Role-based Generic Model Refactoring

Jan Reimann, Mirko Seifert, Uwe Aßmann

Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
jan.reimann|mirko.seifert|uwe.assmann @tu-dresden.de

Abstract. Refactorings can be used to improve the structure of software artifacts while preserving the semantics of the encapsulated information. Various types of refactorings have been proposed and implemented for programming languages such as Java or C#. With the advent of Model-Driven Software Development (MDSD), the need for restructuring models similar to programs has emerged. Previous work in this field [1,2] indicates that refactorings can be specified generically to foster their reuse. However, existing approaches can handle only certain types of modelling languages and reuse refactorings only once per language.
In this paper a novel approach based on role models to specify generic refactorings is presented. We discuss how this resolves the limitations of previous works, as well as how specific refactorings can be defined as extensions to generic ones. The approach was implemented based on the Eclipse Modeling Framework (EMF) [3] and evaluated using multiple modelling languages and refactorings.

1 Introduction

During the development and evolution of software systems new functionality is added or existing one is adjusted to new requirements. This challenges existing software architectures in terms of their flexibility. For some changes the architecture might be perfectly suitable. In other cases, earlier design decisions can render the implementation complicated. To enable the evolution of systems while keeping their architecture clear, there is constant need to change a system's structure, which is the essence of Lehman's Laws of Software Evolution [4].

To improve their software designs, developers spend a lot of time manually restructuring existing code. However, the manual restructuring has a big disadvantage—it has a high probability to introduce bugs. No matter how much attention is payed by the developer, if a system is large enough, there is a good chance that the restructuring changes the behaviour of the system. Thus, one should strive for restructurings that preserve the behaviour, for which [5] introduced the term *refactoring*. Ever since then, refactorings have been an essential tool for developers [6].

In parallel, the extension of code-centric development to MDSD brought many problems once examined in the context of programming languages to new

attention. Developers that are used to restructure code both easy and safely do not want to give up this powerful tool when operating on models.

When looking at the differences between code and models, there is a number of things that come to mind. First, the structure of code is represented by grammars while models are prescribed by metamodels [7]. The former are based on abstract syntax trees, while the latter rely on graph structures. Second, the level of concreteness of code is supposed to be higher than the one of models. Models must abstract some details of a system's implementation. Third, metamodelling allows for quickly defining new modelling languages. Domain-Specific Languages (DSLs) can be created with small effort. This will create a whole landscape of languages, which is significantly different for traditional programming languages, of which only a few are used in industry.

Especially the third point (i.e., the large variety of modelling languages) demands for generic and reusable methods and tools. The effort to develop and maintain the growing number of DSLs can only be reduced by reusing tools across different languages. This does of course also apply to refactorings. If one wants to quickly establish refactoring support for new DSLs, a technology to reuse existing refactorings is needed [8].

A closer look at the reuse of refactorings raises the question about what can be reused across multiple languages and what cannot. Previous work in this area has shown that there is potential for reuse, but it is limited in one way or the other (see Sect. 2). But this research also indicated that some aspects cannot be captured in the reusable part of the refactoring. For example, a generic refactoring cannot make assumptions about the semantics of a language. Thus, reusing refactorings requires a combination of adapting existing generic parts to the modelling language of interest and the additional specification of language-specific information. This paper presents a new technique for this combination. Our contributions are the following: 1) limitations of existing work on generic refactorings are identified, 2) a role-based approach is proposed to overcome these limitations, 3) the novel approach is evaluated by modelling refactorings for different modelling and meta-modelling languages (e.g., Unified Modeling Language (UML) [9] and Ecore), 4) an extensible refactoring framework based on the EMF is presented, serving as a basis for evaluation and future work.

The structure of this paper is as follows. Before introducing the conceptual ideas of our approach in Sect. 3, we discuss the limitations of existing work in the area in Sect. 2. Based on an implementation of our approach, we evaluated the reuse of refactorings. Results of this evaluation can be found in Sect. 4. In Sect. 5 we draw conclusions and outline future work.

2 Related Work

To analyse the limitations of existing model refactoring approaches, in particular those that are generic, we will classify related work into three categories. Depending on the Meta Object Facility (MOF) meta layer [10] at which refactorings are specified (i.e., **M3**, **M2**, or **M1**), different observations can be made.

