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

Thomas Baar and Slaviša Marković

Technical Report No. LGL-REPORT-2006-001
February 2006

Software Engineering Laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland

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: Refactorings, Graph Transformations, Semantic Preserving Model Transformations, UML, OCL, QVT

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 a schematic way. So far, many approaches and tool are tailored for refactoring of programming code. Only recently, refactoring rules became also applicable on more abstract software models such as UML class diagrams. The literature (e.g. [1, 2]), however, ignores so far the fact that UML class diagrams usually contain additional OCL annotations in form of invariants, pre and post conditions and that these OCL annotations are often affected by a refactoring of the underlying class diagram.

There are two important criteria each refactoring rule should meet. 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 should be syntactically correct. Secondly, a rule should be *semantic preserving*, i.e. the semantics of source and target model should coincide. Proving the semantic preservation is often seen to be difficult (cf. [3]), also because for many modeling languages lack a formal semantics and criteria for the semantical equivalence of two models.

In this paper we define a general criterion of semantic preservation for refactoring rules of UML class diagrams with OCL annotations (called *UML/OCL models* in the remainder of this paper). In addition, we describe a technique for proving the semantic preservation according to our criterion. Our proofs are comparably easy to understand since the refer only to definitions given in a graphical way that are easy to grasp by humans.

In Sect. 2 we give a brief introduction to graph transformation rules. Section 3 introduces UML/OCL refactoring rules, our criterion for their semantic preservation, and our approach to prove it. Section 4 concludes paper and gives an outlook for future work.

1.1 Related Work

In his seminal work [4], Opdyke gives a catalog of refactoring rules for C++ programs. Opdykes defines semantic preservation (also called *behavioral preservation* when refactoring rules are tailored for implementation code) 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* emerged (a good overview is given by Mens et al. in [5]).

Unfortunately, the criteria for semantic preservation of refactoring rules for implementation code are not applicable for UML/OCL refactoring rules because the 'domain of refactorings' are different. When refactoring implementation code, one is interested to keep the (observable) behavior of the program implemented by this code (cmp. Opdyke above). When refactoring UML class diagrams, Opdyke's criterion is not applicable. What should be kept unchanged are the possibilities to instantiate the class diagram, so here 'structural preservation' is more important. The basic idea of our approach goes back on works on equivalent data structure representations by Hoare, e.g. [6]

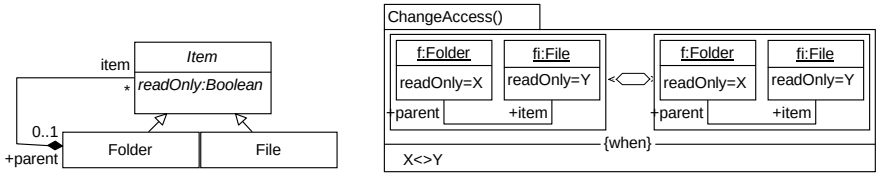
2 Graph Transformation Rules

Graph transformation rules are a popular formalism to specify refactoring rules because of their expressive power and their graphical syntax. The source and target model of a refactoring step are seen as a typed graph, more precisely, as an instance of the modeling language's metamodel. We assume the reader to be familiar with the technique of metamodeling (a good introduction is [7]) and illustrate this technique here merely with a toy example of a FileFolder-language.

A graph transformation rule consists of two patterns: left hand side (LHS) and right hand side (RHS). When applying a rule on a given source model, a LHS-matching region in the source model is first searched and then substituted by RHS under the same matching. A matching is a binding to concrete values for all variables occurring in the pattern. Patterns are specified in a generalized form of object diagrams where variables are used to label objects and to represent

values of attributes. Patterns might contain also further syntactic structures, such as negative application conditions or multiobjects, but these extensions depend on the concrete graph transformation system being used. In this paper, we rely on the OMG standard QVT whose detailed syntax and semantics is given in [8]. QVT allows to restrict the values for pattern variables by a when-clause. When applying a rule for which more than one LHS-matching regions in the source model exist, one of them 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.

We illustrate the graph transformation approach on instances of the class diagram shown in Fig. 1(a) (that serves as a metamodel in this example). Instances of this metamodel form tree-like structures of folders and files. Each file or folder has an attribute `readOnly` of type `Boolean`.



