MODEL REFACTORING USING TRANSFORMATIONS

PAR

Slaviša MARKOVIĆ

Bachelor of Science in Information Systems Engineering, University of Belgrade, Serbie
et de nationalité serbe

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Lausanne, EPFL
2008

Abstract

Modern software is reaching levels of complexity encountered in biological systems; sometimes comprising systems of systems each of which may include tens of millions of lines of code. Model Driven Engineering (MDE) advocates raising the level of abstraction as an instrument to deal with software complexity. It promotes usage of software models as primary artifacts in a software development process. Traditionally, these MDE models are specified by Unified Modeling Language (UML) or by a modeling language created for a specific domain.

However, in the vast area of software engineering there are other techniques used to improve quality of software under development. One of such techniques is refactoring which represents introducing structured changes in software in order to improve its readability, extensibility, and maintainability, while preserving behavior of the software. The main application area for refactorings is still programming code, despite the fact that modeling languages and techniques has significantly gained in popularity, in recent years.

The main topic of this thesis is making an alliance between the two virtually orthogonal techniques: software modeling and refactoring. In this thesis we have investigated how to raise the level of abstraction of programming code refactorings to the modeling level. This resulted in a catalog of model refactorings each specified as a model transformation rule. In addition, we have investigated synchronization problems between different models used to describe one software system, i.e. when one model is refactored what is the impact on all dependent models and how this impact can be formalized.

We have concentrated on UML class diagrams as domain of refactorings. As models dependent on class diagrams, we have selected Object Constraint Language (OCL) annotations, and object diagrams. This thesis formalizes the most important refactoring rules for UML class diagrams and classifies them with respect to their impact on object diagrams and annotated OCL constraints. For refactoring rules

that have an impact on dependent artifacts we formalize the necessary changes of these artifacts.

Moreover, in this thesis, we present a simple criterion and a proof technique for the semantic preservation of refactoring rules that are defined for UML class and object diagrams, and OCL constraints. In order to be able to prove semantic preservation, we propose a model transformation approach to specify the semantics of constraint languages.

Keywords

Model Driven Engineering, UML, Model Transformations, Refactoring, QVT, OCL Semantics, Semantics Preservation

Résumé

La complexité des logiciels actuels atteint des niveaux comparables à ceux rencontrés chez les systèmes vivants. Parfois, des systèmes logiciels peuvent en englober d'autres déjà eux-mêmes composés de dizaines de millions de lignes de code. L'Ingénierie Dirigée par les Modèles (IDM) pose l'abstraction en instrument de gestion de cette complexité. Pour ce faire, l'IDM promeut les modèles comme composants principaux d'un processus de développement logiciel. Souvent, ces modèles sont exprimés soit en utilisant le langage UML (Unified Modeling Language), soit dans un langage spécifique au domaine concerné.

Cependant, le domaine du génie logiciel est très vaste, et il existe d'autres techniques à même d'améliorer la qualité d'un système logiciel lors de son développement. Une de ces techniques est le réusinage (ou refactorisation), plus connue sous sa dénomination anglaise de refactoring. Cette technique permet de restructurer un code logiciel de manière à le rendre plus lisible, plus souple, ou plus facile à maintenir. Cependant, malgré le récent succès des langages et techniques de modélisation, la principale cible d'application du refactoring reste le code programme.

Le sujet principal de cette thèse est d'associer ces deux techniques a priori orthogonales que sont la modélisation et le refactoring. Dans cette thèse, nous avons porté la technique refactoring pour la mettre au service de la modélisation. Comme résultat, nous proposons ici un catalogue de refactorings de modèle, tous exprimés par une transformation de modèle. De plus, nous avons pris en compte le problème de la cohérence entres les différents modèles décrivant un système logiciel. Il est en effet nécessaire de définir comment un refactoring doit impacter les modèles dépendants du modèle initialement restructuré.

Nous nous sommes ici concentrés sur le refactoring des diagrammes de classes du langage UML, dont sont dépendants les modèles de contraintes en OCL (Object Constraint Language) et les diagrammes d'objets. Cette thèse formalise des refactorings parmi les plus importants pour les diagrammes de classes UML, tout

en les triant suivant leur impact sur les diagrammes d'objets ainsi que les modèles de contraintes OCL, impact que nous formaliserons également.

En outre, nous présentons ici une technique de preuve, basée sur un critère simple, pour la préservation sémantique des diagrammes de classes, des diagrammes d'objets et des contraintes, une fois appliqué un refactoring. Cependant, afin prouver formellement une préservation sémantique, il est nécessaire d'avoir à disposition une description formelle de ladite sémantique; nous proposons donc également un approche de description de la sémantique basée sur les transformations de modèles que nous appliquons au langage OCL.

Mot Clefs

Ingénierie Dirigée par les Modèles, UML, Transformations de Modèles, Refactorisation, QVT, Sémantique d'OCL, Préservation de la Sémantique

Acknowledgments

First of all, I wish to express my sincere gratitude to my advisor, Dr. Thomas Baar, for his counsel, guidelines, and help that he provided during all these years I spent at EPFL. Thomas managed to be not only an excellent supervisor but at the same time a good friend always ready to help. This combination made my work at EPFL a pleasant experience.

I would like to thank all the jury members Prof. Didier Buchs, Prof. Jean-Marc Jézéquel, and Prof. Alain Wegmann for having taken the time and effort to review my thesis and to serve on my examination board.

I am greatly indebted to Prof. Alfred Strohmeier for accepting me as a student in his lab, and giving me a chance to work on a PhD.

Special thanks to Frédéric Fondement for reviewing this thesis and giving a lot of precious comments that improved its quality, and to Shane Sendall for helping me "surviving" my first steps as a PhD student.

I would like to thank members of the ROCLET team Cédric Jeanneret and Leander Eyer for implementing ideas expressed in this thesis.

I was very fortunate to work in the pleasant and motivating scientific atmosphere of the Software Engineering Laboratory (LGL). I would like to thank previous LGL members for their support and friendliness: Mohamed Kandé, Jarle Hulaas, Rodrigo Garcìa, Raul Silaghi, Andrea Camesi, Benjamin Barras, Isabelle Moinon, and Jörg Kienzle. Also, I would like to thank members of the Software Modeling and Verification Group from University of Geneva, and the Systemic Modeling Laboratory from EPFL, for their encouragement and help in the very finish of my PhD studies.

I would like to thank friends from (not so) small Serbian community at EPFL for making my life in Switzerland much easier.

Finally, I would like to thank mama Desanka, tata Milorad, Nina, and Nikola for their unconditional support, encouragement, and patience during my everlasting studies. This work is dedicated to them.

Contents

List of Figures

Chapter 1

Introduction

Growth of software systems, increasing complexity and diversity of software platforms, and constant changes of the market demands for software, have led to extensive development of methods and techniques that, in a formalized way, deal with overall increase in software complexity.

One of available approaches for taming development of complex software is Model Driven Engineering (MDE). The intent is to use models as primary artifacts when building software systems, and in that way to raise the level of abstraction by discarding all information that are not relevant for a given viewpoint.

Ideally, all work of system design would be performed on the level of software models that depict different aspects of some software systems (e.g. structure, behavior) and leave dedicated tools to perform generation of executable code.

This type of abstraction, in domain of software engineering, is not a new paradigm. The similar process was applied when passing to high level programming languages leaving compilers to generate instructions that are understandable by machines.

Modern software development processes, such as Rational Unified Process (RUP) [55] and eXtreme Programming (XP)[12], promote the application of refactoring [42] to support iterative and incremental software development. Refactoring is a structured technique to improve the quality of software artifacts.

Although, artifacts produced in all phases of the software development life cycle could become a subject of refactoring, existing techniques and tools mainly target implementation code.

Current trend is to raise the level of abstraction of refactoring techniques so that it becomes feasible to apply refactorings on software artifacts that have higher level of abstraction that programming code.

1.1   Motivation

With the success of the UML, refactoring techniques have found a new domain of applications. The work presented in this thesis was initiated with the fact that contrary to well know catalogs for refactoring of programs, there was no comprehensive collection of refactorings for artifacts situated on higher level of abstractions, like UML models. In order to fill this lack, the first step that we performed was investigating application of refactoring rules on UML class diagrams. In addition to class diagrams refactorings, we have started research on how to propagate refactorings to dependant artifacts so the overall model remains consistent. An example for this type of refactoring propagation is automatic update of OCL constraints and UML object diagrams, once a UML class diagram is refactored.

For representation of refactoring rules, the initial formalism of our choice was OCL, similarly to [82]. However, we have abandoned this solution because of the size of the OCL expressions necessary to describe refactoring rules. Not only the rules were hard to read, but they were hard to maintain as well. In order to be able to specify our refactoring rules in a concise and a readable manner, we have changed the used formalism: instead of "pure" OCL expressions we have described our refactoring rules using model transformations.

A widely accepted definition of refactoring is given in [42] as "*A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*" A similar definition can be found in [69]: "*Refactorings do not change the behavior of a program; that is, 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.*"

When reasoning about refactorings of UML class diagrams annotated with OCL constraints, this definition of refactorings can't be applied because the UML/OCL diagrams represent only the static structure of the system without behavior as specified using programming languages. In order to still be able to apply the model refactorings, the refactoring definition has to be altered to cover "semantic preservation" of refactored models.

The problem with semantic preservation was that the semantic preservation criterion has not been defined precisely. Our goal was to specify such a criterion for refactorings of UML class diagrams, UML object diagrams and OCL constraints in a formal way so that it becomes feasible to check if one refactoring rule actually preserves semantics or not. In the context of UML and OCL we have defined semantics preservation as preservation of evaluation of OCL constraints.

This step in our work has triggered research on semantics of OCL because the OCL semantics was in the core of our semantics preservation criterion. Existing OCL semantics descriptions, given with set theory [68, Annex A], or with UML [68, Chapter 10], were not suitable for application in semantic preservation criterion for the refactoring rules defined as model transformations, because we wanted to have description of OCL semantics that is directly executable. In order to specify semantics of OCL we have used a novel approach of applying model transformation rules to describe effects of evaluation of OCL expressions.

1.2 Scope

As the title suggests, this work is about "Model Refactoring using Transformations". Hence, it is necessary to narrow the scope of the thesis, which lies somewhere within the broad area of model transformations.

Model Refactorings In this thesis, the main artifacts that are subject of refactorings are UML class and object diagrams, and OCL constraints. The same methodology applied on these types of UML diagrams could be successfully applied to any other types of UML diagrams as well.

Model Transformations Model transformation techniques used in this thesis are seen as means for performing model refactorings and specification of language semantics. Our intent was neither to define a new model transformation language nor to advocate usage of some existing specification. We have chosen and slightly adapted Query/View/Transformation (QVT) formalism for describing refactoring and evaluation rules, but there are no obstacles to perform the same tasks using any other available transformation specifications like, for example, Fujaba Story Diagrams [39].

Language Semantics One of the contributions of this thesis is the specification of the semantics of OCL using model transformations. Although, by using this technique, it would be possible to specify semantics of any other constraint language, the OCL is chosen in the context of UML/OCL refactorings.

This thesis does not contain the complete semantics of OCL and its standard library, but covers the most important parts of the language, and provides a guide on how to specify the complete semantics of OCL. Nevertheless, we make available the complete OCL semantics as implemented in our ROCLET tool, by means of the tool's website [88].

1.3   Thesis Contributions and Outline

1.3.1   Contributions

The goal of this thesis was to study UML/OCL models as subject of model refactorings, seen as a special usage for model transformations.

The main contributions of this thesis are the following:

1. Definition, representation, and classification of semantics-preserving refactoring rules for UML class diagrams annotated with OCL constraints, and corresponding object diagrams.

2. Specification of semantics of OCL in a way based on model transformations that is easy to grasp and to understand.

3. Definition of the criterion for semantics preservation of refactoring rules and application of that criterion.

1.3.2   Plan

Chapter 2: *Key Concepts*

This chapter defines some key concepts for this work. Key concepts include explanations of notations and techniques used in the rest of the thesis.

Chapter 3: *Refactoring UML/OCL Diagrams*

This chapter contains a catalog of refactoring rules for class diagrams classified with respect to their impact on annotated OCL constraints. For refactoring rules that have an impact on OCL constraints, we formalize the necessary changes in the OCL constraints. In this chapter we argue that the specified refactoring rules preserve the syntax of models under refactoring. For one of the specified rules, by using the KeY tool, we formally prove that it is syntax preserving. Semantics preservation is the topic of Chapter 5.

Chapter 4: *Model Transformations for Describing Semantics of OCL*

In this chapter we present a metamodel-based approach to define the semantics of constraints languages. The semantics of a constraint language is, roughly speaking, given by an evaluation function whose input is both an expression and a state, in which this expression should be evaluated. Our approach

uses model transformation rules to specify formally every single step in the evaluation process.

This chapter also contains a section on tailoring the semantics for OCL towards the needs of a Domain Specific Language (DSL), i.e. we illustrate on a concrete example how the problems of defining the semantics of a constraint language for a given DSL can be overcome.

Chapter 5: *Semantics Preservation of Refactoring Rules*

In this chapter we give a simple criterion and a proof technique for the semantic preservation of refactoring rules that are defined for UML class diagrams and OCL constraints. Refactoring rules presented in Chapter 3 are extended to cover UML object diagrams so it becomes possible to reason about their semantic preservation. The criterion itself is based on results of evaluation of OCL constraints on corresponding system snapshots depicted by the object diagrams.

Chapter 6: *Conclusions*

In this chapter, we conclude the thesis and summarize the key results. This chapter also contains a discussion of future research work.

Appendix A: *Tool Support*

This appendix describes the RoclET tool that implements all concepts investigated in this thesis: RoclET allows creation of object and class diagrams, parsing and syntax highlighting of OCL constraints, refactoring of UML/OCL models, and evaluation of OCL constraints against object diagrams.

Chapter 2

Key Concepts

2.1   The Unified Modeling Language

The Unified Modeling Language (UML)[19, 75, 43] is a visual language used to describe and design software systems. It provides a set of notations that can be used to model different aspects of one software system, like its static structure, or its behavior.

UML was born as a fusion between different object-oriented modeling techniques like OMT [74], Booch [18], and OOSE [50], and since its very beginning, it is under constant evolution that leads to improvements in preciseness and expressiveness.

UML itself is not a single "monolithic" language but a set of different notations that are used in different contexts. For example use case diagrams can be used to capture requirements, state chart and activity diagrams are commonly used in describing dynamic aspects of software systems, class and object diagrams are used to model structural relationships between various elements, component and deployment diagrams are means to describe software architecture, etc.

Advantages of using UML are numerous. As a standardized language it improves communication between different stakeholders, and provides a mean to document software intensive systems. By offering a common language it allows tool interoperation between different vendors.

The purpose of UML spans from a language used to describe software on different levels of abstraction, to a language used in tools for software simulation. A common misapprehension is that UML is a methodology. UML is just a notation used by some methodologies like RUP [55] or Fondue [79].

In the scope of this thesis two types of UML diagrams (Class Diagrams and Object Diagrams), and OCL expressions, are of the greatest importance and will be

described in more details, in the following subsections.

2.1.1 Class Diagrams

Class diagrams are the most widely used type of UML diagrams. They can show types of objects and various relationships between them, and usually are used to capture the static structure of software. They commonly contain elements like Classes and Interfaces, and relationships between them like Dependencies, Generalizations, and Associations. For each Class it is possible to show their attributes and operations. An example of a Class diagram is shown in Fig. 2.1.
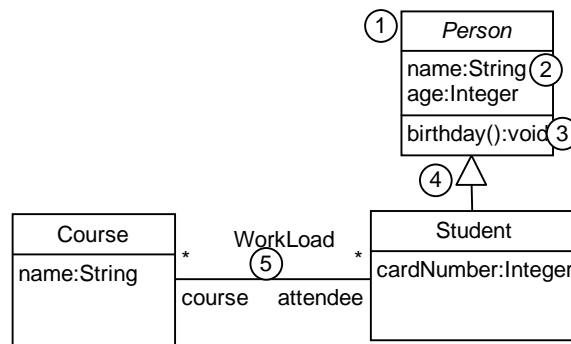
Figure 2.1: An example of a Class Diagram

The Fig. 2.1 shows three classes ① (`Course`, `Student`, and `Person`) together with attributes ② and operations ③ they contain. Moreover, the same diagram captures information about relationships between depicted model elements (An association ⑤ between `Course` and `Student`, and a generalization ④ between `Student` and `Person`).

2.1.2 Object Diagrams

Object diagrams show class instances depicted as objects ① having their slots ③ , and links ② between objects. Usually they are used to represent one system snapshot. For the example class diagram from Fig. 2.1, one possible snapshot is shown in Fig. 2.2.

It is important to stress that there is a conceptual dependency between a class diagram (as shown in Fig. 2.1) and object diagrams that represent its snapshots (see Fig. 2.2). This means that any change made on a class diagrams could make its corresponding object diagrams invalid.
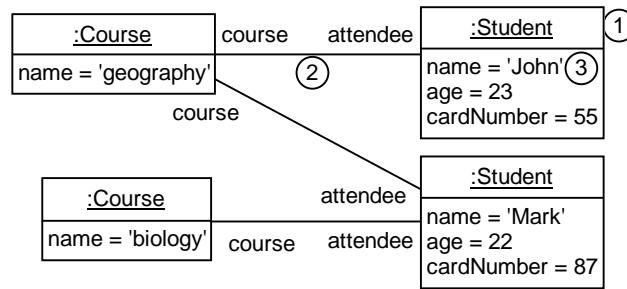
Figure 2.2: An example of an Object Diagram

2.1.3 Object Constraint Language

Object Constraint Language (OCL) (see [53, 68]) is a formal language used to add precision to UML models beyond the capabilities of graphical diagrams. In that case, OCL expressions can be used to specify additional constraints of the UML model.

Traditionally, these additional constraints are written in natural language what usually leads to ambiguous and imprecise model specifications. Many formal languages have been proposed for addressing these problems. The disadvantage of traditional formal languages in comparison with OCL is that they require strong mathematical background from anyone who wants to read or write additional constraints. This makes them often inapplicable by modelers without such a background. OCL is filling this gap by being, on one hand a formal language, interpretable by machines, and on the other hand a user friendly textual language easy to understand and write.

OCL has two main characteristics:

- OCL is a declarative language. OCL expressions do not have side effects, i.e. evaluation of OCL expressions can't alter the state of the corresponding entity.

- OCL is a typed language. Each OCL expression has a type and in order to be well formed it must conform to the type conformance of the language.

In the context of class diagrams, OCL has two common uses: 1) definition of invariants on class model, and 2) definition of pre/post-conditions for operations.

