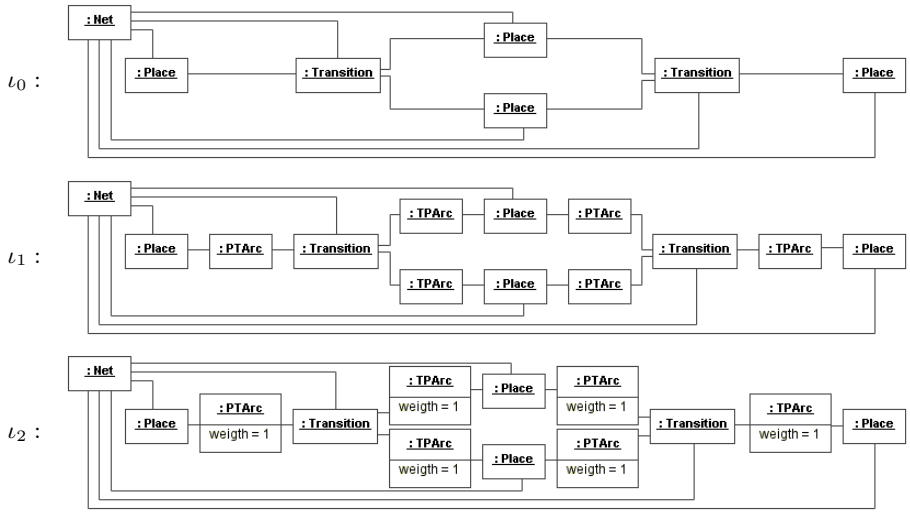


Fig. 3. Petri net instance co-evolution.

Transformational approach. In this paper, we propose a transformational approach to assist metamodel evolution by stepwise adaptation. Transformational metamodel adaptation has several advantages over manual ad hoc adaptation. First, changes become explicit. Thus, transformations provide documentation and traceability. Second, we state several preservation properties of transformations. This allows one to qualify an adaptation according to semantics- or instance-preservation. The co-adaptation of models is achieved automatically by co-transformations. Finally, adaptation scripts are pieces of software on their own. They can be reused in similar adaptation scenarios or be modified to alter adaptation decisions. Generalisations of those scripts define new transformations.

Structure of the paper. In Section 2, the origins of our work, i.e. object-oriented refactoring and language engineering, are discussed. In Section 3, we present a set of binary metamodel relations. Based on these relations, we define several forms of semantics- and instance-preservation properties for metamodel transformations. In Section 4, we develop a set of QVT Relations that assist the evolution of MOF 2.0 compliant metamodels. In Section 5, we address co-evolution of models. In Section 6, we discuss the benefits of a transformational approach in the context of metamodel design, metamodel implementation, metamodel maintenance, and metamodel recovery. The paper is concluded in Section 7.

2 Background

In this section, we discuss the origins of our work. These are the refactoring of object-oriented software and the adaptation of context-free grammars.

2.1 Object-oriented refactoring

Refactoring object-oriented code. Nowadays, software refactoring [11] is a common practice. It forms a central concept for agile development processes, e.g. eXtreme Programming [12] and Rational Unified Process [13]. As the first author, Opdyke formalised refactorings for object-oriented frameworks concerned with behaviour preservation [5]. Roberts carried on these ideas and designed the Refactoring Browser for Smalltalk programs [6]. In another thesis, Bravo developed a method for automatic detection of design flaws in object-oriented software [14]. These *bad smells* advise refactorings between which a user can choose. Fowler et al. captured these ideas and common refactorings in a practical guide to improve the design of object-oriented code [7].

The refactoring of an object-oriented program might result in the evolution of the schema for persistent data. Thus, persistent data needs to co-evolve. The problem of schema evolution was tackled for object-oriented database management systems [15, 16] as well as for object-oriented programs [17, 18].

Refactoring UML class diagrams. Most examples in the book by Fowler et al. are illustrated by UML class diagrams. Boger et al. extended the idea of refactoring to UML models and developed a refactoring browser for UML diagrams [19]. Sunye et al. presented behaviour preserving refactorings for UML class and interaction diagrams [20]. Then, Markovic took OCL annotations in UML class diagrams into account [21]. He presented a set of QVT Relations for the refactoring of OCL annotated class diagrams. All these works address software development. Before, refactoring techniques dealt mainly with implementation code. Now, refactoring can be applied in the design of object oriented software, too. In the tradition of code refactoring, UML diagram refactoring is concerned with behaviour preservation.

MOF compliant metamodels are closely related to UML class diagrams. Since metamodels describe the structure of models, behaviour-preservation properties do not characterise metamodel refactoring accordingly. In this paper, we discuss semantics-preservation for such structural descriptions. We define semantics-preservation properties in terms of modelling concepts of a metamodel and its set of possible instances. Instance-preservation properties are useful to characterise co-evolution problems. Moreover, pure refactoring is insufficient to assist metamodel evolution. We extend structural metamodel refactoring with construction and destruction operators. Thereby, we rely on ideas from grammar engineering. These are discussed next.

2.2 Grammar engineering

Grammar adaptation. In their efforts to establish an engineering discipline for grammarware [22], Klint et al. suppose a transformational setting for stepwise grammar adaptation as a central concept. In his works, Lämmel gives a formalisation for this approach [8]. He develops several relations between grammars to characterise a framework of grammar transformations by its preservation properties. The framework proved to be valuable for the semi-formal recovery of a VS-Cobol-II grammar [23, 9]. Besides work on generic refactorings [24, 25], Lämmel suggests the approach to be applicable to any structure description formalism, e.g. algebraic type declarations or UML class diagrams. In this paper, we follow this suggestion and adopt these ideas for metamodels. We combine these ideas with object-oriented refactoring techniques to provide a transformation library for stepwise metamodel adaptation.

Co-adaptation. Co-adaptation is a well-known problem in grammar engineering. In format evolution, documents need to co-evolve with evolving structure descriptions [26]. Grammar transformation rules need to be migrated after grammar extension [27]. Lämmel stated the problem of coupled transformations in a more general context [10].

2.3 Metamodel evolution

Metamodel evolution and coupled co-evolution of other software artefacts like constraints, transformation rules, and models are well-known problems in model-driven software development [2, 28]. Hößler et al. propose a generic instance model to handle evolution on all meta levels [29]. Other approaches suppose difference models as a solution to handle co-evolution [30, 31]. Metamodels are changed manually. Then, the difference to the last version is calculated. This difference model is used to derive automatic transformations for instance co-evolution.

In contrast to these approaches, we envision a transformational setting for stepwise metamodel adaptation. Each transformation implements a typical adaptation step typically performed manually. We classify these transformations according to preservation properties. Thus, the effect of each adaptation step is made explicit. Differences between metamodels are traceable without calculation. For each transformation, we provide corresponding co-transformations. Thereby, we provide instant co-adaptation of models. At each step these models conform to the actual metamodel version.

3 Preservation properties

We are interested in preservation properties of metamodel transformations. We generalise ideas from grammar engineering [8] and define various metamodel relations starting from equivalence. In a next step, we employ these relations to define miscellaneous forms of semantics- and instance-preservation. Finally, we point out the correlation between semantics- and instance-preservation.