M3 In [2] a generic approach to specify refactorings is presented. The authors introduce a meta-metamodel (GenericMT), which enables the definition of refactorings on the MOF layer M3. This meta-metamodel contains structural commonalities of object-oriented M2 models (e.g., classes, methods, attributes and parameters). Refactorings are then specified on top of the GenericMT. To activate them for a specific metamodel (i.e., a model on M2) a target adaptation is needed. Once such an adaptation exists, every defined refactoring can be applied to instances of the adapted metamodel. The adaptation contains the specification of derived properties declared in the GenericMT which are not defined in the metamodel of interest. By this, an aspect-oriented approach is achieved and the newly defined properties are woven into each target metamodel.

However, this approach is restrictive w.r.t. the structures of GenericMT, because this contains object-oriented elements only. DSLs that expose such structure and that have a similar semantics can be treated, while other DSLs cannot. Refactorings that require other structures cannot be implemented. Furthermore, metamodels are mapped once for all refactorings. This does not allow to map the same structure twice (e.g., if a DSL contains two concepts similar to *method*).

A related approach was published by Zhang et al. in [11], where refactorings are specified on top of a custom meta-metamodel similar to MOF. With the specification on M3 a metamodel-independent solution of generic model refactorings is achieved, but again the DSL designer has limited control over the elements that participate in a refactoring. In summary one can say that the approaches targeting M3 are limited w.r.t. to the structures refactorings operate on.

M2 Other approaches target the MOF layer where metamodels reside—M2. Defining refactorings on top of these metamodels implies that refactorings can work only on one specific language. [12] and [13] follow this approach and introduce the graphical definition of refactorings. DSL designers can graphically define a pre-condition model, a post-condition model and a model containing prohibited structures. All models can share links to express their relations and those which are not subject to modification during the transformation. By this technique, a graph is constructed and therefore further analysis can be conducted. The advantage of specifying refactorings on M2 is that the target structures of the metamodel can be controlled. However, reuse and the generic specification are sacrificed. Refactorings for specific metamodels cannot be reused for other languages, but must be defined again although the core steps are the same.

M1 In [14] and [15] a refactoring-by-example approach is presented, which enables the user of refactorings to specify them on concrete instances of a specific metamodel (i.e., on level M1). Modifications are recorded and then abstracted and propagated to the specific metamodel. Since this metamodel is situated on the M2 layer again, this approach does not allow for reuse either.

Based on the analysis of related works in the field, we observed that the specification of refactorings on a single MOF layer does not yield a satisfactory result. Therefore, a technique is needed that is able to combine the advantages of layer M3 and M2 specifications. Such a technique must solve the problem that M3 approaches are limited to a specific group of languages, by allowing

to use multiple structural patterns rather than a single one (e.g., the object-oriented concepts). It must also address the limitation of M2 approaches, which are specific to one language. A dedicated solution should allow DSL designers to reuse individual generic refactorings for multiple languages.

3 Specifying Refactorings with Role Models

Previously, we have argued that there is a strong need to reuse refactorings, in particular in the context of DSLs. To enable such reuse, the parts of refactorings that can be generalised must be separated from the ones that are specific to a particular language. Consider for example the basic refactoring RenameElement. The steps needed to perform this refactoring are equal no matter what kind of element needs to be renamed. After changing the value of a string-typed attribute, all references to the element need to be updated. The concrete attribute may vary for different languages, but the general procedure is the same. Depending on constraints that apply to particular languages (e.g., whether unique names are required) some renamings may be valid, while others need to be rejected.

From the simple example we gained some initial insights. First, the structural part of a refactoring (i.e., the elements that are transformed) is a good candidate for reuse. Second, the semantics of a language can render concrete refactorings invalid. Also, the example shows that semantics—both static and dynamic—are language-specific and therefore, cannot be part of a generic refactoring. We have also seen that the execution of a refactoring can be a composition of a generic transformation and specific steps that differ from language to language. We will postpone the language specifics for a moment (see Sect. 3.4) and look at the structural and transformational aspects of refactorings first.

To reuse the structural part of refactorings, a model of this structure is needed. More precisely, the structure of models that can be handled by a refactoring must be specified. For the RenameElement example, this structure basically consists of an attribute in the metamodel. Any model element that has such an attribute (i.e., it is an instance of the metaclass that defines the attribute) can be renamed. Other refactorings (e.g., ExtractMethod), have more complex structural constraints, such as requiring that the elements to extract (i.e., statements) need to be contained in a container object (i.e., a method). To model such structural properties—one may also consider these as structural requirements—we choose to use *Role Models* as they were proposed in [16].

