Refactorings in Language Development with Asymmetric Bidirectional Model Transformations

Martin Schmidt[1], Arif Wider[2], Markus Scheidgen[2], Joachim Fischer[2], and Sebastian von Klinski[1]

[1] Beuth Hochschule für Technik Berlin
- University of Applied Sciences -
Luxemburger Straße 10, D-13353 Berlin, Germany
`{maschmidt,klinski}@beuth-hochschule.de`
[2] Humboldt-Universität zu Berlin
Department of Computer Science
Unter den Linden 6, D-10099 Berlin, Germany
`{wider,scheidge,fische}@informatik.hu-berlin.de`

Abstract. Software language descriptions comprise several heterogeneous interdependent artifacts that cover different aspects of languages (abstract syntax, notation and semantics). The dependencies between those artifacts demand the simultaneous adaptation of all artifacts when the language is changed. Changes to a language that do not change semantics are referred to as refactorings. This class of changes can be handled automatically by applying predefined types of refactorings. Refactorings are therefore considered a valuable tool for evolving a language.

We present a model transformation based approach for the refactoring of software language descriptions. We use asymmetric bidirectional model transformations to synchronize the various artifacts of language descriptions with a refactoring model that contains all elements that are changed in a particular refactoring. This allows for automatic, type-safe refactorings that also includes the language tooling. We apply this approach to an Ecore, Xtext, Xtend based language description and describe the implementation of a non-trivial refactoring.

Keywords: DSL evolution, language description, refactoring, bidirectional model transformations.

1 Introduction

Software languages evolve continuously [1] and software language engineering does not only include the initial development but also the continuous adaptation of software languages. Especially the engineering of domain-specific languages (DSLs) requires an agile process to evolve a language along rapidly changing user requirements.

During this process, two distinct problem sets arise. First, languages are already used while they evolve, and artifacts written in a language need to be co-adapted to language adaptations. Secondly, the different artifacts that constitute the description of a language (and eventually the tooling of that language) need to be co-adapted when one of those artifacts changes. We call the former *vertical* and the latter *horizontal co-adaptation.*

In this paper, we are only concerned with horizontal co-adaptation. A language description (depending on the nature of that language) consists of several artifacts: an abstract syntax (i.e., a metamodel, e.g., an Ecore model), concrete syntax (e.g., Xtext grammar, GMF model), and description of semantics (e.g., model transformation rules or code generator). Fig. 1 depicts the different artifacts in language development and their dependencies. When one of these artifacts is changed to evolve the language (e.g., the multiplicity of a metamodel feature is changed), the other artifacts need to be changed, too (e.g., the code generator rules need to be adapted towards the changed metamodel).

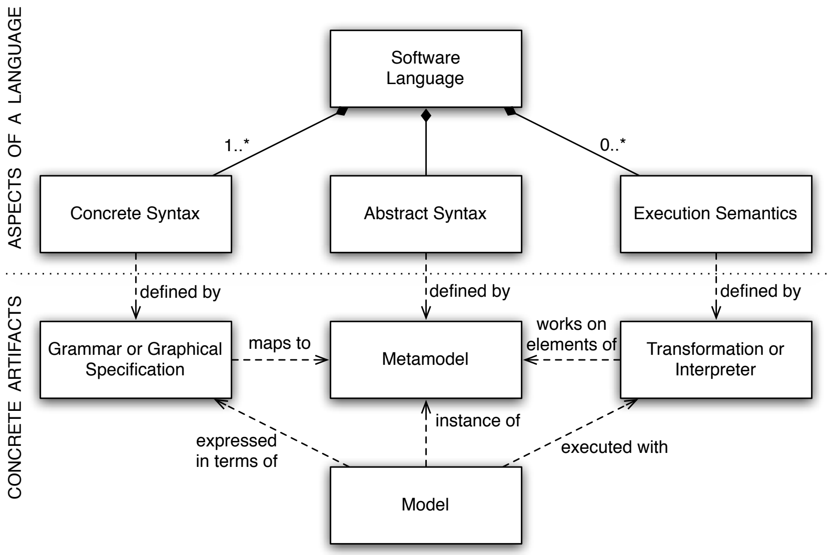


Fig. 1. Aspects of a language and concrete artifacts in metamodel-based language development

Refactorings play an integral role during the evolution of a language. Whereby a *refactoring* is described as a semantically invariant change of the language description [2]. This includes for example changing metamodel identifiers, moving features within the inheritance hierarchy (pull-up, push-down), changing the organization of grammar and transformation rules, etc.

In this paper, we present an approach that allows the refactoring of language description artifacts with automated co-refactoring of depending artifacts of the same language description. We model these co-refactorings as *view-update relations* between a common view capturing the refactoring-specific information and the dependent artifacts using asymmetric bidirectional model transformations [3]. This allows for a more declarative and modular description of refactorings which allows for more possibilities of verification and better reuse. We demonstrate the practicability of our approach by implementing refactorings on Ecore metamodels, Xtext grammars and Xtend-based code generators as artifacts.