3.1 Metamodels

Though we deal particularly with MOF 2.0 metamodels in this paper, we do not rely on a concrete metamodel formalism in this section. Generally, the set of all metamodels conforming to a given metamodel formalism M is denoted as:

$$\mathcal{M}_M := \{\mu \models M\}$$

We use $\mathcal{C}_M(\mu)$ to denote the *concepts* defined by a metamodel $\mu \in \mathcal{M}_M$. For MOF 2.0 [3], we treat qualified names of non-abstract metaclasses as those concepts:

Definition 1 (MOF 2.0 concepts). *The set of* concepts $C_{MOF}(\mu)$ *defined by a MOF 2.0 compliant metamodel $\mu \in \mathcal{M}_{MOF}$ is the result of the OCL query*

```
Class.allInstances() ->
      reject(c | c.isAbstract) ->
        collect(c | c.qualifiedName)
```

The set of all *metamodel instances* conforming to a metamodel μ is denoted as:

$$\mathcal{I}(\mu) := \{\iota \models \mu\}$$

This set might be restricted to those instances relying only on a given set of concepts $\mathcal{C}$:

$$\mathcal{I}_{\mathcal{C}}(\mu) \subseteq \mathcal{I}(\mu)$$

Example 3 (Instance set restriction). In Figure 1, it holds

$$\mathcal{C}_{MOF}(\mu_4) = \{\texttt{Net}, \texttt{Place}, \texttt{Transition}, \texttt{PTArc}, \texttt{TPArc}\}$$
$$\mathcal{C}_{MOF}(\mu_5) = \mathcal{C}_{MOF}(\mu_4) \cup \{\texttt{Token}\}$$

The restriction of $\mathcal{I}(\mu_5)$ to instances only relying on concepts in $\mathcal{C}_{MOF}(\mu_4)$ yields all models not instantiating `Token`. These are exactly the instances of μ_4, i.e. it holds

$$\mathcal{I}_{\mathcal{C}_{MOF}(\mu_4)}(\mu_5) = \mathcal{I}(\mu_4)$$

Note that the suggested notion can be applied to a wider range of metamodel formalisms. For example, $\mathcal{M}_G$ might be the set of context-free grammars with $\mathcal{C}_G(\gamma)$ yielding the nonterminals occurring in a grammar γ.

3.2 Metamodel relations

We now define some relations between metamodels. The metamodels presented in Figure 1 (cf. Example 1) exemplify these relations.

Definition 2 (Equivalence ($\equiv$)). *$\mu_1 \in \mathcal{M}$ and $\mu_2 \in \mathcal{M}$ are equivalent ($\mu_1 \equiv \mu_2$) iff:*

1. $\mathcal{I}(\mu_1) = \mathcal{I}(\mu_2)$.

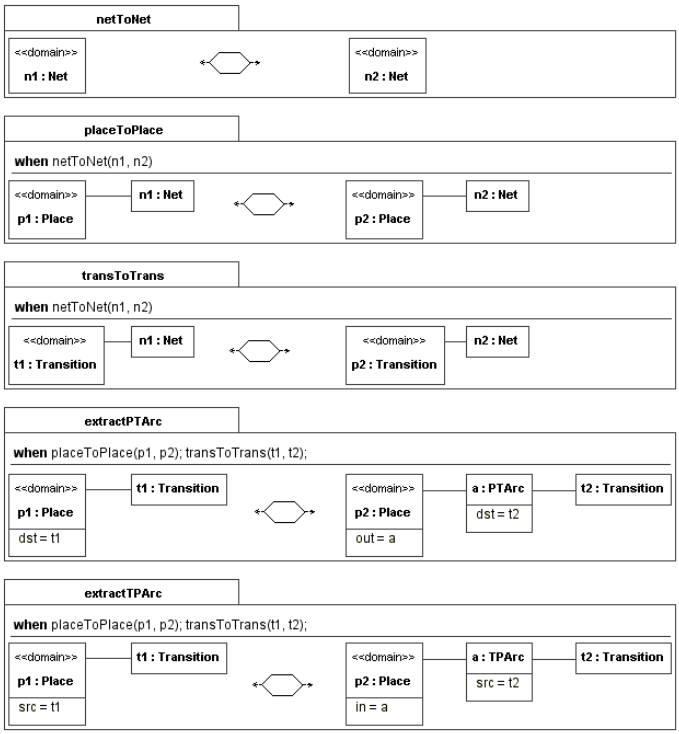


Fig. 4. Bidirectional transformation between model instances of μ_1 and μ_2.

In Figure 1, it holds $\mu_2 \equiv \mu_3$, since the abstract generalisation does not affect the set of instances. A less strict definition of equivalence can be obtained by claiming a bijective mapping between instance sets instead of equality.

Definition 3 (Variation relation ($\equiv_\varphi$)). *$\mu_1 \in \mathcal{M}$ and $\mu_2 \in \mathcal{M}$ are variants modulo φ ($\mu_1 \equiv_\varphi \mu_2$) iff:*

1. *$\varphi : \mathcal{I}(\mu_1) \to \mathcal{I}(\mu_2)$ is a bijective function.*

Variation is useful to characterise extraction and inlining of properties. For example, $\mu_1 \equiv_\varphi \mu_2$ applies to Figure 1 where φ is presented in Figure 4 as a set of QVT Relations. Equivalence and variation both relate metamodels with isomorphic sets of instances. This is often too restrictive to characterise related metamodels. We now define some relations between metamodels with non-isomorphic instance sets.

Definition 4 (Sub-metamodel relation ($\overset{\leq}{\equiv}$)). *$\mu_1 \in \mathcal{M}$ is a sub-metamodel of $\mu_2 \in \mathcal{M}$ ($\mu_1 \overset{\leq}{\equiv} \mu_2$) iff:*

1. *$\mathcal{C}(\mu_1) \subseteq \mathcal{C}(\mu_2)$,*

2. $\mathcal{I}(\mu_1) = \mathcal{I}_{\mathcal{C}(\mu_1)}(\mu_2)$.

A sub-metamodel offers only some of the concepts of its super-metamodel. The instance set of the super-metamodel is restricted to those instances, only instantiating concepts offered by the sub-metamodel. A sub-metamodel lacks only instances of its super-metamodel because of the lack of concepts. For example, in Figure 1, it holds $\mu_4 \overset{\leq}{\equiv} \mu_5$. A new metaclass `Token` is defined in μ_5. Instances of μ_5 instantiating this metaclass cannot be instances of μ_4. On the other hand, all instances of μ_5 not instantiating the new metaclass are instances of μ_4.

A super-metamodel allows for more instances by providing more concepts. In contrast, enrichment and extension are concerned with metamodels that provide the same concepts.

Definition 5 (Enrichment relation ($\overset{\subseteq}{\equiv}$)). *$\mu_2 \in \mathcal{M}$ is richer than $\mu_1 \in \mathcal{M}$ ($\mu_1 \overset{\subseteq}{\equiv} \mu_2$) iff:*

1. $\mathcal{C}(\mu_1) = \mathcal{C}(\mu_2)$,
2. $\mathcal{I}(\mu_1) \subseteq \mathcal{I}(\mu_2)$.