Roles encapsulate the behaviour of a model element with regard to a context. Roles appear in pairs or sets, so-called collaborations. In the context of this work, roles can be considered as the types of elements that are subject to a refactoring. The role collaborations model the references between the elements required to execute a refactoring. More details about this will be presented in Sect. 3.1.

To map the abstract structural definition of the input required by a refactoring to concrete languages, a relation between the structural definition (i.e., the role model) and the metamodel of the language must be established. This mapping defines which elements of the metamodel are valid substitutions for el-

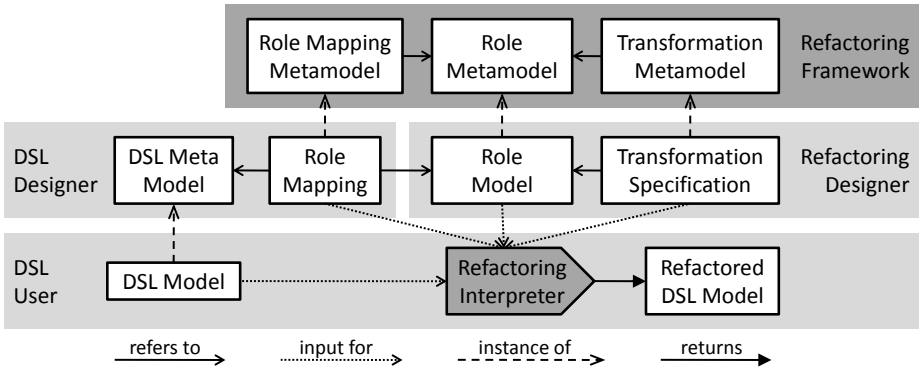


Fig. 1. Metamodels, Relations and Stakeholders

ements in the role model. We call this relation a *Role Mapping*. We will discuss such mappings in Sect. 3.2.

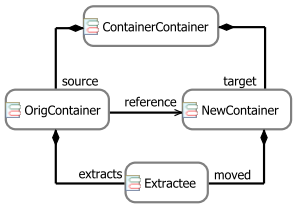
To reuse the transformational part of refactorings a third model is needed—a transformation specification. This model defines the concrete steps needed to perform the refactoring. In the example of renaming an element, this model must contain instructions to change the value of an attribute. Of course, the transformation needs to change the value of the attribute specified by the role model (i.e., it refers to the role model). Section 3.3 contains details about the transformation operators and the interpretation thereof.

The models mentioned so far are depicted in upper right part of Fig. 1. Concrete models are related to their metamodel by instance-of links. The metamodels are provided as part of our framework. Two kinds of models (i.e., role models and transformation specifications) are provided by the refactoring designer. She creates generic refactorings, which consist of pairs of such models. DSL designers can then bind the generic refactorings to their languages by creating role mappings. This enables DSL users to apply refactorings to concrete models.

3.1 Specifying Structural Constraints using Role Models

To explain role models in more detail, consider a concrete refactoring for Ecore models—Extract Superclass. This refactoring can be applied to a set of structural features (i.e., attributes or references) and moves the given set of features to a new EClass. The new EClass is then added to the list of super types for the original EClass. Suppose we are convinced that this type of refactoring (i.e., moving a set of elements to a new container) is useful for other languages too and we want to derive a generic refactoring—ExtractX.

To capture the structural properties of our refactoring we use a role model. For the ExtractX refactoring, we consider elements that are extracted, their container and a new container where the elements are moved to. Elements participating in a refactoring form groups, where all elements within one group are handled equally. In other words, elements *play* a certain role w.r.t. a refactoring.



(a) Graphical representation

```
RoleModel ExtractX {
  ROLE OrigContainer;
  input super ROLE Extractee;
  ROLE NewContainer (newName);
  ROLE ContainerContainer;
  transitive : OrigContainer       [1..1]
    <>- Extractee extracts         [0..*];
  transitive : NewContainer        [1..1]
    <>- Extractee moved            [0..*];
  transitive : ContainerContainer  [1..1]
    <>- NewContainer target        [1..1];
  transitive : ContainerContainer  [1..1]
    <>- OrigContainer source       [1..1];
  transitive : OrigContainer       [1..*]
    -- NewContainer reference      [1..*];
}
```

(b) Textual representation

Fig. 2. Role Model for the ExtractX refactoring

The elements that are extracted (e.g., structural features) play the *Extractee* role. The object that contains the extractees plays the *OrigContainer* role. The object the extractees are moved to, plays the *NewContainer* role. In other words, roles abstract from the concrete metaclasses that are allowed as participants for a refactoring. Instead of referring to concrete metaclasses, the structure that can be transformed is specified by roles. Later on, when a refactoring is enabled for a concrete language, role mapping models map roles to concrete metaclasses.