For the example Class diagram from Fig. 2.1 imagine that we want to express that card number of all students must always be a positive number. This constraint can be easily expressed using the following OCL invariant:

```
context Student inv:
    self.cardNumber>0
```

Besides attributes of a class, OCL invariants can constrain possible associations between classes. For instance, expressing that each Student has taken at most 5 courses can be expressed as:

```
context Student inv:
    self.course−>size()<6
```

OCL constraints can be used for specifying pre and post-conditions for operations.

```
context Student:birthday()
    pre: self.age>0
    post:age=age@pre+1
```

OCL allows definition of "helper" variables and operations that can be reused over multiple OCL expressions. One such helper variable can express that minimal number of courses is 2 as:

```
context Student def:
    minCourseNum: Integer=2
```

Helper variables defined this way, can be used in other OCL expressions, like in:

```
context Student inv:
    self.course−>size()>=minCourseNum
```

As can be seen from the OCL examples provided in this subsection, all OCL expressions are highly dependent on the underlying class diagram and that any change on the class diagram can make annotated OCL expressions syntactically incorrect.

2.2 Refactorings

Refactorings (see [60] for an overview) represent structured changes introduced in software artifacts in order to improve its readability, extensibility, and maintainability. The main characteristic of refactorings is that they represent small, atomic changes used to improve programming code, architecture and design models. Each refactoring is described by one refactoring rule.

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 depend on parts 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.

Examples and catalogs of refactorings can be found for various programming languages, and notations. Despite the vast application domain of refactoring techniques, one distinguishable property of refactoring is semantic preservation. Refactorings, regardless the artifact they are applied on, must preserve semantics of the artifact.
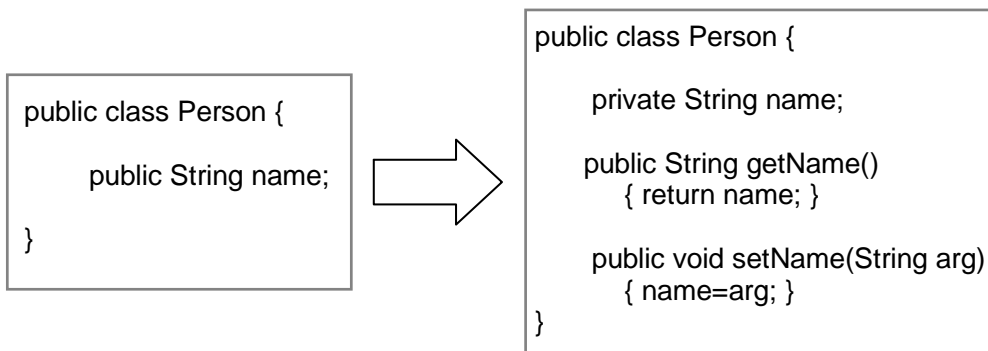
Figure 2.3: Java code refactoring example

The refactoring named "encapsulate field", applied to Java, changes modifier of one attribute from "public" to "private" and adds two methods for accessing and modifying the attribute. An example for application of this refactoring is shown in Fig. 2.3. Part of the refactoring that is not shown in the figure is that all references to the modified attribute must be updated to refer to the two newly created methods. After application of this refactoring, the refactored program "behaves" on the same way as before refactoring.

The application of refactoring rules, called *refactoring steps*, is most often pattern-driven. A design that is an instance of Design Patterns [45] can usually be extended and maintained much better than a design that is less structured. Pattern-driven refactoring steps have been thoroughly studied in [51, 62].

2.3   Metamodeling

In order to be efficiently used, every modeling language must at least have precisely defined syntax. One way to define syntax of modeling languages is by applying metamodeling. Metamodeling is well defined technique for specification of modeling languages (their abstract syntax). It is heavily used in the context of UML, OCL, and related concepts.

Metamodeling is a technique for describing abstract syntax of modeling languages by models. This technique is in the core of Meta Object Facility (MOF) [67] 4-layer architecture shown in Fig. 2.4

LEVEL	ARTIFACT	ROLE
M3	MOF	Meta-metamodel
M2	UML-MM	Metamodel
M1	Class Diagram Object Diagram	Model
M0	Object Diagram	Running System
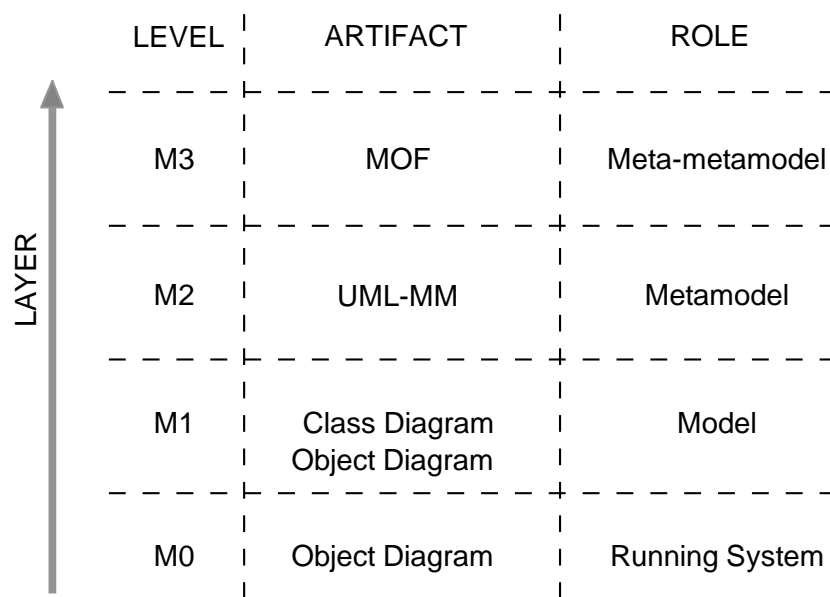
Figure 2.4: 4-layer architecture

Each layer from the Fig. 2.4 can be seen as an instance of the first upper layer in the layer hierarchy, i.e. UML metamodel is instance of MOF, UML class diagram is instance of UML metamodel, etc. This means that it must conform to the "rules" expressed by the model on the higher level.

In Sect. 2.4 we show parts of official UML and OCL metamodels that describe class diagrams, object diagrams, and OCL expressions.

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 transformation rules presented in this thesis.

2.4.1   Declaration of Metaclasses

Figures 2.5, 2.6, 2.7, and 2.8 show relevant parts of the official metamodel for UML 1.5 (for a complete definition see [64]). The chosen fragment of the UML-part of the metamodel concentrates on the main concepts of class and object diagrams.

Figures 2.5 and 2.6 define class diagrams. Each *Classifier* (subclassed by *Class*) can contain *StructuralFeatures* (like *Attributes*) or *BehavioralFeatures* (like *Operations*). At the same time as being a *Namespace*, each *Classifier* is a *GeneralizableElement*, meaning that can be connected to other *Classifiers* by *Generalizations*.

Each *Association* can have two or more *AssociationEnds*, each one connected with one *Classifier*.
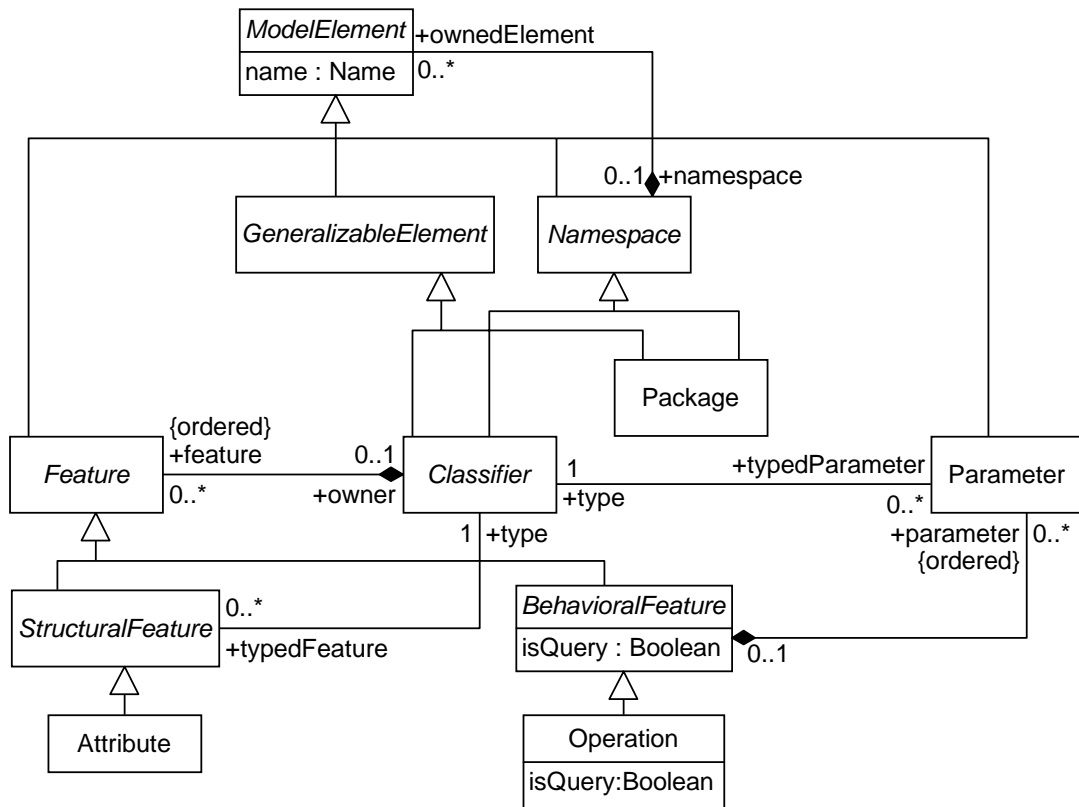
Figure 2.5: UML - Backbone

Figures 2.7 and 2.8 show a "language" for abstract syntax of UML object diagrams.

The OCL-part shown in figures 2.9, 2.10, 2.11, and 2.12 covers abstract syntax for the most important OCL expressions (a complete definition of OCL 2.0 can be found in [68]). The gray background is used to denote classes that belong the the OCL metamodel.
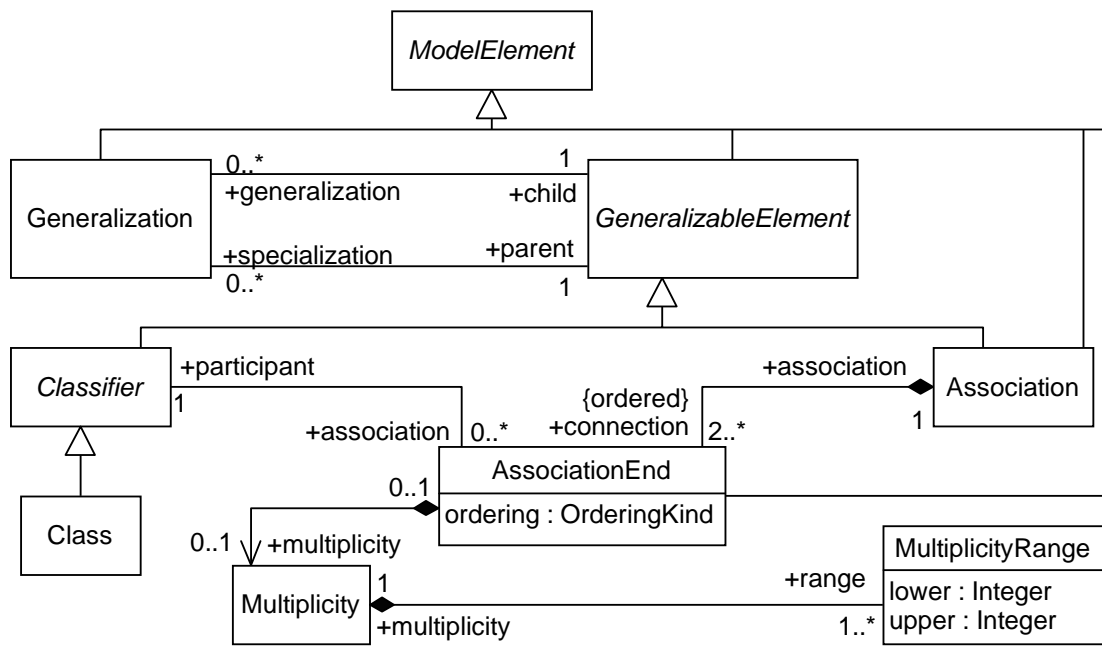
Figure 2.6: UML - Relationships

2.4.2 Well-formedness Rules

The metamodel for UML and OCL contains hundreds of well-formedness rules. 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 approach and the readability of model transformation rules, which grow when more well-formedness rules have to be preserved.

Since the UML/OCL refactorings considered in this thesis 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:

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


context Classifier def:
    allUsedNames():Bag(String)=
```
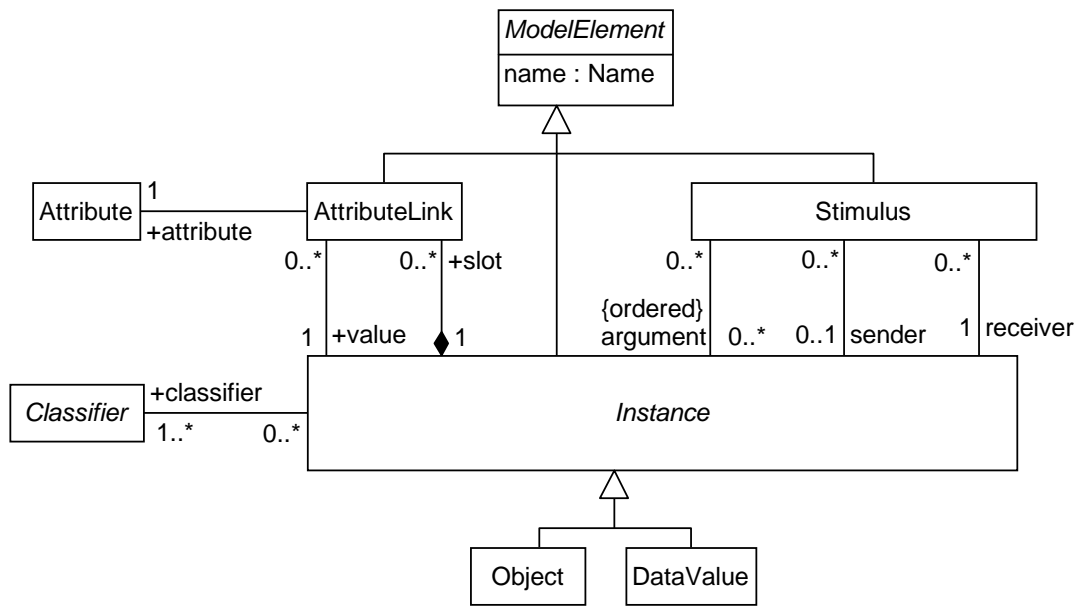
Figure 2.7: UML - Instances

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

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 [64, 68].

A second important well-formedness rule in the metamodel of UML 1.5 is that
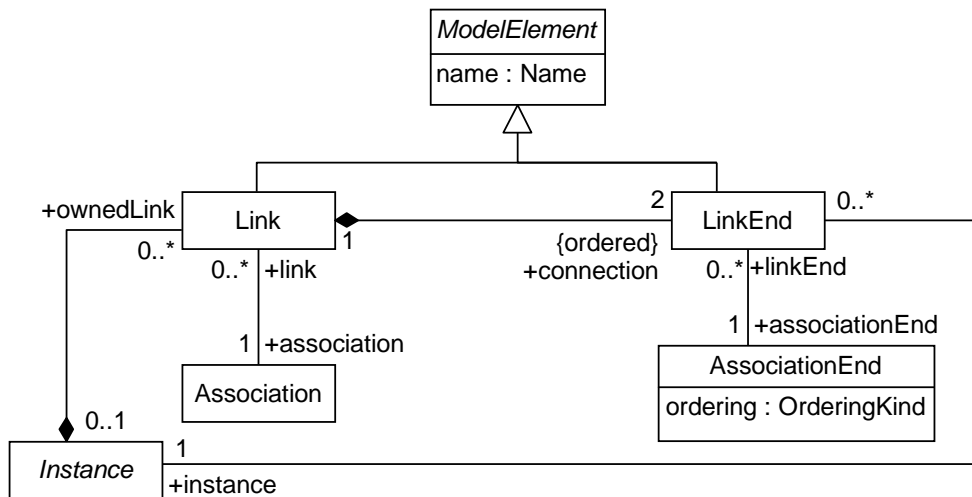
Figure 2.8: UML - Links
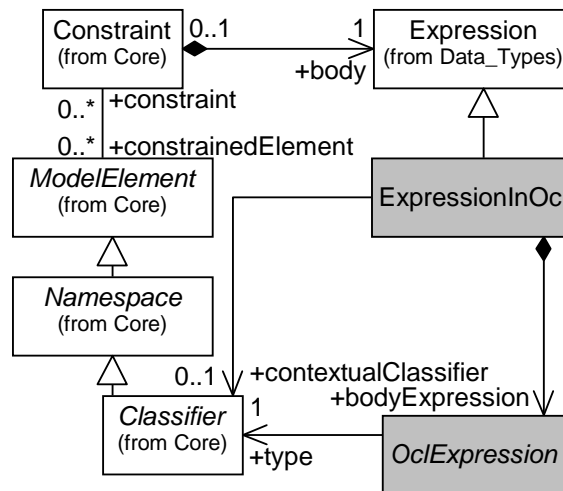
Figure 2.9: OCL - Expressions

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)
```

2.5   Model Transformations and QVT

Model transformations are widely recognized as the *heart and soul* of Model Driven
Architecture [77]. Refactoring rules can be seen as a special form of model trans-
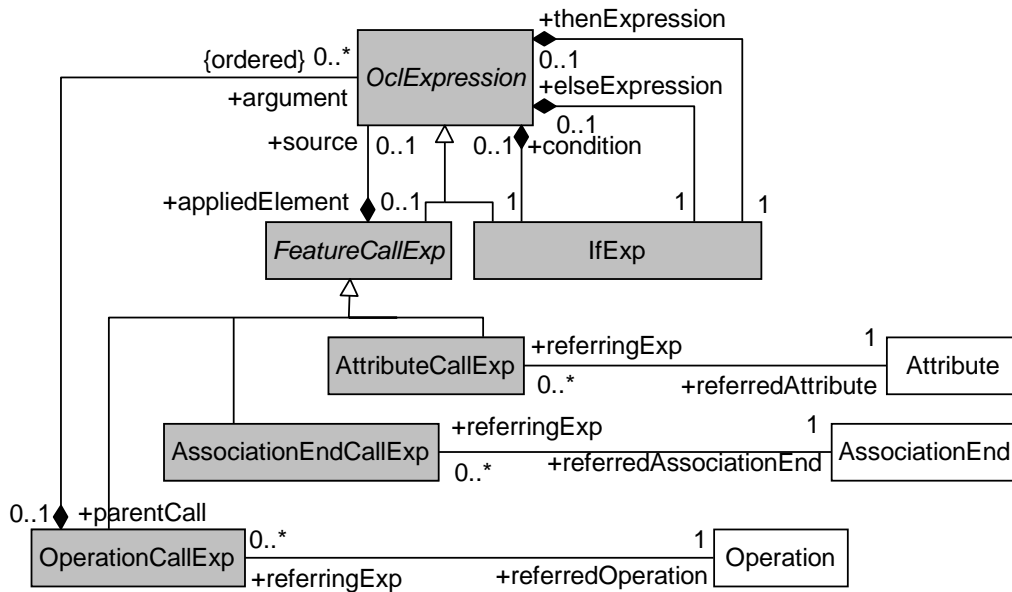
Figure 2.10: OCL - Navigation and If Expressions
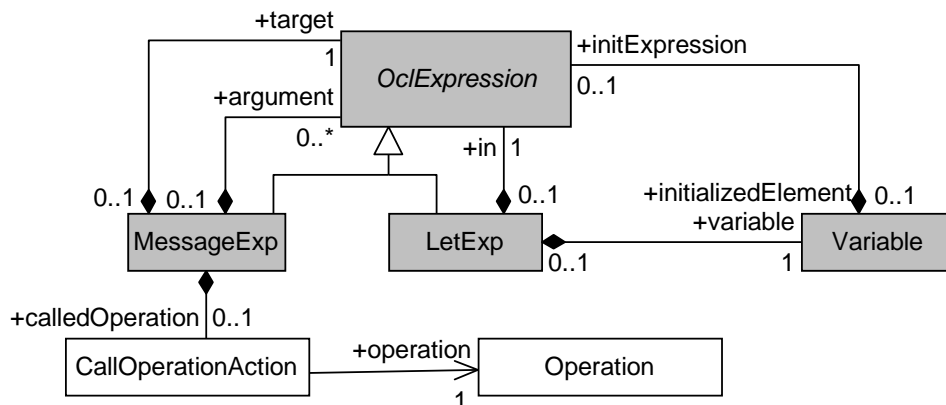
Figure 2.11: OCL - Message and Let Expressions

formations for which the source and the target model are expressed using the same language. In this thesis, we describe refactoring rules in a graphical formalism inspired by the QVT [66].

