A Graphical Approach to Prove the Semantic Preservation of UML/OCL Refactoring Rules

Thomas Baar and Slaviša Marković

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

Abstract. Refactoring is a powerful technique to improve the quality of software models including implementation code. The software developer applies successively so-called refactoring rules on the current software model and transforms it into a new model. Ideally, the application of a refactoring rule preserves the semantics of the model, on which it is applied. In this paper, we present a simple criterion and a proof technique for the semantic preservation of refactoring rules that are defined for UML class diagrams and OCL constraints. Our approach is based on a novel formalization of the OCL semantics in form of graph transformation rules. We illustrate our approach using the refactoring rule `MoveAttribute`.

Keywords: Refactoring, Semantic Preservation, UML, OCL

1 Introduction

Modern software processes advocate the frequent application of so-called *refactoring rules* in order to improve the quality of software under development. A refactoring step is typically a small change made in a schematic way. Many approaches and tools have been developed for refactoring of implementation code but refactoring of more abstract software models, such as UML class diagrams (e.g. [1, 2]) became only recently a research topic. In our previous paper [3] we have formalized refactoring rules for UML class diagrams and OCL invariants (called *UML/OCL models* in the remainder of this paper) using a graph-transformation based formalism. In this paper, we present a technique to prove the correctness of our refactoring rules.

There are two important criteria for the correctness of refactoring rules. Firstly, a rule should be *syntactic preserving*, i.e., whenever the rule is applicable on a source model then the target model obtained by the application of the rule is syntactically correct, i.e., the target model is an instance of the UML/OCL metamodel and obeys all of the metamodel's multiplicity constraints and well-formedness rules. Secondly, a rule should be *semantic preserving*, i.e., the semantics of source and target model should coincide. The proof of both syntactic and semantic preservation can be challenging (see [4]). This paper concentrates on proving semantic preservation.

A proof for semantic preservation must rely on a formal semantics of source and target models and a criterion for their semantic equivalence. For UML/OCL models, a formal semantics based on set theory is given in [5] but this semantics is clumsy when arguing on the semantic preservation of a graphically defined refactoring rule. For this reason, we propose here a novel formalization of OCL's semantics in form of graph-transformation rules. Moreover, we give a simple criterion for the semantic equivalence of two UML/OCL models and show how this criterion is met by the refactoring rule `MoveAttribute`.

The rest of the paper is structured as follows. In Section 2, we give based on an example a brief introduction to graph transformations. Section 3 applies graph transformations for the formalization of the `MoveAttribute` refactoring and defines a criterion for semantic preservation. The section closes with two, more complicated versions of `MoveAttribute` whose formalization requires the usage of semantic preconditions. Section 4 presents a graphical definition of OCL's semantics and applies this semantics for proving the semantic preservation of `MoveAttribute`. Section 5 concludes the paper.

1.1 Related work

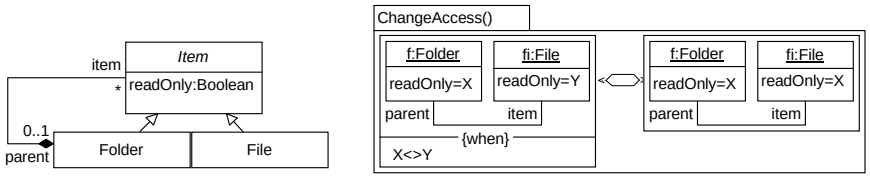
In his seminal work [6], Opdyke gives a catalog of refactoring rules for C++ programs. Opdykes defines semantic preservation (also called *behavioral preservation* if implementation code is refactored) as "...if the program is called twice (before and after a refactoring) with the same set of inputs, the resulting set of output values will be the same". In practice, it turned out that this simple criterion is hard to prove. Thus, more fine grained criteria such as *access preservation*, *update preservation*, and *call preservation* has been discussed in literature (an overview is given by Mens et al. in [7]).

2 Graph Transformation Rules

A graph transformation rule defines how source models are transformed into target models. A model is seen here as a typed graph, more precisely, as an instance of the modeling language's metamodel (see App. A for the relevant part of the UML/OCL metamodel). We assume the reader to be familiar with the technique of metamodeling (a good introduction is [8]).

A graph transformation rule consists of two patterns called *left hand side* (LHS) and *right hand side* (RHS), which are denoted in a generalized form of object diagrams over the metamodel for the transformed modeling language. A graph transformation rule is applied on a given source model by (1) searching a LHS-matching region and (2) substituting the matched region by RHS under the same matching. If LHS matches with more than one region in the source model, one of the regions is non-deterministically chosen and rewritten by RHS. The application of the rule is repeated until the current model does not contain any LHS-matching region. A matching is a binding of all pattern variables to concrete values. Pattern variables are used in LHS and RHS in order to identify

objects or as a representation of attribute values. The value of pattern variables are possibly restricted by the when-clause of the rule.