Between the roles that form the nodes in our structural description, certain relations hold (the collaboration). For example, the *OrigContainer* must hold a containment reference to the *Extractee* elements. Also, a reference between the original and the new container is needed to connect the new container to the old one. We use collaborations to model such structural dependencies between roles.

The complete role model for the ExtractX refactoring is depicted in Fig. 2. One can identify the roles mentioned above (shown as boxes) as well as their collaborations, which are depicted by links between the boxes. Besides the roles mentioned above, there is a role *ContainerContainer* which models the fact that the new and the original container must be contained in a third element. For the ExtractSuperclass refactoring this role is played by the class EPackage, which is specified in the role mapping (see Sect. 3.2).

The example gives an impression how role models specify the structural constraints that must be met by a language's metamodel to qualify for the generic ExtractX refactoring. More concepts that can be used in role models are defined by the Role metamodel shown in Fig. 3.

The concepts `Role` and `Collaboration` are contained in a `RoleModel`. Roles may be annotated by several modifiers. An `optional` role is not needed to be mapped to a specific metaclass (e.g., if a DSL's metamodel doesn't contain such a metaclass in the desired context). Roles which must serve as input of a refactor-

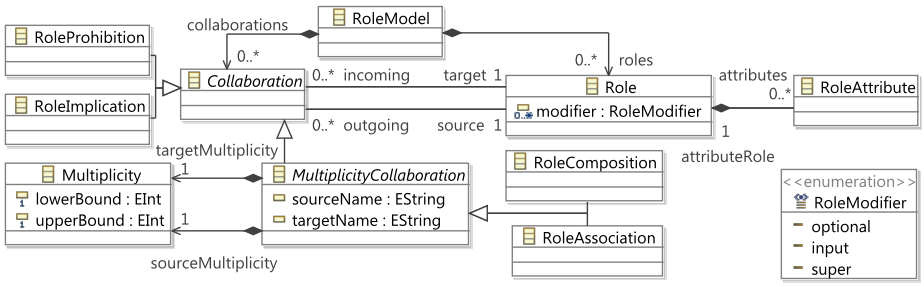


Fig. 3. Role Metamodel

ing are marked with the `input` modifier. In the ExtractX example the *Extractee* must be provided as input. The `super` modifier expresses that, at runtime, all subclasses of the mapped metaclass play this role, too. For example, Extract-Superclass for Ecore must allow to extract both attributes and references of EClasses. Therefore *Extractee* is modelled as super role to enable the mapping to EStructuralFeature by which EAttribute and EReference are comprised. Furthermore, a `Role` can contain `RoleAttribute`s to express that it can own observable properties (e.g., a name) which may change during the role's life cycle.

Collaborations connect two `Role`s—`source` and `target`. They are further distinguished into different types. A `RoleImplication` constraint can be used to express that one role must also play another role. A `RoleProhibition` constraint states that two roles are mutually exclusive—an element playing one role must not play the other role. The other types are characterised by multiplicities, expressing that one role can collaborate with a specific number of elements playing the related role. To distinguish between containment and non-containment references, `RoleAssociation`s and `RoleComposition`s can be used. Using this metamodel we were able to model the structures required by all refactorings presented in Sect. 4.

The next step in the development of a generic refactoring is writing a transformation specification. Such specifications do only refer to the role model of the refactoring, not to a concrete metamodel. However, we will first look at role mappings, which DSL designers can use to bind role models to concrete languages. This is needed to understand the actual execution of refactorings as this can only be performed in the context of a mapping.

