Generic Metamodel Refactoring with Automatic Detection of Applicability and Co-evolution of Artefacts

Henning Berg and Ingrid Chieh Yu
Research report 461, February 9, 2017

ISBN 978-82-7368-426-4
ISSN 0806-3036
Abstract

Refactoring is an essential operation in software engineering, with the purpose of improving the structural qualities of software. An emerging trend in software engineering is the use of model-driven approaches in which models are used as first-class entities. The problem domain concepts used in models are formalised by a metamodel. Refactoring of metamodels is an important task in increasing the extensibility and maintainability of metamodels. However, changing the definition of a metamodel comes at the cost of introducing inconsistencies in modelling artefacts that are defined relatively to the metamodel. In this report, we present a framework extension for refactoring of metamodels that allows re-establishing consistency between the artefacts and the metamodel. We also discuss how refactoring patterns can be applied automatically on metamodels. The framework has been formally defined by a deductive system.

1 Introduction

Code refactoring is an established approach in software engineering for improving the structural properties of programs. Some of the promised benefits of code refactoring are improved reusability, readability and maintainability of the source code. A derivative approach of code refactoring is model refactoring [1] [2]. In model-driven engineering [3] models are first-class entities. That is, models are the key artefact in the design, implementation and testing of software. Hence, the ability to refactor models yields several of the same advantages as for refactoring of source code. Model refactoring is a current topic of research within the model-driven engineering community [4]. The ability to apply metamodel refactorings with automated conversion of existing models has been identified as an interesting research direction in [1].

A metamodel is a model describing models. The MetaObject Facility (MOF) [5] is a commonly used architecture for classification of models used in model-driven engineering disciplines. It comprises four layers, where a given model element at one layer is an instance of a class defined in the above layer\(^1\). The layers are referred to as \(M_0\), \(M_1\), \(M_2\) and \(M_3\), where \(M_2\) and \(M_3\) are the metamodel layers. Specifically, the \(M_2\) layer comprises a language’s metamodel that describes the abstract syntax (and in some cases the operational semantics). A model or program in a language resides on the \(M_1\) layer. The model consists of structural elements that are formally defined by the metamodel. A model conforms to a metamodel if all the model elements are valid instances of classes (or concepts) in the metamodel [6].

By definition, refactoring is the process of improving the structure of code or models while preserving behavioural properties [1]. Refactoring can be described by one or more endogenous, bijective transformations [4]. That

\(^1\)The upper level is meta-circular, i.e. it is self-describing.
is, transformations concerning code or models defined at the same abstraction level, where the code or models are instances of the same grammar or metamodel, respectively. Traditionally, refactoring requires behavioural equivalence between the source and target code/models. One way of assuring that behavioural equivalence is preserved is by defining a set of invariants that can be evaluated before and after the refactoring process. As an example, in test-driven development this is realised by utilising a test suite [1]. In some situations, the requirement for behavioural equivalence may be replaced by a requirement for functional equivalence, e.g. that a program is capable of performing a pre-determined (traceable) set of high-level functions before and after a refactoring process; though the exact fulfillment of such functions may differ between two versions of code/a model.

An important concern in software engineering is software evolution. Software evolution is the continuous process of changing and refining software to accommodate new and revised requirements, changed problem domains and new technology. Software evolution works on all abstraction layers of the MOF architecture. However, the degree of required changes decreases as the abstraction level is increased. This means that changes to models on \( M_1 \) are much more likely than changes to metamodels on \( M_2 \) and so on. Refactoring can be seen as a \textit{perfective evolution} of source code or models [2]. That is, a modification of a design to enhance its features. Even though refactoring is not a critical process, it is often highly desirable and necessary for increasing the value and quality of software [7], and ensuring longevity.

The problem of metamodel refactoring can not be seen in isolation from co-evolution. In fact, refactoring of metamodels impacts all modelling artefacts that are defined relatively to the metamodel. This includes the models, model transformations, code generators, editors, concrete syntax and analysis tools. The motivation for our work is to support refactoring of metamodels and at the same time address the impacts on other artefacts in the metamodelling ecosystem. As a first step, we focus on co-evolving existing models. Similar concerns are motivated in [8] [9] [10]. A related problem of ensuring consistency in the metamodelling ecosystem is that of synchronising different views of a model [1]. Our contribution is an analysis-based approach for refactoring of metamodels that suggests applicable refactoring patterns and calculate required changes based on the applied patterns to ensure co-evolution of artefacts.