A model transformation is defined as a set of *transformation rules*. A transformation rule consists of two patterns, LHS (left hand side) and RHS (right hand side), which are connected with the symbol ≪⟩≫. Optionally, a rule can have parameters and a when-clause containing a constraint written in imperative 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 for each occurrence – during the application of the
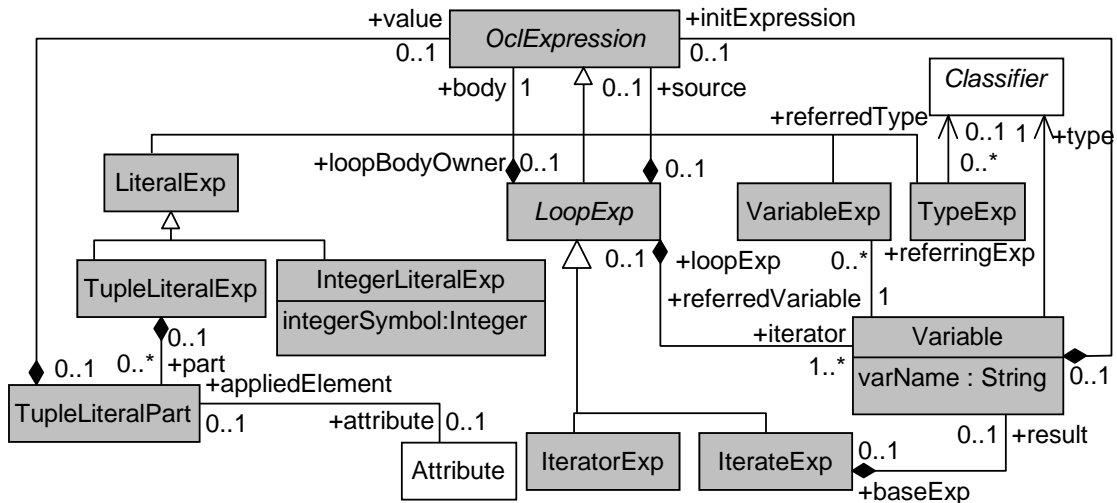
Figure 2.12: OCL - Literal, Loop, Variable, and Type Expressions

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). 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 both by LHS pattern, and 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' (i.e. the links that were connected to the deleted object). 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.

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.

2.5.1 How to Write Syntax Preserving QVT Rules

The purpose of this subsection is twofold. Firstly, the section should illustrate on concrete examples the 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. 3.2 for the refactoring of UML/OCL models.

Example: Item-View World

In order to explain QVT's basic concepts, we start with refactoring rules for a tiny *Item-View* language, Fig. 2.13 shows its metamodel.
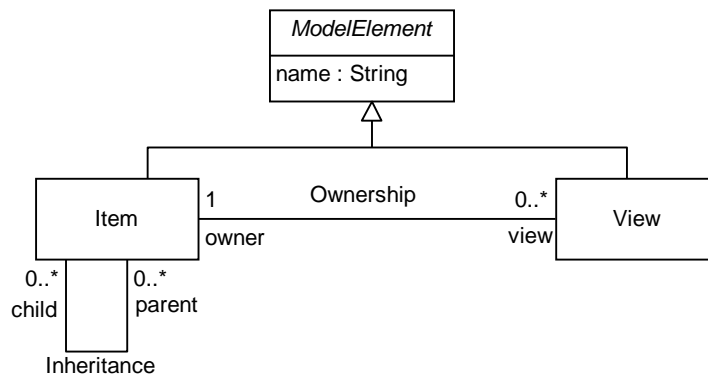
Figure 2.13: Metamodel for simple *Item-View* language

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)
```

```
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 [6] for a detailed justification).

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. 2.14. 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) Item selected by user   (b) Item identified by name
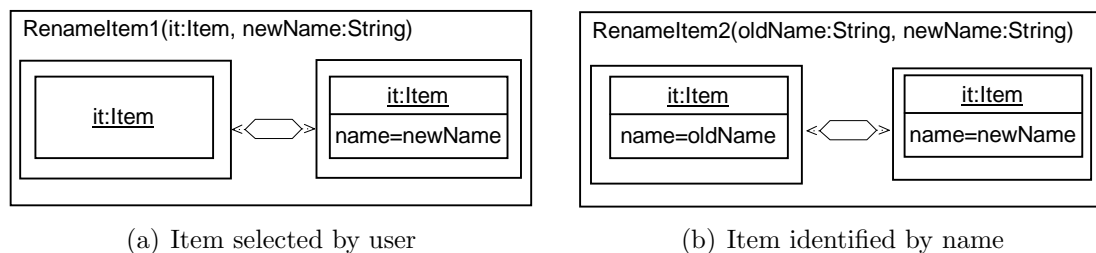
Figure 2.14: Two versions of *RenameItem* refactoring rule

A second version of the *Rename*-refactoring is formalized by rule *RenameItem2* shown in the right part of Fig. 2.14. 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.

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 cor-

rect models are exactly the valid instances of the metamodel, what boils down to the following three criteria: (1) all model elements 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 be 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* can be given as:

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.

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, an interesting approach to automate this filtering has been developed by Cabot [25, 26]. 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. 3.3 on one example, how such an argumentation can be formalized by using the KeY-system.

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*. Often, a refactoring rule can (potentially) lead to a model that does not satisfy 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. 2.15.
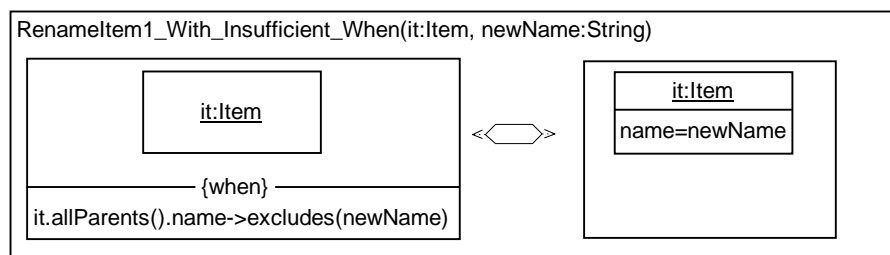
Figure 2.15: Renaming of selected item – when-clause is not sufficient to preserve *UniqueNameInInheritance*

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, *Unique-NameInInheritance* 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.

```
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. 2.16. 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.

2.5.2   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))
```
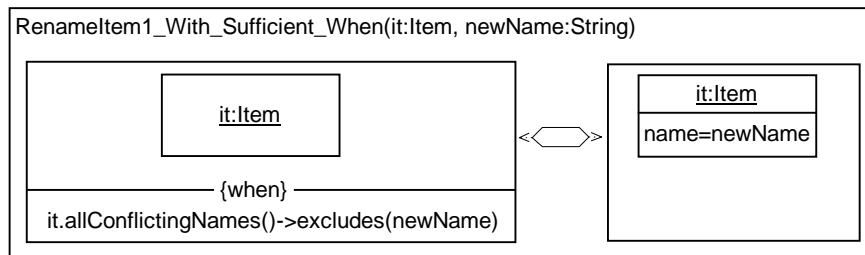
Figure 2.16: Renaming of selected item – correct version for *UniqueNameInInheritance*

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. 2.16 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.
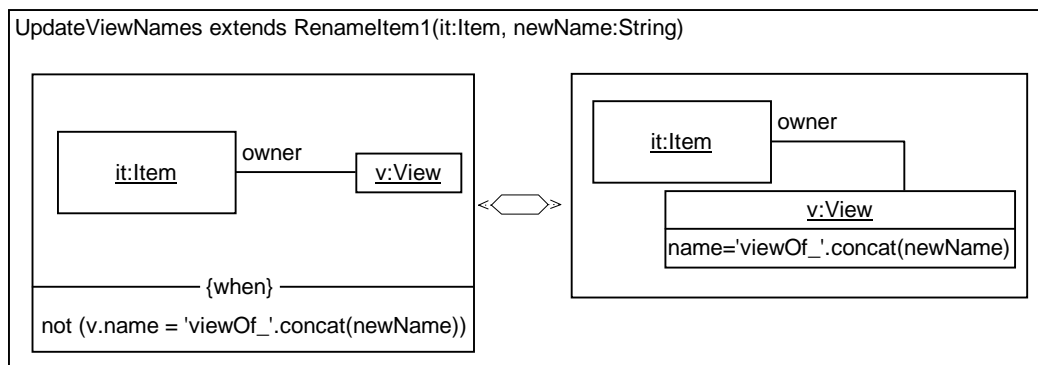
Figure 2.17: Extension of *RenameItem1*

Figure 2.17 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` ob-

ject 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*, that prevents rule's application, ensures the termination of the rule application.

Chapter 3

Refactoring UML/OCL Diagrams

This chapter 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. We finally discuss for our refactoring rules the problem of syntax preservation and show, by using the KeY-system, how this can be enforced.

Work presented in this chapter was firstly published on MoDELS 2005 conference [56], and then an extended version was published in Software and Systems Modeling (SoSym) journal [58].

3.1 Introduction

Existing techniques for refactoring UML class diagrams neglect artifacts that are dependent on class diagrams and that become invalid once the corresponding class diagram is refactored. Examples of such artifacts are OCL constraints and UML object diagrams. In this chapter we formalize UML class diagrams refactorings and show how attached OCL constraints have to be altered in order to preserve their syntax (semantics preservation of OCL constraints and necessary changes of object diagrams are discussed separately in Chapter 5)

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. However, we have formally proved the syntax preservation property only for the rule *ExtractClass* (see Sect. 3.3).

UML class diagrams and OCL constraints can be seen as instances of corre-

sponding metamodels. In this chapter we formalize UML/OCL model refactoring
rules by specifying QVT transformations on the UML and OCL metamodels.

The chapter is organized as follows. In Sect. 3.2 we present a catalog of UML/OCL
refactoring rules. In Sect. 3.3 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 3.4 contains an overview of related work. Section 3.5
concludes the chapter.

3.2   A Catalog of UML/OCL Refactoring Rules

In this section, we present some of the most important refactoring rules already
provided for Java language in [42], translated to UML 1.5 class diagrams. These
rules handle OCL 2.0 constraints that are attached to the refactored class diagram
(handling UML object diagrams is explained in Chapter 5). In Subsection 3.2.3,
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 syntax preserving only with
respect to the part of the UML 1.5 metamodel given in the Section 2.4. 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 Fig. 3.1 for an overview) is inspired by the refactoring rules for
the static structure of Java programs given by Fowler in [42]. 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 [42]
also relevant fields are moved).

Not all class diagram refactoring rules have an influence on attached OCL con-
straints. Fig. 3.1 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.

3.2.1   Rules Without Influence on OCL

RenameClass/Attribute/Operation/AssociationEnd

These rules are very similar to each other and only *RenameAttribute* (see Fig. 3.2) is
discussed here in detail. The *Rename*-rules differ mostly in the when-clause, whose

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**	No	No
*PushDownOperation**	No	No
*PushDownAssociationEnd**	No	No
ExtractClass	No	No
ExtractSuperclass	No	No
MoveAttribute	Yes	Yes
MoveOperation	Yes	Yes
MoveAssociationEnd	Yes	Yes

* only *push down* to one subclass is considered in this thesis

Figure 3.1: Overview of UML/OCL refactoring rules

purpose is to check whether the proposed new name is already in use in the enclosing *Namespace* of the renamed element.

In rule *RenameAttribute* (see Fig. 3.2), 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.
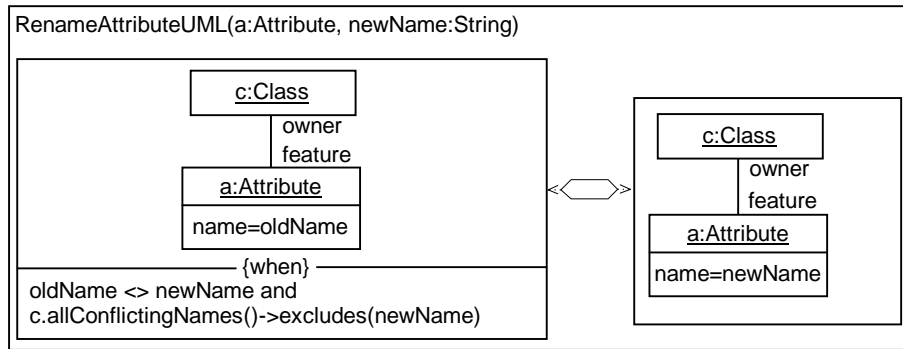
Figure 3.2: Formalization of *RenameAttribute* refactoring
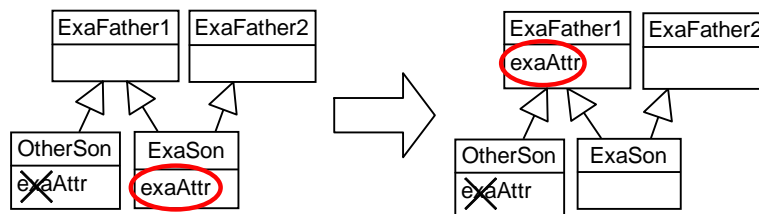
Figure 3.3: Example of applying *PullUpAttribute*

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. 3.3.

The LHS of the rule (Fig. 3.4) 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 refac-
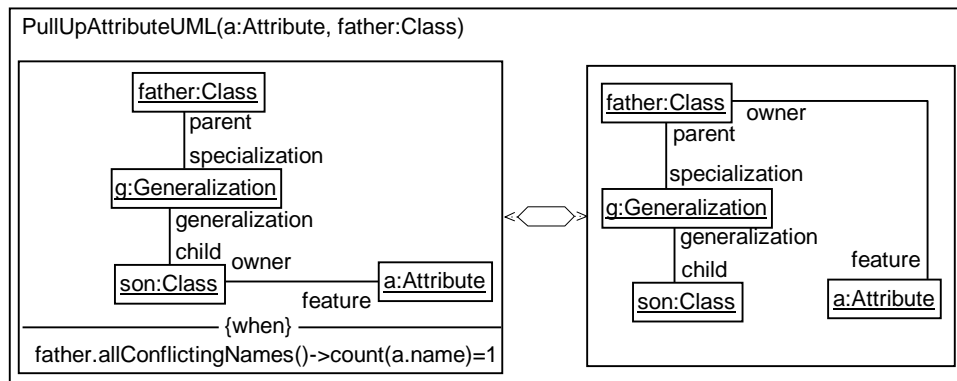
Figure 3.4: *PullUpAttribute* refactoring rule
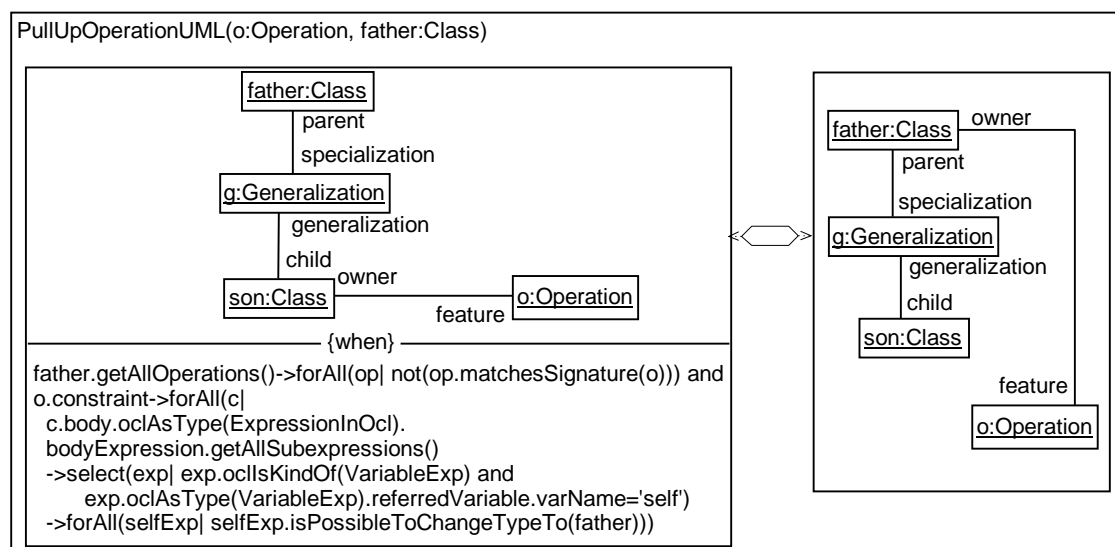
Figure 3.5: *PullUpOperation* refactoring rule

toring 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`. 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. 3.5 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 oper-

ation `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 [88]. *getAllSubexpressions()* results in a collection containing all subexpressions of one OCL expression. *isPossibleToChangeTypeTo()* checks if a *self* expression can change its type depending on navigation expressions whose source is the *self* expression (like *self*.`attSon`).

Fowler has faced in [42] 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.

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. 3.6 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*).

