Refactoring OCL Annotated UML Class Diagrams*

Slaviša Marković and Thomas Baar

École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
{slavisa.markovic, thomas.baar}@epfl.ch

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 annotated OCL constraints. For refactoring rules, whose application on class diagrams could make attached OCL constraints incorrect, we formally describe how the OCL constraints have to be refactored to preserve their syntactical correctness. Our refactoring rules are defined in the graph-grammar based formalism proposed by the QVT Merge Group for the specification of model transformations.

1 Introduction

Modern software development processes, such as Rational Unified Process (RUP) [1] and eXtreme Programming (XP)[2] propagate the application of refactoring to support iterative software development. Refactoring (see [3] for an overview) is a structured technique to improve the quality of artifacts.

Artifacts produced in all phases of the software development lifecycle could become a subject of refactoring. However, existing techniques and tools mainly target the implementation code. Due to the increase in popularity of XP, the tool support for refactoring has been improved considerably over the last years. An up-to-date list of existing tools can be found at [4].

As the first author, Opdyke has tackled refactoring of implementation code in [5]. He defines refactorings as "*... reorganization plans that support change at an intermediate level*" and identifies 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).

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

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 call became syntactically incorrect in step 2 and must be rewritten.

There are several catalogs of refactoring rules for different languages. The most complete and influential was published by Fowler in [6] for refactoring of Java code. The refactoring of artifacts more abstract than implementation code has become only recently a research topic. Some initial catalogs of refactoring rules for UML diagrams, mostly adaptations from the Java refactorings given by Fowler, are presented in [7–9]. Only few tools are currently available to support UML refactorings [10, 11]. None of these catalogs or tools takes OCL constraints into account, which might be attached to diagrams. Thus, applying these refactoring rules on diagrams that have constraints attached can make them syntactically incorrect. As spoken in terms of the above shown *MoveAttribute* example, the first two steps have been realized but the last step is ignored. The only refactoring approach of OCL we are aware of is by Correa and Werner [12], but here the focus is on improving badly structured OCL constraints and only to a very limited extent the relationship between OCL constraints and the underlying class diagram.

In this paper, the most important refactoring rules for class diagrams including attached OCL constraints are described formally. Not all class diagram refactoring rules have an impact on attached OCL constraints, so we first answer the question which of the rules for class diagrams can destroy the syntactical correctness of attached OCL constraints. If a rule has no impact on OCL we informally give the reasons for this. For the rules, whose application can influence the syntactical correctness of OCL constraints, we formalize the necessary changes on the OCL code. Up to now, we are not able to argue for all rules that they preserve the semantics of the refactored OCL constraint. This important topic will be addressed in our future research.

The formal description of refactoring rules is done on the level of the metamodel for UML and OCL. Unlike other approaches to describe refactoring rules formally [12, 7, 13] we do not use OCL pre/post-conditions for this purpose. The formalism of our choice is a slight adaptation of the QVT Merge Group [14] proposal to describe model transformations (note that refactoring can be seen as a special case of model transformation) that is based on graph grammars. Hence, our catalog of refactoring rules can also be seen as a case study for QVT.

In Sect. 2 we give preliminaries to understand our rules formally defined in Sect. 3. The insights gained during the formalization of the refactoring rules are summarized in Sect. 4 whereas Sect. 5 concludes the paper and gives an outlook of future research activities.

2 Description of Refactoring Rules with QVT

Model transformations are widely recognized now as the 'heart and soul' of model driven development [15]. The Object Management Group (OMG) is currently in the process to standardize the notation for the formal description of model transformations and has launched a corresponding *Query/Views/Transformations* Request for Proposals in 2002. In this paper, we mainly use the notation suggested by the QVT Merge Group in the subsequent proposal [14]. Since our aim is to refactor UML class diagrams annotated with OCL, our refactoring rules are based on the metamodels of UML and OCL. The following subsection recalls those parts of these metamodels that are used in our refactoring rules. Afterwards, a brief introduction to the QVT notation for model transformations is given.