A metamodel is richer than another one, if it has at least the same instance set and the same set of concepts. For MOF compliant metamodels, this is useful to characterise generalisation and restriction of properties. In Figure 1, it holds $\mu_1 \overset{\subseteq}{\equiv} \mu_0$. Both metamodels define the same metaclasses and all Petri net models conforming to μ_1 conform also to μ_0. Because μ_0 allows to model a Petri net without any transition or place, μ_0 is richer than μ_1 which restricts its instances to have at least one place and transition.

Definition 6 (Extension relation ($\overset{\subseteq}{\equiv}_\varphi$)). *$\mu_2 \in \mathcal{M}$ extends $\mu_1 \in \mathcal{M}$ by φ ($\mu_1 \overset{\subseteq}{\equiv}_\varphi \mu_2$) iff:*

1. $\mathcal{C}(\mu_1) = \mathcal{C}(\mu_2)$,
2. $\varphi : \mathcal{I}(\mu_1) \to \mathcal{I}(\mu_2)$ *is an injective function.*

Extension does for enrichment what variation does for equivalence. The instances of the extended metamodel are not instances of the extension (as for enrichment), but they are mapped into the new instance set by an injection φ. In Figure 1, it holds $\mu_3 \overset{\subseteq}{\equiv}_\varphi \mu_4$. Instances of μ_3 are no longer instances of μ_4 because they do not provide the new mandatory weight for arcs. Nevertheless, those instances can be easily mapped into the new instance set by providing a default weight.

The relations presented so far are useful to characterise metamodel adaptation. To characterise co-adaptation, only the instance sets of two metamodels have to be considered. Therefore, we define two more metamodel relations.

Definition 7 (Instance-preservation relation ($\overset{\sqsubseteq}{\equiv}$)). *$\mu_2 \in \mathcal{M}$ preserves instances of $\mu_1 \in \mathcal{M}$ ($\mu_1 \overset{\sqsubseteq}{\equiv} \mu_2$) iff:*

1. $\mathcal{I}(\mu_1) \subseteq \mathcal{I}(\mu_2)$.

Definition 8 (Instance-variation relation ($\overset{\sqsubseteq}{\equiv}_\varphi$)). *$\mu_2 \in \mathcal{M}$ varies instances of $\mu_1 \in \mathcal{M}$ by φ ($\mu_1 \overset{\sqsubseteq}{\equiv}_\varphi \mu_2$) iff:*

1. *$\varphi : \mathcal{I}(\mu_1) \to \mathcal{I}(\mu_2)$ is an injective function.*

Again, we distinguish strict preservation and injection. The metamodel relations discussed so far correlate with both of these new relations.

Theorem 1 (Correlation of metamodel relations).

1. $\equiv \subset \overset{\sqsubseteq}{\equiv}$,
2. $\overset{\leq}{\equiv} \subset \overset{\sqsubseteq}{\equiv}$,
3. $\overset{\subseteq}{\equiv} \subset \overset{\sqsubseteq}{\equiv}$,
4. $\equiv_\varphi \subset \overset{\sqsubseteq}{\equiv}_\varphi$,
5. $\overset{\subseteq}{\equiv}_\varphi \subset \overset{\sqsubseteq}{\equiv}_\varphi$.

These correlations result directly from the definitions given in this section. For example, it holds $\mathcal{I}(\mu_1) = \mathcal{I}_{\mathcal{C}(\mu_1)}(\mu_2)$ for $\mu_1 \overset{\leq}{\equiv} \mu_2$ accordingly to Definition 4. This implies $\mathcal{I}(\mu_1) \subseteq \mathcal{I}(\mu_2)$ since a restriction of an instance set subsets the complete instance set (ref. Section 3.1). Thus, it follows $\mu_1 \overset{\sqsubseteq}{\equiv} \mu_2$ accordingly to Definition 7. Other correlations stated in Theorem 1 can be proven in a similar way.

3.3 Semantics-preservation

We can now employ the metamodel relations defined so far in this section to define properties concerning semantics-preservation for metamodel transformations. We model a metamodel transformation as a relation between metamodels.

Definition 9 (Semantics-preservation properties). *A metamodel relation $R \subset \mathcal{M} \times \mathcal{M}$ is*

1. strictly semantics-preserving *iff* $R \subseteq \equiv$,
2. semantics-preserving modulo variation φ *iff* $R \subseteq \equiv_\varphi$,
3. introducing *iff* $R \subseteq \overset{\leq}{\equiv}$,
4. eliminating *iff* $R \subseteq \overset{\leq}{\equiv}^{-1}$,
5. increasing *iff* $R \subseteq \overset{\subseteq}{\equiv}$,
6. decreasing *iff* $R \subseteq \overset{\subseteq}{\equiv}^{-1}$,
7. increasing modulo variation φ *iff* $R \subseteq \overset{\subseteq}{\equiv}_\varphi$,
8. decreasing modulo variation φ *iff* $R \subseteq \overset{\subseteq}{\equiv}^{-1}_{\varphi^{-1}}$.

A metamodel transformation is *strictly semantics-preserving* iff it results always in an equivalent metamodel. It is *semantics-preserving modulo variation* iff it results always in a variant of the original metamodel. A transformation is *introducing* (respectively *eliminating*) iff it results always in a super-metamodel (respectively sub-metamodel) of its input. It is *increasing* (respectively *decreasing*) iff its result is always richer (respectively less rich) than its input metamodel. The transformation is *introducing modulo variation* iff its result is always an extension of the original metamodel. Respectively, it is *eliminating modulo variation* iff the original metamodel is always an extension of the result.

In the next section of the paper, we present a library of transformations between MOF compliant metamodels which separate these preservation properties.

3.4 Instance-preservation

Semantics-preservation properties characterise metamodel transformations accordingly to the offered modelling concepts and possible instances. With respect to the need for co-adaptation, we need to characterise the preservation of existing instances. We now define some properties concerning instance-preservation.

Definition 10 (Instance-preservation properties). *A metamodel relation* $R \subset \mathcal{M} \times \mathcal{M}$ *is*

1. strictly instance-preserving *iff* $R \subseteq \overset{\sqsubset}{\equiv}$,
2. partially instance-preserving *iff* $R \subseteq \overset{\sqsubset}{\equiv}^{-1}$,
3. instance-preserving modulo variation φ *iff* $R \subseteq \overset{\sqsubset}{\equiv}_{\varphi}$,
4. partially instance-preserving modulo variation φ *iff* $R \subseteq \overset{\sqsubset}{\equiv}^{-1}_{\varphi^{-1}}$,

A transformation is *strictly instance-preserving* iff its result preserves always the instances of the original metamodel. It is *partially instance-preserving* iff all instances of the resulting metamodel are always preserved instances of the input metamodel. The transformation is *instance-preserving modulo variation* iff its result always varies the instances of the original metamodel. It is *partially instance-preserving modulo variation* iff all instances of the resulting metamodel are always varied instances of the input metamodel.

