Softw Syst Model (2008) 7:25–47
DOI 10.1007/s10270-007-0056-x

REGULAR PAPER

Refactoring OCL annotated UML class diagrams

Slaviša Marković · Thomas Baar

Received: 1 March 2006 / Revised: 20 August 2006 / Accepted: 21 February 2007 / Published online: 15 May 2007

Abstract Refactoring of UML class diagrams is an emerging research topic and heavily inspired by refactoring of program code written in object-oriented implementation languages. Current class diagram refactoring techniques concentrate on the diagrammatic part but neglect OCL constraints that might become syntactically incorrect by changing the underlying class diagram. This paper formalizes the most important refactoring rules for class diagrams and classifies them with respect to their impact on attached OCL constraints. For refactoring rules that have an impact on OCL constraints, we formalize the necessary changes of the attached constraints. Our refactoring rules are specified in a graph-grammar inspired formalism. They have been implemented as QVT transformation rules. We finally discuss for our refactoring rules the problem of syntax preservation and show, by using the KeY-system, how this can be resolved.

Keywords Refactoring · QVT · Imperative OCL · Graph-transformations · Syntax preserving refactoring rules · Source code verification

1 Introduction

Modern software development processes, such as Rational Unified Process (RUP) [17] and eXtreme Programming (XP) [6], propagate the application of refactoring to support iterative software development. Refactoring (see [20] for an overview) is a structured technique to improve the quality of software artifacts.

Artifacts produced in all phases of the software development life cycle could become a subject of refactoring. Existing techniques and tools, however, still mainly target implementation code. An up-to-date list of existing tools and their application domains can be found in [30].

In his seminal work [28], Opdyke has introduced the concept of refactoring of implementation code. He defined refactorings as "*... reorganization plans that support change at an intermediate level*" and identified 26 of such reorganization plans; now better known as refactoring rules. A refactoring rule for implementation code describes usually three main activities:

1. Identify the parts of the program that should be refactored (code smells).
2. Improve the quality of the identified part by applying refactoring rules, e.g. the rule *MoveAttribute* moves one attribute to another class. As the result of this activity, code smells such as *LargeClass* disappear.
3. Change the program at all other locations which are affected by the refactoring done in step 2. For example, if at some location in the code the moved attribute is accessed, this attribute call became syntactically incorrect in step 2 and must be rewritten.

The application of refactoring rules, called *refactoring steps*, is most often pattern-driven. A design that is an instance of Design Patterns [14] can usually be extended and maintained much better than a design that is less structured. Thus, it is often beneficiary to transform the current design towards an

Communicated by Dr. Lionel Briand.

S. Marković (✉) · T. Baar
École Polytechnique Fédérale de Lausanne (EPFL),
School of Computer and Communication Sciences,
1015 Lausanne, Switzerland
e-mail: slavisa.markovic@epfl.ch

T. Baar
e-mail: thomas.baar@epfl.ch

instance of Design Patterns. Pattern-driven refactoring steps have been thoroughly studied in [16,21].

To a certain degree, refactoring rules depend on the language they are applied on. This explains why there are many catalogs of refactoring rules for different languages. The most complete and influential catalog was published by Fowler in [13] for the refactoring of Java code. The refactoring of artifacts that are more abstract than implementation code is a relatively new research topic that became urgent with the success of the UML. Some initial catalogs of refactoring rules for UML diagrams are presented in [2,31,33]. However, neither these catalogs nor any of the existing UML refactoring tools [8,9,29] support—apart from some simple *Rename*-refactorings—the refactoring of attached OCL constraints once the underlying UML class diagram has changed. Speaking in terms of the above given *MoveAttribute* example, the first two steps have been realized by many tools but the last step is usually ignored. For the refactoring of OCL, we are aware of only one approach. Correa and Werner present in [12] some refactoring rules for OCL, but these rules focus on the improvement of poorly structured OCL constraints and take only to a very limited extent the relationship between OCL constraints and the underlying class diagram into account.

We give in this paper a formal specification of the most important refactoring rules for UML class diagrams including the necessary changes on attached OCL constraints. Trivially, it is always necessary to change an OCL constraint if the refactoring of the underlying class diagram would make this constraint syntactically invalid. We believe that our rules are *syntax preserving*, i.e. each rule preserves the syntactical correctness of the UML/OCL model it is applied on. Section 2 contains some 'design guidelines' for the development of syntax preserving refactoring rules. However, we have formally proved the syntax preservation property only for the rule *ExtractClass* (see Sect. 5). Another criterion for refactoring rules is *semantics preservation*, i.e. the meaning (semantics) of the model remains the same whenever the refactoring rule is applied. We do not discuss the problem of semantics preservation in this paper, but refer the interested reader to [4].

Our formal description of refactoring rules is done on the level of the metamodel for UML and OCL. Unlike other approaches that describe refactoring rules formally [12,15,31,33], we do not use OCL pre-/postconditions for this purpose. The formalism of our choice is the graph-grammar inspired notation proposed by the QVT Merge Group in an early submission for the OMG standard Query/View/Transformation (QVT) [25]. Unfortunately, the graphical notation for QVT has changed in the final adopted specification [26], but we keep using the initial graphical notation in order to keep the rules presented in this paper similar to those presented in the conference version of this paper [18].

All refactoring rules presented in this paper have been implemented using Together Architect 2006 for Eclipse, a commercial CASE tool that supports development and execution of QVT transformations [9]. The implementation of our rules can be downloaded from [19].

The paper is organized as follows. In Sect. 2 we give preliminaries necessary to understand our rules, which are formally defined in Sect. 3. An overview of implementation steps is given in Sect. 4. In Sect. 5 we show on one example how syntax preservation of refactoring rules can be proven formally. For this task, the KeY-system has been successfully applied. Section 6 concludes the paper.

2 Formalizing refactoring rules in QVT

Model transformations are widely recognized as the *heart and soul* of model-driven development [32]. Refactoring rules are a special form of model transformations for which the source and the target model are expressed using the same language. In this paper, we describe refactoring rules in a graphical formalism that has been suggested by the QVT Merge Group in [25].

The following subsections give a brief introduction to the most important elements of QVT. We discuss using a very simple example the general structure of refactoring rules and give some guidelines for achieving syntax preservation. Finally, since our aim is to refactor UML class diagrams together with attached OCL constraints, we recall the relevant parts of the official UML/OCL metamodel in order to facilitate the understanding of refactoring rules presented in Sect. 3.

2.1 Basic concepts

A model transformation is defined as a set of *transformation rules*. In the graphical notation proposed in [25], a transformation rule consists of two patterns, left hand side (LHS) and right hand side (RHS), which are connected with the symbol ‹⬭›. Optionally, a rule can have parameters and a when-clause containing a constraint written in OCL.

The LHS and RHS patterns are denoted by a generalized form of object diagrams. In addition to the normal object diagrams, free variables can be used in order to indicate object identifiers and values of attributes. The same variable can occur both in LHS and RHS and refers at all occurrences – during the application of the rule – to the same value. In order to distinguish between objects/links occurring in patterns and objects/links occurring in concrete models we will use the terms *pattern objects/links* and *concrete objects/links*, respectively.

A rule is applied on a source model (represented as an instance of the metamodel, i.e. as a graph) as follows: In the

source model, a subgraph that matches with LHS is searched and rewritten by a new subgraph derived from RHS under the same matching. If the obtained target model still contains subgraphs matching with LHS, the rule is applied iteratively as long as it is applicable (possibly infinitely often). A *matching* is an assignment of all variables occurring in LHS/RHS to concrete values. When applying a rule, the matching must obey the restrictions imposed by the when-clause. This semantics of QVT rules has the following consequences: If a pattern object appears in the rule's RHS but not in its LHS (i.e., in LHS there is no pattern object identified by the same variable) then—when applying the rule—a corresponding, concrete object is created. If there is a pattern object in LHS but not in RHS, then the matching object in the source model is deleted together with all 'dangling links'. Similarly, a link is created/deleted if the corresponding pattern link does not appear in both LHS and RHS (pattern links are identified by their role names and the pattern objects they connect). The value of an attribute for a concrete object is changed only if the attribute is shown on the corresponding pattern object in RHS. The attribute's new value for the concrete object is obtained by the expression shown as value for the attribute in RHS under the current matching. Values of attributes that are not mentioned in RHS remain unchanged.

2.2 How to write syntax preserving QVT rules

The purpose of this subsection is twofold. Firstly, the section should illustrate on concrete examples the above given basic concepts of QVT, which already allow to write quite expressive transformation rules. Secondly, some basic principles of the design of syntax preserving refactoring rules are explained. These principles have been frequently applied for the design of the (more complex) rules presented in Sect. 3 for the refactoring of UML/OCL models.

2.2.1 Example: Item-View world

In order to explain QVT's basic concepts, we start with refactoring rules for a tiny *Item-View* language, Fig. 1 shows its metamodel.

There are two non-abstract language concepts *Item* and *View*, which both inherit the metaattribute *name* from *ModelElement*. The metaassociation between *Item* and *View* indicates that arbitrarily many views can be attached to one item (which is called the *owner* of the view). Furthermore, the self-association on *Item* indicates that each item can have an arbitrary number of parent- and child-items. Moreover, we assume that the parent-child relationship is acyclic. This can be expressed in OCL by the following invariant:

```
context Item inv CycleFree:
  self.allParents()->excludes(self)
```

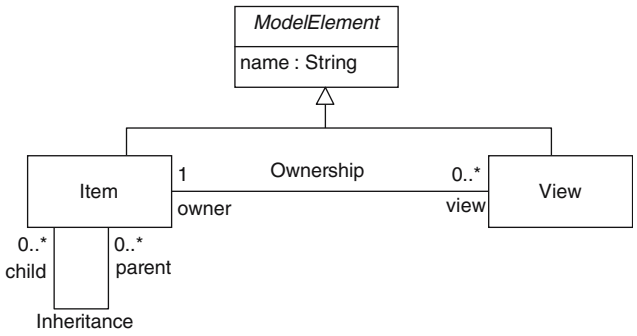

Fig. 1 Metamodel for simple *Item-View* language

```
context Item def:
  allParents():Set(Item)=
    self.parent->union(self.parent.allParents()->asSet())
```

Please note that the additional query *allParents()*, which represents the transitive closure of the parent-relationship, is well-defined despite its recursive definition (see [3] for a detailed justification).

2.2.2 Two simple refactoring rules

As a first example, the renaming of an *item*, which has been selected by the user, is formalized by the QVT rule *RenameItem1* as shown in the left part of Fig. 2. The selected item is passed as the first parameter of the rule and the rule's LHS checks whether the passed item really exists in the source model (what should, trivially, be always the case). The pattern RHS is identical to LHS except for attribute *name*, whose value is set to *newName*, the rule's second parameter.

A second version of the *Rename*-refactoring is formalized by rule *RenameItem2* shown in the right part of Fig. 2. Here, the item that should be renamed is determined by a match of its name with the first parameter of the rule (*oldName*). Applied on a given source model, this rule would iteratively make the following two steps as long as possible: (1) search for an item with name *oldName* in the current model and (2) rename the found item to *newName*. Please note that the application of this rule might not terminate if there is an item with *oldName* in the source model and *newName* is the same as *oldName*. Also the first rule *RenameItem1* suffers from the same problem. We will see later, how termination problems can be avoided by adding a when-clause to the QVT rule.

2.2.3 Checking syntax preservation of a given rule

A refactoring rule is called *syntax preserving* if for every syntactically correct source model the obtained target model is syntactically correct as well. Syntactically correct models are exactly the valid instances of the metamodel, what boils down to the following three criteria: (1) all model elements

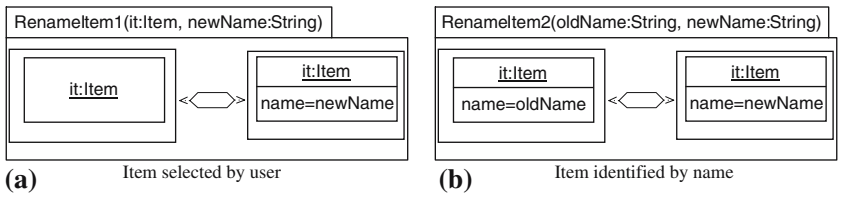

Fig. 2 Two versions of *RenameItem* refactoring rule

are well-typed, (2) all multiplicity constraints are met, and (3) all well-formedness rules are obeyed. Invalid instances of the *Item-View* metamodel would be, for example, an *Item* object having a value of type *Integer* for attribute *name* (fails to meet criterion (1) due to type declaration of *name*), a *View* object that is linked to two *Item* objects (see criterion (2) and multiplicity for *owner*), and an *Item* object having a self-link for association *Inheritance* (see criterion (3) and well-formedness rule *CycleFree*).