In [11], we formalised an approach that enables automatic co-evolution of metamodels and models, with focus on composition and adaptation of metamodels. The approach comprises a set of rules that define basic adaptation operations. The basic adaptation operations can be combined to form adaptation strategies, which describe more complex adaptations. In this report, we will build on this foundation and discuss an approach for realising metamodel refactoring with co-evolution of models. We also discuss how to detect applicable refactoring patterns in a metamodel, and how these may
be applied semi-automatically.

The report is organised in three main parts. Part 1 (Sections 2 and 3) positions our work in the field and motivates why our approach is important. We discuss a selection of refactoring patterns and the requirements of metamodel refactoring. Part 2 (Sections 4, 5 and 6) comprises our contribution to the field. We describe the framework for refactoring and illustrate application of refactoring by the use of an example. We also discuss generic metamodel refactoring using the proposed framework. Part 3 (Sections 7 and 8) puts the work in context with existing research, and concludes the report.

2 Towards Metamodel Refactoring

A metamodel formalises a selection of concepts and structures of a problem domain. Other artefacts in a metamodelling ecosystem are often defined relatively to a metamodel. Hence, changes to a metamodel need to be propagated to the other artefacts to avoid creating an inconsistent system. Applying these changes are often referred to as co-evolution, e.g. metamodel-model co-evolution and metamodel-tool co-evolution [12]. An approach for refactoring of metamodels therefore has to take all of the artefacts of the ecosystem into consideration, including existing models. There are, however, refactoring patterns that do not necessarily introduce inconsistencies. Specifically, the conformance relation between the existing models and their metamodel is not broken; which means that existing models are still valid. This implies that artefacts which read and process the models are valid as well if we assume that the artefacts ignore metamodel elements that are optional, i.e. elements that may optionally be instantiated in models. Specifically, we assume that artefacts are defined in such a way that they ignore references and attributes, that are added to a metamodel, whose multiplicities have a zero lower bound, or when the upper bounds of existing attributes’ and references’ multiplicities are incremented. By this assumption we are able to differentiate between two types of metamodel refactoring patterns according to the degree of impact on the ecosystem: conformance preserving patterns and conformance breaking patterns. Conformance preserving refactoring patterns yield an intermediate stable state of the ecosystem. This state has more of a theoretical value, as the artefacts eventually have to be updated to support co-evolved models according to the changes applied to the metamodel. Also, by ignoring our assumption, changing a metamodel may very well require updating the artefacts immediately to avoid compatibility issues. However, the intermediate state allows us to categorise refactoring patterns according to how severe they impact the ecosystem. Our approach supports both kinds of refactorings. In this report we focus on co-evolution of models, that is, we address how refinements to a metamodel
may be reflected upon the conforming models. However, the approach may be generalised to other artefacts in the metamodelling ecosystem. Therefore, we discuss metamodel refactoring in a broader context where we find it appropriate.

Metamodels are typically created using class models [5]. A class model has two main components: classes and relationships. A class comprises attributes and operations, and allows modelling an (atomic) concept of the problem domain. The relationships allow defining composite structures. There are two types of relationships that can be used in class models: subtyping (inheritance) relations and associations. A subtyping relation describes that a class is a subtype of another class. According to the notion of subtyping [13] this allows substituting an object of one class with an object of the other class. This is also known as subtyping polymorphism. Behaviour is achieved either by using translational semantics, i.e. models are translated to an underlying executable language, or by executing models directly by an interpreter. There are metamodelling approaches that consider operational semantics as being part of the metamodel, e.g. Kermeta [14], the Epsilon Object Language (EOL) [15] and the Eclipse Modeling Environment (EMF) [16].

We will first discuss a selection of metamodel refactoring patterns and some of the technical challenges that emerge when implementing rules for analysing these. All of the patterns are supported by our framework. We do not aim to have a complete list of refactoring patterns. All patterns selected allow for metamodel-model conformance to be re-established. According to the principle of behavioural preservation of refactoring patterns, we argue that a refactoring is only valid if conformance can be re-established. Otherwise, an existing system is made inconsistent beyond automatic repair. This is an important observation. The framework is designed in such a way that it can easily be elaborated to support additional refactoring patterns. Note that a refactoring pattern may have different nuances. It is out of the scope of this report to discuss the reasons for applying certain refactoring patterns. Hence, we will focus on the technical challenges of applying the different patterns. Argumentation on why a specific pattern should be used and examples can be found in [17]. Assessing whether the quality or value of a metamodel has increased or decreased after refactoring is explored in e.g. [18]. An approach for evaluating the effect of a refactoring pattern is to define metrics according to the various dimensions and criteria for metamodel quality [7]. We will not delve into details on these subjects, but conclude that refactoring of metamodels addresses shortcomings of earlier design decisions that may not be suitable for future usages of the metamodels.
2.1 Conformance Preserving Refactoring