2.1 Metamodels of UML/OCL

Metamodeling is a powerful technique to describe the abstract syntax of languages in a concise way. A metamodel for a language can roughly be seen as a class model whose classes and associations encode the concepts of the language and the relationships between them. Each syntactically correct sentence of the language can be represented in form of an instance of the metamodel.[1]

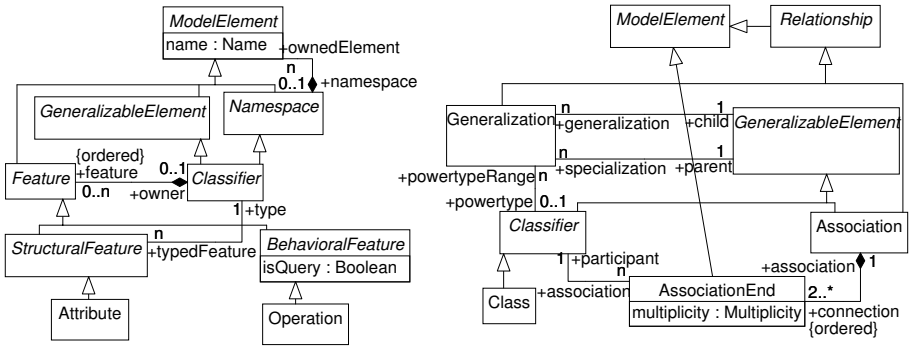

Fig. 1. Relevant parts of UML metamodel - Backbone and Relationships

Fig. 1 and Fig. 2 show relevant parts of the official metamodels for UML1.5 and OCL2.0 (for a complete definition see [16, 17]).[2] In addition to what is shown in Fig. 1 and Fig. 2, some of the refactoring rules refer to additional operations such as *Classifier.allParents:Set(Classifier)*. The definition for these operations are omitted here for the sake of brevity but can be found in the official metamodels [16, 17].

[1] In the remaining paper, such representations are called *MM-representations*.

[2] We have chosen UML1.5 as a basis, because in time of writing this paper, the OCL2.0 metamodel was not aligned yet to UML2.0 and still relied on UML1.5.

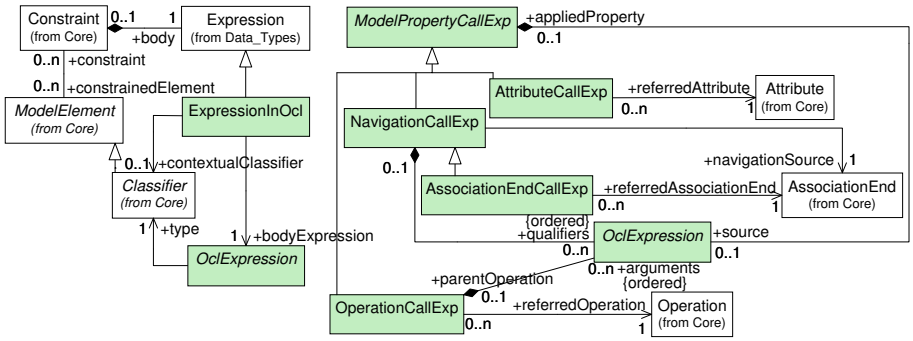

Fig. 2. Relevant part of OCL metamodel - Overview and ModelPropertyCallExp

2.2 Introduction to QVT

The QVT Merge Group proposal [14] aims at providing a standardized graphical notation to define model transformations.[3] A model transformation is defined as a set of *transformation rules* that, when applied on a source model, transform this into a target model. Source and target models are assumed to be represented as instances of metamodels. In general, QVT can handle the case of different metamodels for the source and target models but refactoring rules need only one metamodel for both source and target model.

A transformation rule in graphical notation consists of two patterns LHS (left hand side), RHS (right hand side) that are connected by a symbol indicating the transformation's type such as general transformation (<···○···>), relation (<⬭>), or mapping (<▭>). Optionally, a rule can have parameters and a when-clause comprising textual constraints.
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 – during the application of the rule – at all occurrences to the same value. Furthermore, links and objects in the pattern can be marked as non-existing (by a cross) what is read when applying the rule as a *negative matching condition*. In order to distinguish between objects/links occurring in the patterns and objects/links occurring in the concrete models we call the former ones as *pattern objects/links* and the later ones as *concrete objects/links*.