(a) FolderFile metamodel (b) ChangeAccess transformation rule

Fig. 1. Metamodel and transformation rule

We illustrate the application of graph transformation rules on models written in a simple FileFolder-language, whose metamodel is given in Fig. 1(a). Instances of this metamodel are tree structures over folders and files. Each file or folder has an attribute `readOnly` of type `Boolean`. Suppose, a transformation should update for each file in the tree the value of its attribute `readOnly` with the `readOnly` value of its parent folder (if such a folder exists). Such a transformation is concisely formalized by the graph transformation rule `ChangeAccess` shown in Fig. 1(b).

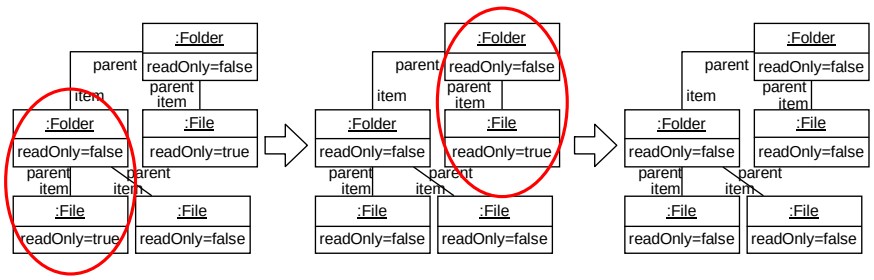


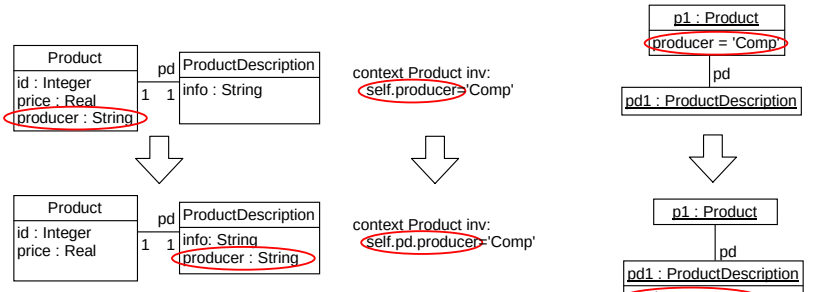
Fig. 2. Sequence of transformations

The LHS of `ChangeAccess` matches in a given source model with each pair of `File-Folder` instances that are connected by a parent-item link and whose values for attribute `readOnly` are different (see when-clause). Due to the RHS, the LHS-matching structure is rewritten by the same pair of `File-Folder` instances but the value for `readOnly` in the file has changed. The rule `ChangeAccess` is applied iteratively as long as LHS-matching structures can be found. Note how termination of this process is ensured by the when-clause. Figure 2 shows an application of `ChangeAccess` on a concrete source model.

3 Formalization of Semantic Preserving Refactoring Rules for UML/OCL

Research on refactoring has focused so far on implementation code but many refactoring rules for (object-oriented) implementation languages can be adapted to UML class diagrams and OCL constraints [3]. Since refactoring rules for UML/OCL models refer to the metamodel defining UML class diagrams and OCL expressions, we have included – for the sake of understandability – the relevant fragments of the metamodel in App. A.

Figure 3(a) shows the application of the refactoring rule `MoveAttribute` on a concrete UML/OCL model. The attribute `producer` is moved over an association with multiplicity 1 on both ends (called 1–1 association in the remainder of the paper) from class `Product` to `ProductDescription`. The attached OCL constraint has to be changed as well since the referred attribute `producer` is not owned any longer by class `Product`.



(a) Refactoring of UML/OCL model

(b) Refactoring of object diagram

Fig. 3. Application of `MoveAttribute` on an example

In the rest of this section we present a graph-transformation based formalization of the refactoring rule `MoveAttribute` and, as a new contribution of this paper, give a correctness criterion for the semantic preservation of UML/OCL refactoring rules. The section closes with a discussion on applying the correctness criterion on more complicated variants of the `MoveAttribute` rule, in which the attribute is moved over an 1–* or *–1 association.