The work of [17] identifies a catalogue of low-level refactoring patterns whose application may improve the structure of metamodels. A low-level pattern represents an atomic refactoring operation (e.g. renaming of a class), on the contrary to a high-level refactoring pattern which combines several low-level patterns. The terms low-level patterns and high-level patterns are used in existing literature, e.g. in [19]. There are three patterns in the catalogue that, in the theoretical intermediate state of the ecosystem, do not necessarily break consistency. These are: add metaclass, extract superclass and add property\(^2\). We also include push down property. Moreover, we have identified three additional patterns (inspired by code refactoring): pull up property, pull up operation and push down operation. Notice that conformance preserving refactoring ensures backward compatibility with existing models/programs.

2.1.1 Add Metaclass

Adding a new class to a metamodel typically requires this class at some point to be related to existing classes of the metamodel. For refactoring, it only makes sense to add empty abstract superclasses that may later contain lifted properties and operations. The reason for this is that refactoring should not alter the domain formalised by the metamodel.

2.1.2 Extract Superclass

An abstract superclass may be created for a number of classes that share common properties. The common properties will then be lifted to the superclass. Extracting a superclass does not break the conformance between existing models and the metamodel, since all definitions of structure referred to by objects of the subclasses are still available through inheritance.

2.1.3 Add Property

A class of the metamodel may be extended with an attribute or reference. A property has a multiplicity. Consistency is preserved if the lower bound of the multiplicity is zero. This states that a value or object for the property is optional. At this point, we assume that artefacts that read or process models are able to ignore added properties whose multiplicities’ lower bound is zero (the property is unknown since the artefact has been defined according to a previous metamodel version).

\(^2\)We do not differentiate between a property’s type, i.e. whether the property is an attribute or reference.
2.1.4 Pull Up Property

An attribute or reference may be lifted from a subclass to a (direct or indirect) superclass.

2.1.5 Push Down Property

Moving a property (attribute or reference) from a superclass to a subclass may create impacts in the ecosystem; in particular if the superclass is concrete. An artefact referring to the property in the superclass will not find it after the refactoring. Therefore, we only allow pushing down a property from a class that is abstract (in order to avoid impacting the ecosystem).

2.1.6 Pull Up Operation

Operations are a bit different than properties in the sense that they are not represented by an object or value in a model. An operation may be lifted to a superclass. It is still available to its subclasses through inheritance.

2.1.7 Push Down Operation

Pushing down an operation from a superclass to a subclass does not break conformance. The reason is that models do not contain an explicit reference to a given operation. The refactoring pattern will in general induce a conflict in other artefacts since the operation that is pushed down is no longer available in the superclass. However, the pattern makes sense in a hierarchy of three or more levels of classes in which one or more subclasses override the operation. The operation defined in a subclass can be pushed down since the operation is still defined in a superclass (direct or indirect). That is, binding to the operation is still possible for all existing objects (at runtime). This refactoring pattern may potentially change the behaviour.

2.2 Conformance Breaking Refactoring

So far we have deliberately discussed the situation where the models are not updated to utilise new features of the metamodel. Clearly, models will eventually take advantage of such features. For the patterns we have studied so far, updating the models will not break conformance between the models and their metamodel, but will induce inconsistencies in artefacts that read and process the models when the models contain objects and values of new metamodel structure. The reason is that the artefacts do not have definitions for treating objects and values according to the added structure, e.g. when a model includes a value for an optional attribute. That is, we leave the intermediate consistent state of the ecosystem by updating the models. We will later discuss how consistency may be re-established in the artefacts.
From the catalogue [17] we have selected two additional patterns: *rename metaclass* and *merge metaclasses*. Moreover, we have identified the pattern *rename property*. As most refactoring patterns induce similar inconsistencies, we will keep the discussion brief.