The paper is organized as follows: In the first part we present different areas of language evolution and give an overview of existing and related work in language evolution. The following section describes a specific case study, which was decisive for this paper. Section 4 describes our approach for handling refactorings of interdependent heterogeneous artifacts in language development. Afterwards, we give an outlook of an implementation using model transformations. Finally, we conclude the paper including some discussion and show up possible directions for future work.

2 Background and Related Work

When we discuss the evolution of languages, the need for co-adaptation arises. We have multiple interdependent artifacts and if we change one the others have to change as well.

2.1 Horizontal vs. Vertical Co-adaptation

We can distinguish two forms of co-adaptation.

- First, when we change the language description (especially the metamodel) language instances have to be changed as well. We call this vertical co-adaptation: Changes need to be propagated from the meta-layer (top) down to the instance layer (bottom). Notable contributions to this kind of co-adaptation comes from Wachsmuth [4] and Hermannsdörfer [5].
- The second form of co-adaptation happens within the same layer, hence horizontal co-adaptation. Software engineers might use several languages to create one piece of software on the instance-layer, and language engineers might use different meta-languages to describe a single language (e.g., a DSL) on the meta-layer. In both cases heterogeneous interdependent artifacts are created. Since we mainly discuss horizontal co-adaptation (more specifically co-refactoring) in this paper, we discuss the related work in this field more detailed in the following subsections.

2.2 Horizontal Co-adaptation in Software Engineering

Software systems in general are often mixed-language systems. They are constructed with declarative descriptions for the user interface, imperative application logic implemented with a general-purpose programming language (GPL), and other specific languages, e.g., configuration scripts, styling or plug-in management. Strein et al. investigated the evolution of a given *inter-language* software system [6]. They described the interdependencies in an object-oriented web application that is implemented with *ASP.NET*, *HTML*, *C#* and *Visual Basic*. They identified that modern *integrated development environments* (IDEs) support the evolution of certain artifacts through offering refactorings or simple adaptations like introducing getters and setters, but these IDEs are limited concerning the co-adaptation of dependent artifacts written in different languages. For transferring adaptations to other parts of a system Strein et al. developed an IDE called X-Develop. They captured refactoring-relevant information, concerning more than just one language, in a model and adapt this model, which is a typical approach for implementing refactorings in software engineering.

2.3 Horizontal Co-adaptation in Language Engineering

As languages evolve too, the development process of a domain-specific language has similarities to general software development [1]. Changes in a language specification can have an impact on the corresponding language tools [7]. In metamodel-based language development (i.e., where the metamodel that describes the abstract syntax is the central artifact) other meta-descriptions often reference the types defined by the metamodel. Therefore, many of the needed co-adaptations that are necessary when the metamodel changes can be detected by checking these references. Although multiple (meta-) languages are involved in DSL development, these languages are interconnected via the metamodel and form the description of one language. Therefore, we call this *intra-language evolution*. In this paper, we are concerned with intra-language evolution of DSLs and other software languages (e.g., modeling languages).

Pizka and Jürgens already captured the difficulties of DSL evolution and the need of co-adaptations for these systems [8]. For handling the evolution of a language they implemented *Lever* (Language Evolver) [9]. Lever provides different integrated DSLs for the description of grammars, the tooling, and the coupled evolution of these parts. Lever focuses on textual DSLs and allows only for adaptation of tooling after the grammar changes.

In this paper we present an approach for describing refactorings and show an exemplary implementation that works with established technologies (e.g., *EMF*, *Xtext*) and allows for co-adaptations resulting from changes to different kinds of artifacts at the metalevel. We also believe that our approach is applicable to the evolution of graphical DSLs.

3 Motivating Example: The NanoWorkbench

The motivation for the work that we present in this paper emerged from practical experiences during the development of an Xtext-based DSL for developing *optical nanostructures* (NanoDSL) and a corresponding integrated tool-suite for that

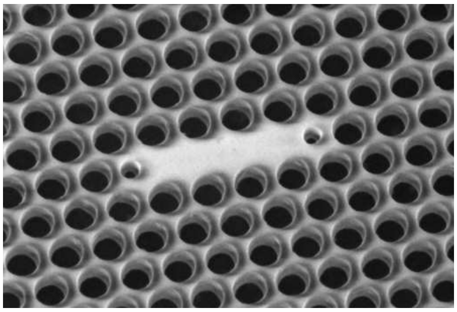
DSL (a *domain-specific workbench* called NanoWorkbench [10]). This project is subject of a cooperation with the nano-optics research group at the physics department.

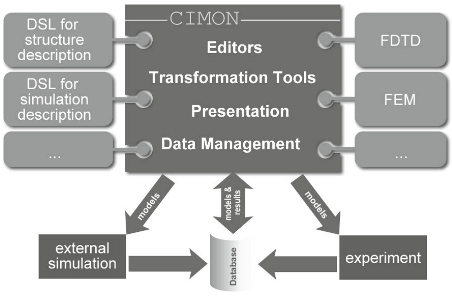
The members of this group design geometrical structures that are smaller than the wavelength of optical light in order to affect the motion of photons in a similar way a semiconductor crystal affects the motion of electrons. The properties of these *photonic crystals* are tested by simulating the propagation of an electromagnetic pulse within the structure. There are different simulation methods for that, e.g., the *finite difference time domain method* (FDTD) or the *finite element method* (FEM). Fig. 2(a) shows a picture of a photonic crystal and Fig. 2(b) shows a schematic overview of the workbench incorporating different DSLs and different simulation methods, as well as a model-driven data management and model-driven communication channels for performing external experiments or computations.