3.1 Formalization of the simple form of `MoveAttribute`

In [3], we have already formalized a number of frequently used refactoring rules for UML class diagrams and analyzed their influence on OCL constraints attached to the refactored class diagram. The formalization of rule `MoveAttribute` is presented in Fig. 4. The refactoring is split into two graph transformation rules,

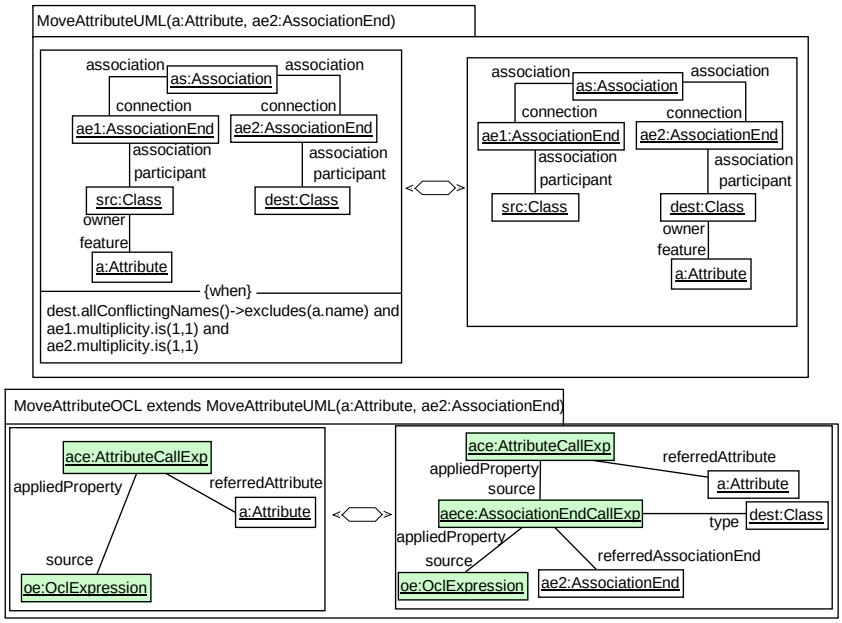


Fig. 4. Influence of `MoveAttribute` on class diagrams and OCL constraints

where the second one, which describes changes on OCL, extends the first rule, which formalizes the changes on the UML class diagram. The two parameters *a* and *ae2* of the first rule determine the attribute to be moved together with the association over which the attribute is moved (note that the parameter *ae2* identifies both the association and the destination class). The when-clause of the first rules prevents rule applications that would yield syntactically incorrect target models (an attribute must not be moved if its name is already used in the destination class). Furthermore, the when-clause explicates the assumption of moving the attribute over an 1–1 association.

Since the second rule is an *extension*, it can refer to elements from the extended rule, e.g. *a:Attribute*. Semantically, rule extension means that the second rule is applied as many times as possible in parallel to each single application of the first rule. For our example: Whenever attribute *a* is moved from class *src* to class *dest* each attribute call expression of form *oe.a*[1] is rewritten by *oe.ae2.a*.

3.2 A correctness criterion for semantic preservation

Semantic preservation, intuitively, means that source and target model express 'the same'. Established criteria for the refactoring of implementation code, where 'the same' usually means that the observable behavior of original and refactored

[1] Here, for the informal argumentation, the attribute call expression mentioned in `MoveAttributeOCL` is rendered in OCL's concrete syntax.

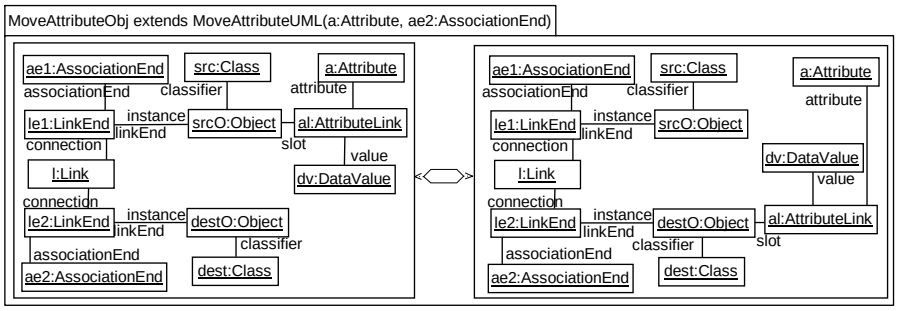


Fig. 5. Influence of `MoveAttribute` on object diagrams

program coincide, cannot be used for UML/OCL models, simply because the refactored UML class diagram with annotated OCL constraints is a static model of a system and does not describe behavior.

We propose to call a UML/OCL refactoring rule *semantic preserving* if the conformance relationship between the refactored UML/OCL model and its instantiations is preserved. An *instantiation* can be represented as an object diagram whose objects, links and attribute slots obey all type declarations made in the class diagram part of the UML/OCL model. An object diagram *conforms to* a UML/OCL model if all OCL invariants evaluate to true and all multiplicity constraints for associations of the class diagram are satisfied. A first – yet coarse and not fully correct (see below) – characterization of conformance preservation is that whenever an object diagram does/does not conform to the source model, it also does/does not conform to the target model.