(a) FolderFile metamodel (b) ChangeAccess transformation rule

Fig. 1. Metamodel and transformation rule

Imagine, we want to unify for each file the value of `readOnly` with the `readOnly` value of its parent folder (if such a folder exists). Such a transformation is elegantly formalized by the graph transformation rule `ChangeAccess` shown in Fig. 1(b).

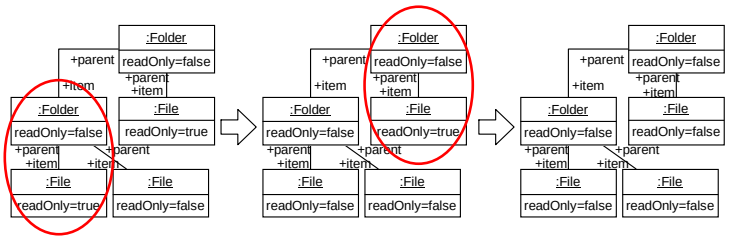


Fig. 2. Sequence of transformations

The LHS of `ChangeAccess` matches in a given source model with all folder-file pairs that are connected by a parent-item link and whose `readOnly` values differ (see when-clause). Due to the RHS, the LHS-matching structure is rewritten by the same folder-file pair but the value for `readOnly` in the file has changed. Note

that `ChangeAccess` is applied iteratively as long as it can find LHS-matching structures (the when-clause ensures in this example the termination of this process). Since the rule `ChangeAccess` is confluent (as well as the rules presented later) the result of the transformation does not depend on the ordering with which the LHS-matching regions where chosen. Figure 2 shows an application of `ChangeAccess` on a concrete source model.

3 Semantic Preserving Refactoring Rules for UML/OCL

The refactoring of software artifacts more abstract than implementation code became only recently a topic in research, but many refactoring rules for (object-oriented) implementation languages can be adapted to UML class diagrams and OCL constraints [9]. Refactoring rules for UML/OCL models refer to the metamodel defining UML class diagrams and OCL expressions (the relevant fragments are shown in App. A). For the sake of readability, the metaclasses from the OCL metamodel are rendered with gray rectangles.

In this paper, we investigate the refactoring rule `MoveAttribute` that moves the attribute *a* from its current class to class *dest* (*dest* and *a* are passed as parameters when `MoveAttribute` is applied). There are two application conditions. Firstly, the class owning attribute *a* in the source model and destination class *dest* must be connected by an association with multiplicity 1-1. Secondly, the name of the moved attribute must not already be used for another attribute in *dest* or in one of its parent or child classes.

Figure 3 shows from top to bottom the application of rule `MoveAttribute` on a concrete UML/OCL model, the attribute named `producer` is moved from class `Product` to `ProductDescription`. Please note that the attached OCL constraint has to be changed as well, otherwise it would become syntactically incorrect.

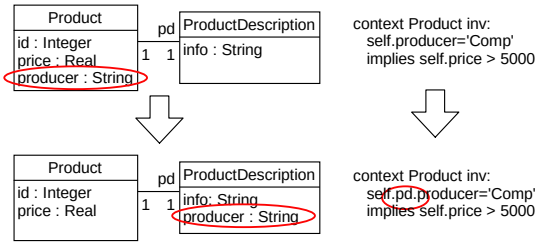


Fig. 3. Application of `MoveAttribute` on an example

In the rest of this section we present a graph-transformation based formalization of the refactoring rule `MoveAttribute` and, as the new contribution of this paper, an argumentation why this refactoring preserves the semantics when applied on a concrete UML/OCL model. Our argumentation includes an intu-

itive criterion for semantic preserving refactoring rules. Moreover, we sketch the proof that this criterion is met for our formalization of `MoveAttribute`.