As we pursued an agile, iterative process to develop the DSL and its domain-specific workbench (with continuous consultation of the domain-experts), we identified problems similar to general software development: When implementing changes requested by the domain experts we had to change the design of the language specification several times. After solving change requests we manually adapted the generator and artifacts concerning the tooling to preserve consistency. Figure 3 gives an overview of corresponding generators that have to be adapted after the language changes.

As a concrete example, the metamodel of the NanoDSL started with a description/class for only adding cylinders as geometrical objects to the photonic crystal. This is the application the physicists mostly used. Later they wanted to add other structures like truncated cones or cuboids. With these structures they wanted to simulate manufacturing faults. For this change request we introduced a superclass for geometrical objects in general. The existing structure was renamed to *Cone* and became a subclass of the introduced class as shown in Figure 4. We identified that some attributes are more general, e.g., height or position in space, than others. To handle this redundancy we started to pull up these attributes.

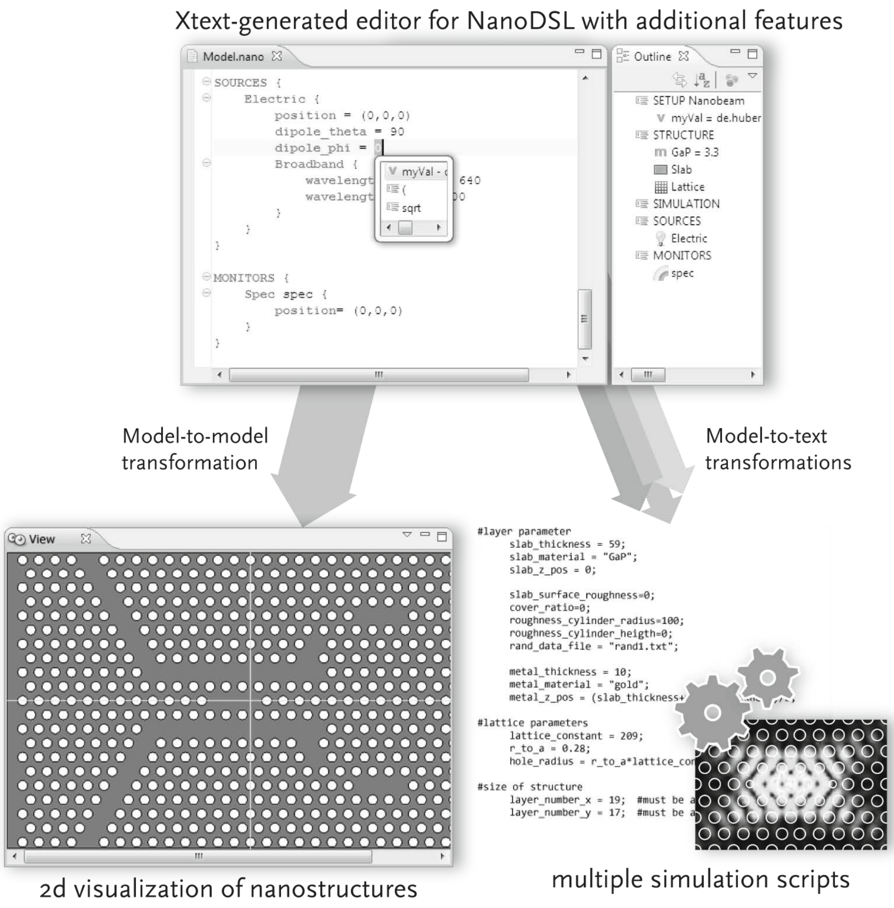
For solving this change request, we applied at least three well-known refactorings - *rename, introduce superclass, pull up feature*. Although we could easily rewrite the core language description, we had to update the existing tooling and in our special case had to adapt two complex model-to-text transformations describing different execution semantics and one model-to-model transformation for a 2D-visualisation. Additionally, we implemented a transformation rule for each added subclass. For calling the rules we need to check the object's type via the `instanceof`-operator first for providing the correct transformation. These manual changes are time-consuming and error-prone. Thus, we identified the practical need for automatic co-refactorings of interdependent artifacts in DSL development.


(a) A photonic crystal [11]



(b) Overview of the domain-specific workbench

Fig. 2. A domain-specific workbench for the development of optical nanostructures