This criterion, however, is still too coarse since it ignores the structural changes of instances of source and target model, e.g., applying `MoveAttribute` changes the owning class of the moved attribute (see Fig. 3(b) for illustration). In order to solve this problem, one has to bridge these structural differences of the model instances. This is realized by the transformation shown in Fig. 5.

Taking the structural differences between instances of source and target model into account, the semantic preservation can now be formulated as:

Definition 1 (Semantic Preservation of UML/OCL Refactorings).

Let cd_o be a class diagram, $constr_o$ be any of the constraints attached to it, od_o be any instantiation of cd_o, and $cd_r, constr_r, od_r$ be the refactored versions of $cd_o, constr_o, od_o$, respectively. The refactoring is called semantic preserving *if and only if*

$$eval(constr_o, od_o) = eval(constr_r, od_r)$$

holds, where $eval(constr, od)$ denotes the evaluation of the OCL constraint constr in the object diagram od.

3.3 Formalization of general forms of `MoveAttribute`

The formalization of `MoveAttribute` covers so far a rather simple case: The attribute *a* is moved from the source to the destination class and in all attached OCL constraints, the attribute call expressions of form *oe.a* are rewritten to *oe.ae2.a*. Semantic preservation of the rule is rather intuitive because for each object *srcO* of source class *src* there exists a unique, corresponding object *destO* of destination class *dest* and the slot *al* for attribute *a* on *srcO* is moved to *destO* (see rule `MoveAttributeObj` in Fig. 5). Before we present in Section 4 a technique to prove semantic preservation, we want to formalize now some versions of rule `MoveAttribute` for other cases than moving over an 1–1 association. As we will see shortly, the semantic preservation of the more general forms of `MoveAttribute` can only be ensured if the conditions for applying the rule (formalized by the when-clause) also refer to object diagrams.

We discuss in the next Subsection 3.3.1 the case that the association keeps multiplicity 1 at the end of the destination class but has an arbitrary multiplicity at the opposite end of the source class. Subsection 3.3.2 discusses the opposite case with multiplicity 1 at the source end and arbitrary multiplicity at the destination end. The last case, arbitrary multiplicity at both ends, is not discussed here explicitly since this case is covered by combining the mechanisms used in the two other cases.

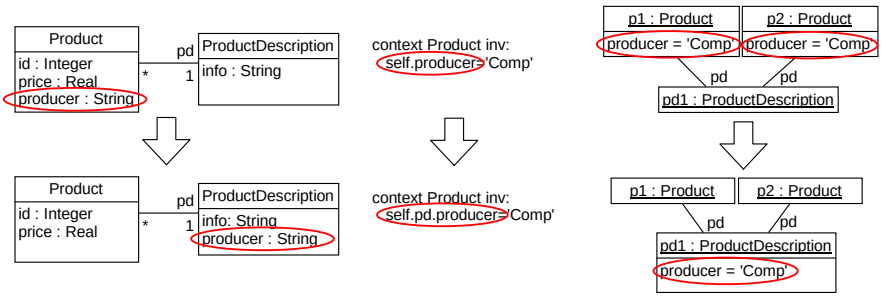


Fig. 6. Example refactoring if connecting association has multiplicities *–1

3.3.1 Multiplicities *–1 The UML and OCL part of the refactoring rule are basically the same as for moving the attribute over an 1–1 association. The only change is a new semantic precondition in order to ensure semantic preservation: All source objects (i.e., objects of the source class), which are connected to the same destination object (in Fig. 6, the source objects *p1*, *p2* are connected to the same object *pd1*), must share the same value for the moved attribute. For this reason, the when-clause of the UML part has changed compared to the previous version shown in Fig. 4 to:

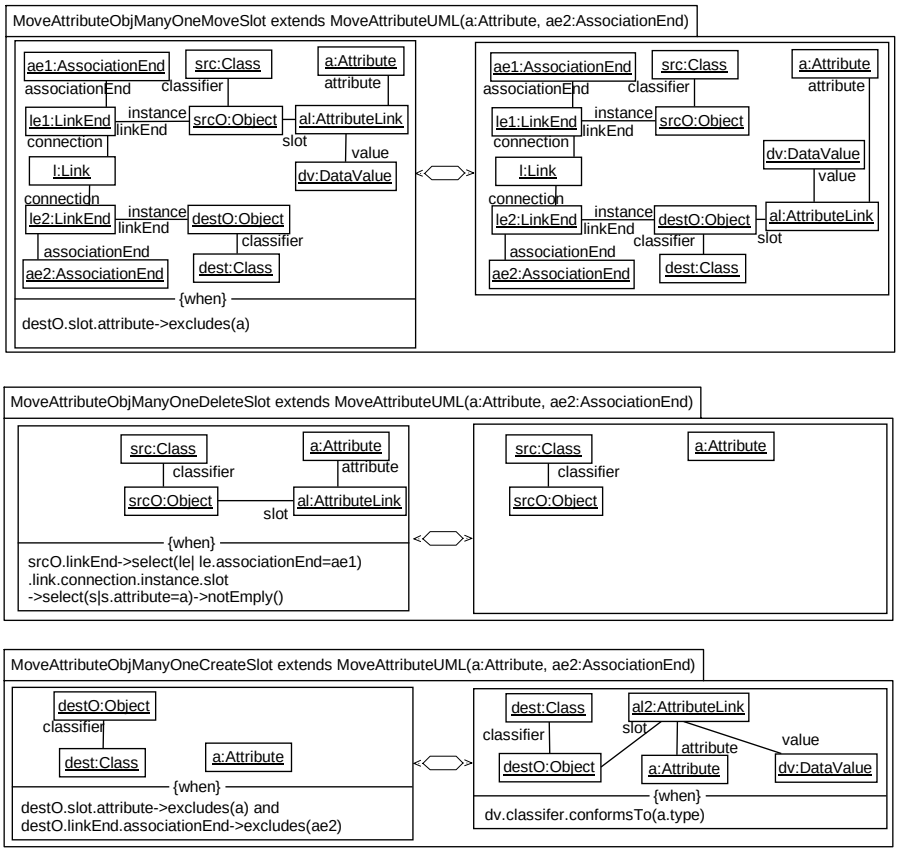


Fig. 7. Object diagram part of refactoring rule if association has multiplicities *–1

```
dest.allConflictingNames()->excludes(a.name) and
ae2.multiplicity.is(1,1) and
dest.instance->forAll(do|
    do.linkEnd->select(le|le.associationEnd=ae2)
    ->collect(ae| ae.oppositeLinkEnd.instance)
    ->forAll(so1,so2|
        a.attributeLink->forAll(al1,al2|
            al1.instance=so1 and al2.instance=so2
            implies
            al1.value=al2.value)))
```

This *semantic precondition* seems, at a first glance, to be put at a wrong place. Is a refactoring of UML/OCL models not by definition a refactoring of the static structure of a system and done when developing the system? And at that time, are system states, i.e. the instantiations of the class diagram, not unavailable? Yes, this is a common scenario in which all refactoring rules, whose

when-clause refers to object diagrams, are not applicable due to semantical problems a refactoring step might cause. But there are also other scenarios, e.g. where a class diagram describes a database schema and an OCL constraint can be seen as a selection criterion for database entries. Here, it would be possible to check whether the content of the database satisfies all semantic preconditions when applying the refactoring. If the refactoring rule is semantic preserving, one can deduce that a refactored database entry satisfies a refactored selection criterion if and only if the original selection criterion is satisfied by the original database entry.

The object diagram part of the refactoring shown in Fig. 7 reflects the fact that slots cannot be moved any longer naively, because the destination object would get in that case as many slots as it has links to source objects (but only one slot is allowed). The first two rules formalize that only one slot is moved to the destination object and all remaining slots at the linked source objects are deleted. The last rule `MoveAttributeObjManyOneCreateSlot` covers the case when a destination object is not linked to any source object. In this case, a slot for the moved attribute is created at the destination object and initialized with an arbitrary value (dv) of appropriate type.

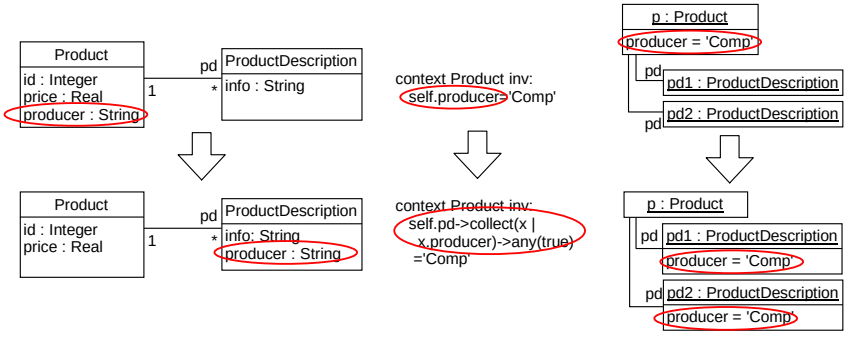


Fig. 8. Example refactoring if connecting association has multiplicities 1–*

3.3.2 Multiplicities 1–* Compared with moving attribute over an 1–1 association, the refactoring has changed in the OCL part and in the object diagram part; the UML part has remained the same (except of a slight extension of the when-clause). In object diagrams, the slot for the moved attribute at each source object is copied to all the associated destination objects (see Fig. 8). Semantic preservation of the rule can only be ensured if for each source object at least one destination object exists, with which the source object is linked (otherwise, the information on the attribute value for the source object would be lost). Thus, the when-clause of the UML part has been rewritten as