Due to Theorem 1, semantics-preservation properties imply a certain instance-preservation property.

Theorem 2 (Correlation of preservation properties). *A metamodel relation* $R \subset \mathcal{M} \times \mathcal{M}$ *is*

1. strictly instance-preserving *if it is* strictly semantics-preserving, introducing, *or* increasing*;*
2. partially instance-preserving *if it is* eliminating, *or* decreasing*;*
3. instance-preserving modulo variation φ *if it is* semantics-preserving modulo variation φ, *or* increasing modulo variation φ*;*
4. partially instance-preserving modulo variation φ *if it is* decreasing modulo variation φ.

Adaptation	Semantics-preservation	Inverse
Refactoring		
rename element	preserving modulo variation	rename element
move property	preserving modulo variation	move property
extract class	preserving modulo variation	inline class
inline class	preserving modulo variation	extract class
association to class	preserving modulo variation	class to association
class to association	preserving modulo variation	association to class
Construction		
introduce class	introducing	eliminate class
introduce property	increasing modulo variation	eliminate property
generalise property	increasing	restrict property
pull property	increasing modulo variation	push property
extract superclass	introducing	flatten hierarchy
Destruction		
eliminate class	eliminating	introduce class
eliminate property	decreasing modulo variation	introduce property
restrict property	decreasing	generalise property
push property	decreasing modulo variation	pull property
flatten hierarchy	eliminating	extract superclass

Table 1. Semantics-preservation properties of presented transformations.

In the remainder of the paper, we will use instance-preservation properties to identify co-adaptation scenarios. There are two cases where co-adaptation is necessary. First, a variation φ hints a co-adaptation. Second, partial instance-preservation might be extended to complete instance-preservation.

4 Transformational adaptation of MOF compliant metamodels

4.1 Overview

In this section, we present a transformation library for the stepwise adaptation of MOF compliant metamodels. The transformations separate semantics-preservation properties introduced in the last section. Thereby, we can distinguish three kinds of transformations. First, we identify transformations for semantics-preserving (by variation) refactoring. Second, introducing and increasing transformations assist metamodel construction. Finally, eliminating and decreasing transformations allow for metamodel destruction. Table 1 groups the transformations presented in this section by this classification. It also gives semantics-preservation properties and inverse transformations.

We give the transformations as QVT Relations [4]. Thereby, we use its graphical notation. In the remainder of this section, we discuss each transformation in detail. We start with constructors and accordant destructors. Since most transformations for refactoring rely on construction and destruction, they are presented subsequently.

4.2 Construction and destruction

Introduce/eliminate class. Introducing a new metaclass into a package is a common step in metamodel construction [5]. Figure 5 shows an implementation of this adaptation. As a precondition, elements in the package must be distinguishable and the package must not own the metaclass already. Afterwards, the package owns the metaclass while elements in the package must stay distinguishable.

In general, this adaptation is *introducing*. The new metaclass offers a new concept and allows thereby for new instances. Instances of other metaclasses are not affected. If the new metaclass is abstract, the adaptation is *strictly semantics-preserving* since the set of instances persists.

The QVT Relation presented in Figure 5 can also be applied right-to-left. This way, a metaclass is eliminated from a package. Afterwards, elements in the package are still distinguishable and the package does not own the metaclass anymore. As a precondition, the metaclass must not have any subclasses and it must not be referred by other classes. This ensures the elimination is reversible by introducing the metaclass. Thus, the adaptation is *strictly semantics-preserving* particularly for abstract metaclasses and *eliminating* in general.

Introduce/eliminate property. Introducing a new property into a metaclass is another common metamodel construction [5]. Its implementation as shown in Figure 5 is quite similar to the one for introducing a new metaclass. As a pre- and postcondition, elements from the namespace defined by the metaclass must be distinguishable. Thus, the new property is distinguishable from other properties owned by the metaclass itself or its superclasses.

Only for the particular case of an abstract metaclass without non-abstract subclasses, this adaptation is *strictly semantics-preserving*. In general, it is *increasing modulo variation* since the concerned metaclass and its subclasses allow for new instances. Only instances of the concerned metaclass and its subclasses are affected. Variation is needed if the property introduced is obligatory. This can be achieved by providing a default value for the property introduced.

Again, the relation can be applied right-to-left eliminating a property from a metaclass. Once more, *strict semantics-preservation* holds only for an abstract metaclass without non-abstract subclasses. The destruction is generally *decreasing modulo variation*. Variation can be performed by eliminating all slots of the property.

Generalise/restrict property. Property generalisation and restriction are two adaptations we adopt from grammar adaptation [8]. A property can be generalised or restricted in terms of its multiplicity and its type. As the implementation in Figure 5 states, types of association ends must not be generalised since this would affect both association ends.

Generalising a property is an *increasing* adaptation. Without offering new concepts, it allows for new instances. Old instances are not affected, so co-adaptation is not needed. Contrarily, restricting a property is *decreasing*. Some