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 (*self* expressions and *exp.element* expressions) and that the when-clause had to make provision

---

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

[2]We consider here only rules that push a model element down to exactly one subclass.
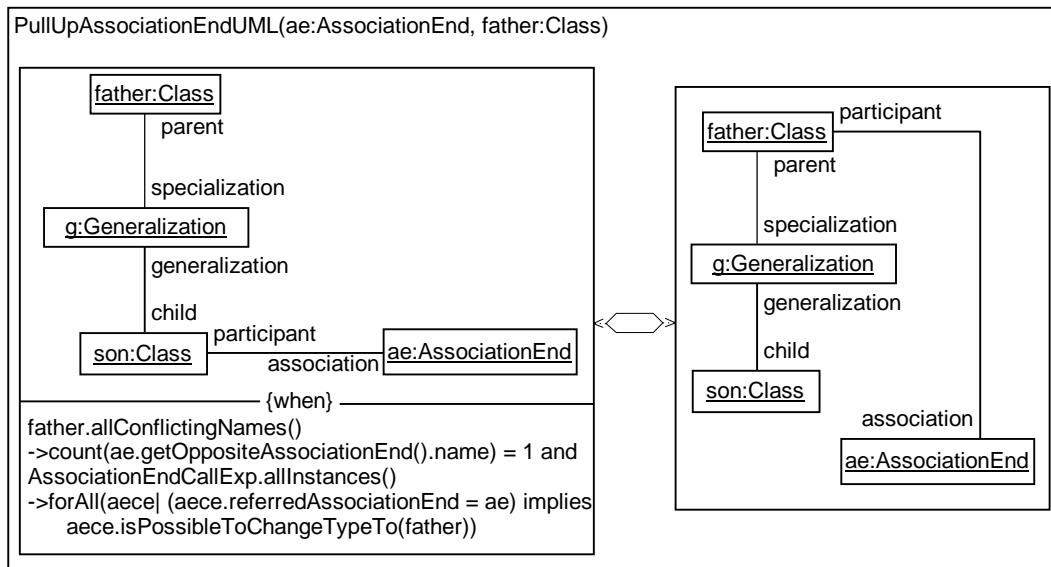
Figure 3.6: *PullUpAssociationEnd* refactoring rule

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. 3.7). As described by Fowler in [42] 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.
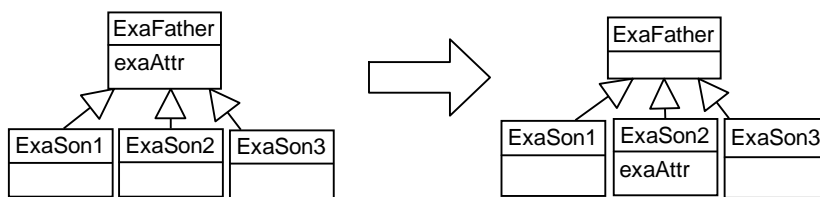
Figure 3.7: Example of applying *PushDownAttribute*

The formalization of *PushDownAttribute* is given in Fig. 3.8. The when-clause has to check possible name conflicts 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. Moreover, the when-clause checks for non-existence of constraints attached to the attribute a whose contextual classifier is father.

The when-clause of rule *PushDownOperation* (see Fig. 3.9) checks for query
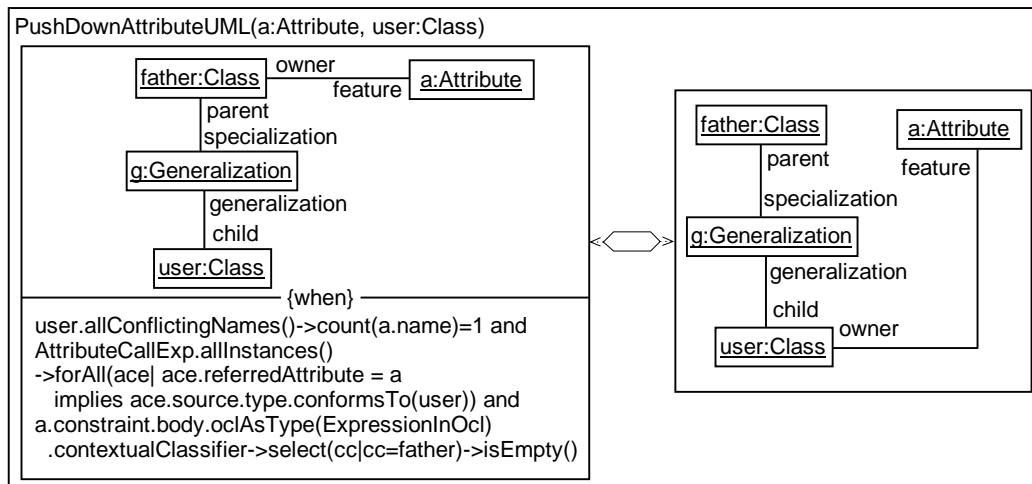
Figure 3.8: *PushDownAttribute* refactoring rule
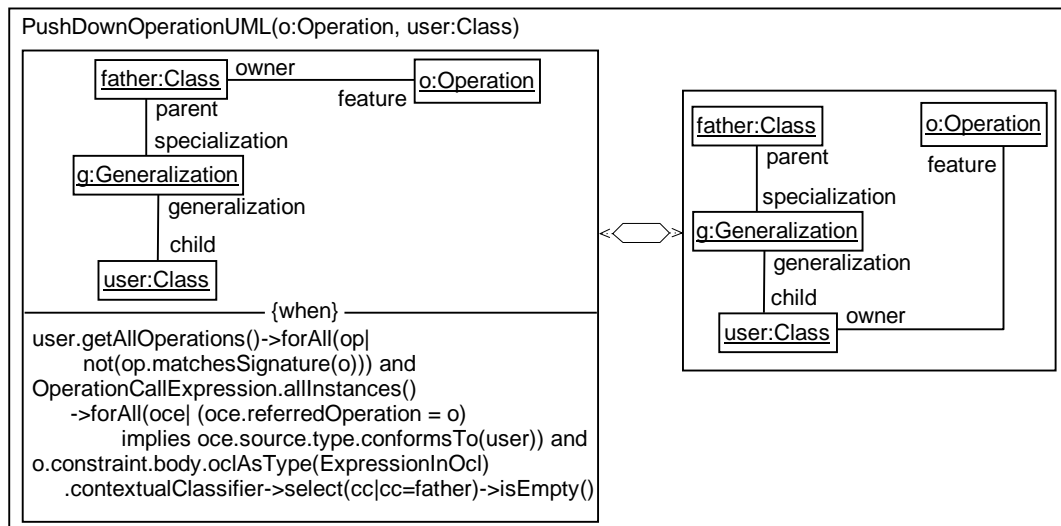
Figure 3.9: *PushDownOperation* refactoring rule

expressions with operation **o** but not for *self*-expressions (note that rule *PullUpOp-eration* checks the opposite).

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

ExtractClass/Superclass

The rule *ExtractClass* (see Fig. 3.10) creates an empty class `extracted` in the same namespace `nsp` as the selected class `src` 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
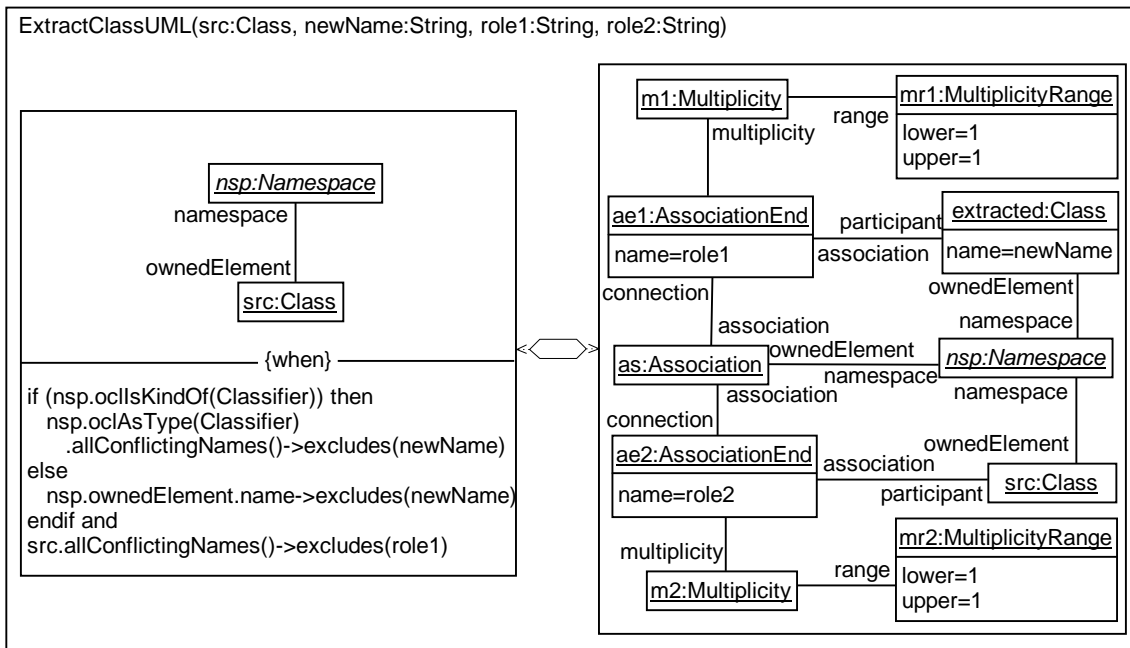
Figure 3.10: *ExtractClass* refactoring rule

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. 2.5). 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. 3.11) 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 [42]. 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 correct-
ness of attached OCL constraints because both rules merely introduce new model
elements and do not delete or change old ones.
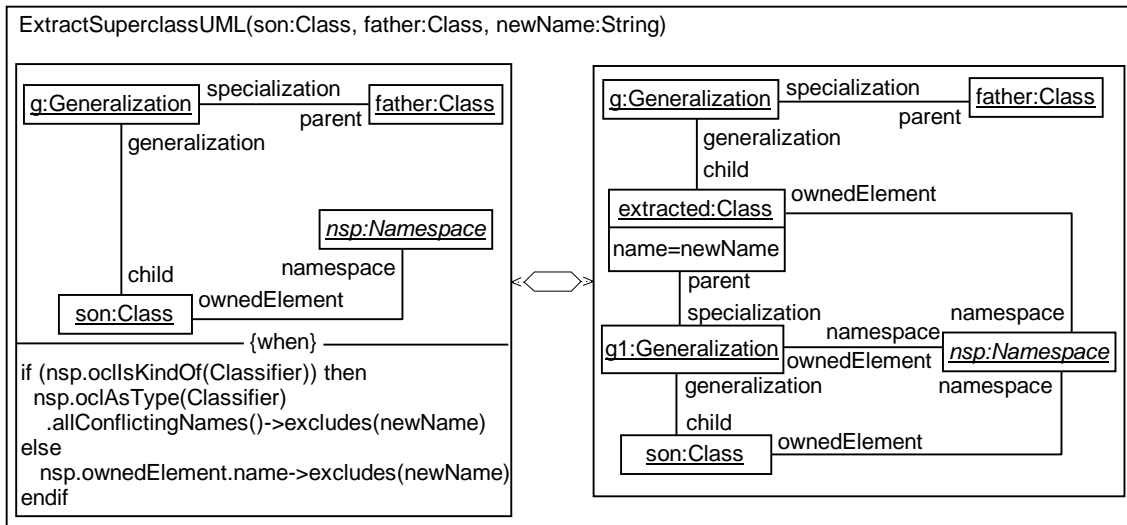
Figure 3.11: *ExtractSuperclass* refactoring rule

3.2.2   Rules With Influence on OCL

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. 3.12.
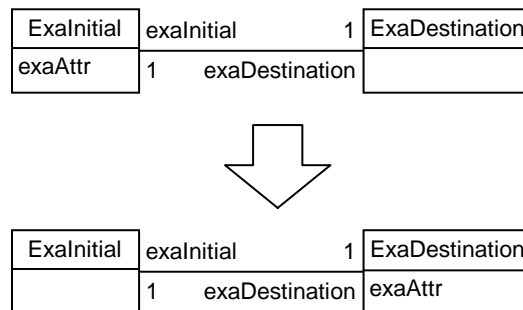
Figure 3.12: Example of applying *MoveAttribute*

The selected attribute is moved from a source to a destination class over an association with multiplicity 1 on both ends (cases of moving attribute over associations with different multiplicities are discussed in Chapter 5). 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. 3.13 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 (the preservation of semantics
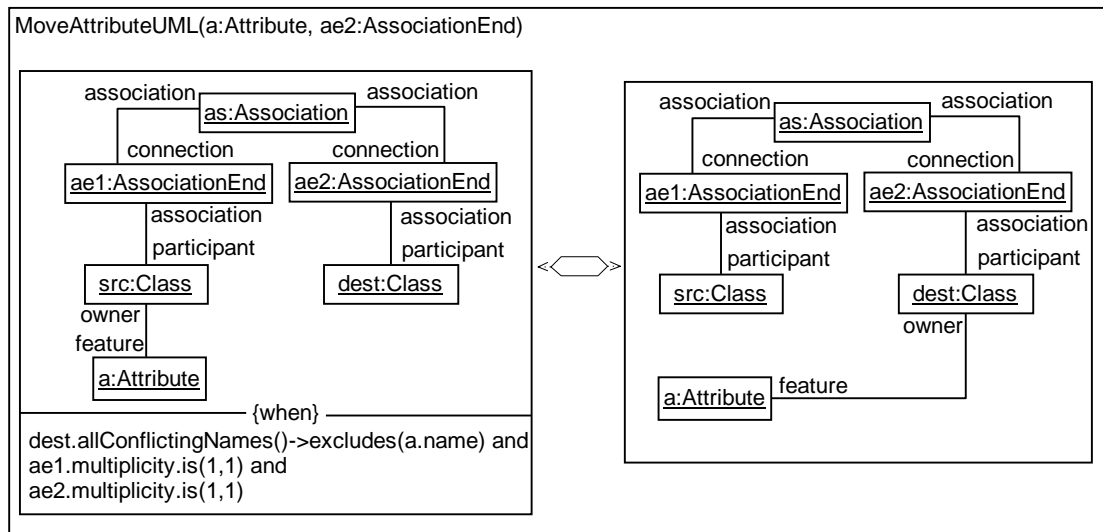
Figure 3.13: UML part of *MoveAttribute* rule

will be discussed in Chapter 5). However, the multiplicity restriction is sufficient but not necessary for semantics preservation of a rule application. The semantics preservation property is analyzed and proven for some variations of *MoveAttribute* refactoring in Chapter 5.

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. 3.12, 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. 3.14), 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. 3.15. As for *MoveAttribute*, the association connecting source and destination class must have on both ends multiplicity 1. The 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
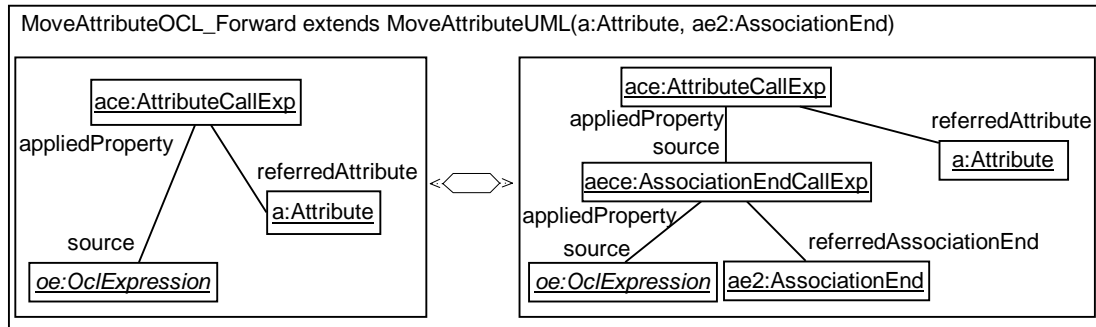
Figure 3.14: OCL part of *MoveAttribute* rule (forward navigation)
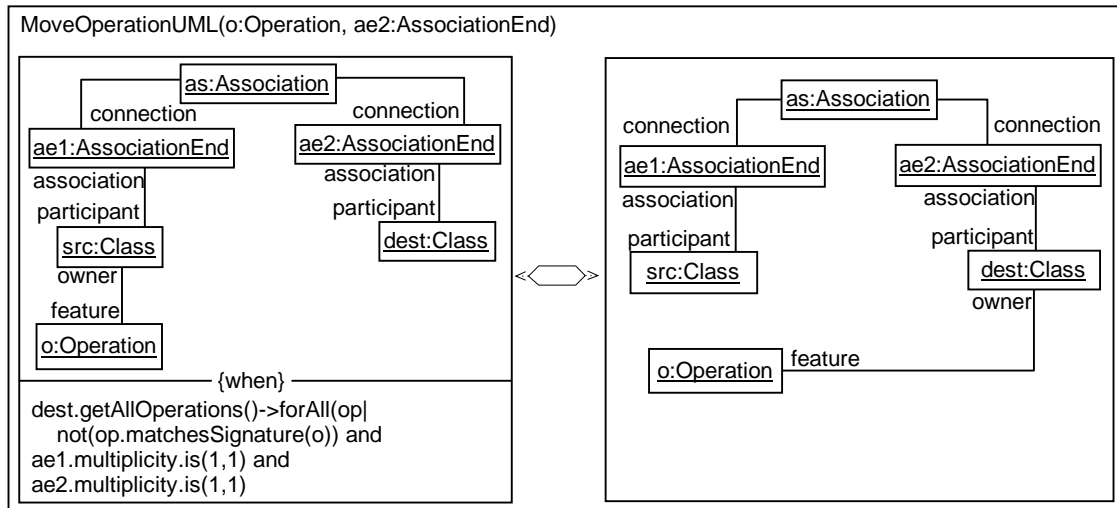
Figure 3.15: UML part of *MoveOperation* rule

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 [42] 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. 3.15.

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. 3.16). 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. 3.17, 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 [35].