Xtext-generated editor for NanoDSL with additional features
Model.nano
Outline
Model-to-model transformation
Model-to-text transformations
View
2d visualization of nanostructures
multiple simulation scripts

```
SOURCES {
    Electric {
        position = (0,0,0)
        dipole_theta = 90
        dipole_phi = 0
        Broadband {
            wavelengt        640
            wavelengt        00
        }
    }
}

MONITORS {
    Spec spec {
        position= (0,0,0)
    }
}
```

```
myVal - 
(
sqrt
```

```
SETUP Nanobeam
    myVal = de.huber
STRUCTURE
    GaP = 3.3
    Slab
    Lattice
SIMULATION
SOURCES
    Electric
MONITORS
    spec
```

```
#layer parameter
      slab_thickness = 59;
      slab_material = "GaP";
      slab_z_pos = 0;

      slab_surface_roughness=0;
      cover_ratio=0;
      roughness_cylinder_radius=100;
      roughness_cylinder_heigth=0;
      rand_data_file = "rand1.txt";

      metal_thickness = 10;
      metal_material = "gold";
      metal_z_pos = (slab_thickness+

#lattice parameters
      lattice_constant = 209;
      r_to_a = 0.28;
      hole_radius = r_to_a*lattice_con

#size of structure
      layer_number_x = 19;  #must be a
      layer_number_y = 17;  #must be a
```

Fig. 3. Implemented parts of the domain-specific workbench

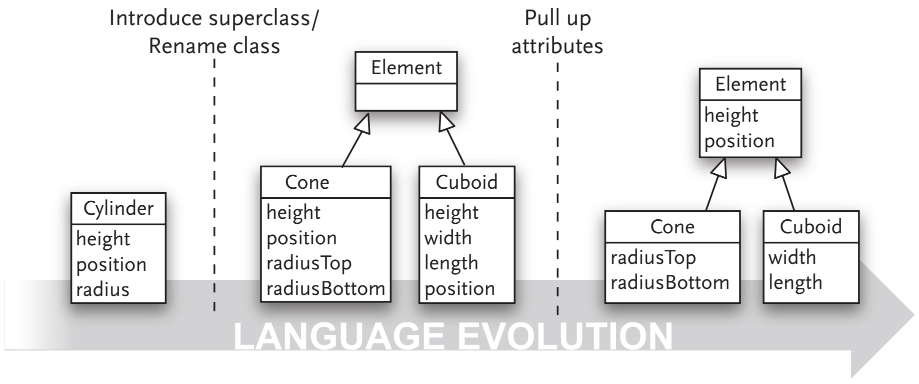


Fig. 4. Example for changes of NanoDSL

4 A Model Transformation-Based Approach for Evolution of Interdependent Artifacts in Language Development

In the following subsections, we present our model transformation based approach to refactoring of software languages. First, we will sketch the traditional, imperative approach that is used to realize refactoring for, e.g., general purpose programming languages like Java (homogeneous artifacts). Secondly, we identify differences between the refactoring of, e.g., Java programs and DSLs. Thirdly, we describe refactoring of software languages as a model-view-synchronization problem. Fourthly, we present *lenses* as one concrete method for model-view-synchronization. Finally, we summarize our approach and identify the components necessary to describe a refactoring within our approach.

4.1 Traditional Imperative Approach

Bäumer, Gamma & Kiezun [12] propose the following three steps to perform a refactoring.

– First, create a *program database* (e.g., an AST) that stores information about declarations and references (independent from a compiler). The database needs to provide a search interface for queries like *What program elements reference a certain method?*. This general step is independent from concrete refactorings.
– Secondly, perform structural program analysis using the previously established database. In this step the *cone of influence* is determined for concrete refactorings: all affected compilation units and program elements are identified and refactoring-specific preconditions are checked.
– Thirdly, the actual changes are performed. Elements of the previously determined cone of influence are changed according to the refactoring.

4.2 Refactoring of Software Languages

It is hard to extend the traditional imperative approach to the refactoring of software languages like DSLs. The existing refactoring capabilities for one type of artifact (homogeneous artifacts, e.g., refactorings for Ecore models) need to be extended to other types of artifacts (heterogeneous artifacts, e.g., Ecore refactorings that also affect related Xtend rules).

There are two specific problems. First, there is no common program database that includes elements for all types of artifacts involved in DSL development. Secondly, there are explicit relations between artifacts of different types and there are also implicit or indirect relations. A code generation rule for example is not only connected to meta-classes it directly references, but also from its super classes and its features. Furthermore, in some cases there are whole artifacts that are implicit. An example is the generated metamodel in Xtext-based DSL development, where the abstract syntax is fully generated from the concrete syntax. Code generator rules are explicitly linked to the generated metamodel, but also indirectly connected to grammar rules.

4.3 Modeling Language-Refactorings as a Model-View-Synchronization Problem

To solve the previously stated problems, we apply the steps of the traditional imperative approach with declarative methods. The information stored in a program database is already contained in the original artifacts and can be extracted through model transformation rules (step 1).

The cone of influence is a view (i.e., an abstraction) on the model that is the sum of all artifacts (i.e., the language description). This *refactoring view* can be described as model transformations between all types of artifacts and the refactoring view (step 2). These transformations filter all elements in all artifacts for those elements that are affected by the refactoring. In that sense, the refactoring view aggregates all refactoring related information scattered in all artifacts into a single view.

The actual manipulation of the model can be described by an in-place model transformation that changes the refactoring view and as model transformations from the refactoring view into all types of artifacts (step 3). Here, the aggregated information about all refactoring related elements is used to change all those elements accordingly.

Pairs of model transformations between artifacts and refactoring view and between refactoring view and artifacts (i.e., forward and backward transformations) can be described as bidirectional model transformations and, thus, the application of a refactoring can be described as a model-view-synchronization problem. Conclusively, each type of refactoring (e.g., a pull down) is described by (1) a metamodel of the refactoring view, (2) one model transformation for altering that view, and (3) bidirectional model transformations between each artifact type and the refactoring view.