### 2.2.1 Rename Metaclass

Renaming a class may result in a typing error in artefacts that are defined relatively to the metamodel. Models with objects of the class need to be updated to reflect the change. In addition, the name of references that point to the renamed class may no longer reflect the meaning and semantics of the class.

### 2.2.2 Merge Metaclasses

Merging of classes induces severe impacts on the artefacts. For models, this requires objects to be merged as well.

### 2.2.3 Rename Property

Renaming a property changes the definition of the class in which the property is defined. The refactoring does only have a moderate impact on the ecosystem if we assume that the new name is unique to/in all artefacts.

### 3 Requirements for Metamodel Refactoring

An essential quality of refactoring patterns is that they should not change the behaviour of the entity they refine. For source code this can be verified, e.g. by running executable tests or by evaluating the source code’s functional capabilities before and after a refactoring. Refactoring of a metamodel has implications for several artefacts. Specifically, since several artefacts in a metamodelling ecosystem have a behavioural aspect, the operational semantics of the system is not confined to the very models/programs, but spread across several of the artefacts (e.g. transformations, code generators, interpreters and analysis tools). As an example, the operational semantics of an analysis tool that is defined relatively to the metamodel may potentially be impacted as the metamodel is refactored. Hence, determining whether behaviour is preserved in an ecosystem of artefacts is a much more complex problem than validating that behaviour is preserved for a piece of source code. There is ongoing research on how to validate preservation of behaviour, e.g. [20] discusses an approach for checking the preservation of behaviour for refactorings defined by graph transformations. In this report, we base our reasoning about behavioural preservation on the following principle:
The overall behaviour of a metamodelling ecosystem is preserved after a metamodel refactoring if each co-evolved artefact, that has a behavioural aspect, fulfills a set of functional requirements as dictated by the previous version of the artefact.

That is, each artefact has to be functionally equivalent before and after a refactoring. Specifically, an artefact should have the exact same capabilities as before, and nothing more. If a refactoring and co-evolution of artefacts increase the expressiveness and capabilities of the artefacts in the ecosystem substantially, we no longer have a refactoring but an extension or adaptation. That is, we do not have a perfective evolution, but an adaptive evolution [2]. For an extension, the purpose of changing the design is not for enhancing existing features; instead the design is changed to accommodate new or altered requirements. Specifically, extension implies that new or refined problems can be modelled, analysed and processed. This fact contradicts the underlying principle of what a refactoring is. Note, however, that the stated principle of preservation of behaviour for metamodel refactorings allows minor extensions to the ecosystem as long as there are no deviations from the core observable functionalities of the artefacts. The principle is deliberately vague due to the complex task of determining whether behaviour is preserved or changed. The reason why we differentiate between refactoring and extension is because the goals of the two processes are completely different. With refactoring we want to improve the structure and/or architecture, increase readability and maintainability, and improve capabilities that will facilitate extensions of a system. The purpose is not to perform actual extensions yielding an increased expressiveness.

4 The Analysis Environment

In [11] we discuss a static analysis approach for co-evolution of models as their metamodel evolves. The analysis is designed for EMOF-compatible metamodels. Figure 1 shows an overview of the supported structural concepts, and defines a grammar for the metamodels. The grammar is inspired from the one used in [21].

Metamodels can be adapted and modified using a set of basic adaptation operations. There are operations for merging of classes, adding a new class, extending a class hierarchy, bridging two classes, and adding new properties and operations to classes.

A sequence of adaptation operations, $\varphi_1 \cdot \varphi_2 \cdot \cdots \cdot \varphi_n$, defines an adaptation strategy $\Phi$ for a particular metamodel adaptation. An adaptation strategy can be seen as a description that differentiates one metamodel variant from another. The sequence of operations is given by the user.

The analysis environment for metamodel adaptations consists of a tuple $(\mathcal{E}, \sigma, \delta)$:
Metamodel ::= package P {ClassDecl}
ClassDecl ::= (abstract)? class C extends C′ {PropDecl OpDecl}
PropDecl ::= t P (Multi)? (containment)? (# P′)?
Multi ::= [lower .. upper]
OpDecl ::= t O (t i P i) t ∈ Type ::= Primitive | C | set ⟨C⟩ | Void
Primitive ::= Boolean | Integer | Real | String