For the backward navigation step, Fowler says: "*… create or use a reference from the target class to the source*".
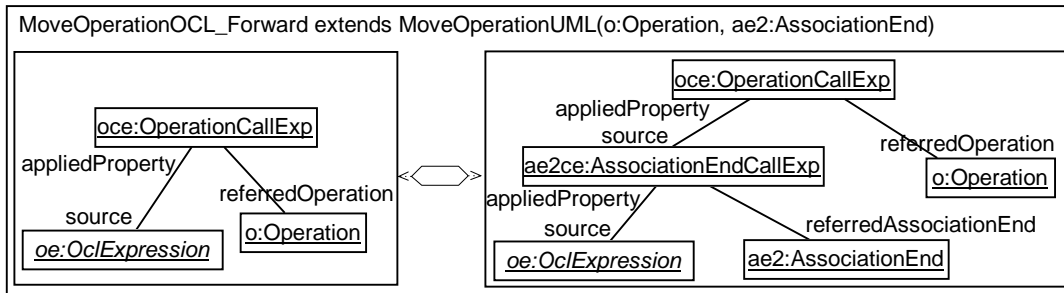
Figure 3.16: OCL part 1 of *MoveOperation* rule (forward navigation/handle query)
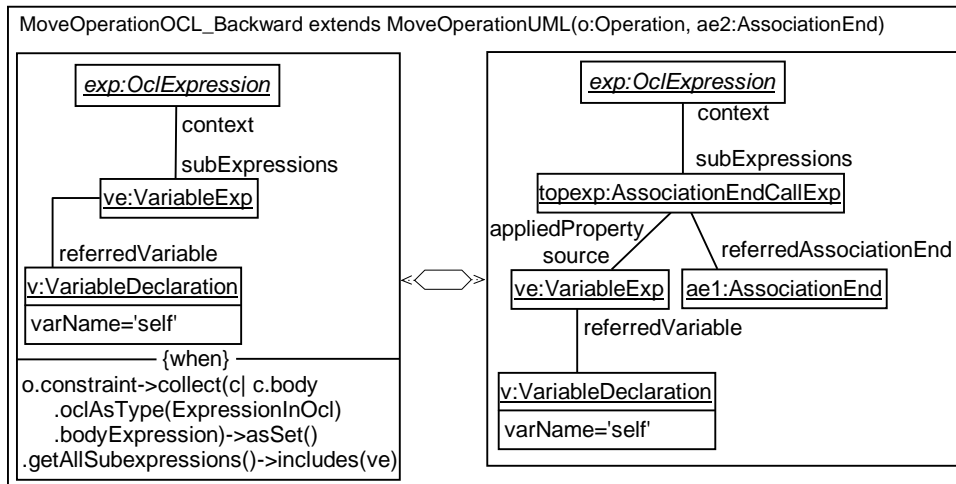
Figure 3.17: OCL part 2 of *MoveOperation* rule (backward navigation)

The rule *MoveAssociationEnd* is very similar to *MoveAttribute* for the refactoring of the UML part; Fig. 3.18 shows an example and Fig. 3.19 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 3.20 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. 3.18.

The backward navigation has to address the problem that the type of expression *exp*.roleA has changed – depending 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.
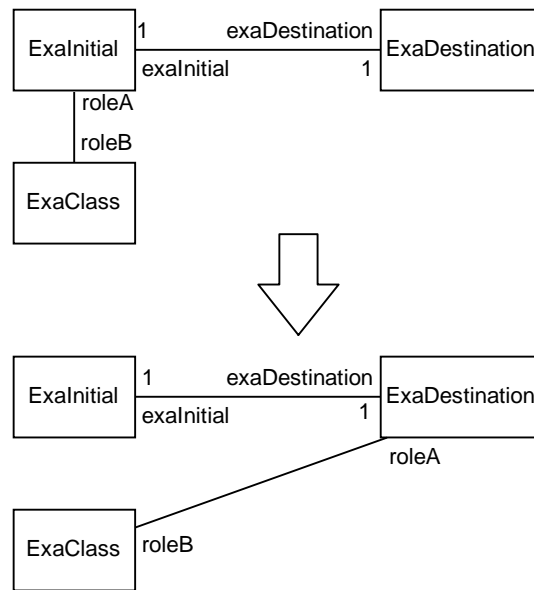
Figure 3.18: Example of applying *MoveAssociationEnd*

The two rules shown in Fig. 3.21 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*.`roleA.exaInitial`. 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*.

3.2.3   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

---

[3]This exceptional case is not reflected in the rule shown in Fig. 3.21 but has been implemented in our tool (see [88]).
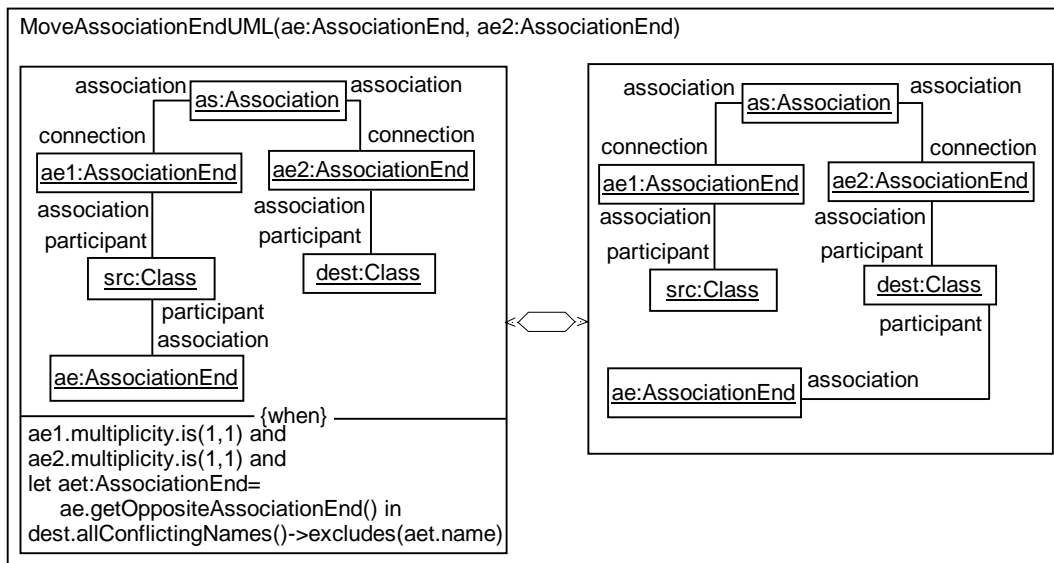
Figure 3.19: UML part of *MoveAssociationEnd* rule
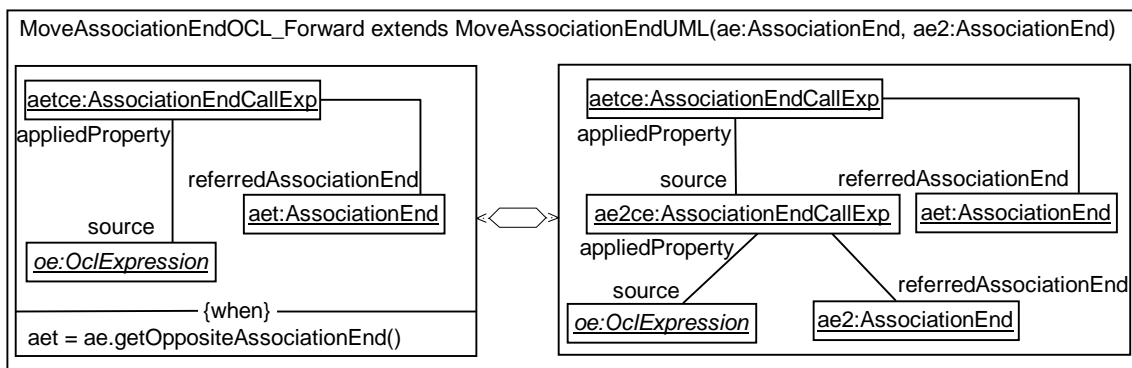
Figure 3.20: OCL part 1 of *MoveAssociationEnd* rule (forward navigation)

class diagrams: the unification of attributes and opposite association ends.

Figure 3.22 shows the relevant part of the UML 2.0 metamodel [65] and the aligned OCL 2.0 metamodel [68] (*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).
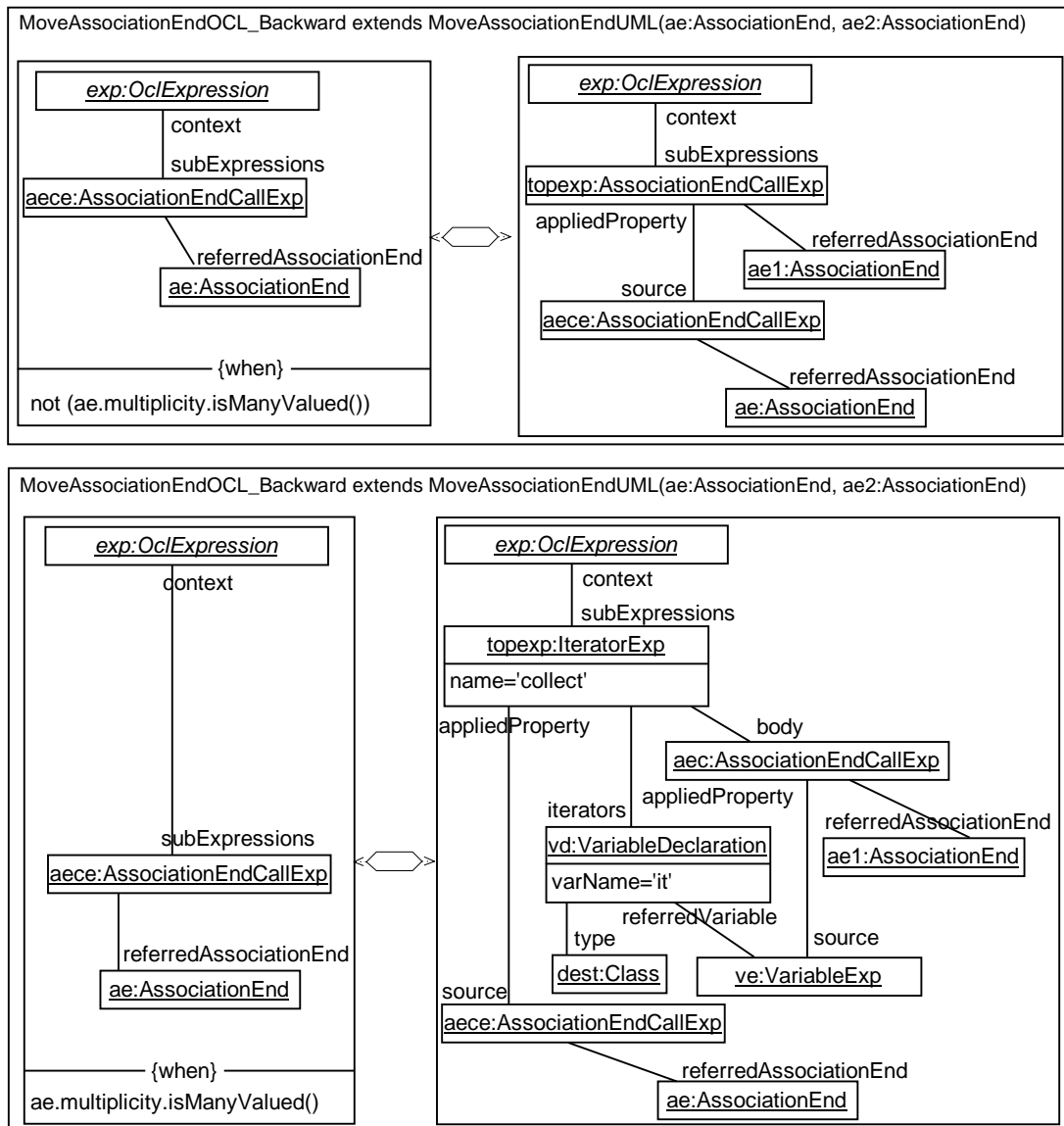
Figure 3.21: OCL part 2 of *MoveAssociationEnd* rule (backward navigation)

MoveProperty

The rule *MoveProperty* is the counterpart of *MoveAttribute* and *MoveAssociationnEnd* 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. 3.23 is very similar to the UML part of *MoveAttribute* and *MoveAssociationEnd* shown in Fig. 3.13 and Fig. 3.19, 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
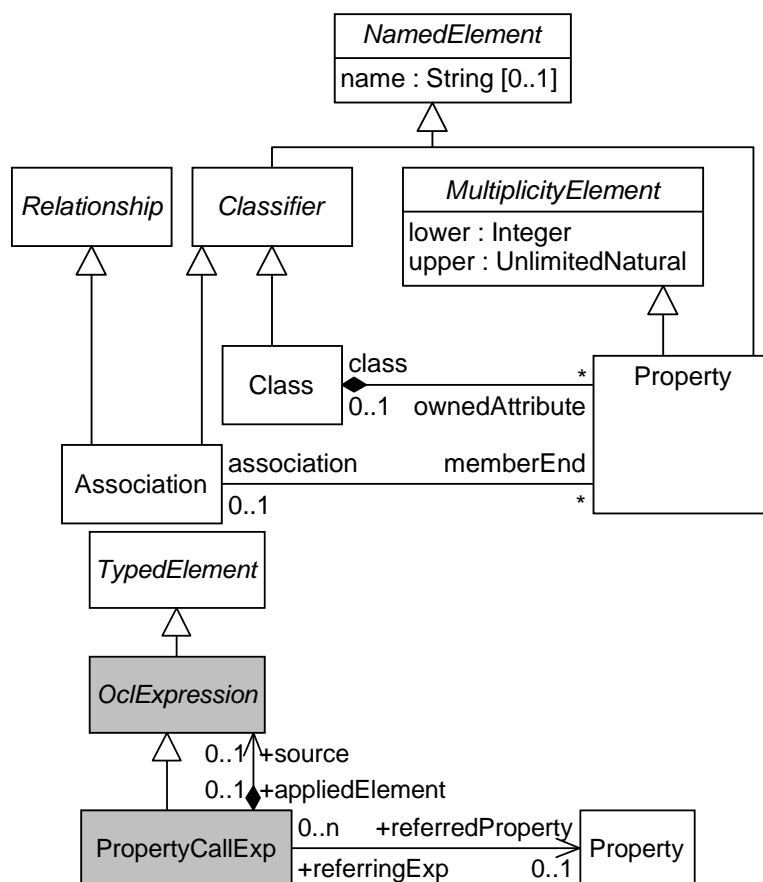
Figure 3.22: Relevant Part of UML2.0 and OCL2.0 for *MoveProperty*

moved property is not in conflict with the destination class.

In Fig. 3.24 and Fig. 3.25, the necessary forward and backward navigation changes on the attached OCL constraints are formalized analogously to *MoveAssociationEnd*.

3.3   Proving the Preservation of Well-formedness Rules

It is not difficult to argue informally that the version of *ExtractClassUML* presented here (Fig. 3.10) 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 *Extract-ClassUML* 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
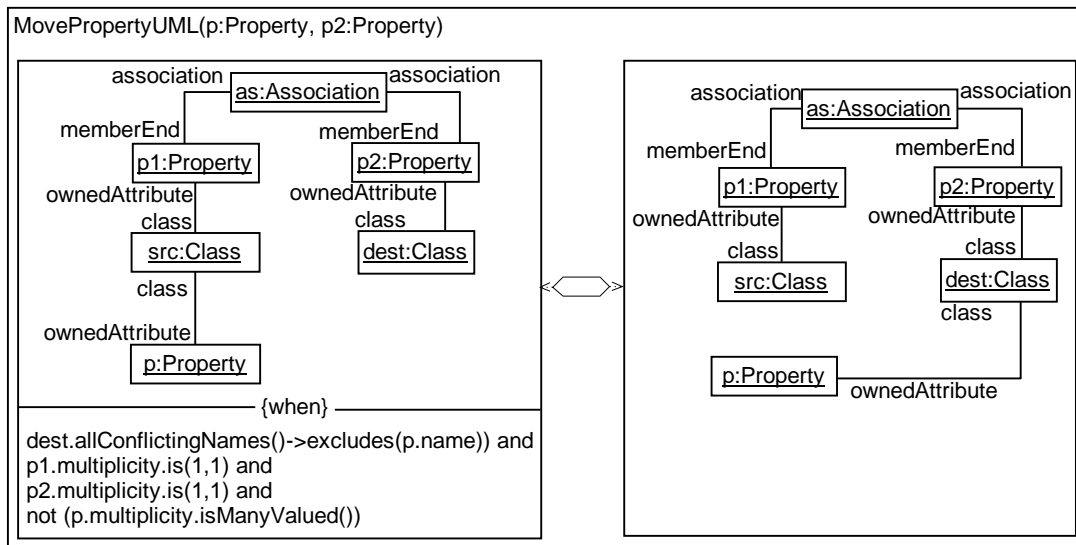
Figure 3.23: UML part of *MoveProperty* rule
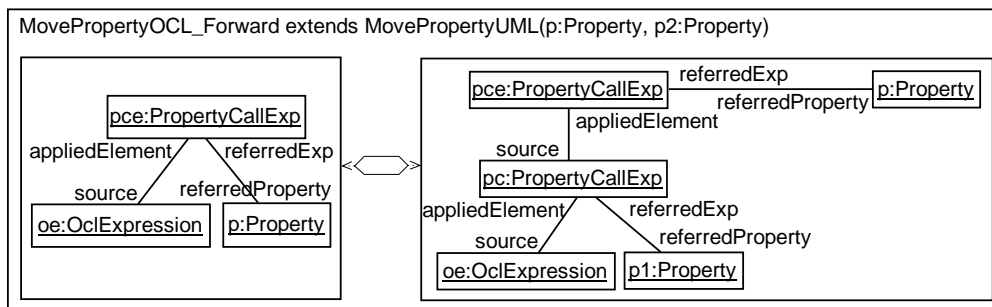
Figure 3.24: OCL part 1 of *MoveProperty* rule (forward navigation)

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.
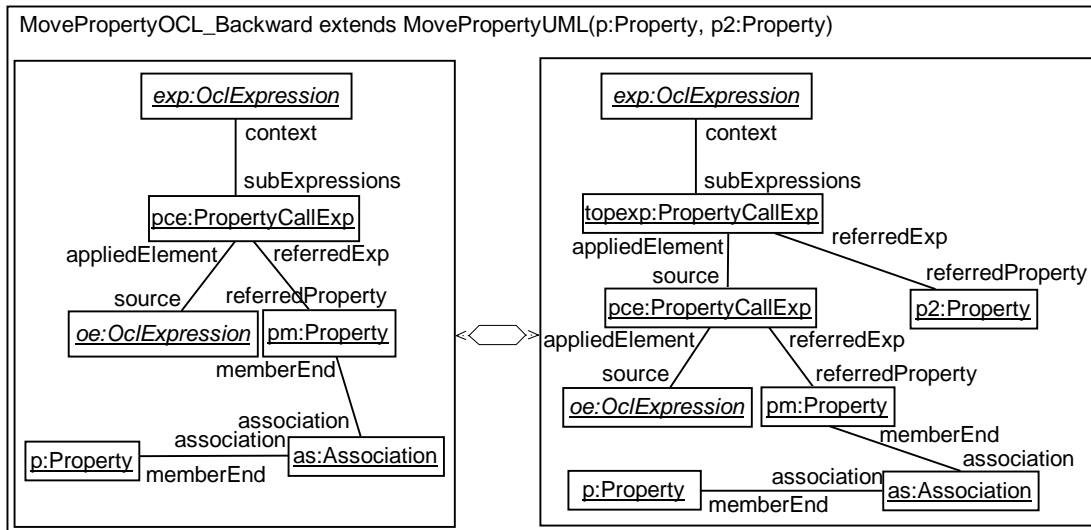
Figure 3.25: OCL part 2 of *MoveProperty* rule (backward navigation)

We show in the rest of the section, how the KeY-system[4], 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. A.3. 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 [13] 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 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. Fig. 3.26 shows how the relevant part of the metamodel is encoded; the metaclasses *Class*, *Association*, etc. became ordinary Java classes.