If a rule is applied on a source model (represented as an instance of the metamodel, i.e. as a graph), then each subgraph that matches with LHS is rewritten by a new subgraph derived from RHS under the same matching. 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-

[3] The proposal defines also a purely textual notation that results, however, into less understandable transformation descriptions.

clause. This semantics of the 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 of the same class and identified by the same variable as in RHS) 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). An attribute value is changed to the value derived from its specification in RHS under the current matching. Values of the attributes that are not mentioned in LHS and RHS remain unchanged. We have now explained the basic principle of rule applications and the fundamental constructs used in patterns. More complicated constructs will be explained later at the places they are needed.

As an example, suppose we want to describe the renaming of some model elements (such as attributes, operations, or classes) in UML models. As a first step, the model element, whose name should be changed, has to be selected. Then, its name can be changed to the new name if it is not already used by another model element of the same type in the same namespace.

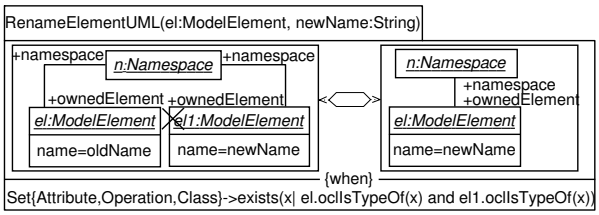

Fig. 3. Formalization of *RenameElement* refactoring

In the left pattern in Fig. 3, the model element *el* is selected by a parameter. If there is no other model element with a name equal to *newName* in the same namespace (indicated by the cross on *el1*), then the RHS pattern describes the change of the name of *el* to *newName*. Furthermore, the model elements *el* and *el1* must be both either attributes, operations, or classes. This is formalized by the when-clause.

3 A Catalog of UML/OCL Refactoring Rules

The rules presented below for refactoring of UML class diagrams and OCL are heavily inspired by refactoring rules for the static structure of Java programs given by Fowler in [6]. We took the freedom to change some rule names introduced by Fowler to indicate UML as their new application domain (e.g. *MoveMethod* became *MoveOperation*). Table 1 gives the list of the formalized

rules. If the rule name has changed compared to the name used by Fowler, the original name is given in parentheses. In few cases, not only the name but also the semantics of the rule has slightly changed. Details on this are given at appropriate places in the text. Furthermore, Table 1 shows which of the rules have an influence on OCL. Note that two rules have an influence only either on the MM-representation or the textual notation of the OCL constraints.

Table 1. Overview of UML/OCL refactoring rules

Refactoring rules	**Influence on syntactical correctness of OCL constraints**	
	MM-Representation	Textual Notation
ExtractClass	No	No
ExtractSuperclass	No	No
RenameElement (*RenameMethod*)	No	Yes
MoveAttribute (*MoveField*)	Yes	Yes
MoveOperation (*MoveMethod*)	Yes	Yes
PullUpOperation (*PullUpMethod*)	No	No
PullUpAttribute (*PullUpField*)	No	No
PushDownOperation (*PushDownMethod*)	Yes	Yes
PushDownAttribute (*PushDownField*)	Yes	No

3.1 Rules Without Influence on OCL

RenameElement The rule *RenameElement* has been already used as an example in Sect. 2. Our version allows changing the name of many model elements (attributes, operations, and classes) whereas Fowler allows in [6] only renaming of methods.[4] This motivates the change of the rule name from *RenameMethod* to *RenameElement*.

At a first glance, renaming of an attribute requires to change all annotated 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 OCL metamodel, then this instance remains the same. Note that the OCL metamodel *refers* only to the UML metamodel but does not comprise it. Thus, the change made in the underlying UML model is automatically propagated to all OCL expressions that use the changed UML element.

PullUpAttribute/PullUpOperation These two rules remove one attribute/ operation from a class and insert it into one of its superclasses, Fig. 4 shows a concrete example. We will concentrate our description on *PullUpAttribute*, the rule *PullUpOperation* is handled analogously.

[4] However, there is no principal obstacle for renaming other declarations in Java. The Eclipse tool [18], for example, provides capability for renaming other model elements, e.g. attributes.