3.2 Role Mappings

To control the structures intended for refactoring, the role model needs to be mapped to the target metamodel (see middle level of Fig. 1), which is performed by the DSL designer. Based on this mapping, refactorings will only be applied to those structures to which they were mapped. An example for a mapping, namely the mapping for ExtractSuperclass is shown in Fig. 4(a). The mapping is written in our DSL for role mappings. If a single refactoring shall be activated

extractSuperClass.rolemapping

```
ROLEMODELMAPPING extractSuperClass FOR
<http://www.eclipse.org/emf/2002/Ecore>

ExtractSuperClass maps <ExtractX> {

  Extractee := EStructuralFeature;

  OrigContainer := EClass {
    extracts := eStructuralFeatures;
    reference := eSuperTypes;
  };

  NewContainer := EClass(newName -> name)
  {
    moved := eStructuralFeatures;
  };

  ContainerContainer := EPackage {
    source := eClassifiers;
    target := eClassifiers;
  };
}
```

(a) Extract Superclass

extractEEnum.rolemapping

```
ROLEMODELMAPPING extractEEnum FOR
<http://www.eclipse.org/emf/2002/Ecore>

ExtractEEnum maps <ExtractX> {

  Extractee := EEnumLiteral;

  OrigContainer := EEnum {
    extracts := eLiterals;
    reference := ePackage;
  };

  NewContainer := EEnum(newName -> name)
  {
    moved := eLiterals;
  };

  ContainerContainer := EPackage {
    source := eClassifiers;
    target := eClassifiers;
  };
}
```

(b) Extract Enum

Fig. 4. Role Mappings for different parts of the Ecore metamodel

for another part of the metamodel it needs to be mapped a second time. In Fig. 4(b) the ExtractX refactoring is instantiated for enumeration literals. In the case of ExtractSuperclass and ExtractEnum, the mapping is all that is required to enable support for these two concrete refactorings.

In general, role mappings conform to the metamodel depicted in Fig. 5. Here, each `RoleMappingModel` refers to an EPackage containing the target metamodel and owns several `RoleMapping`s referencing the refactoring's role model. Each `ConcreteMapping` specifies which role is played by which metaclass from the target metamodel. If roles collaborate, the mapping must specify at least one `ReferenceMetaClassPair`. Each pair points to a concrete EReference (contained in the target metamodel), which must lead to the specified EClass. By defining multiple `ReferenceMetaClassPair`s it is possible to constitute a path from one EClass to another. This enables the control of the structures in a more flexible way. Besides mapping collaborations of a role, attributes can be mapped as well. With this mapping the possibilities of annotating a metamodel with a role model are completed—all structural features can be mapped.

3.3 Executing Refactorings

Returning to the viewpoint of the refactoring designer, the next step to consider is the execution of a refactoring. To abstract the transformation from concrete languages it must refer to the corresponding role model only. For this concern

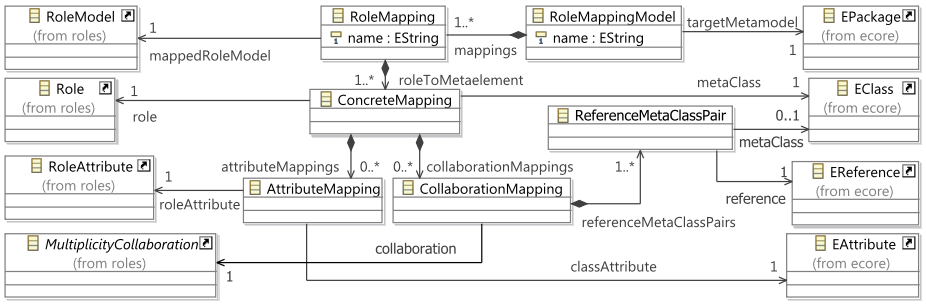


Fig. 5. Metamodel of role mapping DSL

an extensible metamodel—the *Refactoring Specification*—and a textual syntax is used by the refactoring designer. To give an example, Fig. 6 shows the refactoring specification for the role model of the ExtractX refactoring (cf. Fig. 2). This specification is given in our refactoring DSL, called *RefSpec*, which is explained in the following.

Before interpretation, the referenced roles and collaborations are resolved using the role mapping. Then, six basic steps are sufficient to execute this refactoring. First, two object definitions bind concrete objects from the input (i.e., the selected elements) to symbolic names. Subsequently, the four core commands can be found. The `create` operator constructs a new object playing the first given role as a child of the object playing the second given role. In the next step `assign` is used to pass a value to the attribute which was mapped to the given role attribute. The `move` command then relocates the elements playing the role of the collaboration with the first given role and the second given role becomes the new parent. The `distinct` operator achieves that duplicates will be rejected.