If the syntax preservation of a given refactoring rule should be shown, it has to been argued for every valid metamodel instance that the refactoring rule is either not applicable or that the target model is a valid metamodel instance as well. Fortunately, only a single step of the rule application has to be taken into account. By a simple induction argument, one can lift the syntax preservation property from a single step to the whole rule application. The argumentation on the syntactical correctness of each possible target model can be split according to the three validity criteria given above. A detailed argumentation for the refactoring *RenameItem1* is given in Table 1. More generally, the following aspects should to be taken into account:

- The target model is well-typed whenever RHS is well-typed. Note that ill-typed model elements can only stem from ill-typed pattern elements. The type correctness of RHS is, however, checked mechanically once the rule is implemented with a QVT editor such as Together Architect 2006.
- Multiplicity constraints should always be checked carefully whenever the rule creates or deletes objects/links. Please note that also all multiplicities from inherited associations have to be obeyed.
- Arguing about the preservation of well-formedness rules requires the most effort. In a first step, one has to identify all those well-formedness rules of the metamodel that might be affected by the refactoring. We have done this task for all UML/OCL refactorings manually, but, recently, an interesting approach to automate this filtering has been developed by Cabot [10,11]. In a second step, convincing arguments have to be found that the filtered well-formedness rules are obeyed in all possible target models. We show in Sect. 5 on one example, how such an argumentation can be formalized by using the KeY-system.

2.2.4 Using when-clauses to ensure syntax preservation

The argumentation on the preservation of well-formedness rules is not always as trivial as the one for *RenameItem1* shown in Table 1. Often, a refactoring rule can (potentially) destroy many of the metamodel's well-formedness rules. In this case, we need a more sophisticated argumentation why the refactoring rule is nevertheless syntax preserving. To illustrate the problem, we add another invariant to the *Item-View* metamodel:

```
context Item inv UniqueNameInInheritance:
  self.allParents().name->excludes(self.name)
```

Informally speaking, this well-formedness rule requires the name of each *Item* object to be different from the name of all its (transitive) parents. Obviously, this well-formedness rule is not always preserved by *RenameItem1* since there is no provision made to ensure that *newName* is not already used by any of the parents. This problem can be fixed by using QVT's when-clause. A first (not fully successful) attempt to correct the rule *RenameItem1* is shown in Fig. 3.

The when-clause adds some new restrictions for the application of the rule. The rule is only applicable on those subgraphs of the source model that (1) match with LHS and (2) for which the expression given in the when-clause is evaluated to true. Note that identifiers for pattern objects (here *it*) can be used within the when-clause. Informally speaking, the rule is now only applicable on such *Item* objects whose parents have not already used *newName* as a name.

Unfortunately, this when-clause does not preserve *UniqueNameInInheritance* in all cases. For example, suppose the rule is applied on concrete *Item* object *it1* whose parents have names different from *newName*. After the rule has been executed (and *it1* has been renamed to *newName*), the well-formedness rule is indeed valid for *it1*. However, it might be the case that the source model contains another object *it2*, which is a child of *it1* and which has the name *newName* as well. Then, *UniqueNameInInheritance* does not hold anymore in the target model for *it2*, because it has now the same name as its parent *it1*.

In order to prevent such cases, the when-clause has to check not only for the parent items but also for the child items whether *newName* is already used as a name. The following additional operations facilitate to write the necessary when-clause in a compact way.

Table 1 Arguing on the syntax preservation of *RenameItem1*

Type declarations	In the RHS of the rule, all pattern objects, their attribute values and links between them are well-typed according to the metamodel.
Multiplicities	Since neither objects nor links are created/deleted by the rule application, all multiplicity constraints are automatically obeyed in the target model.
Well-formedness rules	The only well-formedness rule is *CycleFree* and the only change on a model that could make it invalid is adding links for the *Inheritance* association to the model. Since this does not happen in the *Rename*-rules, the invariant *CycleFree* is preserved.

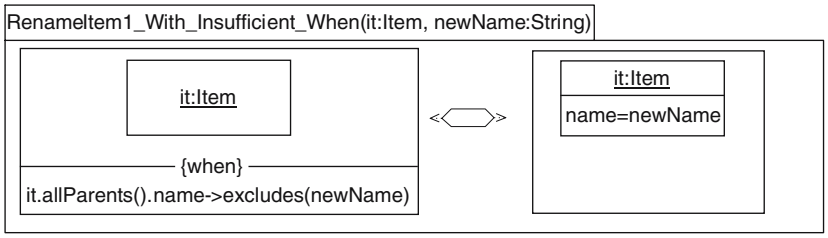

Fig. 3 Renaming of selected item—when-clause is not sufficient to preserve *UniqueNameInInheritance*

```
context Item def:
  allChildren():Set(Item)=
    self.child->union(self.child.allChildren()->asSet())

context Item def:
  allConflictingNames():Bag(String)=
    self.allParents().name
    ->union(self.allChildren().name)
    ->including(self.name)
```

The corrected version of the *RenameItem1* refactoring is shown in Fig. 4. Note that the rule is applicable at most once and, thus, termination of the rule application is always ensured.

Actually, many refactoring rules for UML/OCL have a very similar when-clause because the UML metamodel contains quite a few well-formedness rules imposing unique names for model elements.

2.3 Extends-relationship between QVT rules

Another important concept of QVT is the *extends*-relationship between rules. The need for extensions of QVT rules is motivated by the next well-formedness rule:

```
context Item inv DerivedViewName:
  self.view->forAll(v| v.name = 'viewOf_'.concat(self.name))
```

Informally speaking, *DerivedViewName* stipulates that all views attached to the same item must share the same name, which can be derived from the item's name.

Again, each of the above given *RenameItem*-refactorings would fail to preserve this invariant. Interestingly, there are now at least three possibilities to fix this problem. One possibility is to disallow renaming of *Item* objects if they have already a view attached. This is easily realized by extending the existing when-clause shown in Fig. 4 by `and it.view->isEmpty()`. A second possibility is to delete all attached `View` objects when an `Item` object is renamed. The third possibility is to rename all attached *View* objects accordingly.

Figure 5 shows the realization of the third possibility in form of an *extension* of *RenameItem1*. The new rule is called *UpdateViewNames* and is applied in the following way: Whenever a match for LHS of the extended rule (*RenameItem1*) is found, all its extensions (here *UpdateViewNames*) are applied on the current LHS-match as often as possible. Note that the patterns LHS/RHS from the extension rule can use elements from the extended rule. For example, the pattern object *it:Item* in LHS of *RenameItem1* refers for every match in the source model to the same model element as the pattern object *it:Item* in LHS of *UpdateViewNames*. The pattern object *v:View* in LHS of *UpdateViewNames* matches iteratively with any *View* object that is attached to *it*. The RHS of *UpdateViewNames* enforces to rewrite the name of all these *View* objects with the value `'viewOf_'.concat(newName)`. The when-clause in *UpdateViewNames* ensures the termination of the rule application.

2.4 Metamodel of UML/OCL

We present now all parts of the official metamodel for UML 1.5 and OCL 2.0 that are relevant for the refactoring rules presented in Sect. 3. Our refactoring rules are still based on UML 1.5 (and not on UML 2.0, which was already the official UML version in time of writing this paper) for many reasons. First of all, we had the goal to stay as close as possible to the conference version of this paper [18], where initial versions of the refactoring rules from Sect. 3 are presented. More

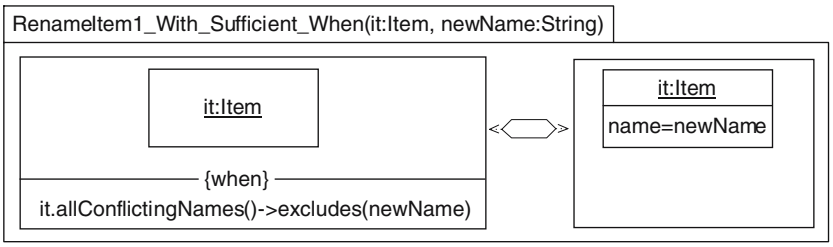

Fig. 4 Renaming of selected item—correct version for *UniqueNameInInheritance*

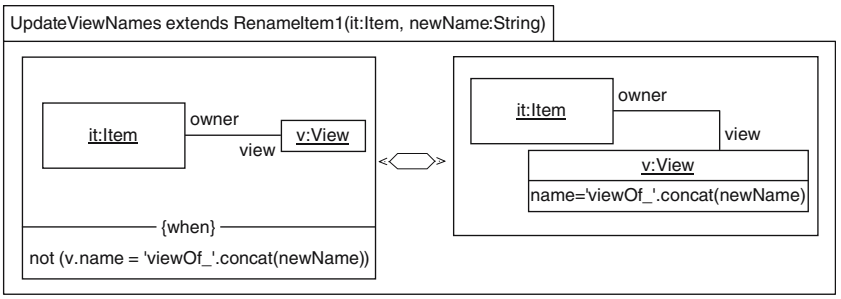

Fig. 5 Extension of *RenameItem1*

importantly, however, the implementation of our approach had topmost priority and we encountered numerous problems when trying to apply the QVT engine we used to repositories containing UML 2.0 models.

The refactoring rules we formulate in Sect. 3 for UML 1.5 are also applicable in a very similar way to UML 2.0 models. In Sect. 3.3, we describe a possible migration process for refactoring rules from UML 1.5 to UML 2.0. As an example, the UML 1.5 refactoring rules *MoveAttribute* and *MoveAssociationEnd* are merged to *MoveProperty* for UML 2.0. So far, however, we were unable to implement this refactoring rule due to the above mentioned technical problems.

2.4.1 Declaration of metaclasses

Figures 6 and 7 show relevant parts of the official metamodel for UML 1.5 and OCL 2.0 (for a complete definition see [22,24]). The chosen fragment of the UML-part of the metamodel concentrates on the main concepts of class diagrams. The OCL-part covers the most important OCL expressions.

2.4.2 Well-formedness rules

The metamodel for UML and OCL contains hundreds of well-formedness rules and, as we have seen above, each well-formedness rule can become crucial if the syntax preservation of the refactoring rule is discussed. Our refactoring rules are designed to preserve only some, but—as we believe—the most important well-formedness rules of UML/OCL. This decision was a trade-off between the completeness of our

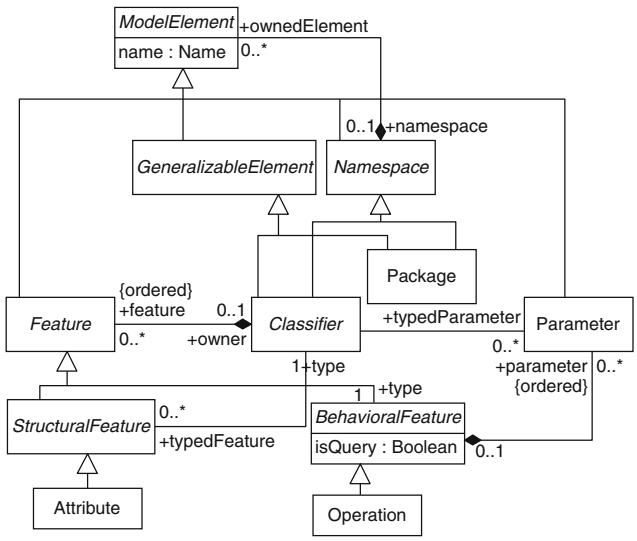

Fig. 6 UML—core backbone and relationships

approach and the readability of when-clauses, which grow when more well-formedness rules have to be preserved.