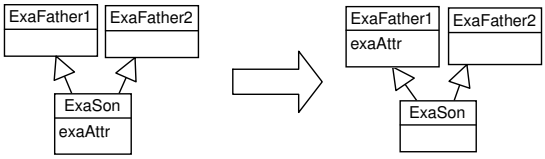

Fig. 4. Example for applying *PullUpAttribute*

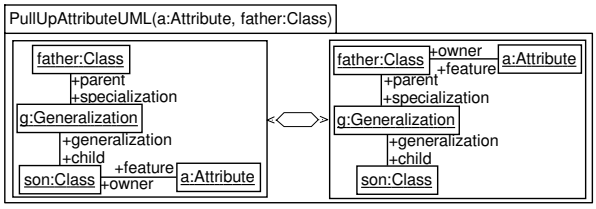

Fig. 5. *PullUpAttribute* refactoring rule

In Fig. 5, the pre-conditions to apply this rule are given: Attribute *a* is owned by class *son* that must have a parent class *father*. The RHS pattern formalizes that the owner of attribute *a* has changed from class *son* to class *father* (link from *a* to *son* is deleted and to *father* created). Unlike the *PullUp* rules for Java, it is not necessary to state as a condition on the LHS, that in the pre-state the class *father* must not have an attribute with the same name as *a*. This is automatically imposed by a well-formedness rule in UML1.5 preventing a class to use names for its attributes which were already taken by one of its ancestor classes (cmp. Sect. 2.5.4.4 in [17]). If the class *father* had an attribute with the same name as attribute *a* then this well-formedness rule would be broken for class *son*. Java is not so strict in this respect; e.g. names for private attributes can be reused in subclasses without problems.

The *PullUpAttribute* rule has no influence on OCL constraints because it widens the applicability of the moved attribute. The attribute `exaAttr` can only occur in attribute call expressions (*AttributeCallExp*) of form *exp*.`exaAttr`. Here, the type of expression *exp* must be compatible with the owner of the attribute *son*. After the refactoring, *exp*.`exaAttr` is still syntactically correct because the type of *exp* is also a subtype of *father* what is the new owner of the attribute.

ExtractClass/ExtractSuperclass The rule *ExtractClass* creates an empty class and connects it with a new association to the source class from where it is extracted. The multiplicity of the new association is 1 on both sides. The *ExtractSuperclass* rule creates an empty class as well but inserts it between the source class and one of its direct parent classes. Note that *ExtractClass/ExtractSuperclass* differ from the corresponding rules given by Fowler in [6]. 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/Operation* or *PullUpAttribute/Operation*.

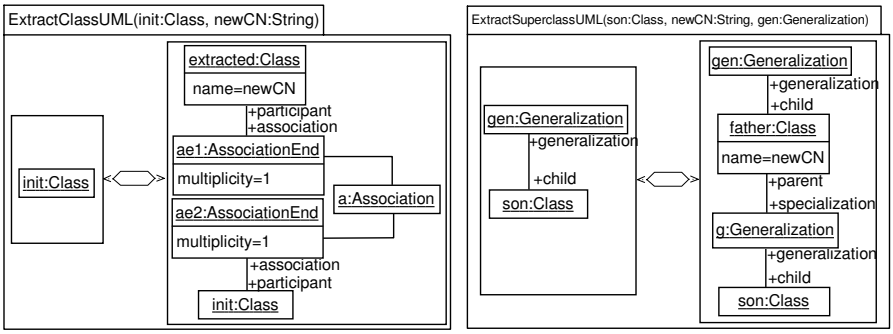

Fig. 6. *ExtractClass/ExtractSuperclass* refactoring rules

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

3.2 Rules With Influence on OCL

PushDownAttribute This rule is the counterpart of the rule *PullUpAttribute* from Fig. 5 and moves an attribute from the parent to some selected subclasses (see Fig. 7). As described by Fowler in [6] for the corresponding rule *PushDownField*, the attribute is moved only to such classes, where it is actually used.

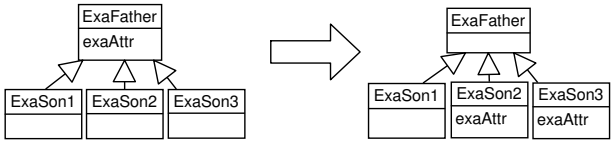

Fig. 7. Example for applying *PushDownAttribute*

The formalization of the *PushDownAttribute* rule is split into a UML and an OCL part shown in Fig. 8 and Fig. 9. It uses some elements of QVT that have not been explained yet as well as some 'private' elements that are missing in QVT.