For the ExtractX example this means that for equal structural features stemming from different EClasses only one instance will be created in the new parent. Finally the `set` command arranges that the object playing the first given role is

ExtractX.refspec

```
REFACTORING ExtractX FOR <ExtractX>

STEPS {
    object containerContainerObject := ContainerContainer from uptree(INPUT);
    object origContainerObjects := OrigContainer as trace(INPUT);

    create new nc:NewContainer in containerContainerObject;
    assign nc.newName;
    move OrigContainer.extracts to nc distinct;
    set use of nc in origContainerObjects;
}
```

Fig. 6. Refactoring Specification for ExtractX

referenced in the mapped collaboration of the object playing the second given role. Due to space restrictions, not all commands can be explained here in greater detail. However, it is important to say that the presented refactoring specifications differ from common model transformations (e.g., Query View Transformation [17] (QVT)). One can interpret the former only within the context of a role mapping, while the latter directly refer to concrete metamodels. This is also the reason for developing a dedicated language here instead of reusing existing ones.

As discussed by Opdyke in [5], certain conditions must be met before and after refactorings. To obtain a valid refactoring, all pre-conditions must be fulfilled before and all post-conditions must hold after the refactoring. Otherwise, the refactoring must be rolled back. In our approach these conditions can also be specified on top of the role models (e.g., using the Object Constraint Language (OCL)). A modified OCL interpreter then must translate the roles and collaborations w.r.t. to a role mapping. Until now, our implementation supports metamodel-specific constraints only. The interpreter for generic constraints has not been realised yet, but we do not foresee conceptual problems to do so.

In summary one can say that the interpretation of generic transformation specifications in the context of role mappings, eventually bridges the gap between generic refactorings and their actual execution. It allows to define execution steps solely based on roles and collaborations between them (i.e., independent of a metamodel), but still enables the execution of refactorings for concrete models.

3.4 Custom Refactoring Extensions

Not all transformation steps specific to a concrete metamodel can be captured in the reusable part of a refactoring. For example, recall the ExtractSuperclass refactoring and suppose we would like to obtain a quite similar refactoring—ExtractInterface. The core steps are the same, but the two refactorings differ in two details. First, after executing ExtractInterface, the values `isAbstract` and `isInterface` of the newly created EClass must be set to `true`.

This is just a small difference, but still it cannot be handled by the generic ExtractX refactoring as it is. To support such language-specific adjustments, we introduced *Post Processors* which execute additional steps after the core refactoring. They can be registered for specific metamodels in combination with a role mapping. Post processors can obtain the runtime objects to which the roles have been resolved and can then invoke further transformation steps. The post processor for ExtractInterface therefore sets `isAbstract` and `isInterface` to `true`.

Alternatively one could create a new generic refactoring, which incorporates the setting of attributes. However, one should not do so before at least a second specific refactoring is found that requires this feature. Then, support for inheritance among role models and refactorings specifications would certainly be useful, but this is subject to future work. In Sect. 4 we present further refactorings which required custom extensions.

3.5 Preserving Semantics

Refactorings are defined as preserving the behaviour of the program, or in our case the model, that is subject to a refactoring. The behaviour of any program (or model) is defined by its static and dynamic semantics. This does immediately imply that preserving behaviour requires a formal specification of this semantics. Without formalisation no guarantees can be given by a refactoring tool. It does also imply that the meaning of a model is highly language-specific and can therefore not be reused. From our point of view, a framework for generic refactorings can solely provide extension points to check the preservation of semantics. DSL designers can use these points to attach custom components that add additional constraints to enforce the correctness of a refactoring.

The need for formal semantics poses a problem for general purpose languages. For complex languages (e.g., Java) there is either no complete formal definition of its semantics, or if there is one, it is rarely used by the respective refactoring tools to prove correctness. However, in the context of modelling languages there is a great chance to provide refactorings that are provable correct w.r.t. the preservation of the model's semantics. Because of their reduced complexity, modelling languages offer the chance to have complete definitions of their formal semantics. In particular for DSLs, which may have a very narrow scope, the reduced semantical complexity may allow to prove the correctness of refactorings. This does of course assume a clean specification of the semantics rather than using loose transformations to source code as often observed in practise.

In summary one can say that there is chance to avoid the limitations observed for general-purpose language refactoring engines, if DSL designers are willing to specify semantics formally. Assuming they do, proofs must be established to show that a refactoring's pre-conditions suffice to guarantee the preservation of semantics. These proofs must be constructed by DSL designers for each DSL, similar to the proofs that were needed for refactorings for programming languages in the past.