`dest.allConflictingNames()->excludes(a.name)` and

```
ae1.multiplicity.is(1,1) and
src.instance->forAll(so|
    so.linkEnd->select(le|le.associationEnd=ae1)->notEmpty())
```

The object diagram part of the refactoring rule is changed as shown by the two upper rules in Fig. 9. The first rule copies the slot *al* for attribute *a* from the source object *srcO* to each of the linked destination objects *destO*. After this has been done, the second rule ensures deletion of slot *al* at the source object *srcO*. Note that this rule is essentially the same as the rule for deletion of slots in the previous subsection.

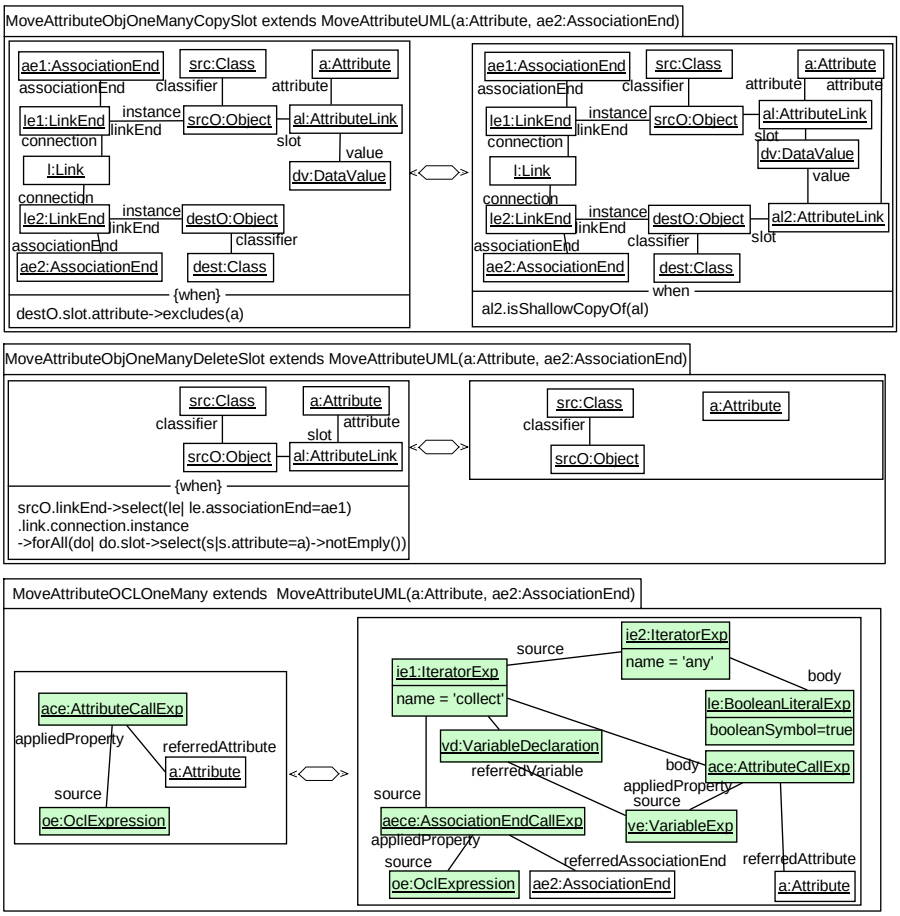


Fig. 9. Object diagram and OCL part of refactoring rule if connecting association has multiplicities 1–*

The third rule in Fig. 9 shows the OCL part of the refactoring rule. If the upper limit of the multiplicity at the destination class is greater than 1,

the rewriting of *oe.a* to *oe.ae2.a*, as it was done in the previous versions of `MoveAttributeOCL`, would cause a type error since the type of subterm *oe.ae2* would be a collection type. However, since oe.ae2 is part of the attribute call expression *oe.ae2.a*, an object type would be expected.

In order to resolve this problem, the expression *oe.ae2* is wrapped by a collect()-expression, which is, in turn, wrapped by an any()-expression. Please note that, despite of the non-deterministic nature of any() in general, the rewritten OCL term oe.ae2−>collect(x|x.a)−>any() is always evaluated deterministically, because the subexpression oe.ae2−>collect(x|x.a) always evaluates in the refactored object diagram to a singleton set.