Since the UML/OCL refactorings considered in this paper mainly rename, move, or add model elements, the well-formedness rule ensuring the uniqueness of used names in a classifier is easily broken when the refactoring rules do not make any provision. According to the UML 1.5 metamodel, all attributes, opposite association ends and other owned elements (e.g. contained classes) of a classifier must have a unique name. Moreover, these names must also not be used by any of the parent classifiers. A (slightly simplified) version of the official well-formedness rule looks as follows:

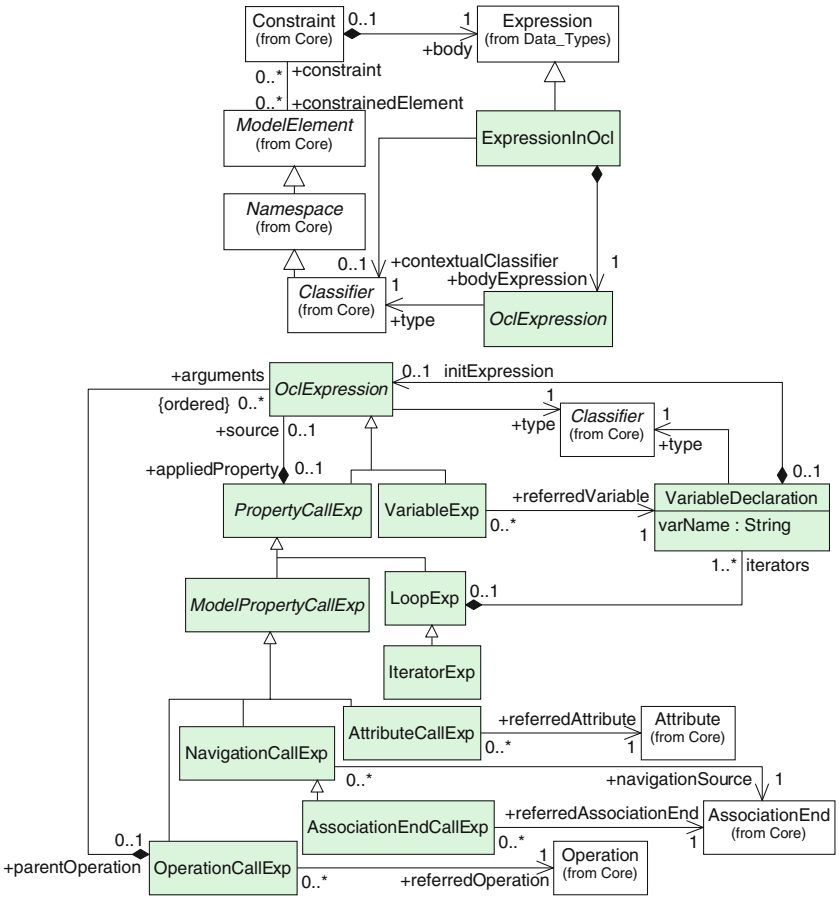

Fig. 7 OCL—overview and *PropertyCallExp*

```
context Classifier inv UniqueUsedName:
  self.allUsedNames()->forAll(n|
                  self.allUsedNames()->count(n)=1)

context Classifier def:
  allUsedNames():Bag(String)=
    self.allParents()->including(self)
    ->iterate(c; acc:Bag(String)=Bag{}|
      acc->union(c.oppositeAssociationEnds().name)
        ->union(c.attributes().name)
        ->union(c.ownedElement.name))
```

As we have seen in the *Item-View* example, it is convenient to define an additional operation that will capture also the names already used in the children of a classifier.

```
context Classifier def:
  allConflictingNames():Bag(String)=
    self.allUsedNames()->union(
      self.allChildren()->including(self)
      ->iterate(c; acc:Bag(String)=Bag{}|
        acc->union(c.oppositeAssociationEnds().name)
          ->union(c.attributes().name)
          ->union(c.ownedElement.name)))
```

Please note that the definition of many additional operations such as

Classifier.allParents():Set(Classifier),
Classifier.allChildren():Set(Classifier),
Classifier.conformsTo(Classifier):Boolean, etc.
is omitted here but can be found in the official definition of the metamodel [22,24].

A second important well-formedness rule in the metamodel of UML 1.5 is that two operations with the same signature can be owned by any two classifiers (even if one of the classifiers is a specialization of the other one), but the two operations cannot be owned by the same classifier.

```
context Classifier inv UniqueMatchingSignature:
  self.operations()->forAll(f,g|
    f.matchesSignature(g) implies f=g)
```

3 A catalog of UML/OCL refactoring rules

In this section, we present the most important refactoring rules for UML 1.5 class diagrams. These rules handle OCL 2.0 constraints that are attached to the refactored class diagram. At the end of this section, in Sect. 3.3, an example for a possible migration from refactoring rules for UML 1.5 to such for UML 2.0 is given. Note that each refactoring rule is

Table 2 Overview of UML/OCL refactoring rules

Refactoring rules	Influence on syntactical correctness of OCL constraints	
	MM-representation	Textual notation
RenameClass	No	Yes
RenameAttribute	No	Yes
RenameOperation	No	Yes
RenameAssociationEnd	No	Yes
PullUpAttribute	No	No
PullUpOperation	No	No
PullUpAssociationEnd	No	No
PushDownAttribute[a]	No	No
PushDownOperation[a]	No	No
PushDownAssociationEnd[a]	No	No
ExtractClass	No	No
ExtractSuperclass	No	No
MoveAttribute	Yes	Yes
MoveOperation	Yes	Yes
MoveAssociationEnd	Yes	Yes

[a]Only *push down* to one subclass is considered in this paper

syntax preserving only with respect to the part of the UML 1.5 metamodel given in the last section. Some of the refactoring rules are designed also for the preservation of some further important well-formedness rules (encoding restrictions for OCL expressions) that are given in the text at appropriate places.

Our catalog (see Table 2 for an overview) is inspired by the refactoring rules for the static structure of Java programs given by Fowler in [13]. We took the freedom to change some of the rule names introduced by Fowler in order to indicate UML as their new application domain (e.g., *MoveMethod* became *MoveOperation*). In few cases, not only the name but also the semantics of the rule has changed (e.g., *PullUpOperation* moves in our version only the selected operation whereas in [13] also relevant fields are moved).

Not all class diagram refactoring rules have an influence on attached OCL constraints. Table 2 classifies the rules according to this criterion. Note that *Rename*-refactorings require to change the textual representation of relevant constraints but not their metamodel-representation.

3.1 Rules without influence on OCL

3.1.1 RenameClass/Attribute/Operation/AssociationEnd

These rules are very similar to each other and only *RenameAttribute* (see Fig. 8) is discussed here in detail. The *Rename*-rules differ mostly in the when-clause, whose purpose is to check whether the proposed new name is already in use in the enclosing *Namespace* of the renamed element.

In rule *RenameAttribute*, the parameter *a* refers to the attribute whose name should be changed. Since *RenameAttribute* is designed to work on class diagrams, we make in LHS the assumption that the owner of *a* is a *Class*, though it could be any *Classifier* according to the metamodel (similar assumptions are made also in all other refactoring rules). The first line of the when-clause is necessary to guarantee termination when applying the rule. The second line ensures the applicability of the rule only in cases, in which the new name of the attribute is not already used within the owning class or one of its parents or children.

At a first glance, renaming an attribute requires to change all attached OCL constraints where the attribute is used. However, these changes are required only for the textual notation. If the attached OCL constraint is seen as an instance of the metamodel, then this instance remains the same. Note that the OCL-part of the metamodel *refers* to the UML-part. Thus, each renaming made within the underlying UML class diagram is automatically propagated to all OCL expressions that use the renamed element.

3.1.2 PullUpAttribute/Operation/AssociationEnd

A *PullUp*-rule never causes a change in the attached OCL constraints. The constraints, however, cannot be ignored when applying *PullUp*-rules (an exception is the very simple *PullUpAttribute* rule). Similarly to a *Rename*-rule, whose application can be prevented by a badly chosen value for parameter *newName*, a *PullUp*-rule becomes non-applicable if certain constraints are attached to the current class diagram. Again, this application condition is expressed in the when-clause of the rule.

The rule *PullUpAttribute* removes one attribute from a class and inserts this attribute into one of its superclasses; a concrete example is shown in Fig. 9.

The LHS of the rule (Fig. 10) requires the owning class *son* of the selected attribute to be a direct subclass of the destination class *father*. The when-clause prevents the applicability of the rule in situations in which another subclass of *father*, i.e. a sibling of *son* or one of its children, already uses the name of the moved attribute. Note that in such situations the query *allConflictingNames()* applied on *father* would yield a bag that contains the name of the moved attribute at least twice. The RHS formalizes that the owner of attribute *a* has changed from class *son* to class *father* (link from *a* to *son* is deleted and link to *father* is created).

The rule *PullUpAttribute* has no influence on OCL constraints because a refactoring step widens the applicability of the moved attribute. In the OCL constraints attached to the source model, the moved attribute *a* can only occur in attribute call expressions (*AttributeCallExp*) of form *exp.a*.

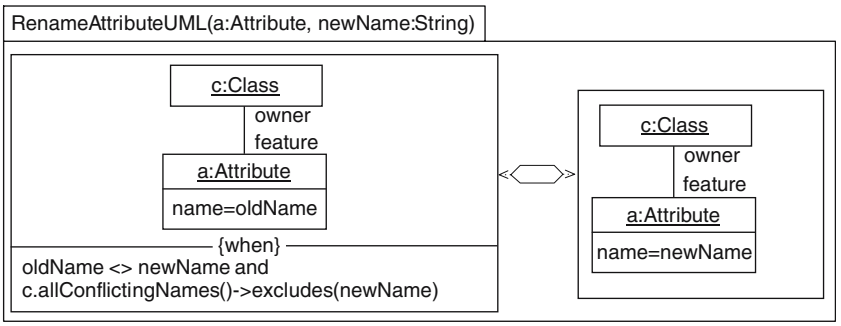

Fig. 8 Formalization of *RenameAttribute* refactoring

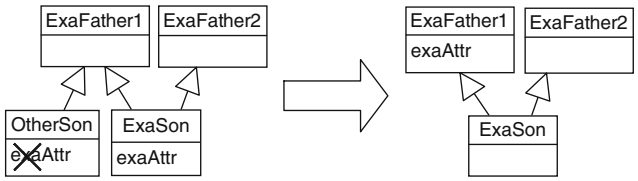

Fig. 9 Example of applying *PullUpAttribute*

Here, the type of expression *exp* must conform to *son*, the owning class of attribute *a*. After the refactoring, *exp.a* is still syntactically correct because the type of *exp* conforms also to *father*, the new owner of attribute *a*.

The rule *PullUpOperation* shown in Fig. 11 is almost identical to *PullUpAttribute* except for the when-clause. Since moving an operation can make the well-formedness rule *UniqueMatchingSignature* (see Sect. 2.4.2) invalid, it is checked in the first line of the when-clause that *father* does not own already an operation whose signature matches with the one of the moved operation *o*.

The rest of the when-clause prevents the following situation: By moving operation *o* from *son* to *father*, also the predefined variable *self*, which is used in the pre-/postconditions attached to *o*, changes its type from *son* to *father*. Consequently, expressions such as *self.attSon*,[1] where *attSon* is an attribute declared in *son*, would become syntactically incorrect after the refactoring. The when-clause checks exactly for the occurrence of these cases. For the sake of a concise description, the when-clause uses queries such as *getAllSubexpressions()*, *isPossibleToChangeTypeTo()* which are not defined in the metamodel of UML/OCL but whose definitions are made available in [19].

Fowler has faced in [13] the same problem for the corresponding refactoring rule *PullUpMethod* and proposes to pull up in such a situation also all used attributes from *son* to *father*. We do not follow this approach here since Fowler's solution could be simulated in our setting by a sequential application of multiple *PullUp*-rules.

[1] Note that OCL allows in the textual notation to suppress *self*. Thus, the variable *self* within *self.feature* is sometimes given only implicitly.

Besides the *self*-expressions within the constraints attached to operation *o*, also query expressions of form *exp.o(...)* are affected by the refactoring (however, such expressions are only possible if *o* is a query). Note that these expressions cannot become syntactically incorrect because OCL's type rules require the type of *exp* to conform to the owner of operation *o* (same argumentation as for *PullUpAttribute*).

The rule *PullUpAssociationEnd* shown in Fig. 12 checks for the absence of expressions of form *exp.ae.attSon*, what corresponds to the check for *self*-expressions in *PullUpOperation*. Also expressions of form *exp.aet*, where *aet* refers to the opposite association end of *ae*, are affected by the refactoring but their syntactical correctness is always preserved (same argumentation as for query expressions *exp.o(...)* in rule *PullUpOperation*).

3.1.3 PushDownAttribute/Operation/AssociationEnd

The *PushDown*-rules[2] are in many respects inverse to the *PullUp*-rules. While *PullUp*-rules change the owner of elements from *son* to *father*, *PushDown*-rules move them from *father* to *son*. We have already observed for *PullUp*-rules that relevant (i.e. affected) OCL expressions can be divided into two groups and that the when-clause had to make provision only for one group of expressions. As we will see now, *PushDown*-rules have to make provision for exactly the opposite group of expressions.