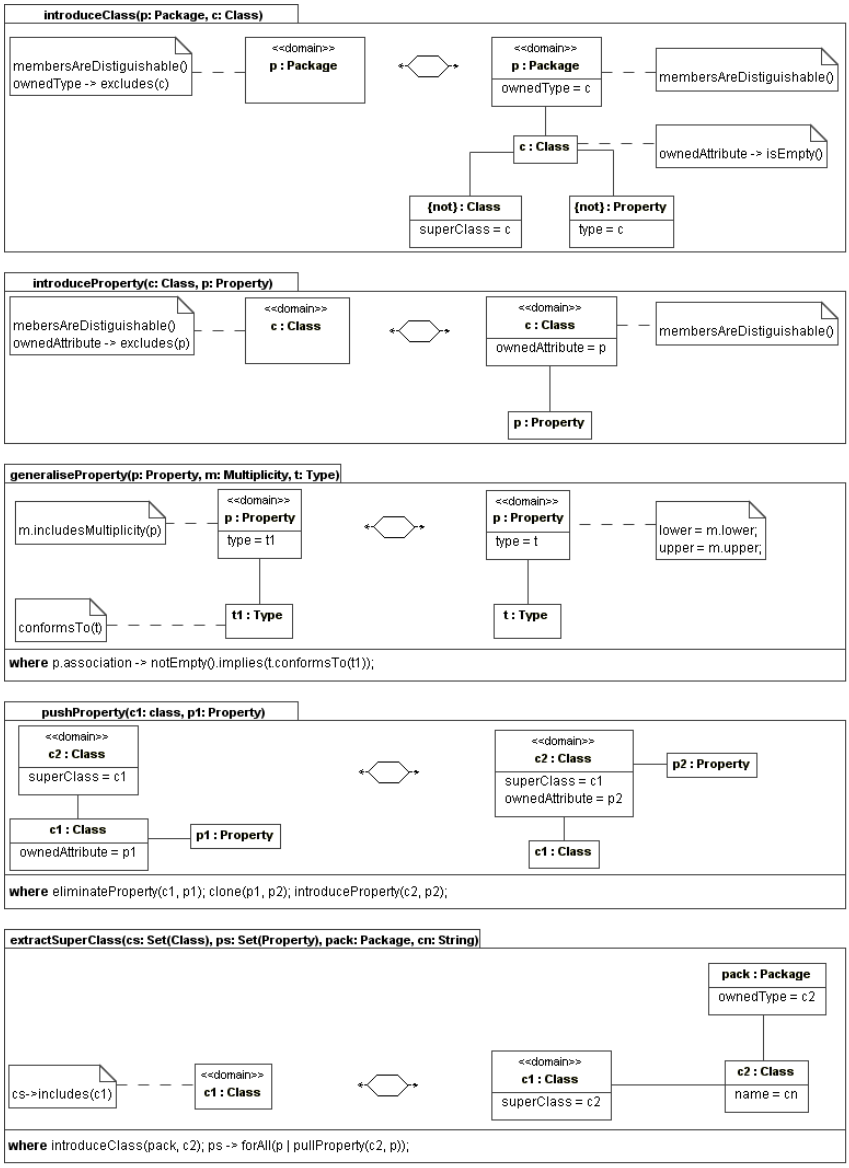


Fig. 5. QVT Relations implementing metamodel construction and destruction.

of the old instances will meet the restriction. Other instances need to be co-adapted. Restricting the upper bound of the multiplicity requires a selection of certain values. This can be achieved automatically. Restricting the lower bound requires new values for the property usually provided manually. Restricting the type of a property requires type conversion for each value.

Pull/push property. Pushing a property into subclasses respectively pulling a property into a subclass are well known object-oriented refactorings [5, 7]. An implementation realising both of them is given in Figure 5. To push a property into subclasses, the property is eliminated in the superclass[1] and a clone of it is introduced in each subclass.

Combining decreasing elimination and increasing introduction, this adaptation is generally *decreasing modulo variation*. Only for abstract superclasses, it is *strictly semantics-preserving*. Otherwise, instances of the superclass are affected by elimination. Instances of subclasses are not affected. Co-adaptation can be performed as for property elimination.

As the inverse adaptation, pulling a property into the superclass can be performed by right-to-left execution. Thereby, the property is introduced into the superclass and its clones in the subclasses are eliminated.

For non-abstract superclasses, the adaptation is *increasing modulo variation*. In this general case, the superclass allows for new instances. Co-adaptation is needed only for old instances of the superclass. As for property introduction, this can be done by providing a default value.

Extract superclass/flatten hierarchy. Superclass extraction and its counterpart of hierarchy flattening are other well-known object-oriented refactorings [5, 7]. Figure 5 gives an implementation for superclass extraction. The transformation extracts a set of properties common to a set of classes into a new superclass. This metaclass is introduced into a specified package. Then, each property is pulled into the new superclass. In another implementation the new superclass might be integrated into the class hierarchy.

This adaptation would be *strictly semantics-preserving* for an abstract superclass. However, to use the implementation right-to-left to flatten hierarchy we abstain from this restriction. Thus, preservation properties can be derived from class introduction. The adaptation is *introducing* since the superclass offers a new concept. Since instances of the subclasses are not affected, no co-adaptation is needed. On the other hand, flatten hierarchy by eliminating a superclass and pushing all its properties into the subclasses is an *eliminating* adaptation. Instances of the subclasses are preserved.

4.3 Refactoring

Rename element. Element renaming is a very simple and common refactoring [5, 7]. In Figure 6, we present a QVT implementation. As a precondition,

[1] Here, `eliminateProperty` refers a right-to-left execution of `introduceProperty`.

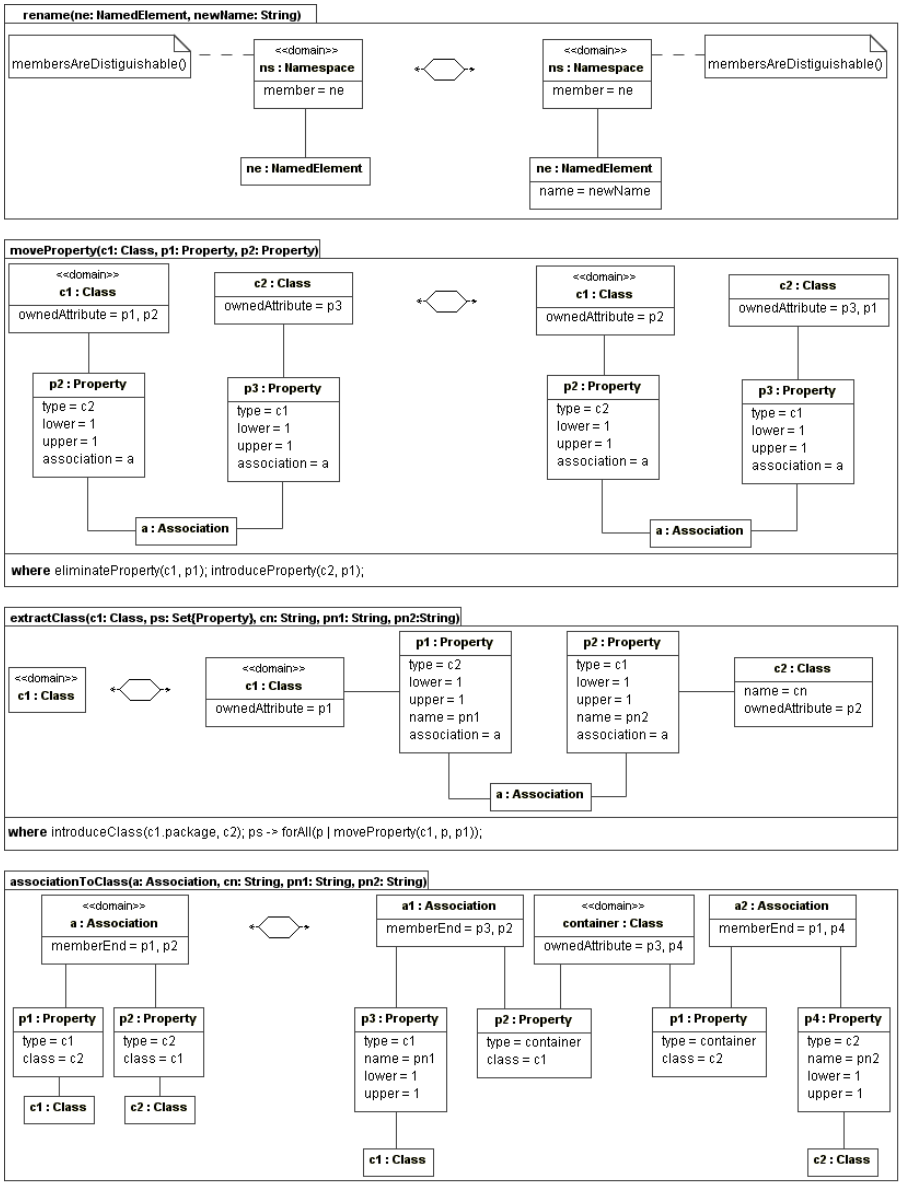


Fig. 6. QVT Relations implementing metamodel refactoring.

elements in each namespace of the element to rename need to be distinguishable. As a postcondition, this still holds after renaming.