Figure 1: Grammar for describing MOF metamodels

Definition 1 The environment mapping \( E \) maps class names \( N_c \) to class definitions: \( E : N_c \rightarrow \text{Class} \), where \( \text{Class} \) is a set of classes \( \langle N_c, \text{Abs}, \text{Inh}, \text{Prop}, \text{Op} \rangle \). \( N_c \) is the name of the class. \( \text{Abs} \) indicates whether the class is abstract. \( \text{Inh} \) is a list of class names defining class inheritance (direct superclasses). \( \text{Prop} \) is a set of properties \( \langle \text{Type}, \text{Np}, \text{Multi}, \text{Cont}, \text{Opp} \rangle \) where \( \text{Type} \) is a type, \( \text{Np} \) a property name, \( \text{Multi} \) the property’s multiplicity comprising a lower and upper bound, \( \text{Cont} \) describing whether the property is of containment type, and \( \text{Opp} \) the name of an optional opposite relation. \( \text{Op} \) is a set of class operations \( \langle \text{Type}, \text{No}, \text{Param} \rangle \) where \( \text{Type} \) is the operation’s return type, \( \text{No} \) is the operation’s name and \( \text{Param} \) is a list of input parameter declarations.

Dot notation is used to access the elements of tuples such as properties and operations; e.g. \( (C, I, P, O) \). \( \text{Prop} = P \), where we use overlines to denote sets or list structures. \( \text{Prop}(N) \) denotes a subset of properties in \( \text{Prop} \) with name \( N \) and \( \text{Prop.Np} \) gives all property names in \( \text{Prop} \). Similar notation is used for \( \text{Op} \). The empty list is denoted \( ϵ \).

Definition 2 The environment mapping \( σ \) consists of a family of mappings \( \langle σ_c, σ_p, σ_o \rangle \):

\[
\begin{align*}
σ_c & : N_c \rightarrow N_c \\
σ_p & : N_p \rightarrow N_p \\
σ_o & : No \rightarrow N_o
\end{align*}
\]

The family of mappings contains substitutions accumulated during the analysis of classes, properties and operations, respectively. Substitutions are introduced in the adaptation analysis to resolve conflicts associated with overlapping metamodel definitions and will be used to ensure conformance for the underlying models. A mapping family \( σ \) is built from the empty mapping family \( \emptyset \).

The adaptation analysis of a syntactic construct \( D \) is formalised by a deductive system for judgements \( \langle E, σ, δ \rangle \vdash D \langle E′, σ′, δ′ \rangle \), where \( \langle E, σ, δ \rangle \) is the analysis environment before and \( \langle E′, σ′, δ′ \rangle \) is the environment after the analysis of \( D \), where \( E′ \) represents a package containing the derived metamodel(s). For updating the analysis environment, we use the associative operator + on mappings with the identity element \( \emptyset \). Let \( E + E′ \) denote \( E \) overridden by \( E′ \).
Definition 3 Let $n$ be a name, $d$ a declaration, $i \in I$ a mapping index, and $[n \mapsto_i d]$ the binding of $n$ to $d$ indexed by $i$. A mapping family $\sigma$ is built from the empty mapping family $\emptyset$ and indexed bindings by the constructor $\mathbf{+}$. The extraction of an indexed mapping $\sigma_i$ from $\sigma$ and application for the mapping $E$, are defined as follows:

$$
\emptyset_i \quad = \quad \varepsilon \\
(\sigma + [n \mapsto_{i'} d])_i \quad = \quad \text{if } i = i' \text{ then } \sigma_i + [n \mapsto_{i'} d] \text{ else } \sigma_i \\
\varepsilon(n) \quad = \quad \perp \\
(E + [n \mapsto d])(n') \quad = \quad \text{if } n = n' \text{ then } d \text{ else } E(n')
$$

The environment $E$ will initially contain well-typed class definitions from one or more metamodels and through a series of adaptation operations, we adapt the metamodel(s) causing modifications to $E$ and $\sigma$, and additions to $\delta$. The final environment that is returned after the analysis is the adapted metamodel(s) and the mappings constructed during the analysis for resolving name conflicts (i.e. substitutions generated by the analysis) in addition to a specification of objects that need to be created due to property multiplicities with a non-zero lower bound. An algorithm using the accumulated effects $\sigma'$ and $\delta'$ for re-establishing model conformance can be found in [11].