The rule *PushDownAttribute* moves an attribute from the parent to a selected subclass (see Fig. 13). As described by Fowler in [13] for the corresponding rule *PushDownField*, the attribute must be moved to that subclass that covers the 'usage' of the attribute. The attribute *a* is *used* in a class *c* if at least one of the constraints attached to the class diagram has a subexpression of form *exp.a* and *exp* has a type conforming to *c*.

The formalization of *PushDownAttribute* is given in Fig. 14. The when-clause has to check possible name conflicts

[2] We consider here only rules that push a model element down to exactly one subclass. For the more general case of pushing down to multiple subclasses, see [18].

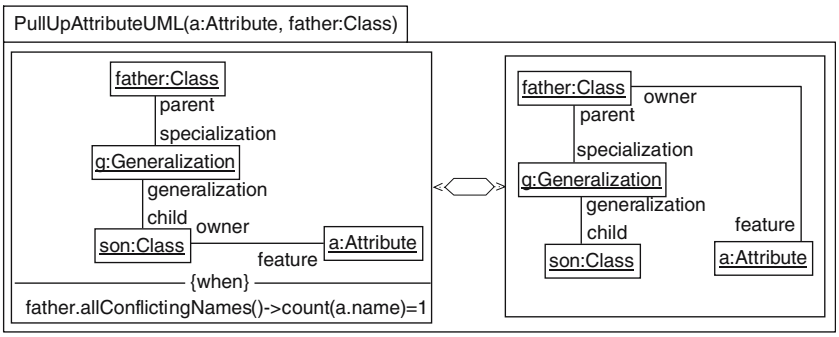

Fig. 10 *PullUpAttribute* refactoring rule

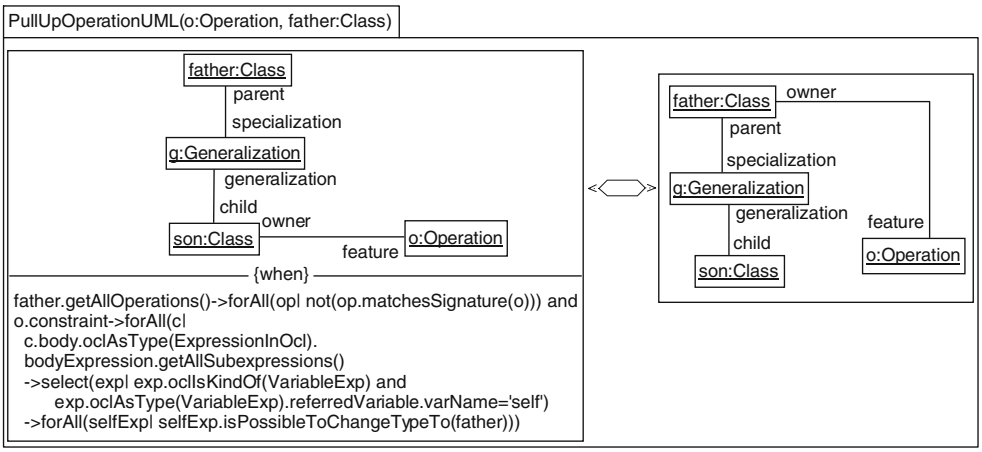

Fig. 11 *PullUpOperation* refactoring rule

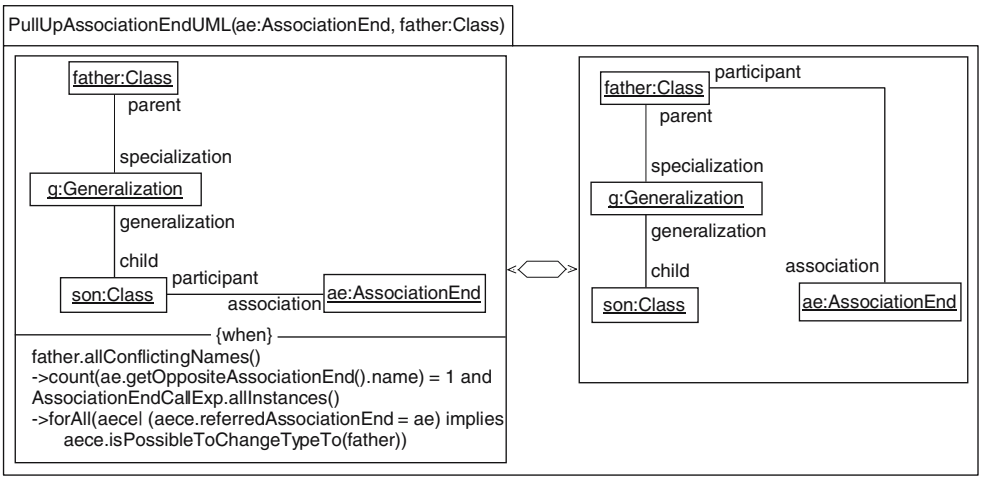

Fig. 12 *PullUpAssociationEnd* refactoring rule

in *user*, but – in addition to the check done in *PullUpAttribute* – also for occurrences of expressions of form *exp.a* where the type of *exp* conforms to *father* but not to *user*.

The when-clause of rule *PushDownOperation* (see Fig. 15) checks for query expressions with operation *o* but not for *self* -expressions (note that rule *PullUpOperation* checks the opposite).

Rule *PushDownAssociationEnd* is defined analogously to *PushDownOperation* and, thus, omitted here.

3.1.4 ExtractClass/Superclass

The rule *ExtractClass* (see Fig. 16) creates an empty class *extracted* in the same namespace *nsp* as the selected class *src*

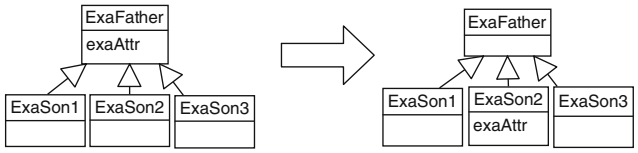

Fig. 13 Example of applying *PushDownAttribute*

and connects both classes with a new association. The multiplicity of the new association is 1 on both sides. Besides the class *src*, also the name for the extracted class and the two role names for the newly created association have to be passed as parameters. The new class name *newName* must not be already used in the enclosing namespace of *src*. To express this formally, the when-clause has to make a case distinction on the actual type of the enclosing namespace (either *Classifier* or *Package* according to our metamodel shown in Fig. 6). While the role name for the association end on *src* can be chosen arbitrarily, the other one must not be in the set of conflicting names for *src*.

The rule *ExtractSuperclass* (see Fig. 17) creates an empty class as well but inserts the newly created class between the source class and one of its direct parent classes. Note that *ExtractClass/Superclass* differ from the corresponding rules given by Fowler in [13]. Our rules are more atomic since they do not move features from the source class to the newly created class. In order to move features to the new class one could apply the refactorings *MoveAttribute/AssociationEnd/Operation* or *PullUpAttribute/AssociationEnd/Operation*.

Applying the rules *ExtractClass/Superclass* cannot alter the syntactical correctness of attached OCL constraints because both rules merely introduce new model elements and do not delete or change old ones.

3.2 Rules with influence on OCL

3.2.1 MoveAttribute/Operation/AssociationEnd

The application of rule *MoveAttribute* is usually driven by the wish to make a class smaller; an example of this refactoring is shown in Fig. 18.

The selected attribute is moved from a source to a destination class over an association with multiplicity 1 on both ends. If source and destination class are connected with more than one such association, it is for the refactoring of the attached OCL constraints important to know, over which association the attribute was moved. Thus, the second parameter of the rule shown in Fig. 19 is an association end that identifies both the destination class and the used association. The restriction that attributes can only be moved over an association with multiplicity 1-1 ensures the semantics preservation of the rule (recall that the preservation of semantics is not discussed in this paper). However, the multiplicity restriction is sufficient but not necessary for semantics preservation of

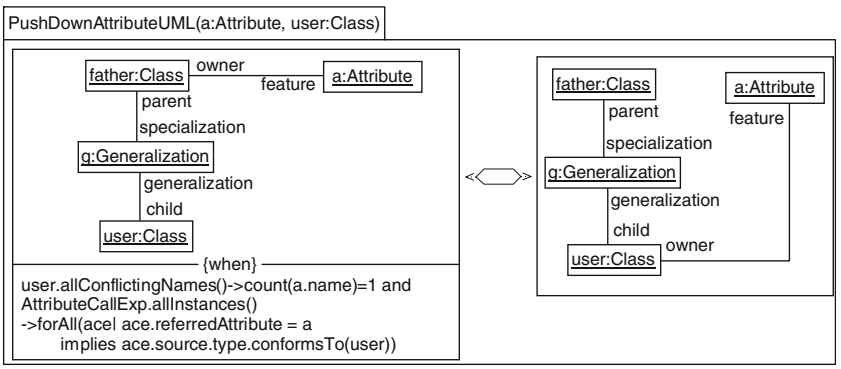

Fig. 14 *PushDownAttribute* refactoring rule

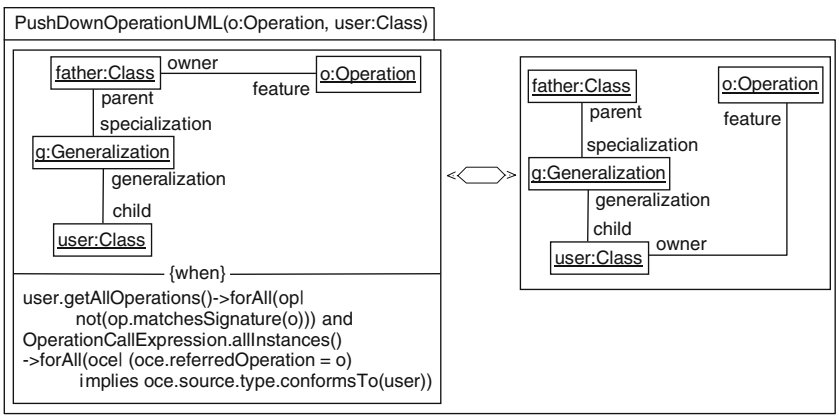

Fig. 15 *PushDownOperation* refactoring rule

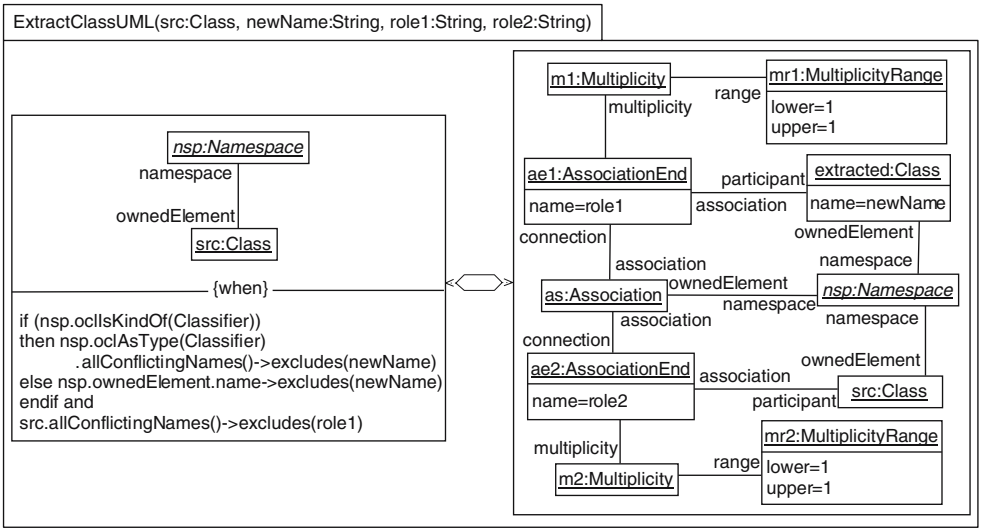

Fig. 16 *ExtractClass* refactoring rule

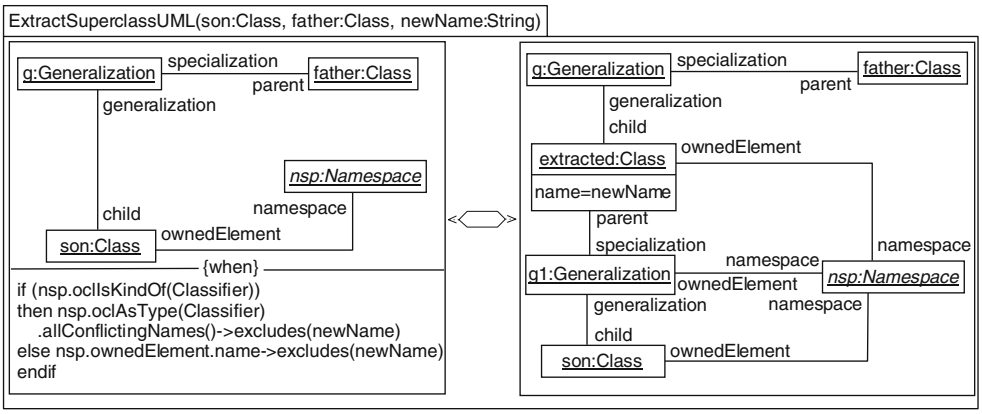

Fig. 17 *ExtractSuperclass* refactoring rule

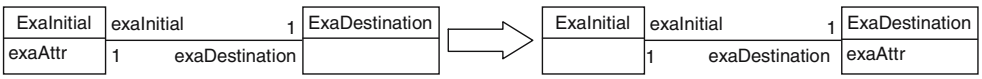

Fig. 18 Example of applying *MoveAttribute*

a rule application. The interested reader is referred to [4], where the semantics preservation property is analyzed and proven for some variations of *MoveAttribute*.