The adaptation is *semantics-preserving modulo variation*. Co-adaptation is achieved automatically by a simple mapping from the old element to the renamed one.

Move property. Moving a property is another simple refactoring. In contrast to the refactoring presented by Fowler et al. [7], we follow Opdyke [5] by moving a property along a one-to-one association as shown in Figure 6. The implementation simply eliminates the property from the source metaclass and introduces it in the target metaclass.

This adaptation is *semantics-preserving modulo variation*. Instances of the affected metaclass can be automatically co-adapted by moving property values along the link between instances of source and target metaclasses.

Extract/inline class. Extraction and inlining are generic refactorings [24]. In object-oriented refactoring, properties are extracted along generalisation or delegation. We already mentioned extraction along generalisation as superclass extraction. Extraction along delegation is often referred to as class extraction [5, 7]. An implementation is given in Figure 6. To extract a set of properties, a new metaclass is introduced. Then, a one-to-one association between this container class and the affected metaclass is established. Finally, the properties are moved along this association into the new class. Extraction of an association between two classes into a new class is a similar refactoring. An implementation is also given in Figure 6.

Both extractions are *semantics-preserving modulo variation*. Instances of the affected metaclasses can be automatically co-adapted by instantiating the container class, linking the affected instances with this new instance, and moving property values into the container instance.

As the inverse transformations, class inlining is achieved by right-to-left execution. Both adaptations are *semantics-preserving modulo variation* as well. Co-adaptation is performed as right-to-left extraction co-adaptation.

5 Co-adaptation of models

In the last section, we presented metamodel transformations for stepwise metamodel adaptation. Thereby, we touched already on co-adaptation of models. We now discuss co-adaptation of models in detail.

5.1 Transformation patterns

Like metamodel adaptation, we exploit transformations to describe model co-adaptation. A co-transformation depends on its triggering metamodel transformation. Therefore, we describe co-transformations by transformation patterns.

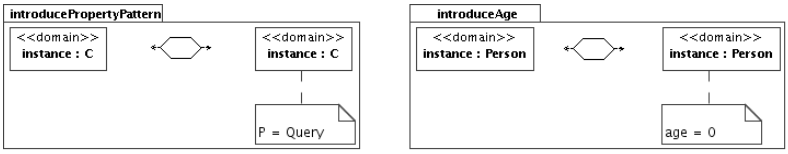


Fig. 7. Co-transformation pattern for property introduction.

A transformation pattern is a QVT Relation with parameters for metamodel elements, e.g. metaclasses or properties. A metamodel transformation instantiates a co-transformation pattern to derive a corresponding co-transformation.

Example 4 (Transformation pattern).

It its left part, Figure 7 shows a co-transformation pattern for property introduction. It contains three parameters: `C` for the affected metaclass, `P` for the introduced property, and `Query` for a OCL query specifying a value. These parameters are instantiated by a concrete property introduction. The right part of Figure 7 shows the resulting QVT Relation for introducing a property `age` into a metaclass `Person`.

5.2 Co-construction

Introduce property and *pull property* are the only constructors concerned with co-adaptation. Instances of the affected metaclass become invalid if a mandatory property is introduced. Therefore, co-adaptation needs to introduce a value for the new property into each instance of the metaclass. The according co-transformation pattern was given in Figure 7. The corresponding metamodel transformation can instantiate the pattern with the affected metaclass and the introduced property. Furthermore, the user has to specify an OCL query providing a value for the introduced property.

Due to Theorem 2, all other constructors are *strictly instance-preserving*. Thus, co-adaptation is needless.

5.3 Co-refactoring

All refactoring transformations are *semantics-preserving modulo variation* and thereby *instance-preserving modulo variation*. We present co-transformation patterns for each transformation in Figure 8. For *rename element*, we give an implementation for property renaming. Implementations for the renaming of other elements are quite similar. To preserve the instances of the metamodel, instances of the renamed element are mapped onto instances of the the new named element.

Co-adaptation for *move property* preserves instances by moving values of the property along a link that instantiates the association from the adaptation.

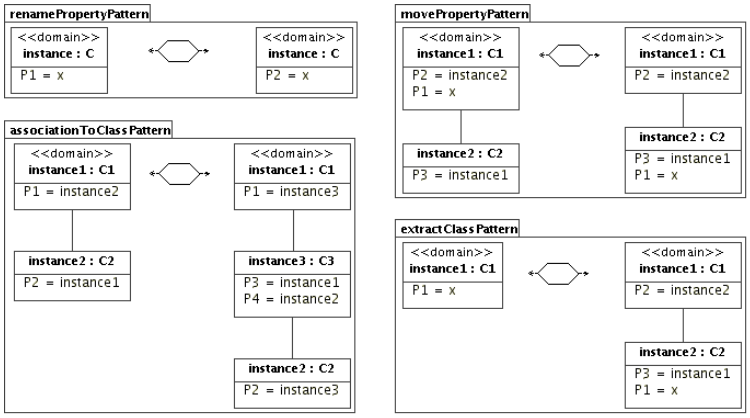


Fig. 8. Transformation patterns for model co-refactoring.

For *extract class*, an instance of the container class is created additionally. The same co-adaptation can be used right-to-left for *inline class*. Co-adaptation for *association to class* and its inverse is similar to this.

5.4 Co-destruction

Eliminate class and *flatten hierarchy* are *partially instance-preserving* destructors. Thus, some metamodel instances are preserved. These are all those metamodel instances that do not contain instances of the eliminated class. Co-adaptation for the remaining metamodel instances is achieved by removing instances of the class from these models. We deal with *eliminate property* and *push property* in a similar way. Both destructors are *partially instance-preserving by variation*. To co-adapt metamodel instances, slots of the eliminated property are removed from objects occurring in a model.

Automatic co-adaptation for the *instance-preserving* destructor *restrict property* is somewhat more difficult. If the upper bound of the property is restricted, metamodel instances containing objects that exceed this bound need to be co-adapted. This can be done automatically by removing elements until the upper bound is met. The user might assist this by an OCL query specifying preserved elements. For lower bound restrictions, new values are needed to co-adapt models containing objects that fall short of this bound. As for *introduce property*, new values are specified by an OCL query given by the user. Considering type restriction, things get even more complicated. A pragmatic approach is to remove all mistyped elements from objects in a co-adapted model. This might introduce lower bound shortfalls. New values might overcome this problem. Again, these values are specified by the user in terms of an OCL query. This query can take the mistyped elements into account. Thus, these elements might be casted to

the restricted type in a user defined way. In a somehow simplier approach, co-adaptation is completely deferred to the user. In this case, the user specifies in an OCL query new elements for affected slots in terms of old elements.