---

[4]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.
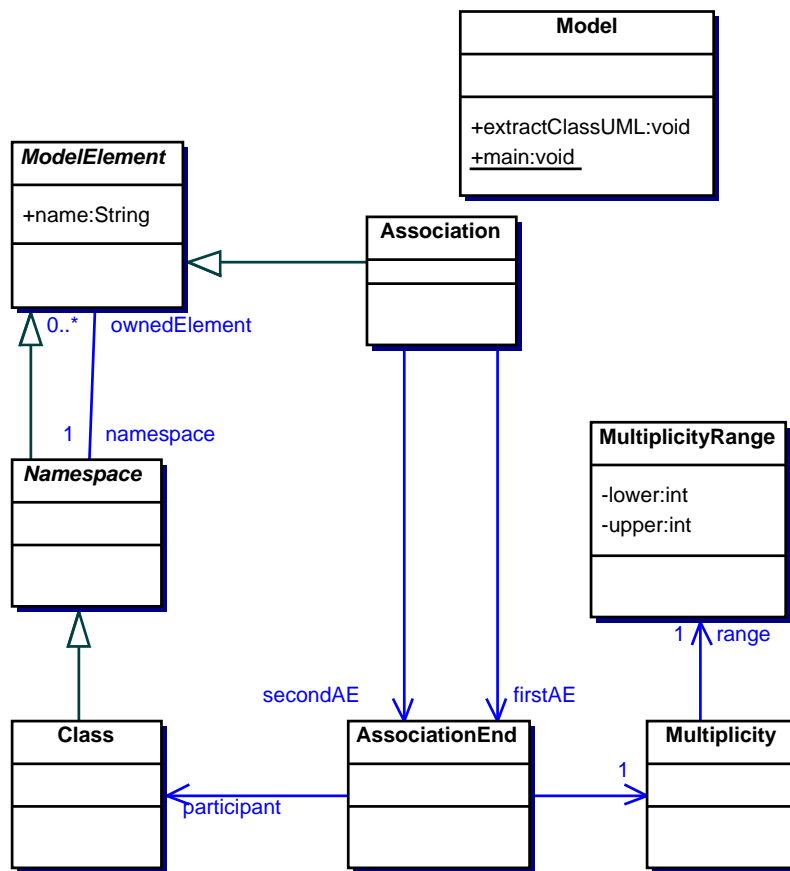
Figure 3.26: Simulation of the UML/OCL metamodel by Java classes

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.

3.4 Related Work

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 [42] 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 and EMF models are presented in [4, 76, 81, 16, 15]. However, neither these catalogs nor any of the existing UML refactoring tools [17, 20, 70] support – apart from some simple *Rename*-refactorings – the refactoring of attached OCL constraints once the underlying UML class diagram has changed.

For the refactoring of OCL, we are aware of only one approach. Correa and Werner present in [35] 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.

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 [35, 46, 76, 81], we do not use OCL pre-/postconditions for this purpose.

In [27], Cabot and Teniente present an approach for transforming existing OCL invariants into semantically equivalent alternatives. While we observe transformations of OCL constraints in the context of UML class diagrams refactorings, the technique discussed in [27] focus on transforming OCL expressions when no changes on underlying class diagram occur. When applying the *MoveOperation* rule, we similarly to [27] change the context of all constraints attached to the moved operation.

3.5 Conclusions

The refactorings considered in this chapter 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 chapter, 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 have formalized all refactoring rules in form of 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. 3.3. Semantics preservation will be discussed in Chapter 5.

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. 3.2 let us conclude that graph transformation systems are an 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 [10], 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 chapter have been fully implemented in the tool RoclET (see Appendix A), 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 precondition to *pull up* agile techniques, which became quite popular over the recent years on the implementation level, also to the modeling level.

Chapter 4

Model Transformations for Describing Semantics of OCL

The Object Constraint Language (OCL) has been for many years formalized both in its syntax and semantics. While the official definition of OCL's syntax is already widely accepted and strictly supported by most OCL tools, there is no such agreement on OCL's semantics, yet.

In the previous chapter we have specified refactoring rules using QVT. In order to prove that the rules are semantics preserving, the first step is to define semantics of artifacts that are subject of refactoring, in our case, UML/OCL models.

In this chapter, we propose an approach based on metamodeling and model transformations for formalizing the semantics of OCL. Similarly to OCL's official semantics, our semantics formalizes the semantic domain of OCL, i.e. the possible values to which OCL expressions can evaluate, by a metamodel. Contrary to OCL's official semantics, the evaluation of OCL expressions is formalized in our approach by model transformations written in QVT. Thanks to the chosen format, it was possible to define a criterion of semantics preservation, and to show that our refactoring rules satisfy this criterion, in a manner that is easy to understand, and reason about.

Our work on the formalization of OCL's semantics resulted also in the identification and better understanding of important semantic concepts, on which OCL relies. These insights are of great help when OCL has to be tailored as a constraint language of a given Domain Specific Language (DSL). We show on an example, how the semantics of OCL has to be redefined in order to become a constraint language in a database domain.

Work presented in this chapter was firstly published on MoDELS 2006 conference [57], and an extended version was accepted for publication in Software and Systems

Modeling (SoSym) journal [59].

4.1   Introduction

The OCL has proven to be a very versatile constraint language that can be used for different purposes in different domains, e.g., for restricting metamodel instances [65], for defining UML profiles [14], for specifying business rules [37], for querying models [28, 2] or databases [36].

Due to the lack of parsers, OCL was used in its early days often in an informal and sketchy style, what had serious and negative consequences as Bauerdick et al. have shown in [11]. Nowadays, a user can choose among many OCL parsers (e.g. OSLO [87], Eclipse Model Developement Tool (MDT) for OCL [84], Dresden OCL Toolkit [83], Octopus [86], Use [89], OCLE [85]), which strictly implement the abstract syntax of OCL defined in the OCL standard [68].

The situation is less satisfactory when it comes to the support of OCL's semantics by current OCL tools. While most of the tools now offer some kind of evaluation of OCL expressions in a given system state, none of the tools is fully compliant with the semantics defined in the OCL standard. We believe that the lack of semantic support in OCL tools is due to the lack of a clear and implementation-friendly specification of OCL's semantics. Interestingly, the *normative* semantics of OCL[1] given in the language standard [68], Section 10: *Semantics Described using UML* is also formalized in form of a metamodel, but, so far, this metamodel seems to be poorly adopted by tool builders.

In this chapter we present a new approach for formulating a metamodel-based semantics of OCL. Defining a semantics for OCL basically means (1) to define the so-called semantic domain, in which OCL expressions are evaluated, and (2) to specify the evaluation process for OCL expressions in a given context.

The semantic domain for OCL is given by all possible system states. Since a system state can be visualized by an object diagram, the semantic domain is (almost) defined by the official UML metamodel for object diagrams. There are two major problems to be solved when defining the semantic domain based on the definition of object diagrams. Firstly, UML's metamodel for object diagrams does not define the semantics of OCL's predefined types, such as *Integer, Real, String, Set(T)*, etc. However, this problem has been already recognized in the OCL standard and an additional package (named *Values*) for the OCL metamodel has been proposed. We

---

[1]There is also an *informative semantics* given in Annex A of [68], which is formulated in a set-theoretical style and goes back to the dissertation of M. Richters [71].

will, to a great extent, reuse the Values package in our approach. Secondly, the metamodel for object diagrams implicitly assumes the existence of solely one object diagram at any moment of time. This becomes a major obstacle as soon as more than one system state is relevant for the definition of OCL's semantics (and this is really the case when defining the semantics of OCL's post-conditions). We propose for this problem a solution which is fundamentally different from the one chosen in the normative semantics and which leads, as we think, to a much simpler metamodel for the semantic domain of OCL.

The *evaluation of OCL expressions* is specified in our approach by model transformations, which are in turn described as QVT rules. All QVT rules presented in this chapter are also available in its textual form. The complete set of rules can be downloaded, together with all relevant metamodels, from [88].

To summarize, our semantics for OCL has the following characteristics:

- The semantics is directly executable. Contrary to a paper-and-pencil semantics, OCL developers can immediately see by using a tool (e.g. RocLET), how the semantics applies in a concrete scenario.

  To our knowledge, only the semantics of OCL given by Brucker and Wolff ([24, 22]) has the same characteristics and can be executed in the OCL tool HOL-OCL.

- The semantics is defined on top of the official metamodels for OCL's abstract syntax and UML class- and object-diagrams. Consequently, the semantic definition becomes an integral part of the already existing language definitions for UML and OCL.

  However, we had to redefine some of the existing metamodels due to some obvious inconsistencies, which would have prevented us from completely implementing our approach.

- The target audience for our semantics are developers, who use OCL in practice. No familiarity with mathematical and logical formalisms is presumed. In order to understand the semantics, only some knowledge of metamodeling and QVT is required.

- The semantics is presented in a modular way. This allows to easily define, starting from our semantics of OCL, the semantics of another constraint language, which is tailored to a given DSL. Similarly, one could also create a new dialect for OCL in the context of UML; for example, one could decide to

abandon OCL's concept of being a three-valued logic and to allow only two
Boolean values *true* and *false*.

The last point highlights the flexibility of our approach. This flexibility is an
important step forward to the vision originally formulated by the PUML group (see,
e.g., [33]) to treat OCL not just as one monolithic language but rather as a family of
languages, which can be applied in many different domains and can adapt easily to
different requirements from these domains while still sharing a substantial amount
of common semantic concepts, libraries, etc.

The rest of the chapter is organized as follows. In Sect. 4.2, we sketch our
approach and show, by way of illustration, a concrete application scenario for our
semantics. The basic evaluation steps are formalized by QVT rules in Sect. 4.3. Sec-
tion 4.4 proposes a list of semantic concepts and discusses their impact on evaluation
rules. Section 4.5 shows the flexibility of our approach and presents a stepwise adap-
tation of OCL's semantics, so that the adapted version can be used as a constraint
language for a given DSL. Related work is given in Section 4.6, while Section 4.7
draws conclusions.

4.2   A Metamodel-Based Approach for OCL Evaluation

In this section we briefly review the technique and concepts on which our approach
relies, and illustrate, with a simple example, the evaluation of OCL constraints. We
concentrate on the evaluation of an invariant constraint in a given state. We finally
describe the difficulties arising from the evaluation of pre-/postconditions.

4.2.1   Changes in the OCL Metamodel

In order to realize our approach in a clear and readable way, we had to add a few
metaassociations, -classes, and -attributes to the *Values* package part of the official
OCL metamodel.

The metaclass *OclExpression* has a new association to *Instance*, what represents
the evaluation of the expression in a given object diagram (see Fig. 4.1). Further-
more, the classes *StringValue*, *IntegerValue*, etc. have now attributes *stringValue*,
integerValue, etc. what makes it possible to clearly distinguish a datatype object
from its value.

We revised slightly the concepts of bindings (association between *OclExpression*
and *NameValueBinding*) and added to class *LoopExp* two associations *current* and
intermediateResult, and one attribute *freshBinding* (see Fig. 4.2).
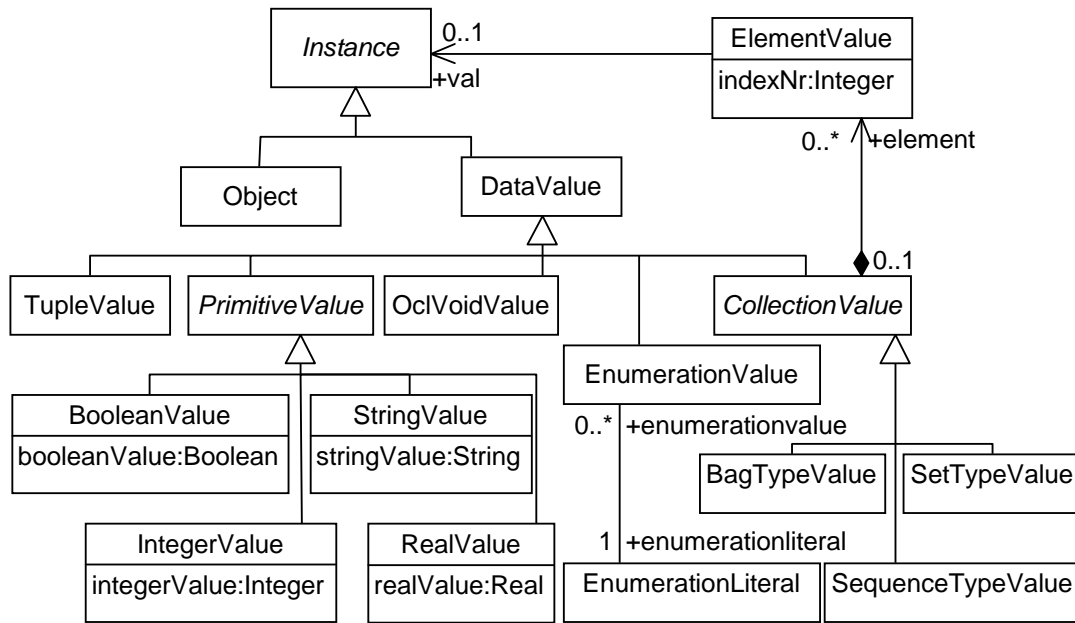
Figure 4.1: Changed metamodel for OCL - Instances
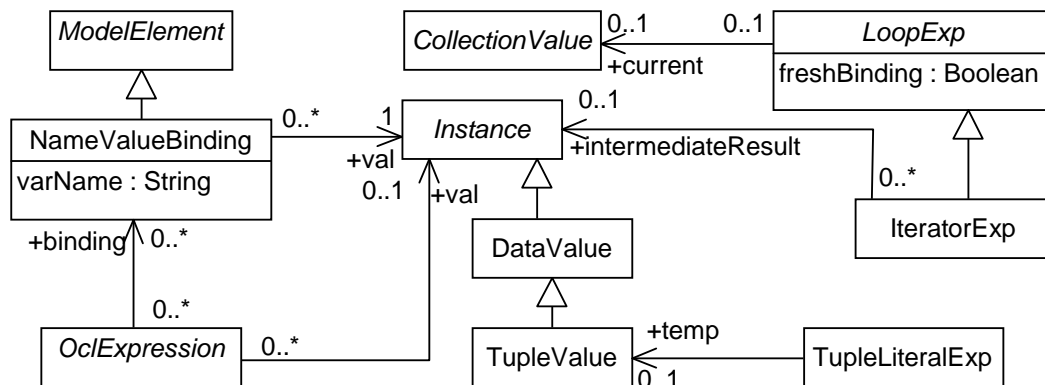
Figure 4.2: Changed metamodel for OCL - Bindings
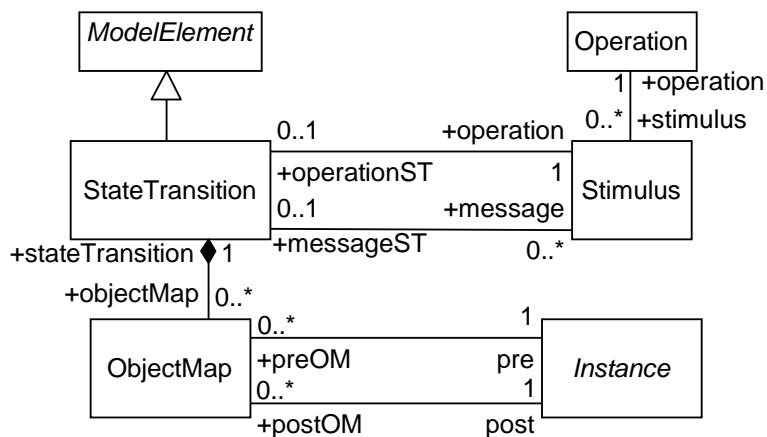
Figure 4.3: Changed metamodel for OCL - State Transitions

We have created two new metaclasses *StateTransition* and *ObjectMap* that are used in evaluations of pre and post-conditions (see Fig. 4.3). Metaclass *ObjectMap* has two metaassociations with metaclass *Instance* and is used to relate two *Instances* in a pre- and a post-state. Metaclass *StateTransition* has two metaassociations with *Stimulus* representing an Operation that corresponds to a given *StateTransition* or a sent message. Stimulus itself is used to keep the track about an operation invocation: receiver and sender of a message, and operation arguments.

4.2.2   Evaluation
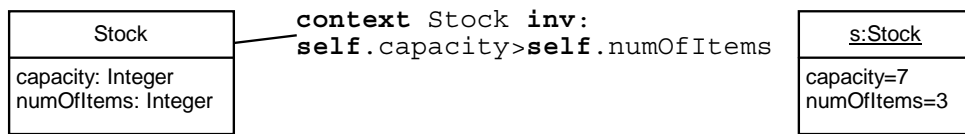
Figure 4.4: Example - Class Diagram and snapshot

We motivate our approach to define OCL's semantics with a small example. In Fig. 4.4, a simple class diagram and one of its possible snapshots is shown. The model consists of one class `Stock` with two attributes: `capacity` and `numOfItems`, both of type *Integer*, representing the capacity of `Stock` and the current number of available items, respectively. The additional constraint attached to the class `Stock` requires that the current number of items in a stock must always be smaller than the capacity. The snapshot shown in the right part of Fig. 4.4 satisfies the attached invariant because for each instance of `Stock` (class `Stock` has only one instance in the snapshot) the value of `numOfItems` is less than the value of attribute `capacity`. In other words, the body of the constraint attached to the class `Stock` is evaluated on object `s` to *true*.