4 `MoveAttribute` is Semantic Preserving

For a proof of the semantic preservation of a UML/OCL refactoring rule it is necessary to have a formal definition on how OCL constraints are evaluated. The evaluation function *eval* is defined with mathematical rigor in the OCL language specification [5]. The mathematical definition is, however, clumsy to apply in our scenario since it does not match the graph-based definitions we used so far for the formalization of our refactoring rules.

For this reason, we propose an alternative formalization of *eval* in form of graph-transformation rules. Due to the lack of space, we present here only the definition of *eval* for attribute call expressions and association end call expressions (a more complete version of OCL's semantics can be found in [9]). Fortunately, these two definitions are sufficient for proving the semantic preservation of `MoveAttribute` if the attribute is moved over an 1–1 association.

The formalization of *eval* given in Fig. 10 refers to a slightly extended version of the OCL metamodel in which the metaclass `OclExpression` has a new association to metaclass `Instance` (with multiplicity 0..1 and role `eval`). A link of this association from an object *oe:OclExpression* to an object *i:Instance* indicates that the expression *oe* is evaluated to *i*. If an expression does not have such a link to `Instance`, then this expression is not evaluated yet.

The first rule `EvalAttributeCallExp` defines the evaluation of expressions of form *oe.a* (where *a* denotes an attribute) in any object diagram that conforms to the underlying class diagram. The rule can informally be read as follows: Within the syntax tree of the OCL constraint to be evaluated, we search successively for expressions of form *oe.a* which are not evaluated yet (when-clause) but whose subexpression *oe* is already evaluated (to an object named *o*). Due to the type rules of OCL we know that object *o* must have a slot for attribute *a*. The lower part of the LHS shows the relevant part of the object diagram in which the OCL constraint is evaluated. The value of the slot for attribute *a* at object *o* is represented by variable *dv*. The RHS of rule `EvalAttributeCallExp` differs from LHS just by an added link from object *ac* (what represents expression *oe.a*) to *dv*. Informally speaking, the expression *oe.a* is now evaluated to *dv*. The second rule `EvalAssociationEndCallExp` is defined analogously. Based on this formalization we can state the following theorem:

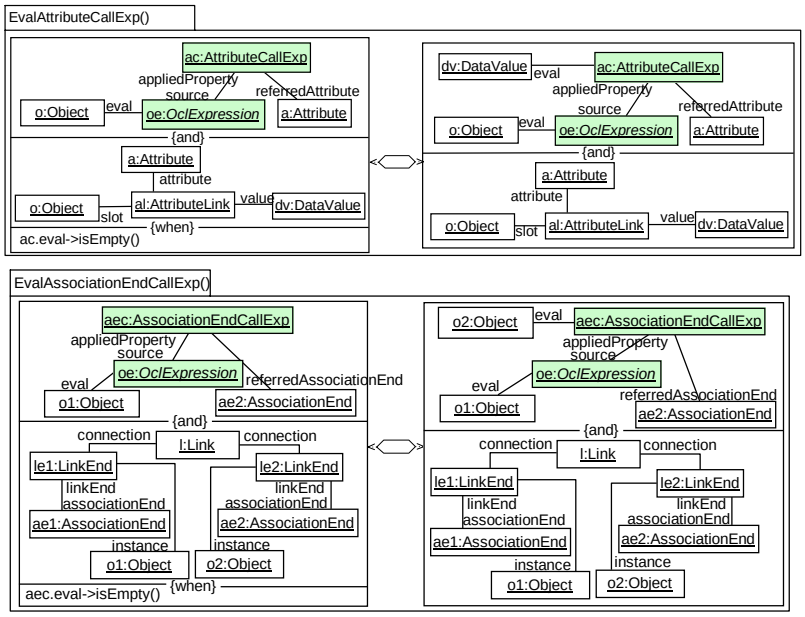


Fig. 10. Evaluation of OCL expressions (attribute call, association navigation)

Theorem 1 (Semantic Preservation of MoveAttribute). *Let $cd_o, constr_o, od_o$ be a concrete class diagram, a concrete OCL invariant, and a concrete object diagram, respectively, and $cd_r, constr_r, od_r$ their version after the refactoring of moving attribute a from class src to dest has been applied. Then,*

$$eval(constr_o, od_o) = eval(constr_r, od_r)$$

Proof: By construction, $constr_o$ and $constr_r$ differ only at places where $constr_o$ contains an expression form $oe.a$. The refactored constraint $constr_r$ has at the same place the expression $oe.ae2.a$. By structural induction, we show that these both expressions are evaluated to the same value. By induction hypothesis, we can assume that oe is evaluated for both expressions to the same value $srcO$. In object diagram od_o, object $srcO$ must have an attribute link for a, whose value is represented by dv. According to `EvalAttributeCallExp`, $oe.a$ is evaluated in od_o to dv. Furthermore, in both od_o and od_r the object $srcO$ is linked to an object $destO$ of class $dest$. According to `EvalAssociationEndCallExp`, the expression $oe.ae2$ is evaluated to $destO$ in od_r. Furthermore, we know by construction of od_r that $destO$ has an attribute slot for a with value dv. Hence, $oe.ae2.a$ is evaluated to dv.