6 Applications

6.1 Metamodel design

The transformational approach facilitates a well-defined stepwise metamodel design. Starting from basic features, new features are introduced by construction. Exhaustive usage of this principle leads to an agile process. Refactoring enables generalisation of metamodel features. This way, common concepts become explicit. Construction allows to reuse these concepts by specialisation. Generalisation and specialisation permit a pattern-based metamodel design [32]. Furthermore, scripts of consecutive adaptation steps document design decisions. By changing particular steps, metamodel designers can alternate designs.

6.2 Metamodel implementation

Metamodel-dependent model processors, e.g. editors, compilers, simulators, debuggers, code generators, documentation generators, or pretty-printers, should be generated semi-automatically. This ensures conformance between tools and metamodels. Metamodel erosion is avoided [2]. Grammar-based tool generators are well known for specification languages relying on restricted grammar formalisms, e.g. by prohibiting left recursion. To use such generators, language engineers need to develop compliant versions of their grammars. The same phenomenon is observable for metamodel-based tool generators and metamodelling tools in general. The MOF specification [3] itself provides two different metametamodels, i.e. Essential MOF (EMOF) as a minimal specification and Complete MOF (CMOF) as an extension to the former one. Several metamodelling tools rely on their own metametamodels, e.g. the Eclipse Modeling Framework on Ecore [33], Kermeta on the Kermeta language [34], and the ATLAS Transformation Language on KM3 [35]. To use these tools, metamodels need to comply to these metametamodels. Refactoring ensures equivalence to original versions. According to the tool, one can switch between metamodel variations. Co-adaptation ensures preservation of models.

6.3 Metamodel refinement

In [36], Staikopoulos and Bordbar propose a method called *One Step Refinement* for bridging technical spaces or domains. The authors suggest a successive refinement of the target metamodel to meet a richer source metamodel. In each step, the target model is extended by a new concept which is constructed out of old concepts. The process is repeated until an extension is created, such that all concepts of the source metamodel can be easily mapped into concepts of the

extended target metamodel. As a side effect, an overall transformation from the source metamodel to the original target metamodel is derived.

The transformational setting presented in this paper assist this approach. The target metamodel is extended by introducing new concepts while the overall transformation can be derived from co-transformations.

6.4 Metamodel maintenance

Like other software, metamodels are subject to maintenance. This includes remedying defects, reengineering to improve design, and meeting changes in requirements. Metamodel maintenance also benefits from a transformational setting. Erroneous features can be corrected by construction and destruction. Due to the local character of transformations, other features stay unchanged. Refactoring provides for reengineering a metamodel design without introducing defects. Scripts of adaptation steps can be adapted and reused in a similar context. Generalisation of those scripts leads to definitions for new transformations. This can be used to implement common redesigns, e.g. subsequent introduction of patterns [32]. Construction and destruction assist adjustment to changing requirements.

6.5 Metamodel recovery

Often, language knowledge resides only in language-dependent tools or semiformal language references. Language recovery is concerned with the derivation of a formal language specification from such sources. This comprises both, grammar recovery [23] and metamodel recovery [2]. For grammar recovery, a transformational approach already proved to be valuable [8]. In a similar way, the presented transformational setting assists metamodel recovery.

7 Conclusion

Contribution. In this paper, we combined ideas from object-oriented refactoring and grammar adaptation to provide a basis for automatic metamodel evolution. We defined several relations between metamodels to characterise metamodel evolution. These were employed to deduce properties for semantics- and instance-preservation of metamodel transformations. We do not restrict metamodel relations and preservation properties to object-oriented, e.g. MOF compliant, metamodels. The notions presented are useful for other structural descriptions, e.g. grammars, as well. Furthermore, we presented a set of QVT Relations to assist automatic metamodel evolution by stepwise adaptation. The transformations were classified in three groups accordingly to their preservation properties, namely refactoring, construction, and destruction. Implementation and preservation properties were discussed for each transformation. The problem of co-evolution was stated. It was shown how automatic co-adaptation can solve this problem for metamodel instances.

To prove the relevance of our work, we proposed applications in metamodel design, metamodel implementation, metamodel refinement, and metamodel recovery.

Future work. From a theoretical point of view, we will focus our ongoing research on the evolution and co-evolution of constraints and transformation rules. Another interesting topic will be the question when a metamodel needs to be adapted. In our practical work, we concentrate on two prototypical implementations. Furthermore, we employ our transformational approach for metamodel-based development of domain-specific languages. In a practical setting, we need to deal with several versions of metamodels and models. This is another interesting research topic. We will now discuss each of the topics mentioned more in detail.

Evolution of constraints and transformations. In this paper, we were concerned with co-adaptation of models. Constraints and transformation rules also co-evolve triggered by metamodel evolution. Furthermore, constraints and transformation rules evolve on their own. Constraints might be adapted to be more or less restrictive. Transformation patterns might be adapted to match more or less instances. Again, a library supporting automatic stepwise adaptation while guaranteeing certain preservation properties will be valuable. For metamodel adaptation, it would be interesting to express co-adaptation in terms of this library.

The smell of structure. As stated in the subtitle of the book by Fowler et al. [7], one of the main goals in object-oriented refactoring is to improve the design of existing code. Often, a refactoring is indicated by a bad smell [14], e.g. duplicate code, long methods, or message chains. These smells are almost all code-centric. For metamodel evolution, we are concerned with the question if structure can indicate an adaptation. How does structure smell? Works on object-oriented metrics [37–39] might be a good starting point for further research.

Tool support. Our first attempts to provide a prototypical implementation relying on ModelMorf [40] failed due to missing support for in-place transformations. We are now working with an implementation of QVT's relational part provided by ikv [41], an industrial partner of our research group. The implementation was applied in a project between ikv and the biggest consumer electronic vendor in Korea in the area of embedded systems for in-place model to model transformations. The implementation was ported by ikv to the Eclipse Modeling Framework (EMF) [33]. This enables us to make metamodel adaptation available for EMF.

Furthermore, we started a prototypical implementation providing metamodel adaptation for CMOF. This tool is implemented as an Eclipse plugin in Java. It is built upon a Java implementation of CMOF [42] and the Eclipse Language Toolkit.

For both tools, we envision four editing modes for metamodels: First, a *free mode* allows for arbitrary manual changes. Second, an *adaptation mode* allows

only for transformational adaptation. Third, a *construction mode* restricts the user to construction and refactoring. Finally, a *refactoring mode* is even more restrictive and allows only for refactoring.

DSL development. We started to employ an adaptive development process for domain-specific languages based on metamodels. In our work, we are concerned with languages from the domain of disaster management [43]. We are working on two case studies. The first study is concerned with the development of a new language. Metamodel adaptation is used to develop alternative designs, to meet requirement changes, and to maintain the language. The second study is concerned with language recovery. Metamodel adaptation is used to capture implicit language knowledge explicitly in a metamodel.

Versioning. In a practical setting, one has to deal with several metamodel versions. For each metamodel version different model versions need to be taken into account. Co-adaptation might affect all these model versions. On the other hand, only few versions of a model might indicate a metamodel adaptation leading to a new metamodel version. Finally, co-adaptation might join different model versions. Thus, further research is needed to integrate metamodel adaptation with versioning approaches [44].