Analogously to the changes of Java code described by Fowler for the corresponding refactoring *MoveField*, this rule must update attached OCL constraints on all locations where the moved attribute is used. The necessary change of the OCL expressions can be seen as a kind of *Forward Navigation*. For the example from Fig. 18, this would mean that an expression of form *exp*.`exaAttr`, where *exp* has a type conforming to the source class `ExaInitial`, is not type correct after the attribute has moved to the destination class. Thus, the term *exp*.`exaAttr` should be rewritten with *exp*.`exaDestination.exaAttr`. This refactoring of OCL constraints is formalized by a second rule (see Fig. 20), which extends the first rule. The RHS inserts between the attribute call expression *ace* and its source expression *oe* a new association end call expression *aece* that realizes the forward navigation.

The rule *MoveOperation* is usually applied when some class has too much behavior or when classes are collaborating too much.

The formalization of the UML part of *MoveOperation* is similar to that of *MoveAttribute* and shown in Fig. 21. As for *MoveAttribute*, the association connecting source and destination class must have on both ends multiplicity 1. The

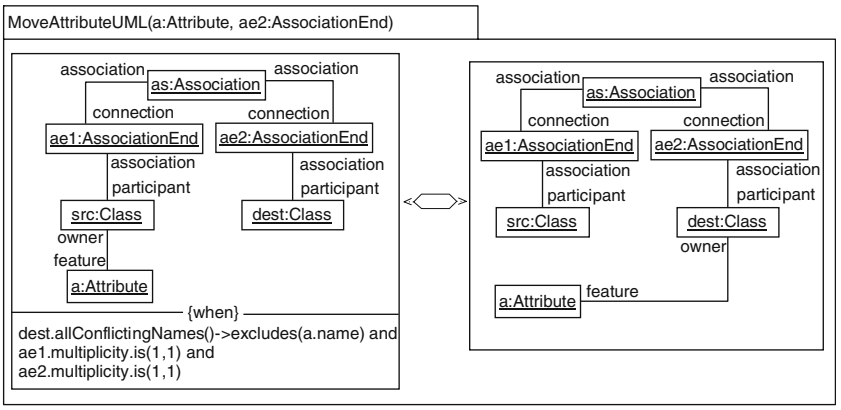

Fig. 19 UML part of *MoveAttribute* rule

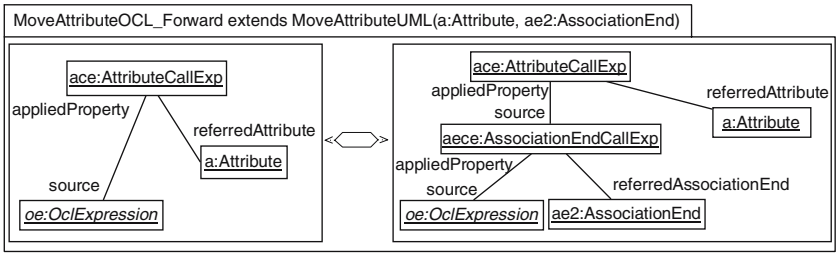

Fig. 20 OCL part of *MoveAttribute* rule (forward navigation)

main difference to *MoveAttribute* is the when-clause that is tailored to preserve the well-formedness rule *UniqueMatchingSignature* (see Sect. 2.4.2).

The changes induced on attached OCL constraints can be described in three steps. The last two steps refer to the different handling of query- and *self*-expressions, what is fully analogous to what we have already observed in *PullUp*- and *PushDown*-rules.

Change Context: If a constraint is attached to the moved operation (e.g. as pre-/postcondition) then the context of this constraint has to be changed, for example from **context** `ExaInitial::exaOp()` to **context** `ExaDestination::exaOp()`. Fowler describes in [13] informally this step as "*Copy the code from the source method to the target*".

The context of a constraint is formalized in the OCL metamodel by the metaassociation from *ExpressionInOcl* to *Classifier* with role name *contextualClassifier*. However, this metaassociation is derived. Consequently, changing the context of a constraint is subsumed in our setting by changing the owner of the moved operation, as formalized in Fig. 21.

Forward Navigation (Handle Query): In case that the moved operation is a query, all operation call expressions have to redirect their references to the moved operation (see Fig. 22). This means to substitute all expressions *exp*.`exaOp()` by *exp*.`exaDestination.exaOp()`. This step corresponds to "*Turn the source method into a delegating method*" from Fowler's book.

Backward Navigation: All occurrences of *self*-expressions in the constraints attached to the moved operations have changed their type from `ExaInitial` to `ExaDestination`. This requires to rewrite the expression *self* by *self*.`exaInitial`. This navigation is possible due to multiplicity 1 on the end of class `ExaInitial`.

What is left to be done is to embed the new expression *self*.`exaInitial` at the same location at which the original expression *self* was placed. In the refactoring rule from Fig. 23, this has been formalized by the link between *ve* and *exp*. Note that there is no such metaassociation *context-subExpressions* in the official UML/OCL metamodel. The metaassociation has been defined here as a derived association that subsumes all existing owning-relationships between OCL expressions, such as *appliedProperty-source*, *parentOperation-arguments*, etc. A similar extension of the OCL 2.0 metamodel has been also proposed by Correa and Werner in [12].

For the backward navigation step, Fowler says: "... *create or use a reference from the target class to the source*".

The rule *MoveAssociationEnd* is very similar to *MoveAttribute* for the refactoring of the UML part; Fig. 24 shows an example and Fig. 25 the formalization as QVT rule.

The refactoring of OCL constraints consists of two parts, one for forward and one for backward navigation. The forward navigation is analogous to *MoveAttribute* and rewrites association end call expressions of form *exp*.`roleB` by *exp*.`exaDestination.roleB`. Figure 26 shows the formalization as QVT rule. Note that the association end *aet*, which is determined by the when-clause of the rule, matches with `roleB` in the example from Fig. 24.

The backward navigation has to address the problem that the type of expression *exp*.`roleA` has changed – depending

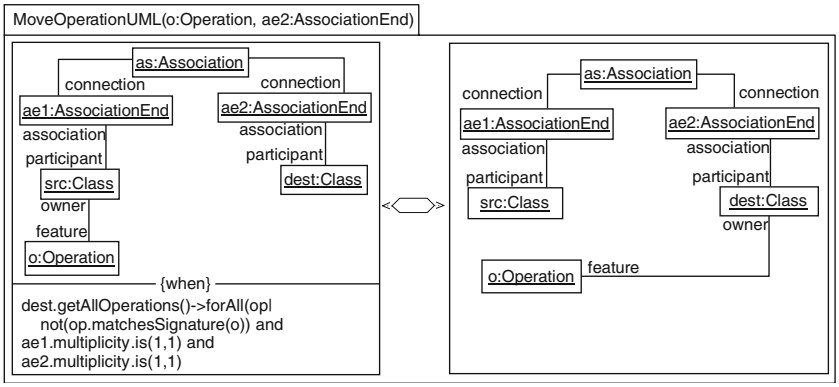

Fig. 21 UML part of *MoveOperation* rule

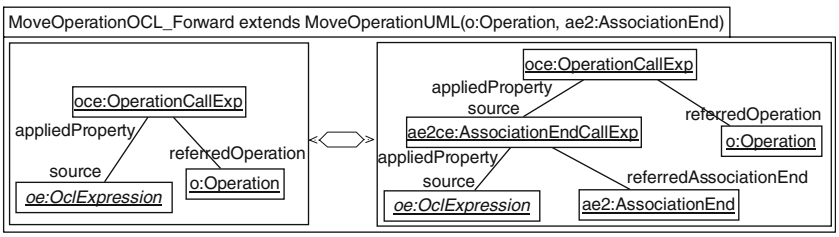

Fig. 22 OCL part 1 of *MoveOperation* rule (forward navigation/handle query)

on the multiplicity and ordering of the association end with name `roleA` – from `ExaInitial`, `Set(ExaInitial)`, `Sequence(ExaInitial)` to `ExaDestination`, `Set(ExaDestination)`, `Sequence (ExaDestination)`, respectively.

The two rules shown in Fig. 27 cover the first and the third case correctly. If the association end with name `roleA` has multiplicity 0..1 or 1..1, then the expression *exp*.`roleA` is rewritten by *exp*.`exaInitial.roleA`. For all other (many-valued) multiplicities, the original expression *exp*.`roleA` is rewritten by *exp*.`roleA->collect(it| it.exaInitial)`. Note that we have chosen `it` as the name for the iterator variable but any other name could have been taken as well. Our rewriting is, however, only fully correct if the association end named `roleA` was ordered and the original and new expressions have type `Sequence(ExaInitial)`. For unordered association ends, the expression *exp*.`roleA` should[3] be rewritten by *exp*.`roleA->collect(it| it.exaInitial)->asSet()` in order to ensure that the original and new expression have the same type `Set(ExaInitial)`.

Finally, the new expression has to be embedded at the same location as the original expression what is formalized analogously to rule *MoveOperation*.

3.3 Reformulation of refactoring rules for UML 2.0

As already mentioned, the above given catalog of refactoring rules is defined on top of the metamodel for UML 1.5 and thus not directly applicable for UML 2.0 models. In this subsection, we discuss how our refactoring rules can be reformulated for UML 2.0. As an example we have chosen one of the most drastic changes in UML 2.0 for class diagrams: the unification of attributes and opposite association ends.

Figure 28 shows the relevant part of the UML 2.0 metamodel [23] and the aligned OCL 2.0 metamodel [27] (*AttributeCallExp* and *AssociationEndCallExp* were unified to *PropertyCallExp*). An instance of metaclass *Property* in the UML 2.0 metamodel can represent either an attribute of a class (in this case, it is an *ownedAttribute* of the class), a navigable association end (encoded as an *ownedAttribute* of the class at the opposite association end and, furthermore, a *memberEnd* of its association), or a non-navigable association end (only a *memberEnd* of its association).

3.3.1 MoveProperty

The rule *MoveProperty* is the counterpart of *MoveAttribute* and *MoveAssociationEnd* already specified for UML 1.5. For the sake of brevity, we assume that the moved property has the multiplicity 0..1 or 1..1. We have already shown in rule *MoveAssociationEnd* how a multiplicity greater than 1 can be handled.

The transformation rule formalized in Fig. 29 is very similar to the UML part of *MoveAttribute* and *MoveAssociationEnd* shown in Figs. 19 and 25, respectively. *MoveProperty* moves one property from the source to the destination class. The when-clause specifies that this rule is applicable only if the name of the moved property is not in conflict with the destination class.

In Fig. 30 and Fig. 31, the necessary forward and backward navigation changes on the attached OCL constraints are formalized analogously to *MoveAssociationEnd*.

[3] This exceptional case is not reflected in the rule shown in Fig. 27 but has been implemented in our tool (see [19]).