In order to show how the evaluation of an OCL constraint is actually performed
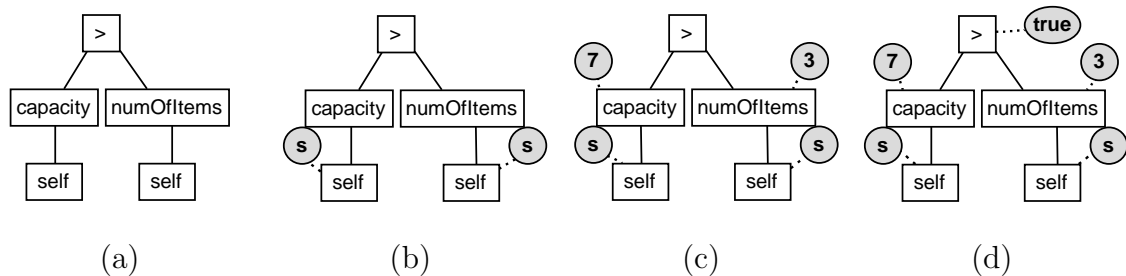
Figure 4.5: Evaluation of OCL expressions seen as an AST: (a) Initial AST (b) Leaf nodes evaluated (c) Middle nodes evaluated (d) Complete AST evaluated

on a given snapshot, we present in Fig. 4.5 the simplified state of the Abstract Syntax Tree as it is manipulated by an OCL evaluator. Step (a)-(b) performs the evaluation of the leaf nodes. Depending on the results of these evaluations, step (b)-(c) performs evaluation of nodes at the middle level. Finally, the last step (c)-(d) performs evaluation of the top-level of the AST. Please note that in this example we were not concerned about concrete binding of the variable *self*. The problem of variable binding is discussed in Sect. 4.2.3.
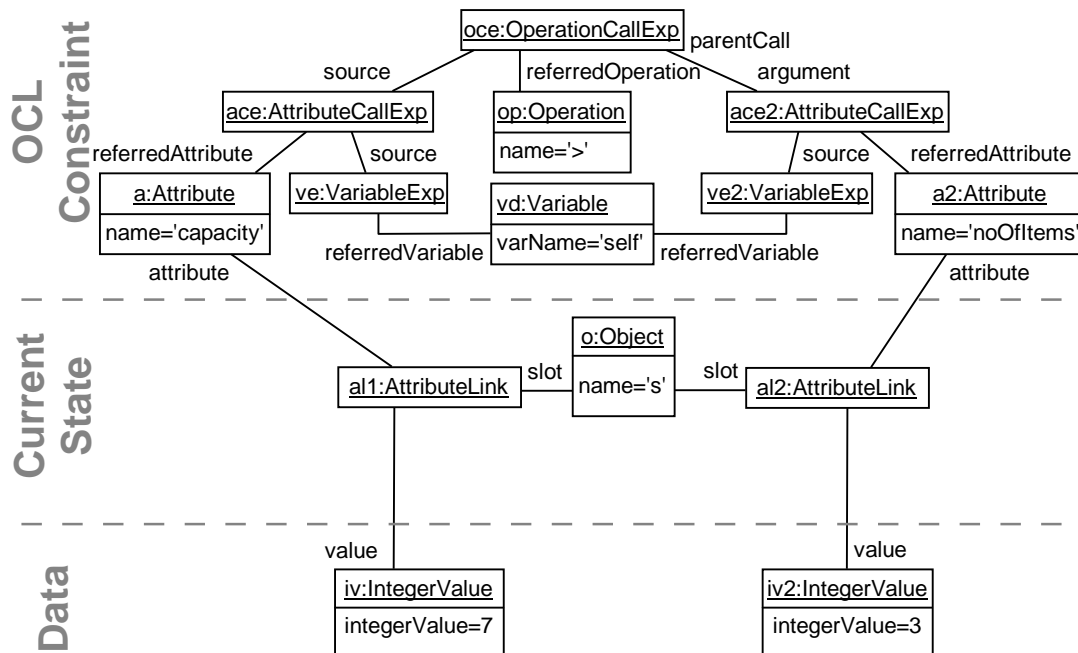
Figure 4.6: OCL constraint before evaluation

Figure 4.6 shows the instance of the OCL metamodel representing the invariant from Fig. 4.4. Here, we stipulate that all expressions have not been evaluated yet because for each expression the link *val* to metaclass *Instance* is missing. Please note that here we assume that in all expressions, variable `self` is bound to the object `o`. For the sake of readability, this information is omitted in Figures 4.6 and 4.7.

The final state of the metamodel instance, i.e. after the last evaluation step has been finished, is shown in Fig. 4.7. What has been added compared to the initial state (Fig. 4.6) is highlighted by thick lines. The evaluation of the top-expression (*OperationCallExp*) is a *BooleanValue* with *booleanValue* attribute set to *true*, the two *AttributeCallExpression*s are evaluated to two *IntegerValue*s with values 7 and 3, and each *VariableExp* is evaluated to *Object* with name `s`.
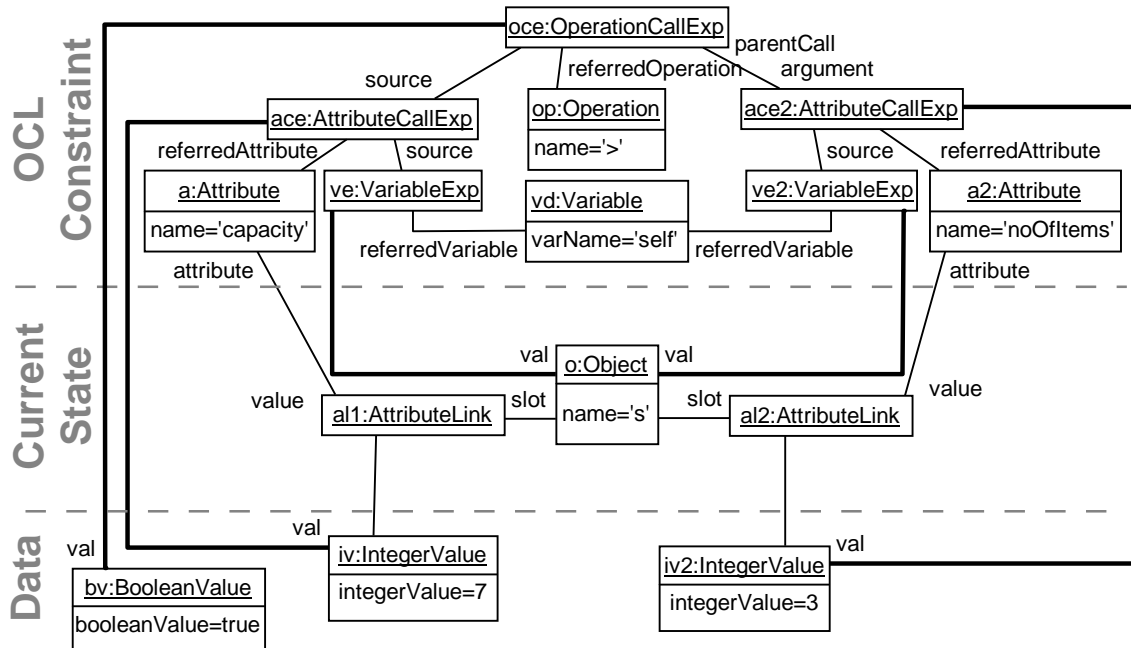
Figure 4.7: OCL constraint after evaluation in a given snapshot

4.2.3 Binding

The evaluation of one OCL expression depends not only on the current system state, on which the evaluation is performed, but also on the binding of free variables to current values. The binding of variables (e.g. *self* variable, *let* variables, *iterator* variables) is realized in the OCL metamodel by the class *NameValueBinding*, which maps one free variable name to one value. Every OCL expression can have arbitrarily many bindings, the only restriction is the uniqueness of variable names within the set of linked *NameValueBinding* instances.

The binding of variables is done in a top-down approach. In other words, variable bindings are passed from an expression to all its sub-expressions. Some expressions do not only pass the current bindings, but also add/change bindings (like *let* and *iterate* expressions). An example for adding new value-name bindings will be presented in more details in Sect. 4.3.

Figure 4.8 shows the process of binding passing on a concrete example. In the upper part, the initial situation is given: The top-expression already has one binding `nvb` for variable *self*. In the lower part of the figure, all subexpressions of the top-expression are bound to the same *NameValueBinding* as the top-expression.
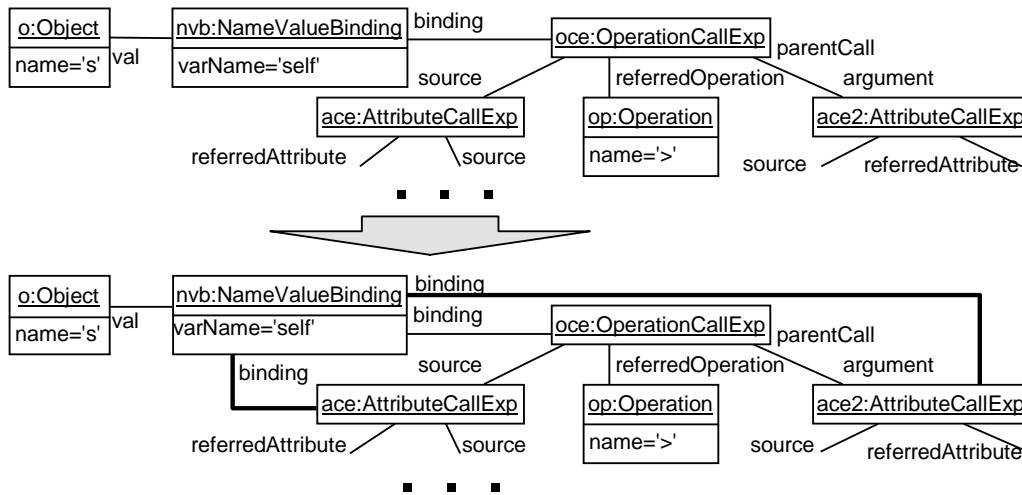
Figure 4.8: Binding passing

4.3 Core Evaluation Rules Formalized as Model Transformations

The previous section has shown the main idea of our approach: we annotate the evaluation result of each (sub)expression directly to the corresponding instance of class *OclExpression* in the OCL metamodel. What has not been specified yet are the evaluation steps themselves, for example, that an *AttributeCallExp* is always evaluated to the attribute value on that object to which the source expression of *AttributeCallExp* evaluates. As shown below, these evaluation steps will be formally given in form of model transformation rules.

Although the model transformation rules are generally nicely readable and understandable, their number can become quite high if one wants to accommodate all peculiarities of OCL (e.g. undefined values, flattening of collections, *@pre* in postconditions, etc.). In order to structure the semantics definition, we will present in this section the core version of evaluation rules for certain types of expressions and will explain in the next Section 4.4 how this core rules have to be extended/adapted in order to reflect all semantic concepts of OCL.

4.3.1 Model Transformation Rules

For the specification of evaluation rules, we use the formalism of model transformations, more precisely QVT (Query/View/Transformation) rules [66]. This formalism is already used in Chapter 3 for specifying refactoring rules.

In order to avoid the redundancy of having the same subpatterns in LHS in RHS,

our evaluation rules contain besides LHS and RHS a third part called *Context*, that specifies the structures in the input, which must be available when applying the rule but which are not changed . The *Context* part is optional. For the core rules presented in this section, the *Context* will encode the assumed structures in the current state, in which the OCL expression is being evaluated. When it comes to the evaluation of pre-/postconditions, we will see in the next section that the *Context* can also contain even more information. Besides the structures that describe the system state, *Context* can also contain an optional part with data values that are necessary for the evaluation of the rules.

4.3.2   Binding Passing

Before the source expression can be evaluated, the current binding of variables has to be passed from the parent expression to all its subexpressions. Figure 4.9 shows the transformation rule for *OperationCallExp*. When applying this rule, the binding of the parent object `oce` (represented by a link from `oce` to the multiobject `nvb` in LHS) is passed to subexpressions `oe` and `aoe` (links from `oe` and `aoe` to `nvb` are established in RHS). Multiobject `nvb`, shown in Fig. 4.9, represents an object template that matches many objects of type *NameValueBinding*. Analogous rules exist for all other kinds of OCL expressions which have subexpressions (e.g. *if* expressions, *let* expressions, etc.). For the (subclasses of) *LoopExp*, one needs also additional rules for handling the binding because the subexpressions are evaluated under a different binding than the parent expression.
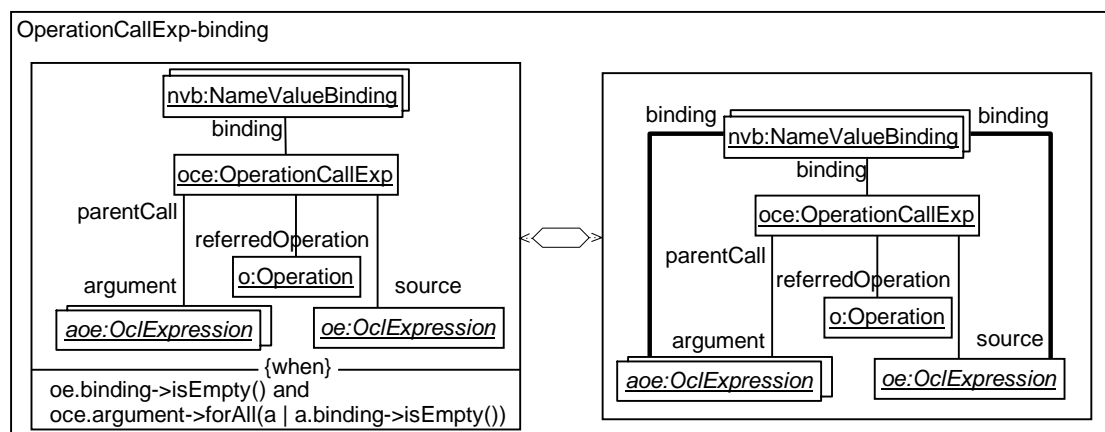
Figure 4.9: Binding of an expression

4.3.3   A Catalog of Core Rules

In order to define the semantics of OCL, one needs to provide at least one evaluation rule for each concrete subclass of *OCLExpression* metaclass from the OCL metamodel.

The semantics of a constraint language such as OCL can be split along this syntactic dimension (in Section 4.4, we will see that it is useful to have also another dimension for the semantics). However, it is not always appropriate to organize a catalog of evaluation rules based on the metaclasses from the abstract syntax metamodel. Sometimes, evaluation rules for different metaclasses are very similar so that these evaluation rules could be put into the same category (for example, *Navigation Expressions*). But there is also the opposite case, where instances of the same metaclass are evaluated using very different mechanisms, what is a sign for a wrong granularity of metaclasses in the metamodel (for example, *OperationCallExp*).

We propose to organize the evaluation rules for OCL, based on *Navigation Expressions*, *Operation Expressions*, *Loop Expressions*, *Variable Expressions*, *Literal Expressions*, *If Expressions*, *Let-Expressions*, *State Expressions*[2], and *Tuple Expressions*. Moreover, regarding *Operation Expressions*, it is useful to distinguish expressions that refer (1) to predefined operations from the OCL library, (2) to queries defined by the user in the underlying class model.

Here, we discuss only the most representative rules. The main goal is to demonstrate that the evaluation of all kinds of OCL expressions can be formulated using graph transformations in an intuitive but precise way. The complete catalog of rules, as implemented in our ROCLET tool, can be downloaded from [88].

Navigation Expressions

OCL expressions of this category are, for example, instances of *AttributeCallExp* and *AssociationEndCallExp*. Such expressions are evaluated by 'navigating' from the object, to which the source expression is evaluated, to that element in the object diagram, which is referenced by the attribute or association end.

AttributeCallExp

The semantics of *AttributeCallExp* is specified by the rule *AttributeCallExp-evaluation* given in Fig. 4.10. The evaluation of `ace` is *DataValue* d, which is also the value of

---

[2]We consider as the semantic domain of our evaluation only object diagrams in which the objects do not have a reference to an explicit state given in a state diagram. Consequently, *State Expressions* are ignored here.

the attribute `a` for object `o`. Note, that we stipulate in the LHS, that `oc`, the source expression of `ace`, has been already evaluated to object `o`.
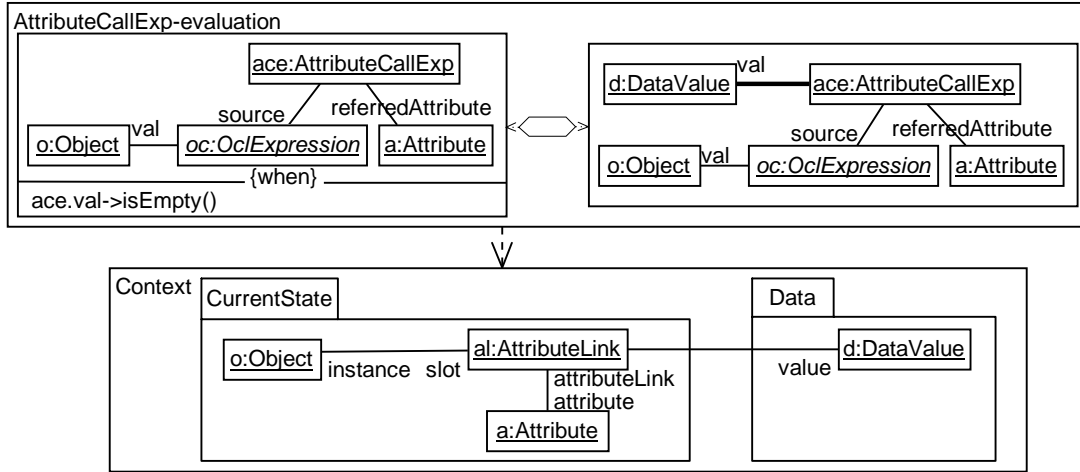
Figure 4.10: Attribute Call Expression evaluation

AssociationEndCallExp

Here we discuss two cases of *AssociationEndCallExp*. The first one is navigation over an association end with multiplicity equal to 1, and the second one is navigation if multiplicity is greater that 1 and the association end is unordered.

The semantics of the former case is specified by the rule given in Fig. 4.11. As can be easily recognized, this rule is very similar to *AttributeCallExp*.