3.1 Refactoring Rules for UML/OCL Models

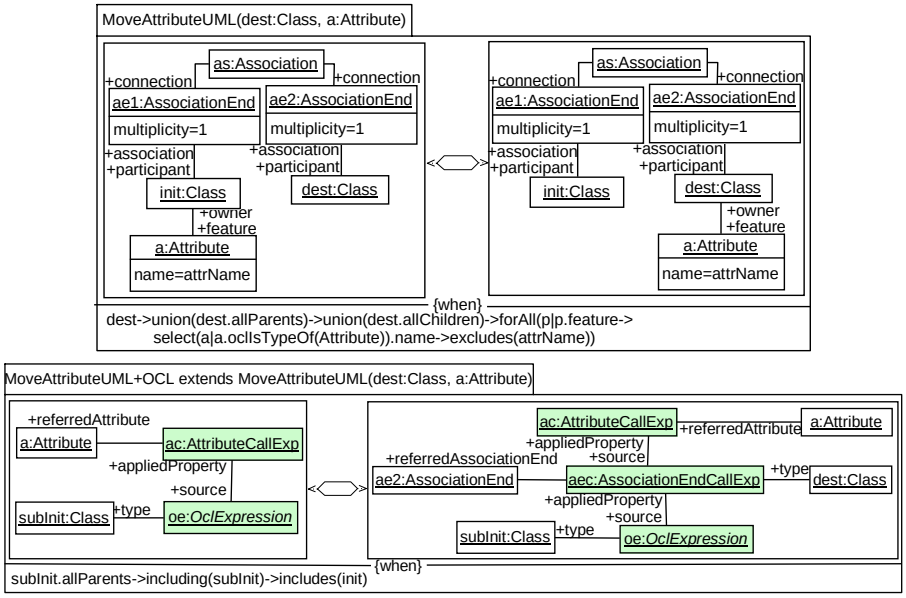


Fig. 4. Influence of `MoveAttribute` on class diagrams and OCL constraints

In [9], 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. Figure 4 shows the formalization of rule `MoveAttribute` as it is given in [9]. The formalization is split into two graph transformation rules where the second one, which describes changes on OCL, extends the first rule, which formalizes the changes of the class diagram. QVT's extension mechanism allows the second rule to refer to elements from the first rule, e.g. `a:Attribute`. Semantically, the second rule is applied as many times as possible in parallel to each single application of the first rule. For our example, this means that whenever attribute *a* is moved from class *init* to class `dest` each attribute call expression of form *oe.a*[1] is rewritten by *oe.ae2.a* where *ae2* is the association end on *dest* of the 1-1 association connecting `init` with *dest*. The when-clause imposes for the type of *oe* the restriction to be a subclass of *init*. This condition ensures termination of the rule application.

[1] Here, for the informal argumentation, we render the attribute call expression mentioned in MoveAttributeUML+OCL with OCL's concrete syntax.

3.2 A Correctness Criterion for Semantic Preserving Refactoring Rules

In order to be *refactoring rule*, a graph transformation rule as the one defined in Fig. 4 should preserve both the syntax and semantics when applied on any possible source model. Syntactic preservation means that the application of the rule terminates and the target model is syntactically correct, i.e. the target model is an instance of the UML/OCL metamodel and obeys its well-formedness rules. For rule `MoveAttribute`, termination is ensured by the when-clauses. Furthermore, the target model indeed conforms to the UML/OCL metamodel (the formal proof requires intimate knowledge of complete UML/OCL metamodel and is skipped here). Semantic preservation intuitively means, that the source and the target model express 'the same'. In case of refactoring of implementation code, 'the same' usually means that the observable behavior of original and refactored program coincide, but the literature lacks so far a commonly agreed criterion on what 'the same' means in the context of UML/OCL refactoring rules.

We propose as a criterion for semantic preservation of UML/OCL transformation rules that the states and state transitions specified by source and target model are 'the same'. A state is an object diagram that conforms to the model (all annotated OCL invariants are satisfied). State transitions are used as the semantic domain for operation specifications and consist of pre- and post-states.