5 The Framework : Rules

A low-level refactoring pattern may be realised by an adaptation operation, whereas a high-level refactoring pattern may be realised in the form of an adaptation strategy. This allows the user to create a set of customisable and highly extensible refactoring patterns. The framework proposed in this report analyses whether a refactoring pattern, described as an adaptation strategy, can be performed on a given metamodel. The analysis can be seen as a set of assertions collected from each of the rule definitions. Hence, a given strategy requires a subset of the assertions to be true for the strategy to be applicable to a metamodel (since analysis of a specific strategy requires utilising a subset of the rules). The subset of assertions yields a set of pre-conditions that must be true for a strategy to be applicable, or in this context, a set of pre-conditions for a high-level refactoring pattern. The generated effects from the analysis yield a set of post-conditions. More specifically, the effects indicate changes that must be applied on the models and artefacts for the refactoring to re-establish consistency in the ecosystem. If a single pre-condition or post-condition is not true/addressed, a given refactoring pattern may not be utilised. The formalisation of the rules also serves as a specification from which an implementation can be made.

The framework supports all of the refactoring patterns identified in Section 2. The new rules are given in Figure 2 with the following explanations. The rules preserve well-typedness according to [5].
• \textit{extractSuper}((C, true, ε, \overline{P}, \overline{O}, N)) adds \( C \) as a new abstract class for the classes \( \overline{N} \). The intersection of properties in the subclasses is lifted to \( C \). The environment is updated with refined class definitions.

• \textit{renameClass}(N, N') renames class \( N \) to class \( N' \). A new name mapping is added to \( \sigma \). Moreover, all references to the old class name is updated to point to the renamed class. This is done by applying the name mapping as a substitution on the environment.

• \textit{renameProp}(C, N, N') renames the property \( N \) in class \( C \) to \( N' \). A new name mapping for properties is added to \( \sigma \).

• \textit{renameOp}(C, N, N') renames the operation \( N \) in class \( C \) to \( N' \). A new name mapping for operations is added to \( \sigma \).

• \textit{pullUpProp}(N, C, C') lifts the property named \( N \) in class \( C \) to a superclass \( C' \). To ensure well-typedness, the rule checks that \( N \) is unique in all subclasses of \( C' \). To ensure conformance, objects of subclasses may need a default value/object corresponding to the lifted property. This is handled by \textit{createInstDesc}.

• \textit{pullUpOp}(N, C, C') lifts the operation \( N \) in class \( C \) to the superclass \( C' \), given that the operation is not previously defined in \( C' \).

• \textit{pushDownProp}(N, C) pushes down the property \( N \) in the abstract class \( C \) to all subclasses of \( C \). The rule ensures that no subclass can inherit the same property from several superclasses.

• \textit{pushDownOp}(N, C, C') pushes down the operation \( N \) in class \( C \) to the subclass \( C' \). The rule ensures that the operation in \( C \) can still be bound to a superclass of \( C \).

In [11], we prove that with an adaptation strategy \( \Phi \) for metamodel adaptation and the accumulated effects \( \sigma \) and \( \delta \) (i.e. \( \mathcal{E}, \emptyset, \emptyset \vdash \Phi(\mathcal{E}', \sigma, \delta) \)), the associated models can be updated so that they conform to the altered metamodel(s) (in \( \mathcal{E}' \)). The proof may be extended to cover the new rules and validates that refactoring of metamodels with co-evolution of models is achievable.

### 5.1 An Example - Graph Metamodel

We will illustrate refactoring using an excerpt from a language for modelling of graphs. We will focus on the \texttt{Node} and \texttt{Edge} classes of a Kermeta metamodel, see Figure 3. The operational semantics of Kermeta metamodels is defined directly in class operations, i.e. the \texttt{print()} operation in this example.

In the metamodel excerpt given, both classes have the property \texttt{size} in common\textsuperscript{3}. To avoid redundancy, we apply the \textit{Extract Superclass} refactoring