4.4 Asymmetric Bidirectional Transformations

We model the refactoring of DSLs as a model-view-update problem. There are many approaches to bidirectional model transformations for solving a model-view-synchronization problem. For refactorings we favor asymmetric bidirectional model transformations (more specifically *lenses*). This specific kind of bidirectional model transformation fits the needs of a model-view-synchronization [14].

Lenses, as introduced by Pierce et al. [15], are asymmetric bidirectional transformations, i.e., one of the two structures that are synchronized has to be an abstraction of the other. This asymmetric approach is inspired by the *view-update problem* known in the database community, where a database view – the abstraction – has to be updated when the database changes and vice versa.

Given a *source* set S of concrete structures and a *view* set V of abstract structures, a lens comprises two functions:

$$\begin{aligned} get &: S \rightarrow V \\ put &: V \times S \rightarrow S \end{aligned}$$

The forward transformation *get* derives an abstract view structure from a given concrete source structure (e.g., filtering for a cone of influence). The backward transformation *put* takes an updated abstract view structure and the original concrete source structure to yield an updated concrete structure (e.g., propagate refactoring related changes into an artifact). Fig. 5 depicts a lens and its two functions.

Lenses, as presented by Pierce et al., is a combinator-based approach to asymmetric bidirectional transformations, i.e., lenses are composed from other lenses. There are primitive and combinator lenses. Formal properties of lenses can be proved for combinations if the used combinator preserves these properties. The

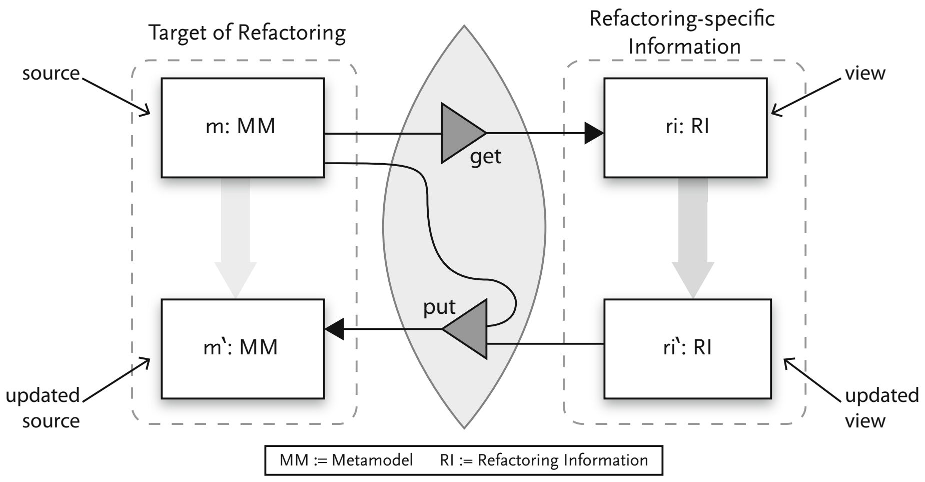


Fig. 5. Asymmetric Bidirectional Model Transformation (based on [13])

lenses framework therefore provides a flexible model transformation technique with strong capabilities for reuse and formal verification. However, the more general notion of a lens (an encapsulated tuple of the two functions) is already useful when modeling a model-view-synchronization problem.

The lenses approach stands in contrast to directly manipulating the source (which would be the naive object-oriented approach), as it describes the synchronization as two side-effect-free functions. However, the lenses approach is only possible when at a given time always only one artifact is changed and immediately synchronized with its interdependent artifacts, i.e., with no concurrent changes. This is always the case in our refactoring scenario.

4.5 Describing Refactorings with Model Transformations

A refactoring type R is described as an x-tuple of one refactoring view metamodel MM_{rv}, a number of asymmetric bidirectional transformations $\leftrightarrows_{sv}$ between the metamodels of all involved artifact types and the refactoring view metamodel, and one unidirectional view-change-transformation $\rightarrow_{vv}$ (view to view) on the refactoring view model.

$$R = \langle MM_{RV}, [\leftrightarrows_{sv}], \rightarrow_{vv} \rangle$$

In principle, this framework is independent of the concrete types of artifacts (unless they are not applicable to the same model transformation method) and independent of the concrete model transformation (unless it does not allow for asymmetric bidirectional transformations). Fig. 6 shows a refactoring type (pull up) based on Ecore and Xtend as artifact types.