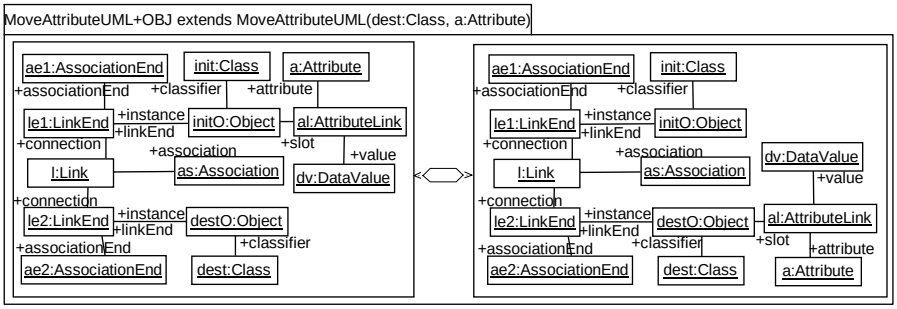


Fig. 5. Influence of `MoveAttribute` on object diagrams

Since the source and target model often have structural differences (for `MoveAttribute`, both models have the same classes and associations but the owning class of the moved attribute has changed) the states of source and target model have structural differences as well. Thus, having 'the same' states cannot mean that source and target model have an identical statespace. Instead, a mapping between the statespaces is needed and this mapping should become part of the graph transformation rule. In case of `MoveAttribute`, this mapping is actually an isomorphism and defined in Fig. 5 by an new extension of the rule given in Fig. 4.

Note that each object diagram of the source model (no matter if the OCL invariants are satisfied) is mapped now to exactly one object diagram of the target model. The correctness criterion means that object diagrams of the source model satisfying the annotated OCL constraints are mapped to exactly those object diagrams of the target model which satisfy the refactored OCL constraints. Thus, in order to verify semantic preservation of a refactoring rule, it would be sufficient to prove the following: Let $cd_o, cd_r, constr_o, constr_r, od_o, od_r$ be the source and target version of a class diagram, OCL constraint, and object diagram. Then[2],

$$eval_{cd_o}(constr_o, od_o) = eval_{cd_r}(constr_r, od_r)$$

3.3 `MoveAttribute` is Semantic Preserving

For an argumentation on the semantic preservation it is necessary to have a formal definition on how OCL constraints are evaluated, i.e. a formal definition of *eval*. The function *eval* is defined with mathematical rigor in the OCL language specification [10]. The mathematical definition is, however, clumsy to read and would not be match the graphical style we used so far suitable for the formalization of the refactoring rules. Thus, we have formalized that part of *eval* needed for the argumentation of `MoveAttribute`'s semantical correctness in a graph-transformation way. As we will see later, we only need the definition of *eval* for attribute call expressions and navigation expressions over an association with multiplicity 1.

The formalization of *eval* given in Fig. 6 refers to a slightly extended version of the OCL metamodel where the metaclass `OclExpression` has a new association to metaclass `Instance` (with multiplicity 0..1 and role `eval`)[3]. A link of this association to an object `i:Instance` indicates for an instance of `OclExpression` that the expression is evaluated to `i`. If an expression has no such link, it means that this expression has not been 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 an attribute link (in former version of UML called *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 attribute link on object *o* for attribute *a* is represented by variable *dv*. The RHS of rule `EvalAttributeCallExp` differs from LHS just

[2] We cover only the case here where $constr_o, constr_r$ are invariants or pre-conditions. The evaluation of post-conditions would require two states. However, if states are mapped correctly, then pairs of states are mapped correctly as well.

[3] This is actually a simplified version of OCL evaluation since we ignore here the binding of free variables. For the two evaluation rules, however, variable bindings can be ignored since they are not affected.