\textsuperscript{3}Multiplicities for the attributes are excluded from the figure.
Figure 2: Operations desirable for refactoring where \( \text{sub}(C,C') \) and \( \text{super}(C,C') \) check if \( C \) is a subclass of \( C' \) and if \( C \) is a superclass of \( C' \), respectively. \( \text{createInstDesc} \) returns the specification of default objects or values that need to be created of a property for all subclasses of the class given as argument to preserve conformance. \( \text{implements} \) checks that an operation is defined in one of the classes provided as argument. \( [\mathcal{E}]_\sigma \) applies the substitution mapping \( \sigma \) on \( \mathcal{E} \)
Figure 3: Excerpt of a metamodel for modelling of graphs

\[ \Phi = \text{extractSuper}(\langle \text{Element}, \text{true}, \epsilon, \emptyset, (\text{size}) \rangle) \cdot \\
\text{rename}_\text{prop}(\text{Element}, \text{size}, \text{dimension}) \cdot \\
\text{pullUp}_\text{OP}(\text{print}, \text{Node}, \text{Element}) \cdot \\
\text{rename}_\text{class}(\text{Node}, \text{Vertex}) \cdot \\
\text{rename}_\text{class}(\text{Size}, \text{Dimension}) \]

Figure 4: Strategy describing refactoring of the graph metamodel

the value of description in Node or the value of id in Edge depending on the type of object on which the operation is invoked). Finally, to better reflect the domain, we rename the Node class to Vertex by applying the Rename Meta-class pattern. Similarly, the Size class is renamed to Dimension (not shown in the figure). The composite high-level refactoring pattern is described by the strategy in Figure 4.

After applying the composite refactoring pattern, we derive the metamodel of Figure 5. Using the accumulated effects in \( \sigma (\delta \text{ is empty}) \), we are able to automatically co-evolve the existing graph models. Figure 6 shows the effects generated by the framework resulting from analysing the refactoring strategy with the graph metamodel in Figure 3 as input. The first two effects state that objects of Node and Size should be updated to be instances of Vertex and Dimension, respectively. The third effect says that the slot referring to the size reference in Node and Edge objects is now a slot corresponding to the dimension reference. (The EMF XMI object specification format does not differentiate between declared and inherited properties.)

Figure 5: Refactored metamodel with added superclass

\[ \sigma = \text{[Node} \rightarrow_e \text{Vertex]} + \text{[Size} \rightarrow_e \text{Dimension]} + \text{[size} \rightarrow_p \text{dimension]} \]

Figure 6: Effects generated by the analysis framework

Figure 7 illustrates how a simplified graph model is co-evolved. Each node/vertex has a description and a size (dimension), e.g. A,1. Each edge has an id and a size (dimension), e.g. 1,1. Notice how the Node (N) objects have been updated to be instances of Vertex (V).
5.2 Tool Support

As discussed, refactoring of metamodels is a comprehensive task due to all the impacts on artefacts in the metamodelling ecosystem. Even a simple low-level refactoring, e.g. a renaming of a class, may require several changes or refinements to be performed. We have illustrated how models may be updated automatically as part of a refactoring process. Although we have merely discussed automatic co-evolution of other artefacts, the work of [22] suggests that such co-evolution is possible as well. The framework rules formalise the algorithm for an analysis engine. The effects generated are used as input to transformation engines that co-evolve the artefacts. Figure 8 illustrates a complete refactoring environment/tool based on the analysis framework. The analysis generates effects which are used by transformation engines to update the respective artefacts (additional artefacts may be added to the environment). Artefacts are not necessarily implemented using the same programming language or technology. Hence, it is likely that each artefact type requires a proprietary transformation engine. This is possible due to the generic format of the generated effects.

6 The Framework: Generic Metamodel Refactoring

The approach defined in [11] requires using adaptation strategies that incorporate information about specific classes, properties and operations. As an example, adding a property to a class requires identifying the class in which the property should be added and providing a definition for the property. Hence, an adaptation strategy is specifically defined for a given set of metamodels. So far we have seen how a specific refactoring pattern can be created and applied on a metamodel. The natural next step is generic refactoring.

There exists a lot of work on how to refine code and models to optimise their structure. The result of this work is the availability of generic refactoring and design patterns. The ability to apply generic refactoring patterns on a metamodel is therefore of high value. Generic patterns may be realised as adaptation strategies and stored in a repository. A strategy can be refined for a specific kind of metamodel, i.e. by incorporating domain-specific information into the strategy.
The deductive rules in [11], and the new ones in this report, formalise the core functionalities of an analysis engine/refactoring engine which can be very complex [23]. By adding two additional components to the analysis engine, we are able to search metamodel structure for matching pre-conditions of pre-defined generic refactoring patterns. The applicable patterns can then be applied semi-automatically to the metamodel (based on choices by the user). Figure 9 shows the components of an analysis engine for automatic detection of pattern applicability. Each refactoring pattern utilises a subset of the rules available. The purpose of the Instantiator is to instatiate these rules with structure found in the metamodel. Such instantiation is possible by using parameterisation. As an example, the conclusion of the rule for extracting a superclass has the following definition:

$$\mathcal{E}, \sigma, \delta \vdash extract_{Super}((C, true, \epsilon, \mathcal{P}, \mathcal{O}), \overline{N})\langle \mathcal{E}', \mathcal{E}'', \sigma, \delta \rangle$$