4 Evaluation

To evaluate the feasibility of our generic refactoring approach, we collected refactorings for different modelling languages and implemented them according to the procedure outlined in Sect. 3. Based on EMF, we created metamodels for role models, role mappings and transformation specifications. We defined textual syntax for all three languages using EMFText [18] to easily create model instances. For the role models an additional editor based on the Graphical Modeling Framework (GMF) [19] was generated. The interpreter for the transformation specification was implemented manually. To apply refactorings to models, we implemented editor adaptors for the EMF tree editors, GMF editors and EMFText editors and a test suite to validate the execution of refactorings automatically.

The goal of this evaluation was to collect information about the number of specific refactorings that can benefit from reusing a generic refactoring. In

addition, the question how many refactorings would require specific extensions to the generic refactoring should be answered. This includes both extensions to the transformation executed by the refactoring, as well as additional pre- and post-conditions. To obtain a representative result, we tried to cover as many refactorings as possible. However, this list can never be complete, which is why the results of our evaluation must always be considered w.r.t. the languages and refactorings under study.

We used languages of different complexity and maturity for our evaluation. The metamodels for UML, Web Ontology Language (OWL) [20], Timed Automata and Java exposed the highest number of classes and structural features. Other languages (e.g., Ecore, ConcreteSyntax, Feature Models and BPMN) have a medium complexity. Some metamodels (e.g., Pl0, Office, Conference, Text Adventure, Sandwich and Forms) were rather small and taken from the EMFText Syntax Zoo[1]. We are aware that this selection of metamodels is by no means exhaustive, but we are still convinced that it supports the idea of generic refactorings. In any case, we will extend the set of languages in the future to obtain a more complete picture of the applicability of generic refactorings.

The concrete results of our evaluation can be found in Table 1. The metamodels to which refactorings were applied are depicted as columns, the generic refactorings form the rows. The numbers in the cells denote how often the generic refactoring was mapped to the metamodel. The numbers in parenthesis indicate how many role mappings needed a post processor to augment the generic refactoring with additional transformation steps.

Table 1: Refactorings applied to metamodels

	Ecore	UML	BPMN	Java	ConcreteSyntax	Feature Models	Timed Automata	Pl/0	OWL	Roles	Conference	Office	Text Adventure	Sandwich	Forms	Simple GUI
Rename X	1	1	1	1	1	1	1	3	1	1	1	2	2	1	1	
Move X	1	4						1								
Introduce Reference Class	1	2				1										
Extract X	5(2)	2														
Extract X with Reference Class		1(1)	1(1)	1(1)	1(1)			1			1		1		1	1
Extract Sub X	1	1														

In total, we applied 6 generic refactorings to 16 metamodels by creating 47 mappings. The refactorings that were reused most often are RenameX and ExtractXwithReferenceClass. The latter is an extended version of ExtractX. Due

[1] http://www.emftext.org/zoo

to space restrictions, we cannot list the names of all the specific refactorings here. The complete list can be found on the web[2].

In Sect. 2 we have identified two types of limitations of previous works. The first limitation—no reuse of refactorings across languages—applies to approaches that define refactorings on meta level M2. When looking at Table 1, one can see that each generic refactoring was applied to at least two metamodels. Some of them were even applicable to the majority of the languages under study.

The second limitation—having a single fixed mapping for each metamodel—was observed for the approaches residing on M3. All cells from Table 1 that contain a value higher than 1 indicate cases where our approach is superior to existing approaches. Here, the same generic refactoring was mapped multiple times to a metamodel to obtain different specific refactorings.

Even w.r.t. to the limited number of languages and refactorings that were evaluated, we think that the results support our approach. First, we were able to instantiate many specific refactorings from few generic ones, which shows that reusing refactorings specifications is beneficial. Second, the low amount of customisation that was needed (6 out of 47 refactorings), indicates that most refactorings can be instantiated without additional effort, besides specifying a role mapping.

5 Conclusion and Future Work

In this paper the limitations of existing work on generic refactorings have been identified and a novel approach to overcome them was presented. Based on role models, structural requirements for refactorings can be generically formalised. Using a mapping specification, such role models can be bound to specific modelling languages. This mapping defines which elements of a language play which role in the context of a refactoring. Based on the mapping, generic transformation specifications are executed to restructure models. Thus, generic refactorings can be reused for different languages only by providing a mapping. The transformation and generic pre- and post-conditions are reused. Furthermore, the same generic refactoring can be repeatedly applied to one language.