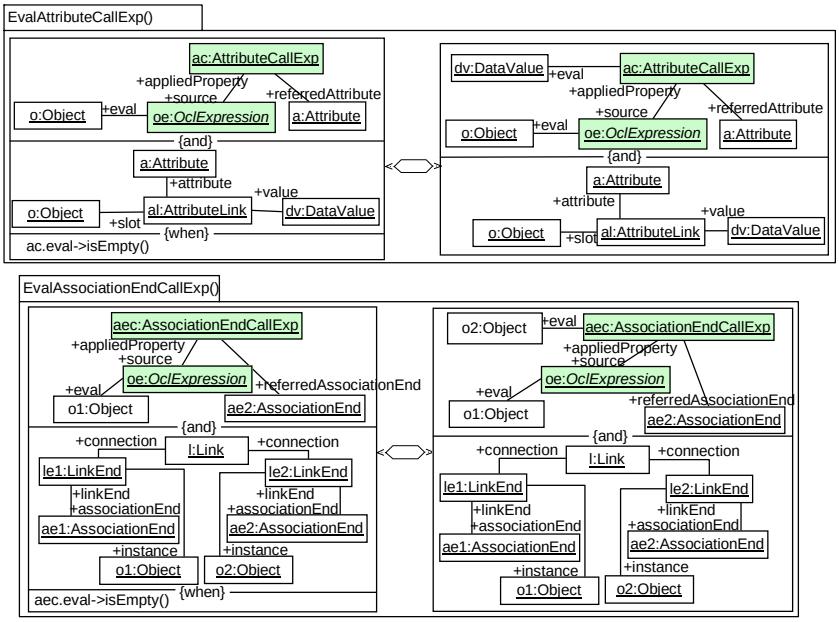


Fig. 6. Evaluation of OCL expressions (attribute call, association navigation)

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

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

Proof (sketch): By construction, $constr_o$ and $constr_r$ differ only at places where $constr_o$ contains an expression $oe.a$. 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 $initO$. In object diagram od_o, object $initO$ 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 $initO$ 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 in this object diagram an attribute slot for a with value dv. Hence, $oe.ae2.a$ is evaluated to dv.

4 Conclusion 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 to show that a transformation rule is semantic preserving. 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. As our example shows, however, to prove the semantic preservation of rules it is sometimes only necessary to have a partial formalization of the involved modeling languages, in case of `MoveAttribute` it is enough to agree on the semantics of attribute call and association end 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) as well as 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 where the structure of complicated OCL expressions is simplified but the underlying class diagram remains the same, see [11]).

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, François Pennaneac'h, Wai-Ming Ho, Alain Le Guennec, and Jean-Marc Jézéquel. Using UML action semantics for executable modeling and beyond. In *CAiSE*, volume 2068 of *LNCS*, pages 433–447. Springer, 2001.
3. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.
4. William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
5. T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal on Softw. Maint. and Evolution*, 2005.
6. C. A. R. Hoare. Proof of correctness of data representation. In *Language Hierarchies and Interfaces*, volume 46 of *LNCS*, pages 183–193. Springer, 1975.
7. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, second edition, 2005.
8. OMG. MOF QVT Final Adopted Specification. OMG Adopted Specification ptc/05-11-01, Nov 2005.
9. Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. In *MoDELS'05*, volume 3713 of *LNCS*, pages 280–294. Springer, 2005.
10. OMG. UML 2.0 OCL Specification – OMG Final Adopted Specification. OMG Document ptc/03-10-14, Oct 2003.
11. Alexandre Correa and Cláudia Werner. Applying refactoring techniques 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 UML1.5 (including object diagrams) and OCL2.0.

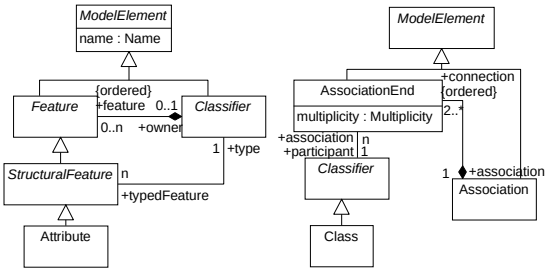


Fig. 7. UML - Core Backbone and Relationships

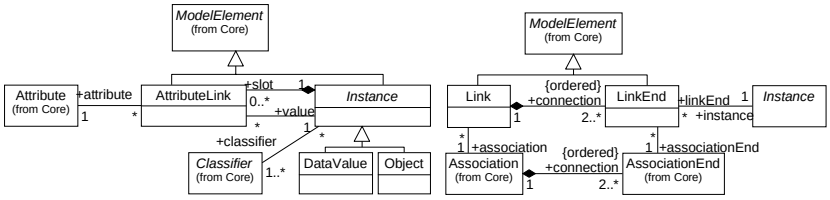


Fig. 8. UML - CommonBehavior Instances and Links

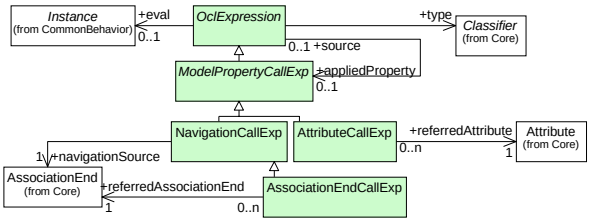


Fig. 9. OCL - ModelPropertyCallExp