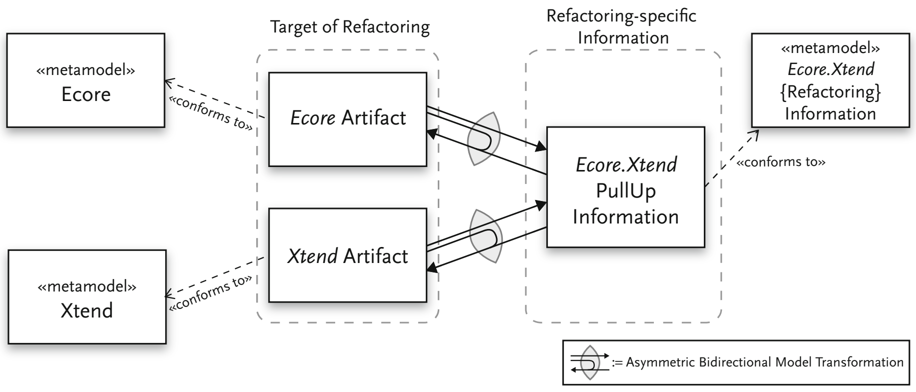

Target of Refactoring
Refactoring-specific Information
«metamodel» Ecore
«conforms to»
Ecore Artifact
Xtend Artifact
«metamodel» Xtend
«conforms to»
Ecore.Xtend PullUp Information
«metamodel» Ecore.Xtend {Refactoring} Information
«conforms to»
:= Asymmetric Bidirectional Model Transformation

Fig. 6. The Pull-up refactoring involving an Ecore and an Xtext artifact

4.6 Advantages and Disadvantages

The proposed approach provides the following advantages compared to traditional imperative approaches. Firstly, the approach allows for more comprehensible and reusable descriptions of refactorings as a clear set of models and transformations. Refactoring related logic is not scattered across multiple parts of the language description anymore. Secondly, the declarative model transformation based approach clearly separates meta-language tooling (e.g., Xtext, Ecore, Xtend tools) from refactorings. The refactoring description only depends on models and not on tools. Thirdly, the approach provides a history of changes and refactorings as a set of model transformation traces. Finally, the declarative and side-effect-free lenses based approach provides good verification capabilities, especially when implemented in a functional manner.

As a disadvantage, a potentially large number of transformations has to be created for each refactoring – up to two times the number of involved artifacts. Firstly, this number can be halved by using special languages for bidirectional transformations that provide special notations for defining both the forward and the backward transformation at the same time. Secondly, we argue that the refactoring logic is the same as in the traditional imperative approach, it is just structured differently. This different structure can result in code duplication when implemented naively, e.g., when transformations for multiple involved artifacts are very similar. However, by defining reusable building blocks for transformations and by composing transformations from them, it is possible to not only keep code duplication within a refactoring minimal, but also code duplication across refactorings, which is one of the advantages of our approach.

5 Implementation with Model Transformations

The general approach presented so far is independent from concrete modeling technologies or model transformation languages. It can be implemented using special languages for the description of bidirectional transformations or pairs of unidirectional transformations which again can be described with special model transformation languages or GPLs.

In this section we demonstrate how to implement our approach using Java as a GPL and how to integrate it with Eclipse-based modeling technologies. As an example, we use the 'Pull Up Item' refactoring in the Ecore, Xtext, and Xtend based scenario that we motivated in Sect. 3.

5.1 Abstract Refactoring Structure

Listing 1.1 shows the abstract structure of refactorings where three artifacts are affected: A grammar describing a textual concrete syntax, a metamodel describing the abstract syntax, and a generator describing one execution semantics. Thus, a refactoring consists of three asymmetric bidirectional model transformations (i.e., lenses, see List. 1.2) that synchronize the grammar, the model, and a generator, respectively, with a refactoring-specific refactoring view. In addition to that, an abstract method for implementing the (typically trivial) change on that refactoring view is provided. It takes a second parameter of type `ChangeInfo`, if there are different possibilities how to perform the change.

```
public abstract class
    Refactoring<RefactoringView,ChangeInfo,Grammar,Model,Generator>
    {

   public Lens<Grammar, RefactoringView> grammarLens;
   public Lens<Model, RefactoringView> modelLens;
   public Lens<Generator, RefactoringView> generatorLens;

   public abstract RefactoringView changeView(RefactoringView
       oldView, ChangeInfo info);
}
```

Listing 1.1. Java implementation of the Refactoring structure

```
public interface Lens<Source, View> {
   public View get(Source src, ISelection sel);
   public Source put(Source src, View view, ISelection sel);
}
```

Listing 1.2. A lens interface with parameterizable lens functions

5.2 Implementation of 'Pull Up Item' for Xtext, Xtend and Ecore

Based on this general structure of a refactoring description, the following listings show parts of an exemplary implementation of the 'Pull Up Item' refactoring [2]. First, a refactoring-specific view type is defined which, in this case, holds the attribute that is to be pulled up, the class it originally belongs to (the subclass), and a list of this class' superclasses (List. 1.3). From the list of superclasses one is to be chosen for the attribute to be pulled up to.

```
public class PullUpRefactoringView {
   public EAttribute selectedAttribute;
   public EClass subClass;
   public List<EClass> superClasses;
}
```

Listing 1.3. RefactoringView for 'Pull up Item'

Next (refer to List. 1.4), the concrete refactoring type is defined by extending the abstract refactoring type and by providing appropriate type parameters: Obviously, the view type is the previously defined `PullUpRefatoringView`. The `ChangeInfo` contains the target superclass. As multi-inheritance in principle is allowed in model-driven engineering, we provide a wizard for selecting the target superclass if there is more than one option. Before an instance of this view can be created, refactoring-specific pre-conditions have to be checked like *Exists at least one superclass?* or *Are there already attributes with the same signature in the selected superclass?*.