Multiobjects as *gs* and *users* are already defined in QVT and represent a set of objects of the same type (here *Generalization* and *Class*). A multiobject that is linked to an ordinary pattern object – in our example, *gs* is linked to *father* – encodes the situation where all elements represented by the multiobject have actually a link to the object represented by the ordinary pattern object. Note

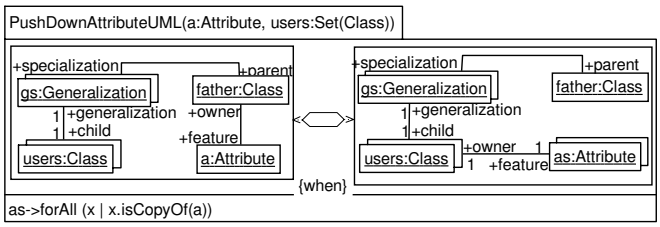

Fig. 8. UML part of *PushDownAttribute* rule

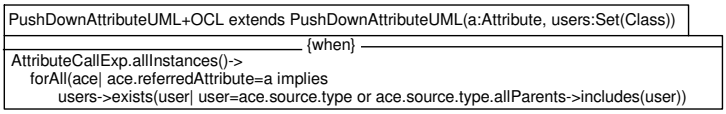

Fig. 9. OCL part of *PushDownAttribute* rule

that in Fig. 8 the variable *a* is passed as an parameter and thus *father* is implicitly determined as the owner of attribute *a*. The multiobject *gs* is determined as the set of generalizations which have *father* as the parent and which are linked with the elements represented by the multiobject *users* as their child. Note that the variable *users* is also passed as a variable to the rule in order to select the subclasses where the attribute *a* is moved to.

The multiplicity 1/1 at the link between *gs* and *users* is a 'private' pattern element and not included in the QVT Merge Group proposal yet. It was added here to enrich QVT's standard semantics of links between two multiobjects. The QVT semantics always assumes that such a link represents the situation where each element of the first multiobject is linked to every element of the second one, and vice versa. This standard semantics is not appropriate to describe the relationship between *gs* and *users* since each element of *gs* should be linked to exactly one element of *users*, and vice versa. Thus we propose to add multiplicities to pattern links between multiobjects what allows to indicate a non-standard semantics of such links in an intuitive way.

Another new element is the usage of operation *isCopyOf()* in the when-clause. Since the multiobject *as* occurs only in the RHS, we already know that all its elements are newly created. The multiplicity 1/1 between *as* and *users* let us further conclude, that for each element of *users* exactly one element of *as* is created. The when-clause and the intended semantics of *isCopyOf()* should ensure that each element of *as* is a shallow copy of attribute *a*. However, the elements of *as* have a different owner than *a* as indicated in RHS.

If *PushDownAttribute* is applied on a class diagram that has attached OCL constraints then we must ensure that in all constraints the attribute is never used in the superclass (*father*) nor in any class which is not compatible with at least one of the selected subclasses (*users*). This has been formalized by the rule shown in Fig. 9 that extends the rule of Fig. 8.

The rule *PushDownAttribute* does not cause changes of the OCL textual notation because instead of calling the attribute that is removed from the superclass, all calls now refer to a copy of this attribute at some of the subclasses. However, this refactoring causes changes on the MM-representation of OCL because every instance of *AttributeCallExp* that was calling the moved attribute has after the refactoring a new link to a newly created copy of the attribute in the subclasses.

MoveAttribute Applying the *MoveAttribute* rule helps to make a class smaller; an example of this refactoring is shown in Fig. 10.

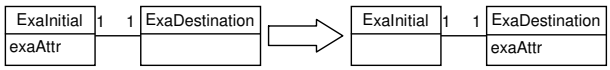

Fig. 10. Example for applying *MoveAttribute*

The attribute can only be moved to a class which is connected with the initial class by an association with multiplicity 1 at both ends. This allows objects of the initial class still to have access to the moved attribute after the refactoring. Not visible in the example but in the formalization in Fig. 11 is that neither the destination class nor one of its parents or children is allowed to have already an attribute with the same name as the moved attribute.