We have discussed the extent to which generic refactorings can be reused and the preservation of semantics. Even though the latter is highly language-specific, a generic refactoring framework can still provide extension points to be parameterised by a language's semantics. An evaluation using an EMF-based implementation has shown that many generic refactorings can be reused. For the custom extensions that were needed in some cases, we observed that these were rather small and that they could be applied in a monotonic fashion (i.e., without making invasive changes to the generic refactorings). We do not claim that all refactorings should be realised as instantiations of generic ones. There will always be refactorings that are highly language-specific and which may therefore not benefit from reusing a generic refactoring. But, we believe that our evaluation has shown that reusing refactorings can be beneficial in many cases.

[2] `http://www.emftext.org/refactoring/catalog`

We identified multiple open issues for future research. First, models are connected over multiple levels of abstraction. If the relations between those levels are known, refactorings should change dependent models to preserve the overall semantics of such a model stack. A similar connection exists between models on different meta levels. Thus, the evolution of metamodels can also be considered as refactorings. Here, the references of all instances of the metamodels must be changed in order to preserve the instance-of relation if a metamodel is changed.

Second, the refactorings presented in this paper were mapped manually to the target languages. As the role models exactly define the structure required to instantiate a generic refactoring, searching for refactoring candidates is an interesting task for further investigations. If such a search is possible, DSL designers could simply ask for all applicable refactorings for a given metamodel and select the ones they find suitable for their language.

Third, the presented evaluation is only meaningful w.r.t. the languages and refactorings under study. To draw conclusions on a more broad basis, we currently perform a survey to collect more refactorings. The outcome of this survey will hopefully be an even bigger set of refactorings and modelling languages, which allows to justify the feasibility of our approach more accurately.

Acknowledgement

This research has been co-funded by the European Commission within the FP6 project MODELPLEX #034081 and the FP7 project MOST #216691.

References

1. Ralf Lämmel: Towards Generic Refactoring. In: Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02, Pittsburgh, USA, ACM Press (October 2002)
2. Naouel Moha, Vincent Mahé, Olivier Barais, Jean-Marc Jézéquel: Generic Model Refactorings. In Schürr, A., Selic, B., eds.: MODELS. Volume 5795 of Lecture Notes in Computer Science., Springer (2009) 628–643
3. Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks: Eclipse Modeling Framework, 2nd Edition. Pearson Education (2008)
4. Meir M. Lehman: On understanding laws, evolution, and conservation in the large-program life cycle. Journal of Systems and Software **1** (1980) 213–221
5. William F. Opdyke: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)
6. Martin Fowler: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman, Amsterdam (1999)
7. Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. In: International Symposium on Distributed Objects and Applications, DOA Federated Conferences, Industrial track, Irvine, 2002. (2002)
8. Tom Mens, Gabriele Taentzer, Dirk Müller: Challenges in Model Refactoring. In: Proc. 1st Workshop on Refactoring Tools, University of Berlin (2007)
9. The Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.2. Technical report (February 2009)

10. The Object Management Group: Meta Object Facility (MOF) Core Specification. Technical report (January 2006)
11. Jing Zhang, Yuehua Lin, Jeff Gray: Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In Beydeda, S., Book, M., Gruhn, V., eds.: Volume II of Research and Practice in Softw. Eng. Springer (2005) 199–218
12. Taentzer, G., Müller, D., Mens, T.: Specifying Domain-Specific Refactorings for AndroMDA Based on Graph Transformation. In: Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers, Berlin, Heidelberg, Springer (2008) 104–119
13. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E. In: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. Volume 4199/2006 of Lecture Notes in Computer Science. Springer (2006) 425–439
14. Petra Brosch, Martina Seidl, Konrad Wieland, Manuel Wimmer, Philip Langer: The Operation Recorder: Specifying Model Refactorings By-Example. In Arora, S., Leavens, G.T., eds.: OOPSLA Companion, ACM (2009) 791–792
15. Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger: An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In: MODELS '09: Proc. of the 12th International Conf. on Model Driven Engineering Languages and Systems, Berlin, Heidelberg, Springer-Verlag (2009) 271–285
16. Dirk Riehle, Thomas Gross: Role Model Based Framework Design and Integration. In: Proc. of OOPSLA '98, New York, NY, USA, ACM (1998) 117–133
17. The Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/-Transformation. Specification Version 1.0 (April 2008)
18. Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Christian Wende: Derivation and Refinement of Textual Syntax for Models. In: Proc. of the 5th Europ. Conf. on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009). Volume 5562 of LNCS., Springer (2009) 114–129
19. Richard C. Gronback: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Pearson Education (April 2009)
20. W3C: OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. Technical report (October 2009)