The remaining type parameters are specific to the involved technologies: An Xtext resource for the grammar description, `EObject` for the root object of the (meta-) model describing the abstract syntax, and again an Xtext resource for the Xtend-based generator because Xtend is based on Xtext. Apart from providing these type parameters (and, thus, typing the three lenses accordingly) the declaration of the refactoring type only provides a concrete implementation of the `changeView`-method, which here, only changes the attribute in the view so that it belongs to the selected superclass of its originally containing class. Listing 1.4 shows the complete definition of the PullUpRefactoring type (except the trivial, field initializing constructor).

```
public class PullUpRefactoring extends
    Refactoring<PullUpRefactoringView, PullUpChangeInfo,
    XtextResource, EObject, XtendResource>{

   public Lens<XtextResource, PullUpRefactoringView> grammarLens;
   public Lens<EObject, PullUpRefactoringView> modelLens;
   public Lens<XtendResource, PullUpRefactoringView> generatorLens;

   @Override
   public PullUpRefactoringView changeView(PullUpRefactoringView
       oldView, PullUpChangeInfo info) {
      // ..
      // checking parameter and relevant preconditions

      PullUpRefactoringView newView = oldView;
      EAttribute changedEAttribute = clone(oldView.getAttribute());

      for (EClass superClass : newView.getSuperClasses()) {
         if (superClass.getName() ==
              info.selectedSuperClass.getName()) {
            // Change the container for the attribute
            superClass.getEStructuralFeatures().add(changedEAttribute);
      }}

      newView.setAttribute(changedEAttribute);
      return newView;
}}
```

Listing 1.4. The PullUpRefactoring class

Now, the vital parts of the refactoring are the bidirectional model transformations, which are defined separately and are then passed as constructor arguments to the refactoring during instantiation. As we show an implementation without the use of special languages for bidirectional transformations, a total of six transformations have to be provided in this case (three forward and three backward transformations.) For brevity, we only show two selected transformations.

Listing 1.5 shows the forward transformation `get` of the lens synchronizing between the Xtend-based generator and the refactoring view. First, we are extracting the element, which is the target of the refactoring from the text selection `sel`. Afterwards we collect the relevant information – the containing class and its superclasses – and build the refactoring view. To gain this information, we navigate through the containment hierarchy of the resolved element. Finally, the refactoring view is returned.

```
public PullUpRefactoringView get(XtextResource src, ISelection sel) {
  PullUpRefactoringView pullUpRefView = new PullUpRefactoringView();

  EObject elementUnderChange = getElement(sel);

  // Attribute which will be pulled up
  pullUpRefView.attribute = (EAttribute)elementUnderChange;
  // Containing class of attribute
  pullUpRefView.subClass = getContainerOfType(elementUnderChange);
  // List of possible superclasses
  pullUpRefView.superClasses = getSuperClasses(elementUnderChange);

  return pullUpRefView;
}
```

Listing 1.5. Forward transformation of the generator lens of 'Pull Up Item'

The backward transformation of the lens synchronizing between the Ecore-based (meta-) model and the refactoring view is shown in Listing 1.6. Additionally to the (potentially altered) refactoring view, this transformation takes the original artifact as the `src` argument – here, the original model. A copy of this model is created and the selected attribute is replaced by the one contained in the refactoring view (including the attribute's updated reference to its containing class).

```
@Override
public EObject put(final EObject src, final PullUpRefactoringView
    view, ISelection sel) {
  EObject newModel = copy(src); // returns a copy of src

  EAttribute selectedEAttribute = getSelectedAttribute(sel);

  // replace the selected attribute with containment hierarchy
  newModel.eSet(selectedEAttribute, view.getAttribute());

  return newModel;
}
```

Listing 1.6. Backward transformation of the model lens of 'Pull Up Item'

5.3 Alternative Implementation Approaches

In the previous subsections, we demonstrated a pragmatic implementation of our model transformation based approach using Java as a GPL, because it is well-known and eases integration with existing EMF-based technologies. However, the advantages are more apparent, when using special model transformation languages for the implementation. For a Java-like integration of such languages with EMF-based technologies, we showed how to implement a unidirectional rule-based model transformation language as an internal DSL in the *Scala* programming language [16]. Using this language or another unidirectional model transformation language like ATL [17], the forward and backward transformations of refactorings could be described more concisely.

However, in order to use our approach to its full capacity, special languages for describing bidirectional model transformations like *QVT Relations* could be used instead of describing pairs of unidirectional transformations. Unfortunately, QVT Relations suffers from weak tool support and from semantic issues regarding non-bijective relations [18]. This especially affects the presented scenario of constructing and synchronizing an abstracted view. The combinator-based approach of lenses that was presented by Pierce et al. is especially strong in such a scenario. Therefore, we are working on an implementation of such combinator-based lenses that integrate well with EMF-based technologies [14] and work on an implementation of our refactoring approach using these lenses.