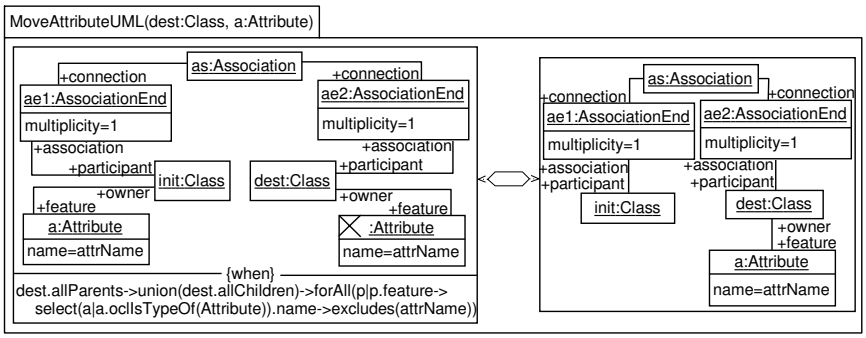

Fig. 11. UML part of *MoveAttribute* rule

Analogously to the changes of Java code described by Fowler for the corresponding refactoring *MoveField*, this rule must update OCL constraints on all locations where the moved attribute is applied. The necessary change of the OCL expressions can be seen as a kind of "Forward Navigation": Terms of form *exp*.`exaAttr` have to be rewritten as *exp*.`destination.exaAttr`. This change of OCL is formalized by the rule in Fig. 12.

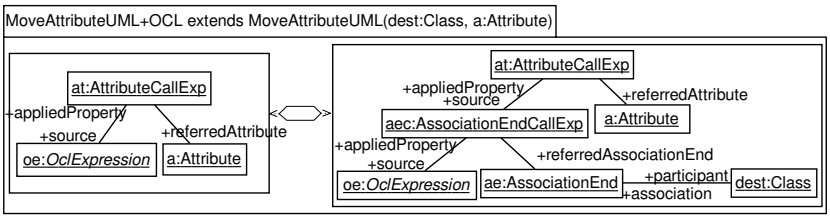

Fig. 12. OCL part of *MoveAttribute* rule

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

The formalization of *MoveOperation* refactoring is similar to that of *MoveAttribute* and shown in Fig. 13. As for *MoveAttribute*, the association must have on both ends multiplicity 1. The main difference is that the name of the moved operation is now allowed to be already used in the parent classes of the destination since UML1.5 allows operations to be refined along the generalization hierarchy.

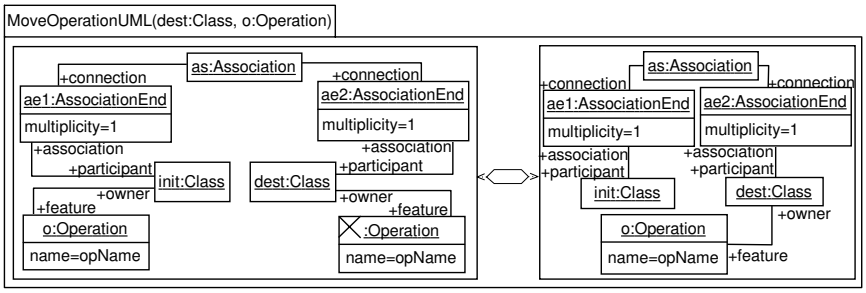

Fig. 13. UML part of *MoveOperation* rule

The changes induced on OCL can be described in three steps:

"Change context": If a constraint is attached to the operation (e.g. as pre/post-condition) then the context of this constraint has to be changed, in the above example from **context** `ExaInitial::exaOp()` to
context `ExaDestination::exaOp()`. Fowler describes in [6] informally this step as "Copy the code from the source method to the target". Note that in this step, we only copy the constraint body, the adaptations of the body will be done in the next steps.

"Backward navigation": After "Change context" the constraint attached to the moved operation still assumes variable `self` to be of type of the original class. At the new location, the variable `self` of the original class can be "simulated" by navigation from the destination class to the original class.

All occurrences of `self`.*propertyCallExp* in the moved constraints[4] have to be rewritten by `self.exaInitial`.*propertyCallExp*. This navigation is made possible by the multiplicity 1 on the end of the original class. For this step, Fowler says: "... create or use a reference from the target class to the source".

"Forward navigation": In case that the moved operation is a query we have to redirect in all operation call expressions the operation reference. This means to substitute all expressions *expression*.`exaOp()` by *expression*.`exaDestination.exaOp()`. This step corresponds to "Turn the source method into a delegating method" from Fowler's book.