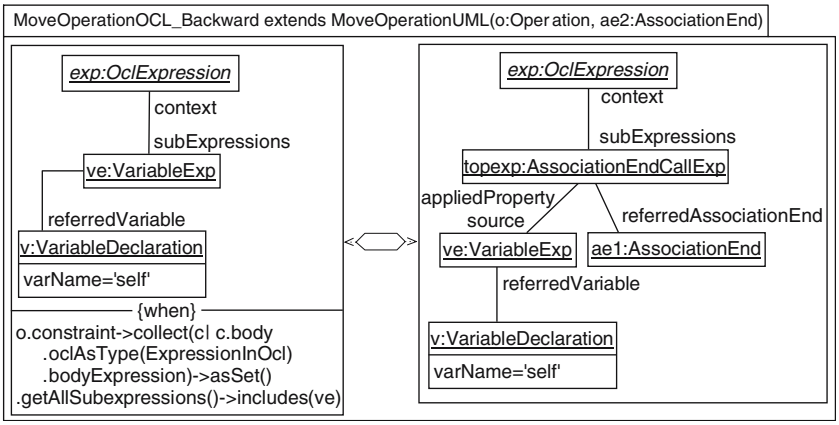

Fig. 23 OCL part 2 of *MoveOperation* rule (backward navigation)

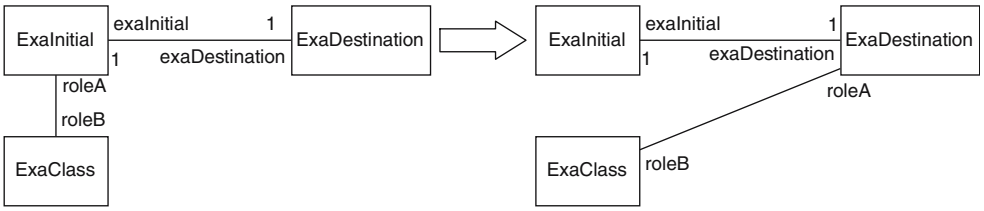

Fig. 24 Example of applying *MoveAssociationEnd*

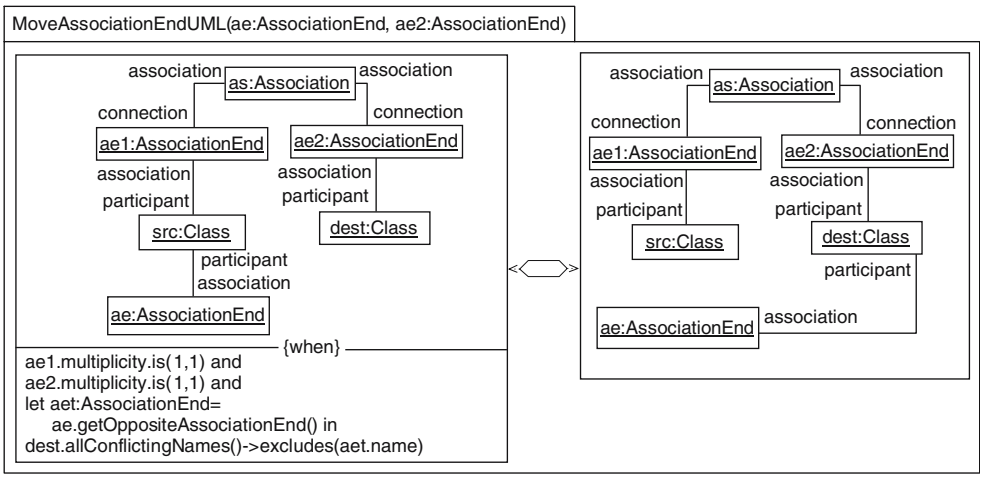

Fig. 25 UML part of *MoveAssociationEnd* rule

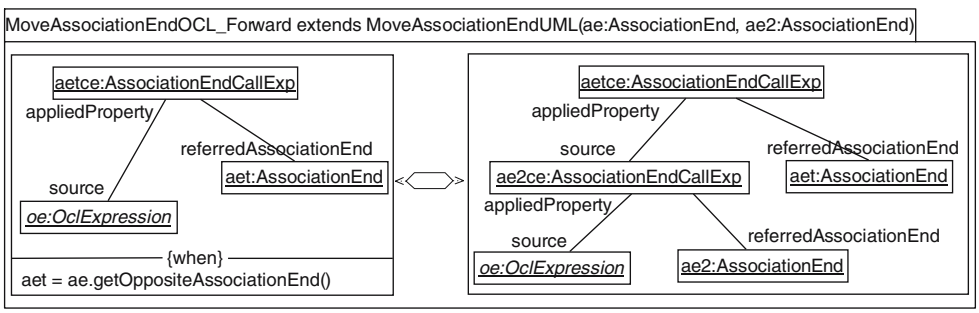

Fig. 26 OCL part 1 of *MoveAssociationEnd* rule (forward navigation)

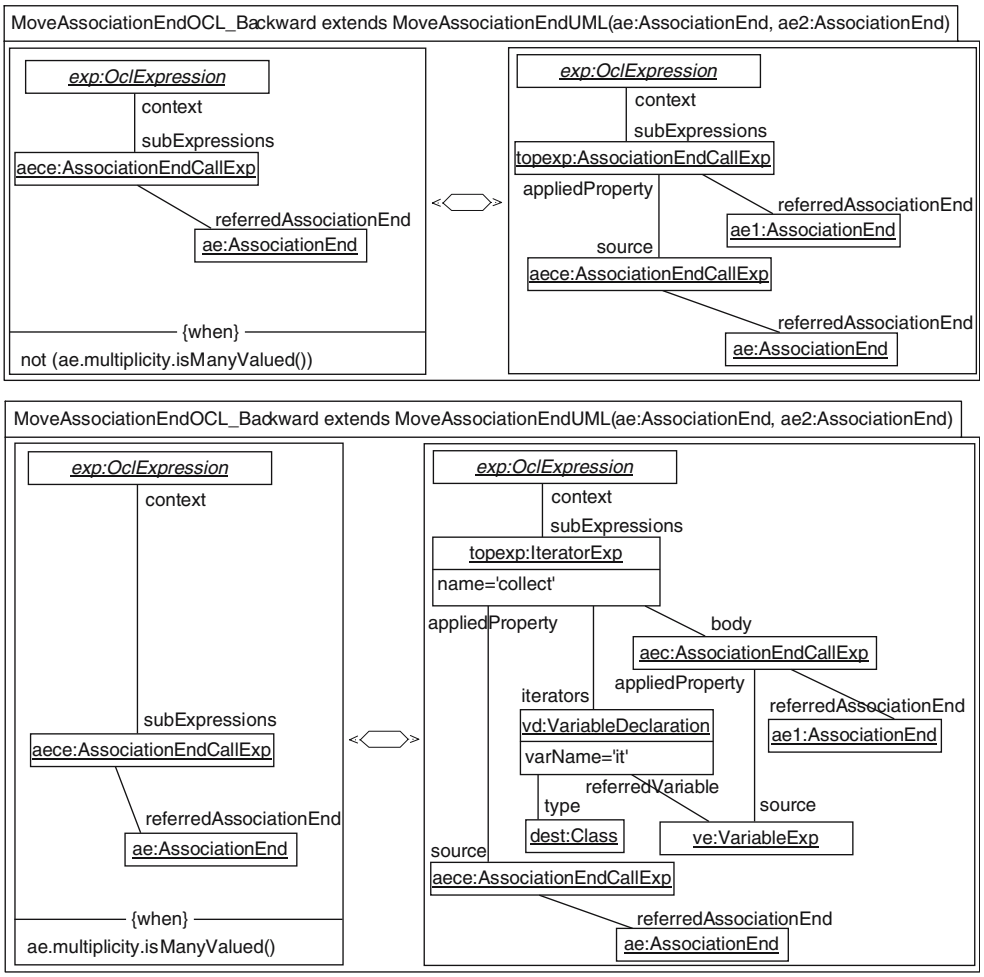

Fig. 27 OCL part 2 of *MoveAssociationEnd* rule (backward navigation)

4 Implementation of refactoring rules in QVT

We implemented all rules of our refactoring catalog using the QVT implementation of Together Architect 2006 for Eclipse [9]. Our rule implementation is based on the metamodel for UML 1.5 class diagrams and OCL 2.0 as shown in Sect. 2.4. The implementation of the rules, together with the used metamodel can be downloaded from [19].

4.1 Overview

Together Architect 2006 for Eclipse implements a large body of the QVT standard.[4] The implemented version of the transformation rules looks at the first glance quite different from what was specified in graphical form in Sect. 3. There are obvious changes on the notational level – Together Architect 2006 supports so far only the textual notation of QVT – but, in general, we made the experience that implementing the refactoring rules in Together Architect 2006 for Eclipse is a straightforward and – thanks to Together's matured editor and debugger for OCL – also a painless task.

Before we describe in more details the transition from a refactoring rule given in graphical notation to an implementation in textual QVT, let us recall the steps to follow when applying a rule on a concrete source model. These steps are

1. Find a substructure in the source model that matches with the LHS of the transformation rule. If no LHS-matching substructure exists, the application of the transformation rule terminates.
2. Rewrite the identified substructure with the RHS under the same matching.
3. Continue with step 1 where the source model is now the model obtained by the last rewriting step (step 2).

Note that, theoretically, it could be the case that the rewriting step 2 adds a new LHS-matching substructure that has not been present in the original source model. For the refactoring

[4] A list of missing features not implemented yet is shipped with Together.

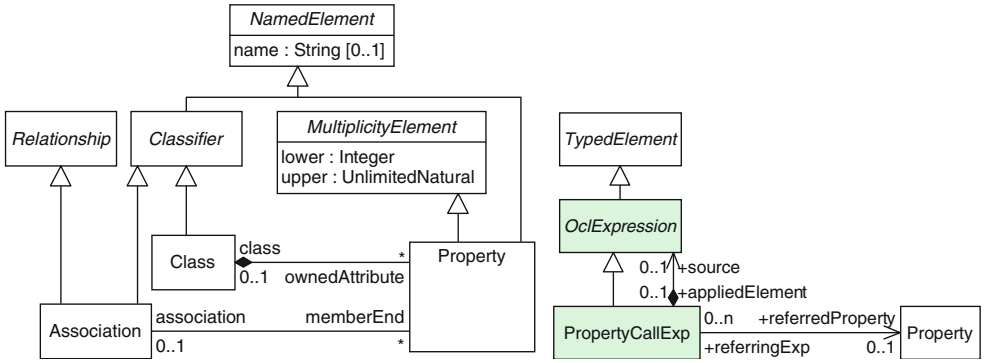

Fig. 28 Relevant part of UML2.0 and OCL2.0 for *MoveProperty*

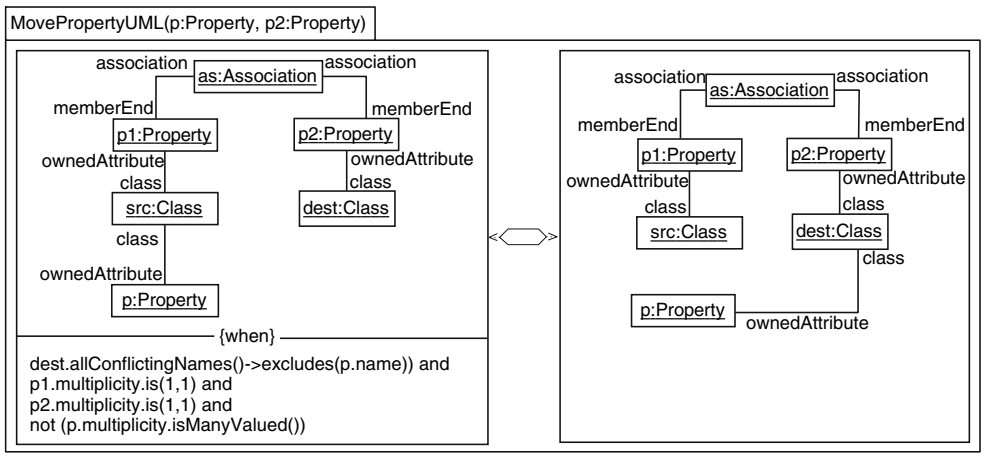

Fig. 29 UML part of *MoveProperty* rule

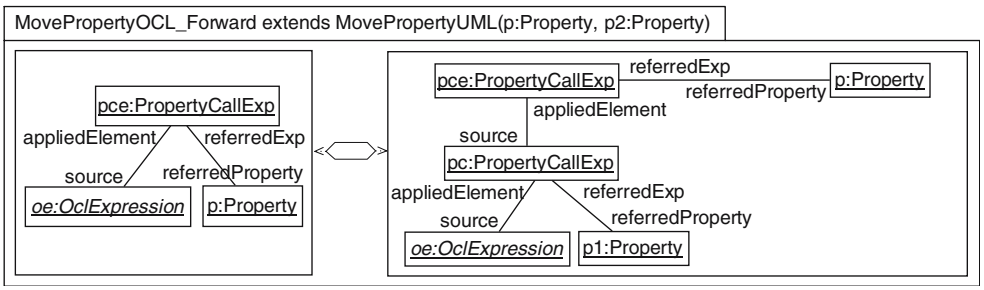

Fig. 30 OCL part 1 of *MoveProperty* rule (forward navigation)

rules we specified in this paper, however, this case does not occur. Please note that each refactoring rule is invoked separately by the user. This is an important difference to other rewrite systems where a model is transformed by a concurrent application of multiple transformation rules.