Acknowledgement. This work is supported by grants from the DFG (German Research Foundation, Graduiertenkolleg METRIK). I am grateful to Eckardt Holz, Ralf Lämmel, Daniel Sadilek, and Markus Scheidgen for encouraging discussion, and for helpful suggestions. I am also thankful to Dirk Fahland, Lena Karg, and Falko Theisselmann for comments on earlier versions of this paper.

References

1. Object Management Group: MDA Guide Version 1.0.1. (2003)
2. Favre, J.M.: Meta-model and model co-evolution within the 3D software space. In: ELISA'03: Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications, Amsterdam, The Netherlands. (2003) 98–109
3. Object Management Group: Meta Object Facility Core Specification, version 2.0. (2006)
4. Object Management Group: MOF QVT Final Adopted Specification. (2005)
5. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1992)
6. Roberts, D.B.: Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1999)
7. Fowler, M., Beck, K., Brant, J., Opdyke, W.F., Roberts, D.B.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (1999)
8. Lämmel, R.: Grammar adaptation. In Oliveira, J.N., Zave, P., eds.: FME '01: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, Proceedings. Volume 2021 of Lecture Notes in Computer Science., Springer-Verlag (2001) 550–570

9. Lämmel, R., Wachsmuth, G.: Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment. Electronical Notes in Theoretical Computer Science **44** (2001)
10. Lämmel, R.: Coupled software transformations (extended abstract). In: SET '04: 1st International Workshop on Software Evolution Transformations, Proceedings. (2004) 31–35
11. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Software Eng. **30** (2004)
12. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)
13. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley (2004)
14. Bravo, F.M.: A Logic Meta-Programming Framework for Supporting the Refactoring Process. PhD thesis, Vrije Universiteit Brussel, Belgium (2003)
15. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. SIGMOD Rec. **16** (1987) 311–322
16. Nguyen, G.T., Rieu, D.: Schema evolution in object-oriented database systems. Data Knowl. Eng. **4** (1989) 43–67
17. Dmitriev, M.: Safe Class and Data Evolution in Large and Long-Lived Java Applications. PhD thesis, University of Glasgow (2001)
18. Meyer, B.: Schema evolution: Concepts, terminology, and solutions. Computer **29** (1996) 119–121
19. Boger, M., Sturm, T., Fragemann, P.: Refactoring browser for UML. In: NODe '02: International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, Revised Papers. Volume 2591 of Lecture Notes in Computer Science., Springer-Verlag (2002) 336–377
20. Sunye, G., Pollet, D., Traon, Y.L., Jezequel, J.M.: Refactoring UML models. In: UML'01: 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, Toronto, Canada. Volume 2185 of Lecture Notes in Computer Science., Springer-Verlag (2001) 134–148
21. Markovic, S., Baar, T.: Refactoring OCL annotated UML class diagrams. In Briand, L.C., Williams, C., eds.: MoDELS'05: Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica. Volume 3713 of Lecture Notes in Computer Science., Springer-Verlag (2005) 280–294
22. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Transactions on Software Engineering and Methodology (TOSEM) **14** (2005) 331–380
23. Lämmel, R., Verhoef, C.: Semi-automatic grammar recovery. Softw. Pract. Exper. **31** (2001) 1395–1438
24. Lämmel, R.: Towards generic refactoring. In: RULE '02: Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming, ACM Press (2002) 15–28
25. Heering, J., Lämmel, R.: Generic software transformations (extended abstract). In: Proceedings of the Software Transformation Systems Workshop. (2004)
26. Lämmel, R., Lohmann, W.: Format evolution. In Kouloumdjian, J., Mayr, H.C., Erkollar, A., eds.: RETIS '01: 7th International Conference on Re-Technologies for Information Systems, Proceedings. Volume 155 of books@ocg.at., OCG (2001) 113–134
27. Lohmann, W., Riedewald, G.: Towards automatical migration of transformation rules after grammar extension. In: CSMR '03: 7th European Conference on Soft-

ware Maintenance and Reengineering, Benevento, Italy, Proceedings, IEEE Computer Society (2003) 30–39
28. Deng, G., Lenz, G., Schmidt, D.C.: Addressing domain evolution challenges in software product lines. In Bruel, J.M., ed.: MoDELS'05: Revised Selected Papers of International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica. Volume 3844 of Lecture Notes in Computer Science., Springer-Verlag (2005) 247–261
29. Hößler, J., Soden, M., Eichler, H.: Coevolution of models, metamodels and transformations. In Bab, S., Gulden, J., Noll, T., Wieczorek, T., eds.: Models and Human Reasoning. Wissenschaft und Technik Verlag, Berlin (2005) 129–154
30. Hearnden, D.: Software evolution with the model driven architecture. (PhD Confirmation report)
31. Gruschko, B.: Towards structured revisions of metamodels and semi-automatic model migration. Position Paper for the Eclipse Modeling Symposium (2006)
32. France, R., Ghosh, S., Song, E., Kim, D.K.: A metamodeling approach to pattern-based model refactoring. IEEE Software **20** (2003) 52–58
33. Budinsky, F., Merks, E., Steinberg, D.: Eclipse Modeling Framework. 2nd edn. Addison-Wesley (2006)
34. Fleurey, F., Drey, Z., Vojtisek, D., Faucher, C.: Kermeta language. (2006)
35. Jouault, F., Bézivin, J.: KM3: A DSL for metamodel specification. In Gorrieri, R., Wehrheim, H., eds.: FMOODS'06: Proceedings of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy. Volume 4037 of Lecture Notes in Computer Science., Springer (2006) 171–185
36. Staikopoulos, A., Bordbar, B.: Bridging technical spaces with a metamodel refinement approach. A BPEL to PN case study. Electronical Notes in Theoretical Computer Science (2006)
37. Lorenz, M., Kidd, J.: Object-oriented software metrics: a practical guide. Object-Oriented Series. Prentice-Hall, Upper Saddle River, NJ, USA (1994)
38. Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity. Object-Oriented Series. Prentice-Hall, Upper Saddle River, NJ, USA (1995)
39. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer-Verlag, Berlin (2006)
40. TATA Research Development and Design Centre: ModelMorf: A model transformer. (available at http://www.tcs-trddc.com/ModelMorf/)
41. ikv: Company home page. (http://www.ikv.de)
42. Scheidgen, M.: CMOF-model semantics and language mapping for MOF 2.0 implementation. In Machado, R.J., Fernandes, J.M., Riebisch, M., Schätz, B., eds.: Joint Meeting of the 4th Workshop on Model-Based Development of Computer Based Systems (MBD) and 3rd International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES), 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), Proceedings, IEEE Computer Society (2006)
43. Sadilek, D., Theisselmann, F., Wachsmuth, G.: Challenges for model-driven development of self-organising disaster management information systems. In Happe, J., Koziolek, H., Rohr, M., Storm, C., Warns, T., eds.: IRTGW'06: Proceedings of the International Research Training Groups Workshop, Dagstuhl, Germany. Volume 3 of Trustworthy Software Systems., Berlin (2006) 24–26
44. Object Management Group: MOF 2.0 Versioning Final Adopted Specification. (2005)