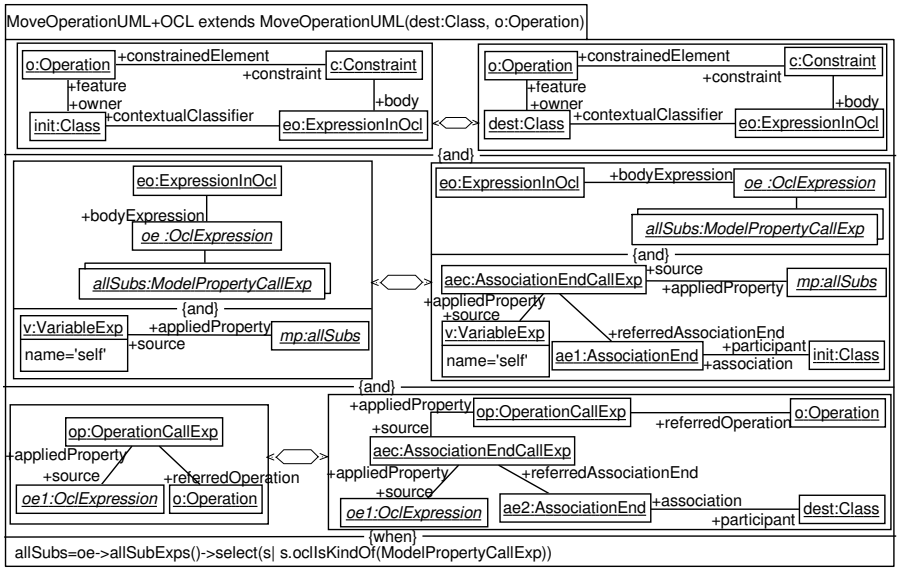

Fig. 14. OCL part of *MoveOperation* rule

As shown in Fig. 14, the formalization of *MoveOperation* refactoring is composed of three smaller transformations. The first sub-transformation is used to change the context of one OCL expression. LHS and RHS in the rule differ only in the class that represents the context for the attached OCL constraint.

In the second sub-transformation, the backward navigation is specified by adding a new instance of *AssociationEndCallExp* to the class from which the operation has moved. The when-clause uses a new operation *OclExpression.allSubExps:Set(OclExpression)* that is not part of the OCL metamodel yet. The intended semantics of *allSubExps* is to return all subexpressions of the OclExpression it is applied to.

The third sub-transformation describes "Forward navigation". The LHS pattern finds all occurrences where the moved operation is called. RHS specifies the insertion of an additional navigation to the destination class.

[4] Note that OCL allows in the textual notation to suppress `self`. Thus, `self` within `self`.*propertyCallExp* is sometimes given only implicitly.

PushDownOperation This rule is very similar to *PushDownAttribute* but, somehow surprisingly, it has influence on the OCL textual notation.

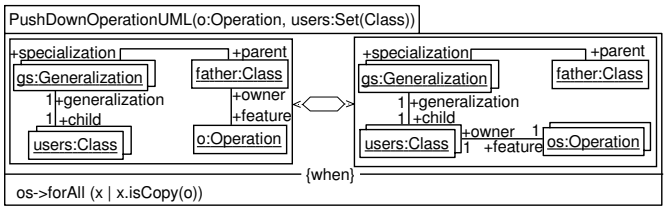

Fig. 15. UML part of *PushDownOperation* rule

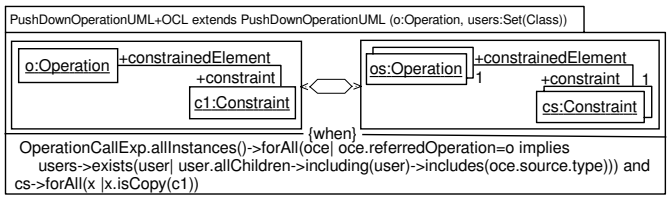

Fig. 16. OCL part of *PushDownOperation* rule

If the moved operation is a query and occurs in operation call expressions then the operation must be moved at least to all children that actually use the query.
No matter whether the moved operation is a query or not all its constraints have to be copied with an adapted context to the new operations in the selected subclasses (see Fig. 16).

4 Lessons Learned

The formalization of refactoring rules for UML/OCL has highlighted some advantages but also some missing elements of the graph-grammar based notation proposed by the QVT Merge Group in [14].