The major obstacle to implement our graphical rules directly in textual QVT is the lack of a pattern-matching mechanism in textual QVT, which would allow to find all substructures of a source model that match with LHS (step 1). The basic entity in QVT to describe a transformation is a *mapping* that works on a certain *domain*. A mapping can call sub-mappings or queries; the latter are implemented by a sequence of OCL expressions. Mappings are written in a dialect of OCL, called *Imperative OCL*. This dialect is no longer side-effect free and adds to standard OCL two facilities, assignment (`:=`) and object creation (`object ...`), for the manipulation of data structures.

The main application scenario for QVT is the description of transformations in which source and target model are instances of different metamodels. When working in this

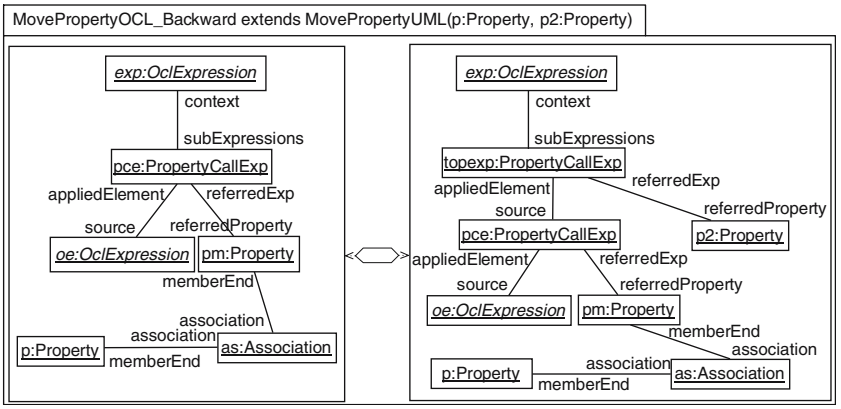

Fig. 31 OCL part 2 of *MoveProperty* rule (backward navigation)

mode, the QVT mapping traverses the source model, normally by calling sub-mappings, and creates successively the target model. In our refactoring scenario, however, we have the special case that the metamodels for source and target model coincide. Moreover, the source and target model themselves coincide except at some locations where substructures have been refactored. QVT supports this special scenario by *inout*-variables which represent both the source and the target of a mapping. Within the mapping, it is only possible to change those parts of the data structure to which the *inout*-variables refer. All other, untouched, parts of the data structure will then be copied automatically from the source to the target model.

The general approach to implement our refactoring rules is as follows. A mapping implements a traversal through the source model in order to find all substructures that match with LHS. Fortunately, due to the simple structure of used LHS patterns, this task is easily programmed manually and does not require to apply sophisticated search algorithms. Then, for each matching substructure, a sub-mapping is invoked that realizes the rewriting step accordingly to RHS.

Figure 32 shows the application of a QVT transformation on a concrete UML/OCL source model in our tool ROCLET,[5] into which the implementation of refactoring rules has been integrated.

4.2 Entry-point mapping

A model transformation is implemented in QVT usually by a set of (sub)mappings, but there is one top-level mapping that represents the whole transformation. In the QVT jargon, this top-level mapping is called *entry-point mapping*. One important restriction is that the entry-point mapping can have only one parameter, representing the model-element on which the transformation is applied. In our case, the chosen parameter is always the root element of the source model.

The fact that the entry-point mapping has just one parameter does not correspond to our graphical refactoring rules. The parameters in our graphical rules encode decisions taken by the user, e.g. for rule *MoveAttribute* the decisions, which attribute should be moved over which association end. If the entry-point mapping has only one parameter, the user decisions can obviously not be passed as arguments. A solution for this problem is to simulate the needed parameters by query calls. The entry-point mapping for rule *ExtractClass* looks as follows:

```
transformation extractClass;

-- import of private QVT library
import utils;

-- declaration of metamodel for source/target model
metamodel 'core';

mapping main(in model: core::Model): core::Model {
  init {
    -- simulation of parameter passing
    var src := getSrc(model);
    var newName := getNewName();
    var role1 := getRole1();
    var role2 := getRole2();

    -- call of sub-mapping with all required parameters
   var d := extractClass(model, src, newName, role1, role2);

   result:=model;
  }
}
```

4.3 Finding the matches for LHS

The first step that has to be realized by the implementation of a refactoring rule is finding the substructure of the source model that matches with the LHS of the rule. Since the class *src* is passed as an argument of the refactoring rule, finding an LHS-match boils down to simply check the when-clause.

```
query extractClass(inout root:core::Package,
                   in src:core::Class,
                   in newName:String,
                   in role1:String,
                   in role2:String):OclVoid{
    if findUMLMatch(src, newName, role1)
```

[5] ROCLET is available from `www.roclet.org`

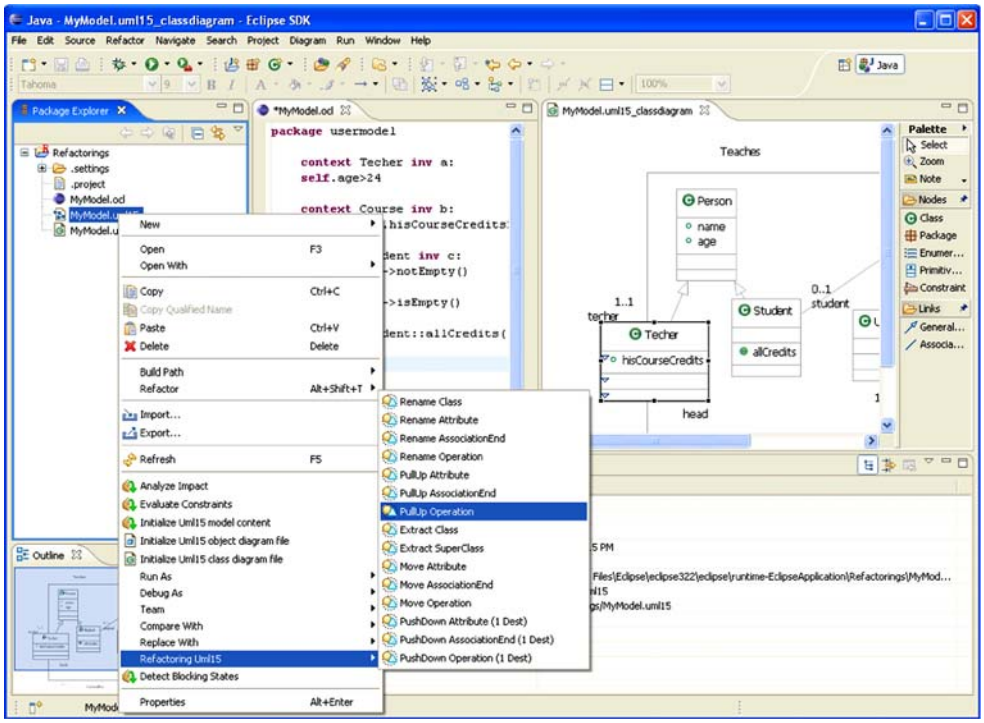

Fig. 32 Application of refactoring rule on a concrete UML/OCL model

```
    then applyRHSUML(src.namespace, src, newName, role1, role2)
    else true
    endif;

  undefined
}

query findUMLMatch(in src:core::Class,
                   in newName:String,
                   in role1:String):Boolean{
  if (whentest1(src.namespace, newName) and whentest2(src, role1))
  then true
  else false
  endif
}

query whentest1(in nsp:core::Namespace,
                in newName:String):Boolean{
    if (nsp.oclIsKindOf(core::Classifier))
    then nsp.oclAsType(core::Classifier)
        .allConflictingNames()->excludes(newName)
    else nsp.ownedElement.name->excludes(newName)
    endif
}
query whentest2(in src:core::Class,
                in role1:String):Boolean{
    src.allConflictingNames()->excludes(role1)
}
```

4.4 Applying RHS

Once a matching substructure is identified, this substructure is passed to `applyRHSUML`, which implements a rewriting of the substructure according to the RHS of the transformation rule. The rewrite step uses extensively the new facilities integrated into Imperative OCL in order to manipulate data structures.

```
mapping applyRHSUML(inout nsp:core::Namespace,
                    in src:core::Class,
                    in newName:String,
                    in role1:String,
                    in role2:String):core::Class{
  init{
  var extracted := object core::Class {
                name := newName
              };
  nsp.ownedElement += extracted;
  var as:core::Association := object core::Association{
                    namespace:=nsp
                  };
  var ae1:core::AssociationEnd :=
     object core::AssociationEnd{
       association := as;
       name := role1;
       participant := extracted;
       multiplicity :=
         object core::Multiplicity{
           range += object core::MultiplicityRange{
                    lower := 1;
                    upper := 1}}
     };
  var ae2:core::AssociationEnd :=
     object core::AssociationEnd{
       association := as;
       name := role2;
       participant := src;
```

```
        multiplicity :=
          object core::Multiplicity{
            range += object core::MultiplicityRange{
                    lower := 1;
                    upper := 1}}
      };

    result:=undefined;
    }
}
```

The most important difference to normal OCL is the usage of keyword `object` in order to express the creation of an object. The first statement, for example, expresses that the local variable `extracted` is assigned to a newly created object of type `Class` whose attribute `name` has the same value as parameter `newName`.

4.5 Summary

The encoding of the graphical refactoring rules as given in Sect. 3 into textual QVT is straightforward. We have used for all refactoring rules the same structure as for rule *ExtractClassUML*. The main difference between graphical and implemented version is that the search for an LHS-match had to be realized by a concrete algorithm. This algorithm, however, is trivial for refactoring rules because the elements from the source model that are affected by the refactoring rule are always passed as parameters. This trait of refactoring rules minimizes the effort to search for the right location in the source model that matches with the LHS of the rule.

Encoding the RHS in textual QVT is straightforward as well; one has just to change the relevant properties of the elements identified by RHS. Please note that the implementation of RHS has only an influence on the current location and does not change anything else in the rest of the model.

5 Using KeY to prove formally the preservation of well-formedness rules

In the conference version of this paper [18], we published a version of rule *ExtractClassUML* whose application could break the following well-formedness rule in the UML metamodel, which ensures that classes residing in the same namespace (usually this is the contextual package) have to have different names:

context c1,c2:Class **inv**:
(c1.namespace=c2.namespace *and* c1.name=c2.name) *implies* c1=c2

Our mistake in [18] was to have forgotten to set up the namespace for the newly created class *extracted* and to ensure in the when-clause, that the name of class *extracted* has not been already taken by another class residing in the same namespace.

It is not difficult to argue informally that the version of *ExtractClassUML* presented in this paper (Fig. 16) preserves the well-formedness rule on the uniqueness of class names: Whenever *ExtractClassUML* is applicable, the when-clause ensures that in the namespace of class *src* there is no class with name *newName*. Applying *ExtractClassUML* means to create exactly one new class *extracted*. Let us assume that the well-formedness rule is broken for the target model. Since it was satisfied in the source model and all classes have kept their names, the only possibility is that there is in the namespace of *extracted* a second class with the same name as *extracted*. This, however, is prevented by the when-clause of *ExtractClassUML* and by the fact that *src* and *extracted* have the same namespace.

Such an argumentation would probably convince most people but, as any informal argumentation, it is prone to errors since it takes the semantics of the implementation language, here Imperative OCL, only informally into account. The argumentation would be more reliable if it would base on a formal semantics of the implementation language and would take literally the implementation of the transformation into account.

To our knowledge, there is no tool available yet, that is based on the formal semantics of Imperative OCL (which is a rather recent dialect of OCL). There are, however, verification tools for other programming languages such as Java, C++, etc. available, that could be used to verify syntactic preservation of transformation rules, if these rules would be implemented in Java, C++, etc. instead of Imperative OCL.