5 Conclusions and Future Work

While the MDA initiative of the OMG has triggered recently much research on model transformations, there is still a lack of proof techniques for proving

the semantic preservation of transformation rules. In the MDA context, this question has been neglected also because many modeling languages do not have an accessible formal semantics yet what seems to make it impossible to define criteria for semantic preservation. However, as our example shows, the semantic preservation of rules can also be proven if the semantics of source/target models is given only partially. In case of `MoveAttribute` it is enough to agree on the semantics of attribute call and association end call expressions.

In this paper, we define and motivate a criterion for the semantic preservation of UML/OCL refactoring rules. Our criterion requires to extend a refactoring rule by a mapping between the semantic domains (states) of source and target model. We argue that our running example `MoveAttribute` preserves the semantics according to our criterion. Our proof refers to the three graphical definitions of the refactoring rule (class diagram, OCL, object diagram) and to a novel, graphical formalization of the relevant parts of OCL's semantics.

As future work, we plan to apply our approach also on pure OCL refactoring rules, i.e., rules, which simplify the structure of complicated OCL expressions but do not change anything in the underlying class diagram (see [10]).

Acknowledgments We would like to thank the anonymous reviewers for their very helpful comments. This work was supported by Swiss Science Foundation (SNF), contract number 200020-109492/1.

References

1. Dave Astels. Refactoring with UML. In *International Conference eXtreme Programming and Flexible Processes in Software Engineering*, pages 67–70, 2002.
2. Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In *UML 2001*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.
3. Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. In *MoDELS'05*, volume 3713 of *LNCS*, pages 280–294. Springer, 2005.
4. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.
5. OMG. UML 2.0 OCL Specification – OMG Final Adopted Specification. OMG Document ptc/03-10-14, Oct 2003.
6. William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
7. T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution*, 17(4):247–276, 2005.
8. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, second edition, 2005.
9. Slaviša Marković and Thomas Baar. An OCL semantics specified with QVT. In *MoDELS'06*, volume 4199 of *LNCS*, pages 660–674. Springer, October 2006.
10. Alexandre Correa and Cláudia Werner. Applying refactoringtechniques to UML/OCL. In *UML 2004*, volume 3273 of *LNCS*, pages 173–187. Springer, 2004.

A Metamodels

This appendix contains the relevant parts of the metamodels for UML 1.5 (including object diagrams) and OCL 2.0. For the sake of readability, the metaclasses from the OCL metamodel are rendered with gray rectangles.

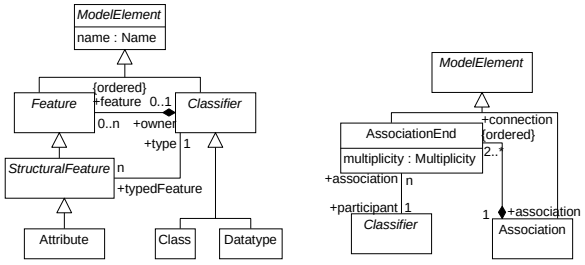


Fig. 11. UML - Core Backbone and Relationships

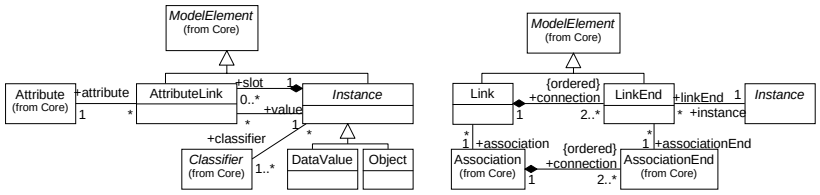


Fig. 12. UML - CommonBehavior Instances and Links

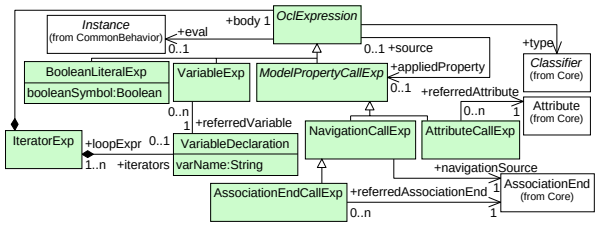


Fig. 13. OCL - Expressions