Furthermore, an issue that arises with our current implementation approach, is that some logic, e.g., for finding all occurrences of an element, is needed in both the forward and the backward transformation (because it takes the original source as an argument) but is currently not shared, resulting in code duplication. Therefore, we are investigating into splitting the process of building a view into two steps: First, the abstraction step that only collects relevant information and, second, the aggregation step that merges redundant information so that it can be used as a shared view on different artifacts. This way, the abstraction step could be shared by forward and backward transformation.

6 Conclusions and Future Work

We presented a declarative approach to horizontal co-refactorings in language development. Our approach employs asymmetric bidirectional model transformations to extract all information important for a refactoring into a refactoring view first, and then to synchronize all artifacts of a language description with the changes made to that refactoring view.

This approach allows for describing types of refactorings independent from tooling and concrete notation of meta-languages. All that comprises a refactoring is put into its description – which is a clear set of models and transformations – and refactoring related logic is not spread over and mixed with meta-tools. This allows for extending refactorings towards new meta-languages. Sequences of changes can be recorded through traces of model transformations. Different model transformation languages can be used to implement our abstract approach as long as they allow for asymmetric bidirectional transformations. However, the advantages of our declarative approach can be leveraged in conjunction with special (often more declarative) transformation languages for bidirectional model transformations. Therefore, our approach should benefit from ongoing research in that area [3,14,19].

We showed the principle feasibility of our approach based on the *'Pull up item'* refactoring, which we applied to an Ecore, Xtext, Xtend based DSL. We are working on a full catalog of refactorings in order to better show advantages of our approach in terms of comprehensibility and reuse of transformations across multiple refactorings. Furthermore, we want to provide an implementation that makes use of special languages and frameworks for bidirectional model transformations. Therefore, we are evaluating which of such languages and frameworks work best for our scenario. Finally, we are investigating how our approach can be generalized towards co-adaptations that are not refactorings.

References

1. Favre, J.-M.: Languages evolve too! changing the software time scale. In: Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE 2005), pp. 33–44. IEEE Computer Society (2005)
2. Fowler, M., Beck, K.: Refactoring – improving the design of existing code. Addison-Wesley Professional (1999)
3. Diskin, Z., Xiong, Y., Czarnecki, K.: From State- to Delta-Based Bidirectional Model Transformations – the Asymmetric Case. Journal of Object Technology 10, 6:1–6:25 (2011), `http://www.jot.fm/issues/issue_2011_01/article6.pdf`
4. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
5. Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G.: Language evolution in practice –The history of GMF. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 3–22. Springer, Heidelberg (2010)
6. Strein, D., Kratz, H., Lowe, W.: Cross-language program analysis and refactoring. In: Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), pp. 207–216. IEEE Computer Society (2006)
7. Favre, J.: Meta-model and model co-evolution within the 3D software space. In: Proceedings of the Interantaional Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), pp. 98–109 (2003), `http://plg.math.uwaterloo.ca/~migod/papers/2003/ELISAproceedings.pdf`
8. Pizka, M., Jürgens, E.: Tool-supported multi-level language evolution. In: Software and Services Variability Management Workshop, vol. 3, pp. 48–67. Helsinki University of Technology (2007), `http://citeseerx.ist.psu.edu/viewdoc/doi=10.1.1.190.9220&rep=rep1&type=pdf`
9. Jürgens, E., Pizka, M.: The Language Evolver Lever-Tool Demonstration. Electronic Notes in Theoretical Computer Science 164(2), 55–60 (2006), `http://dx.doi.org/10.1016/j.entcs.2006.10.004`
10. Wider, A., Schmidt, M., Kühnlenz, F., Fischer, J.: A Model-Driven Workbench for Simulation-Based Development of Optical Nanostructures. In: Proceedings of the 2nd International Conference on Computer Modelling and Simulation (CSSim 2011) (2011) IEEE CD with ISBN 978-80-214-4320-4

11. Barth, M., Kouba, J., Stingl, J., Löchel, B., Benson, O.: Modification of visible spontaneous emission with silicon nitride photonic crystal nanocavities. Optics Express 15(25), 17231–17240 (2007), `http://dx.doi.org/10.1364/OE.15.017231`
12. Bäumer, D., Gamma, E., Kiezun, A.: Integrating Refactoring Support into a Java Development Tool. In: OOPSLA 2001 Companion. ACM (2001), `http://people.csail.mit.edu/akiezun/companion.pdf`
13. Foster, J.: Bidirectional Programming Languages. PhD thesis, University of Pennsylvania (2009), `http://repository.upenn.edu/cgi/viewcontent.cgi?article=1967&context=cis_reports`
14. Wider, A.: Towards Combinators for Bidirectional Model Transformations in Scala. In: Sloane, A., Aßmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 367–377. Springer, Heidelberg (2012)
15. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005), pp. 233–246. ACM (2005)
16. George, L., Wider, A., Scheidgen, M.: Type-Safe Model Transformation Languages as Internal DSLs in Scala. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 160–175. Springer, Heidelberg (2012)
17. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL – A model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
18. Stevens, P.: Bidirectional model transformations in QVT – Semantic issues and open questions. Software and Systems Modeling 9(1), 7–20 (2010)
19. Sasano, I., Hu, Z., Hidaka, S., Inaba, K., Kato, H., Nakano, K.: Toward Bidirectionalization of ATL with GRoundTram. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 138–151. Springer, Heidelberg (2011)