We show in the rest of the section, how the KeY-system,[6] a verification system for Java, could be used to prove syntactic preservation for a Java-implementation of *ExtractClassUML*, which looks literally the same as the implementation in Imperative OCL discussed in Sect. 4. Note that this is still not a formal proof yet for the version of *ExtractClassUML* implemented in Imperative OCL since the KeY-system can currently verify only Java implementations. There are, however, no fundamental obstacles to adapt the KeY-system, so that it can handle in addition to implementations written in Java also those written in Imperative OCL.

The KeY-system, see [1] for an overview and [7] for a complete introduction, allows software developers to prove formally that the implementation of a Java method satisfies a method contract (pre-/postcondition together with invariants written in OCL). In particular, one can formally show that whenever a method is invoked in a system state, in which all invariants and the method's precondition hold, the execution of the method will terminate in a state, in which the invariants

[6] The KeY-system is published under the GNU Public License (GPL) and can be downloaded from `www.key-project.org`. KeY is available both as a stand-alone tool and as a *TogetherCC* plugin.

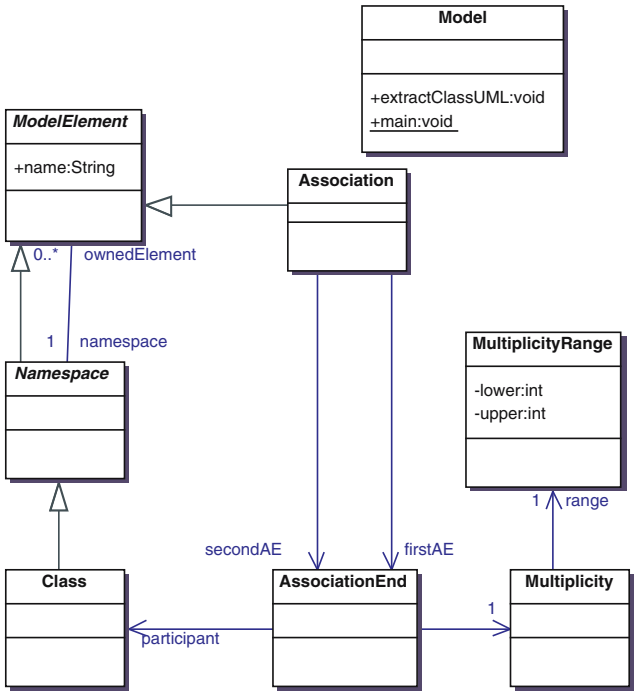

Fig. 33 Simulation of the UML/OCL metamodel by Java classes

hold as well (this functionality is available as *PreserveInvariant* in the KeY menu).

In order to apply the KeY-system to prove the preservation of the well-formedness rule on unique class names, the metamodel and the effect of applying *ExtractClassUML* had to be encoded in Java. Figure 33 shows how the relevant part of the metamodel is encoded; the metaclasses *Class*, *Association*, etc. became ordinary Java classes.

The Java class `Model` has been added just to serve as a container for the refactoring rules. The effect of the rule *ExtractClassUML* would be implemented in Java as follows:

```
/**
 * @ preconditions Class.allInstances->forAll(c|
 * c.namespace=src.namespace implies c.name<>newName)
 */
public void extractClass(Class src, String newName) {

   if (src!=null)  {

   Class extracted=new Class(newName);
   extracted.namespace=src.namespace;
   Association as=new Association();
   as.firstAE= new AssociationEnd(extracted,
     new Multiplicity(new MultiplicityRange(1,1)));
   as.secondAE=new AssociationEnd(src,
   new Multiplicity(new MultiplicityRange(1,1)));
   as.namespace=src.namespace;
   }
}
```

The listing shows the source code as it is managed by *TogetherCC*. The JavaDoc comment *@preconditions* contains the condition on the pre-state in OCL syntax (comparable with the when-clause). The body of `extractClassUML` resembles the implementation of `applyRHSUML` written in Imperative OCL. The only exception is that the Java version encodes the additional pre-condition `src!=null` as an if-statement. For the version implemented in Imperative OCL, the same condition is automatically stipulated by the semantics of OCL.

The invariant to be proven is exactly the same as in the real metamodel and attached in OCL syntax to class `Model`.

```
/**
 * @invariants Class.allInstances->forAll(c1,c2|
 *    c1.namespace=c2.namespace and
 *    c1.name=c2.name
 *    implies c1=c2)
 */
public class Model {  ...  }
```

The KeY-system is able to prove fully automatically for the shown implementation of *ExtractClass* that the invariant is preserved.

6 Conclusions

The refactorings considered in this paper realize a special form of model synchronization: a change in a UML class diagram should trigger an automatic update of attached OCL constraints. A minimal requirement for a refactoring rule is that a refactoring step does not destroy the syntactic correctness of the manipulated UML/OCL model. Ideally, the semantics of the model remains unchanged as well.

In this paper, five groups of refactoring rules (*Rename*, *PullUp*, *PushDown*, *Extract*, *Move*) are investigated and classified with respect to their influence on attached OCL constraints. Only *Move*-refactorings require to update attached OCL constraints but the applicability of *PullUp*- and *PushDown*-refactorings depends on the absence of certain OCL constraints. The rules targeting attributes are generally less complex than the rules for operations and association ends.

We formalized all refactoring rules in form of QVT model transformations that are based on graph-transformations. This formalization allows a precise argumentation that the refactoring rules preserve the syntax and the semantics of the models they are applied on. Syntax preservation is formally shown for one example in Sect. 5. Semantics preservation was not in the scope of this paper but has been discussed in [4].

The understandability of graphical QVT rules depends, most likely, on the degree of familiarity with the underlying metamodel and on personal preferences. The experiences we gained when writing up and discussing multiple versions of the refactoring rules presented in Sect. 3 let us conclude

that graph-transformation systems are a excellent choice for the formalization of refactoring rules. The readability of our rules can, however, be even improved for persons who are only vaguely familiar with the metamodel for UML/OCL. As shown in [5], this can be achieved by defining a concrete syntax for refactoring rules. This concrete syntax can hide many internals of the UML/OCL metamodel from the reader, but keeps the expressive power of ordinary QVT rules.

All rules presented in this paper have been fully implemented in the tool RoclET, a versatile tool for the development and analysis of OCL specifications. RoclET's refactoring functionality takes from the user the burden to correct manually all OCL constraints that became syntactically incorrect when the underlying UML diagram has been refactored. We see this as a necessary pre-condition to *pull up* agile techniques, which became quite popular over the recent years on the implementation level, also to the modeling level.

Acknowledgments This work was supported by Swiss National Scientific Research Fund under reference number 2000-067917.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Softw. Syst. Model. **4**(1), 32–54 (2005)
2. Astels, D.: Refactoring with UML. In: Proceedings of 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering, pp. 67–70 (2002)
3. Baar, T.: The definition of transitive closure with OCL— limitations and applications. In: Broy, M., Zamulin, A.V. (ed.) Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, 9-12 July 2003, Revised Papers, volume 2890 of LNCS, pp. 358–365. Springer, Heidelberg (2003)
4. Baar, T., Marković, S.: A graphical approach to prove the semantic preservation of UML/OCL refactoring rules. In: Virbitskaite, I., Voronkov, A. (eds.) Proceedings, 6th International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI 2006), Akademgorodok near Novosibirsk, Russia, vol 4378 of LNCS, pp. 70–83. Springer, Heidelberg (2007)
5. Baar, T., Whittle, J.: On the usage of concrete syntax in model transformation rules. In: Virbitskaite, I., Voronkov, A. (eds.) Proceedings, 6th International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI 2006), Akademgorodok near Novosibirsk, Russia, vol. 4378 of LNCS, pp. 84–97. Springer, Heidelberg (2007)
6. Beck, K.: Extreme programming explained: embrace change. Addison-Wesley, Reading (2000)
7. Beckert, B., Hähnle, R., Schmitt, P.H.: (eds.) Verification of Object-Oriented Software: The KeY Approach. LNAI 4334. Springer, Heidelberg (2007)
8. Boger, M., Sturm, T., Fragemann, P.: Refactoring browser for UML. In: Proceedings of 3rd International Conference on extreme Programming and Flexible Processes in Software Engineering, pp. 77–81 (2002)
9. Borland: Together technologies. http://www.borland.com/together/ (2007)
10. Cabot, J., Teniente, E.: Computing the relevant instances that may violate an OCL constraint. In: Pastor, O., Falcão e Cunha, J. (eds.) 17th International Conference on Advanced Information Systems Engineering, CAiSE 2005, Porto, vol. 3520 of LNCS, pp. 48–62. Springer, Heidelberg (2005)
11. Cabot, J., Teniente, E.: Incremental evaluation of OCL constraints. In: Dubois, E., Pohl, K. (eds.) Advanced Information Systems Engineering, 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg, Proceedings, vol. 4001 of LNCS, 5–9 June 2006, pp. 81–95. Springer, Heidelberg (2006)
12. Correa, A., Werner, C.: Applying refactoring techniques to UML/OCL. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004—the Unified Modeling Language. Model Languages and Applications, Lisbon, Portugal, vol. 3273 of LNCS, pp. 173–187. Springer, Heidelberg (2004)
13. Fowler, M.: Refactoring: Improving the Design of Existing Programs. Addison-Wesley, Reading (1999)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading (1995)
15. Van Gorp, P., Stenten, H., Mens, T., Demeyer, S.: Towards automating source-consistent UML refactorings. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003—The Unified Modeling Language, Modeling Languages and Applications, San Francisco, CA, USA, vol. 2863 of LNCS, pp. 144–158. Springer, Heidelberg (2003)
16. Kerievsky, J.: Refactoring to Patterns. Addison-Wesley, Reading (2004)
17. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley, Reading (2004)
18. Marković, S., Baar, T.: Refactoring OCL annotated UML class diagrams. In: Briand, L., Williams, C. (eds.) Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, 2–7 October 2005, Proceedings, vol. 3713 of LNCS, pp. 280–294. Springer, Heidelberg (2005)
19. Marković, S., Baar, T.: Documentation of UML/OCL refactoring rules. http://www.roclet.org/publications/SoSymSpecialIssue-Models05/, 2007
20. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2), 126–139 (2004)
21. O'Cinneide, M.: Automated application of design patterns: a refactoring approach. Ph.D. Thesis, University of Dublin, Trinity College (2001)
22. OMG: UML 1.5 Specification. OMG Document formal/03-03-01 (2003)
23. OMG: UML 2.0 Infrastructure Specification. OMG Document ptc/03-09-15 (2003)
24. OMG: UML 2.0 OCL specification—OMG final adopted specification. OMG Document ptc/03-10-14 (2003)
25. OMG: Revised submission for MOF 2.0, query/views/transformations, version 1.8. OMG document ad/04-10-11 (2004)
26. OMG: Meta object facility (MOF) 2.0 query/view/transformation specification. OMG document ptc/05-11-01, Nov 2005
27. OMG: Object constraint language—OMG available specification, version 2.0. OMG document formal/06-05-01 (2006)
28. Opdyke, W.F.: Refactoring: A program restructuring aid in designing object-oriented application frameworks. Ph.D. Thesis, University of Illinois at Urbana-Champaign (1992)
29. Porres, I.: Model refactorings as rule-based update transformations. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003—The Unified Modeling Language, Modeling Languages and Applications, San Francisco, CA, USA, vol. 2863 of LNCS, pp. 159–174. Springer, Heidelberg (2003)
30. Refactoring community: refactoring homepage. http://www.refactoring.com (2007)

31. Rumpe, B.: Agile Modellierung mit UML. Springer, Heidelberg (2005) In German
32. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw **20**(5), 42–45 (2003)
33. Sunyé, G., Pennaneac'h, F., Ho, W.-M., Guennec, A.L., Jézéquel, J.-M.: Using UML action semantics for executable modeling and beyond. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) Advanced Information Systems Engineering, 13th International Conference, CAiSE 2001, Interlaken, Switzerland, Proceedings, vol. 2068 of LNCS, 4–8 June 2001, pp. 433–447. Springer, Heidelberg (2001)

Author Biographies

Slaviša Marković graduated from the University of Belgrade and is currently a PhD student and research assistant at the Software Engineering Laboratory (LGL), Swiss Federal Institute of Technology in Lausanne (EPFL). His research interests include model transformations, model refactorings, and semantics of constraint languages.

Thomas Baar is Senior Researcher and Lecturer for software engineering at the École Polytechnique Fédérale de Lausanne (EPFL). His research interests include quality-oriented software processes, (semi-) formal specification techniques, and automatic verification of system implementations. Dr. Baar holds a diploma degree in computer science from Humboldt-University Berlin and a doctoral degree from University Karlsruhe. He is a member of the ACM.