Specifically, $C, \mathcal{P}, \mathcal{O}$ and $\overline{N}$ are variables that can be parameterised. The purpose of extracting a superclass is to simplify maintenance and avoid
redundancy when a number of classes have identical properties. Hence, the properties may be lifted up to a superclass. By using an empty class definition for the superclass, represented by \( C \), and parameterising \( N \) with classes found in a metamodel, it is possible to perform an automatic search for structure that may be improved by the pattern. The Pattern Collector collects all patterns that are applicable to the metamodel given as input to the analysis engine. Additional work has to be performed on sequencing of refactoring patterns (rule scheduling), and finding an optimal search/instantiation algorithm. Specifically, the sequence of refactoring patterns is important. A refactoring pattern may not be possible to perform before another refactoring pattern is used.

![Diagram](image)

Figure 10: Overview of generic metamodel refactoring

Figure 10 shows an overview of the process of generic metamodel refactoring. The process goes as follows: A list of refactoring patterns, associated constraints and a source metamodel are given as input to the Analysis Engine (1-3). The rules are instantiated with candidate segments of the metamodel structure and analysis is performed to detect which patterns that are applicable (4). The applicable patterns are presented to the user who may choose which of them to apply (5). A new execution of the analysis engine applies the selected patterns to the source metamodel yielding a refactored metamodel (6). Co-evolution of artefacts is performed based on generated effects and the selected patterns (7). A constraint specifies the minimum set of pre-conditions for a pattern. That is, the requirements that have to be fulfilled before analysing whether a pattern is applicable on a candidate metamodel segment. Constraints are specified manually (inspired by the rule assertions). It is also possible to create variants of a refactoring pattern by adding additional custom constraints to the pattern.

7 Related Work

The work of [25] addresses how model editors expressed in the Eclipse Graphical Modeling Framework (GMF) [24] may co-evolve. The approach is cen-
tered around constructing strategies that fix a broken GMF editor, as a consequence of changing the domain model (metamodel) of a language. Adaptation of GMF models (required to generate an editor) is achieved by calculating the differences between the initial and adapted versions of the domain model. The calculated differences are then used to generate adapted GMF models. The work resembles our approach. One major difference is that we use a synthetic way of generating adaptation effects, whereas the approach in [25] uses an analytic approach. The work supports destructive operations, e.g. deleting a concrete class and deleting a property. Hence, they do not focus on metamodel refactoring according to our definition of the concept. The approach does not address co-evolution of existing models created in a model editor whose definition changes.

EMFMigrate is a language for handling co-evolution of a metamodel and related artefacts [22]. The language allows defining migration strategies comprising migration rules. Libraries of customisable strategies are supported. Executing a migration strategy requires utilising a difference model, which resembles the effects of our approach.

An approach for refactoring of models is discussed in [26], though the work does not consider refactoring of metamodels. The work of [27] describes an approach for generic metamodel refactoring. However, the approach does not address co-evolution of artefacts.

8 Conclusion and Future Work

Metamodel refactoring is an important research topic. Contrary to source code and model refactoring, metamodel refactoring has greater impacts on artefacts in the metamodelling ecosystem. These artefacts may be rendered invalid as a consequence of applying a refactoring pattern to a metamodel. In this report we have presented a framework and a mechanism for refactoring of metamodels, and discussed how artefacts in the ecosystem, with focus on models, may co-evolve with the metamodel. The mechanism supports both specific and generic metamodel refactoring, with automatic detection and application of patterns. This allows refactoring a metamodel according to best practices and well-proven design principles.

The next step is to implement an analysis engine (refactoring engine) based on the formalisation of the refactoring patterns. Also, more work has to be performed on how to schedule refactoring patterns. We believe that the analysis rules provided can easily be adapted for generating additional effects required to co-evolve other artefacts than models. For instance, it may be desirable to keep a record of how properties and operations are pulled up or pushed down in class hierarchies. Another interesting research direction is to see whether corrective refactoring can be used when a metamodel is extended.
References