Since our refactoring rules are described in a graphical formalism they are much more accessible and understandable than existing formalizations of UML refactoring rules in form of pure OCL pre/post-conditions. Another advantage compared to purely OCL-based formalizations is the elegant solution of the *Frame problem* that is provided by the QVT semantics: only structures of the source model which match the LHS pattern of the rule are processed and substituted by the RHS under the same matching. The source model and the result of the refactoring can only differ in the elements that were made explicit in the RHS whereas an OCL formalization of the rules has to be read as "everything can change unless it is not explicitly stated that it remains the same".

Compared to corresponding refactoring rules for Java, the rules for UML and OCL are sometimes simpler to formulate because, for example, the visibility of model elements is ignored in the OCL syntax. Also the assumption in UML1.5 on the uniqueness of attributes names along the generalization hierarchy helps to keep the formulation of refactoring rules elegant. On the contrary, other concepts of UML such as multiple inheritance make the formulation of refactoring rules often more difficult.

Another interesting insight is that, not all class diagram refactoring rules can simply be classified in such a way that keep the OCL code untouched and in a way that can require a change in OCL. There is a group of rules in between which do not influence the OCL but whose applicability depends on some properties of the OCL constraints attached to the class diagram (e.g., in *PushDownAttribute* the LHS of the rule states that terms of a certain type do not appear).

Proposed change to QVT and OCL We have encountered some elements that are missing in the current QVT proposal and OCL metamodel:

- Sometimes, it is inevitable to express, that an object is the (shallow) copy of another object but an operation such as *OclAny.isCopyOf(OclAny):Boolean* is not available in OCL yet although its semantics is clear.
- The pattern language of QVT should allow to express a 1-1 relationship between objects of two multiobjects. As an intuitive way, we propose to add multiplicities to links connecting two multiobjects.
- The operation *OclExpression.allSubExps():Set(OclExpression)* is needed to access all subexpressions of an expression and, thus, should be added to the OCL metamodel as an additional operation.

5 Conclusions and Future Work

In the literature, refactoring rules for UML class diagrams have been described so far only informally or in form of pure OCL pre/post-conditions. In this paper, we formalized these rules in a precise and very readable way by using the formalism proposed by the QVT Merge Group. As the main contribution, the impact of changing class diagrams on annotated OCL constraints has been investigated. For the rules having an impact on OCL, the class diagram refactoring rules have been extended by additional transformation rules for OCL expressions. The extended rules now allow keeping class diagrams, which are often subject of change, easily in sync with annotated OCL constraints. Note that the OCL constraints play an important role in modern model-based software development paradigms.

So far, we are only able to argue that the presented refactoring rules preserve the syntactical correctness of OCL constraints. In a next step we will investigate whether or not the given refactoring rules are also behavior preserving (or rather semantics preserving). As another activity, we are currently developing a tool that is capable to perform the described UML refactorings and propagate these refactorings to annotations given in OCL.

References

1. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley (2004)
2. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)
3. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Software Eng. **30** (2004) 126–139
4. Refactoring community: Refactoring homepage. www.refactoring.com (2005)
5. Opdyke, W.F.: Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)
6. Fowler, M.: Refactoring: Improving the Design of Existing Programs. Addison-Wesley (1999)
7. Rumpe, B.: Agile Modellierung mit UML. Springer (2005) In German.
8. Astels, D.: Refactoring with UML. In: International Conference eXtreme Programming and Flexible Processes in Software Engineering. (2002) 67–70
9. 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.: CAiSE. Volume 2068 of LNCS., Springer (2001) 433–447
10. Boger, M., Sturm, T., Fragemann, P.: Refactoring browser for UML. In: International Conference eXtreme Programming and Flexible Processes in Software Engineering. (2002) 77–81
11. 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. Volume 2863 of LNCS., Springer (2003) 159–174
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. Volume 3273 of LNCS., Springer (2004) 173–187
13. Gorp, P.V., 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. Volume 2863 of LNCS., Springer (2003) 144–158
14. OMG: Revised submission for MOF 2.0, Query/Views/Transformations, version 1.8. OMG Document ad/04-10-11 (2004)
15. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Software **20** (2003) 42–45
16. OMG: UML 2.0 OCL Specification – OMG Final Adopted Specification. OMG Document ptc/03-10-14 (2003)
17. OMG: UML 1.5 Specification. OMG Document formal/03-03-01 (2003)
18. Eclipse community: Eclipse homepage. http: //www.eclipse.org (2005)