The latter case shown in Fig. 4.12 specifies that the value of `aece` is a newly created object of type *SetTypeValue* whose elements refer to all objects `o2` that can be reached from object `o` via a link for `ae`. Again, object `o` is the evaluation of source expression `oe`. Note that in Fig. 4.12 the newly created objects are marked with grey color. The rule shown in Fig. 4.12 contains at few locations the multiplicities 1-1 at the link between two multiobjects, for example at the link between `le2` and `l`. This is an enrichment of the official QVT semantics on links between two multiobjects. Standard QVT semantics assumes that a link between two multiobject means that each object from the first multiobject is linked to every object from the second multiobject, and vice versa. This semantics is not appropriate for the situation shown in Fig. 4.12 where each element of multiobject `l` must be connected only to one element from multiobject `le2`, and vice versa. By using 1-1 multiplicities, we indicate a non-standard semantics of links between two multiobjects.
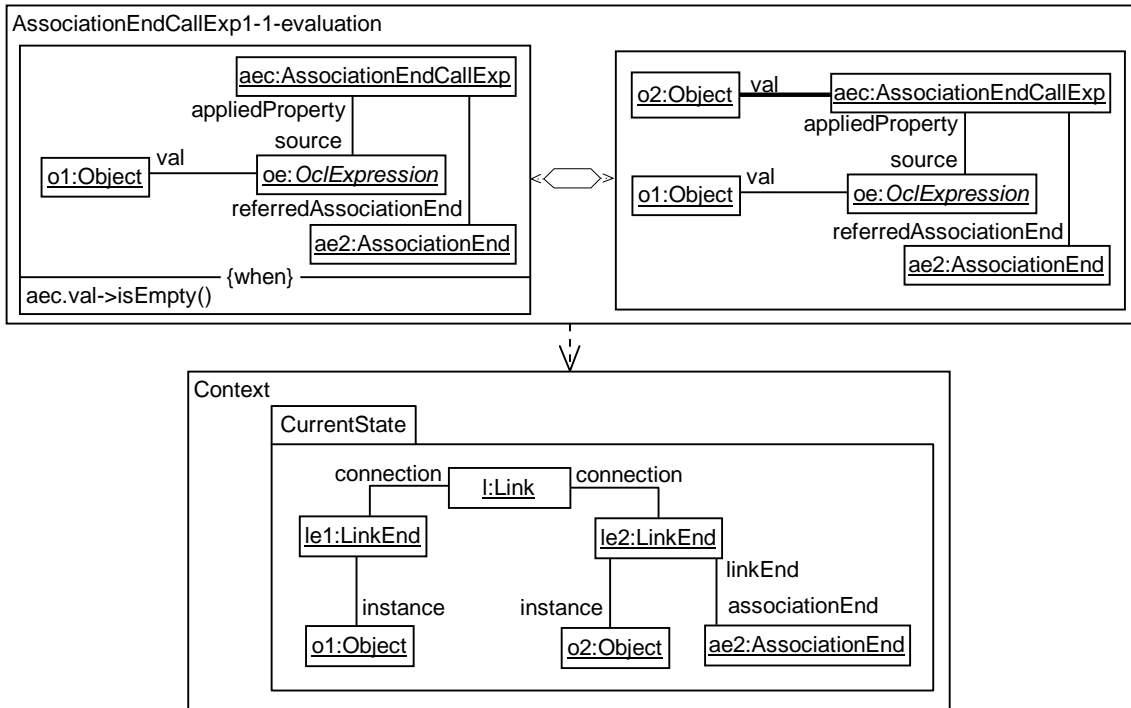
Figure 4.11: Association End Call Expression evaluation that results in an object
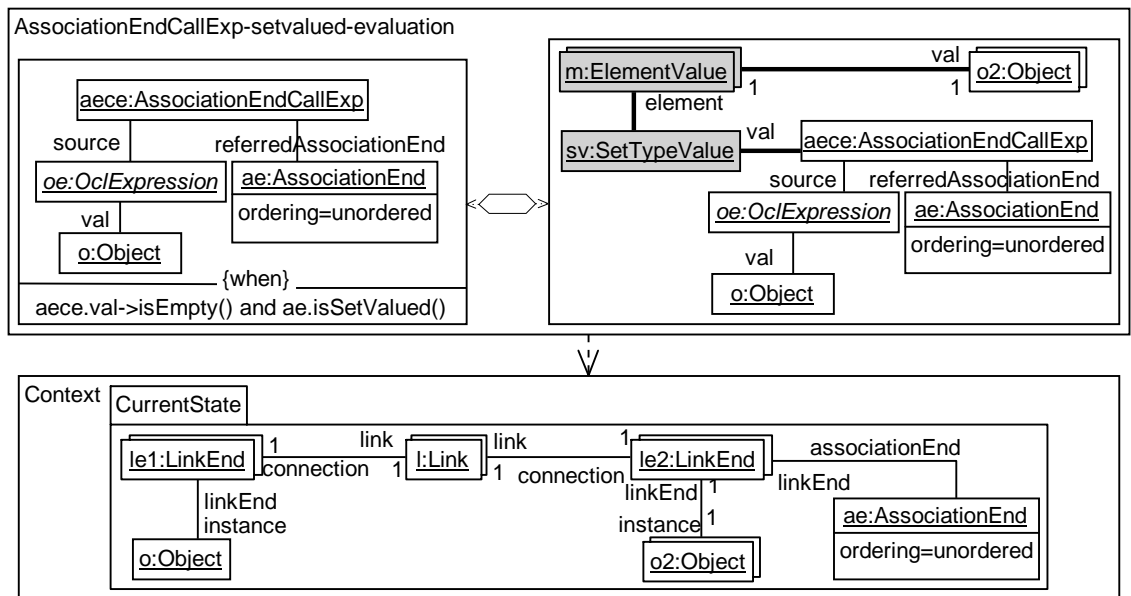
Figure 4.12: Association End Call Expression evaluation that results in set of objects

Operation Expressions

Expressions Referring to Predefined Operations

Expressions from this category are instances of the metaclass *OperationCallExp* but the called operation is a predefined one, such as $+$, $=$. These operations are declared and informally explained in [68, Chapter 11]. As an example, we explain in the following the semantics of operation "$=$" (equals). We show only two rules here, one specifies the evaluation of equations between two objects, and the other the evaluation of equations between two integers.

In Fig. 4.13, the evaluation is shown for the case that both subexpressions `oe1`, `oe2` are evaluated to two objects `o1` and `o2`, respectively. In this case, the result of the evaluation is `bv`, of type *BooleanValue*, with attribute *booleanValue* `b`, which is *true* if the evaluations of `oe1` and `oe2` are the same object, and *false* otherwise. Evaluation to undefined is discussed in Sect. 4.4.
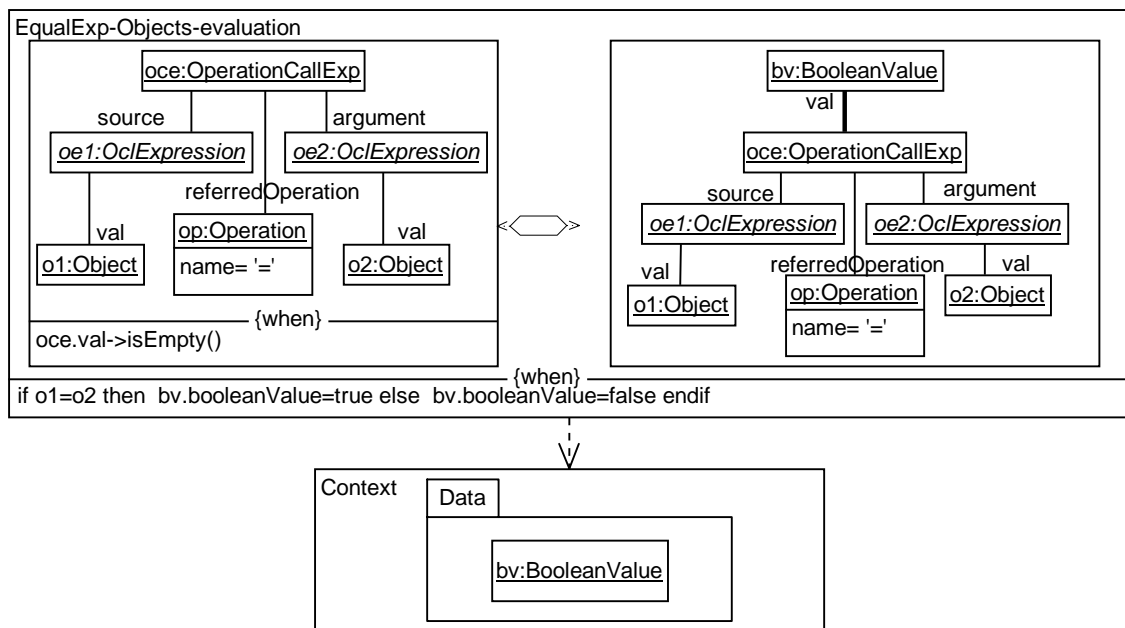
Figure 4.13: Equal Operation evaluation for objects

If `oe1` and `oe2` evaluate to *IntegerValue*, the second QVT rule shown in Fig. 4.14 is applicable and the result of evaluation will be an instance of *BooleanValue* with attribute *booleanValue* set to *true* if the attribute *integerValue* of `iv1` is equal to *integerValue* of `i2`, and to *false* otherwise.

Rules shown in Fig.4.13 and Fig. 4.14 can be re-formulated using two rules, each; one rule that specifies evaluation to `true` and the other that specifies evaluation to
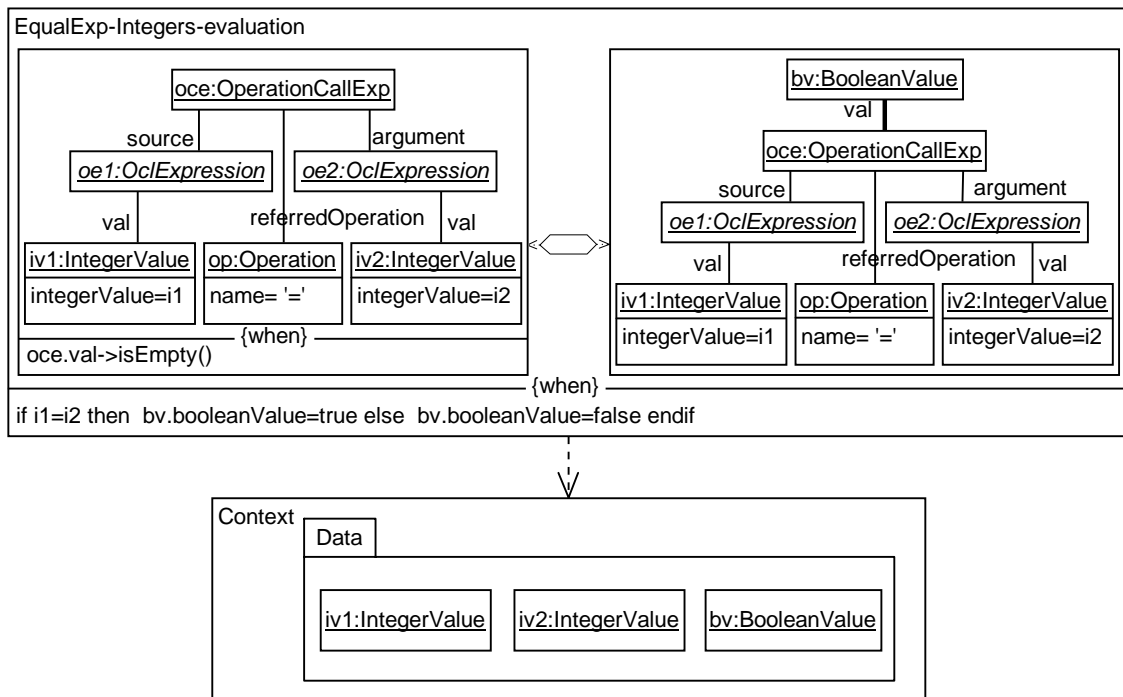
Figure 4.14: Equal Operation evaluation for integers

`false`. This way we could avoid usage of `if` expression in the *when* clauses of the rules, but this would lead to bigger evaluation rules, which are harder to understand.

Expressions Referring to a User-defined Query

If a user-defined query is used in an OCL constraint, then the semantics of the used query must be specified by a body-clause (or def-clause), which is attached to the query. The query might also have attached a pre-condition, which must evaluate to *true* in the current situation. Otherwise, the query-expression is evaluated to *undefined*. If the pre-condition evaluates to *true*, then the value of the *OperationCallExp* is the same as the evaluation of the body-clause under the current argument binding.

Fig. 4.15 shows evaluation rules for user-defined queries specified with a body-clause. The first rule creates a set of *NameValueBindings* for the expressions in *precondition* and *body*. Every *NameValueBinding* from this set corresponds to exactly one argument of the *OperationCallExp* `opce`. The second rule performs evaluation of the query in such a way that, if the *precondition* does not evaluate to *true*, the result of the evaluation will be *undefined*, otherwise the result is the result of evaluation of the *body*. One problem, however, is, that the body-expression might contain again an *OperationCallExp* referring to `op`, i.e. the definition of `op` is recursive. Recursive query definitions can lead in some cases to infinite loops during the evaluation. Brucker et al. propose in [23] that recursive query definitions should
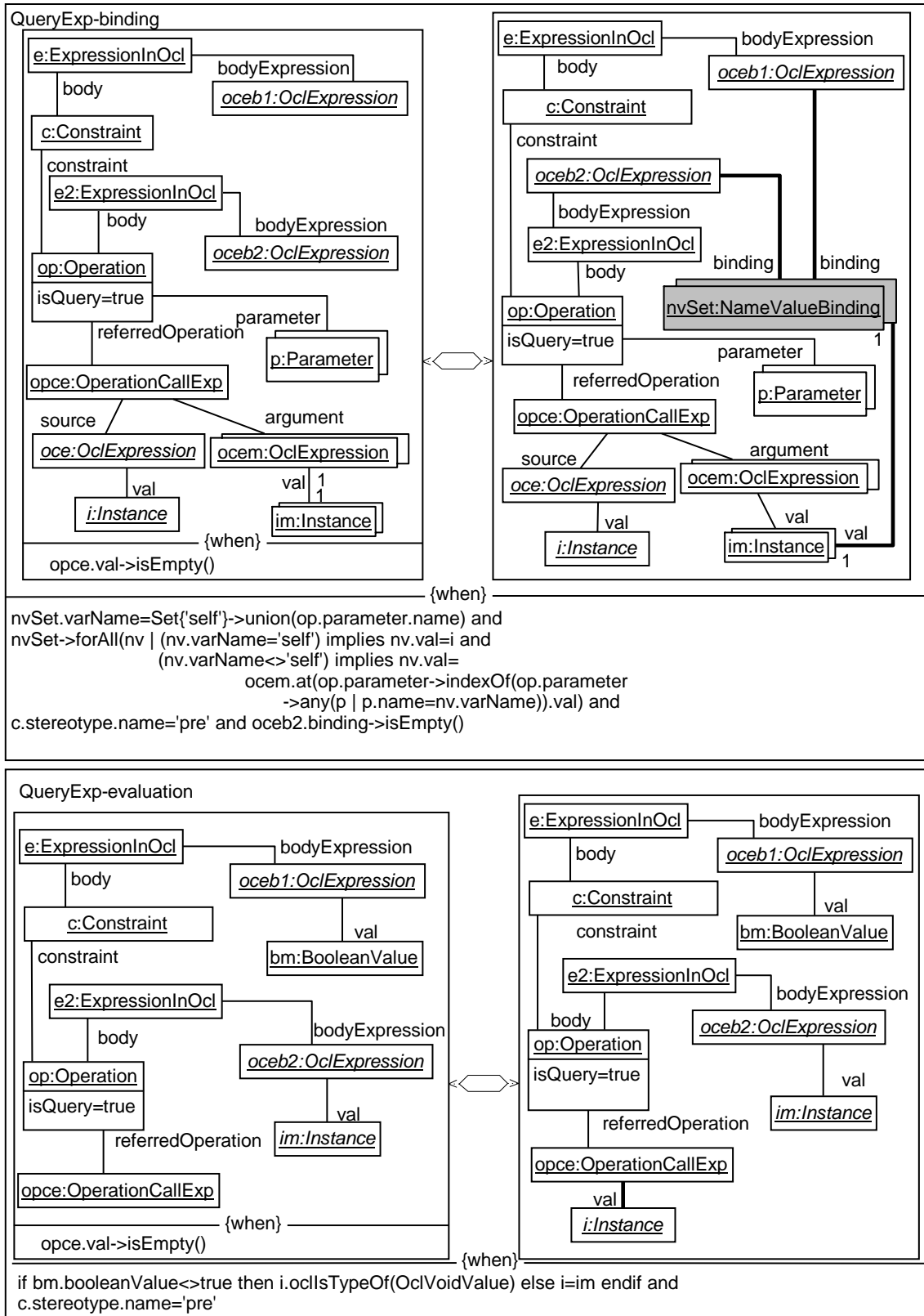
Figure 4.15: Evaluation of an expression referring to a query

be checked by the user for unfounded recursions, but this would require substantial analysis effort.

Expressions for Typecheck and Typecast

To this group belong all *OperationCallExps* referring to the predefined operation *oclAsType*, *oclIsTypeOf*, and *oclIsKindOf*. The operation *oclAsType* makes a cast of the source expression to the type specified in the argument. If this cast is successful, the whole expression is evaluated to the same object as the source expression. If the cast is not successful (i.e., the evaluation of the source expression is an object whose type does not conform to the type given in the argument), then the whole expression is evaluated to *undefined*. Because we treat the evaluation to undefined in the next Section 4.4 in a general manner, we skip the rule for *oclAsType* here.

The rules for *oclIsTypeOf* and *oclIsKindOf* are very similar; Fig. 4.16 shows the rule for *oclIsKindOf*. Please, note that operation *conformsTo()* is omitted here but can be found in the official definition of the OCL metamodel [68].
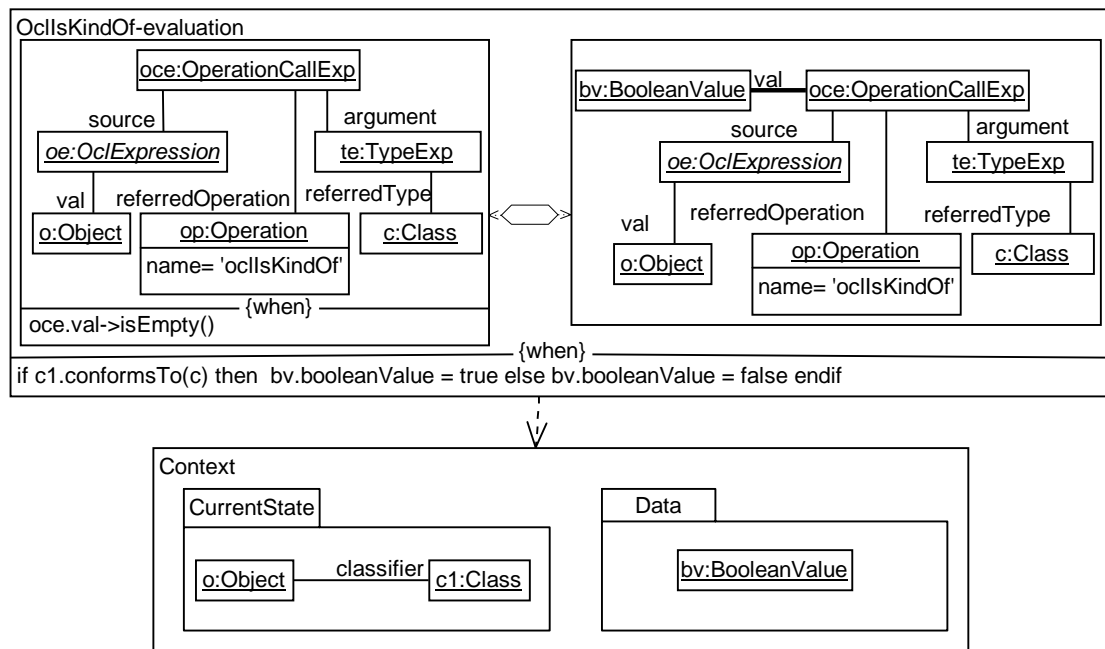
Figure 4.16: Evaluation rule for oclIsKindOf

allInstances()-Expressions

The predefined operation *allInstances()* yields all existing objects of the specified type and all its subtypes. The rule is shown in Fig. 4.17. Note that the multiobject **os** represents, according to the QVT semantics, the maximal set of objects **o**, for which the condition given in the when-clause of the *Context* holds.
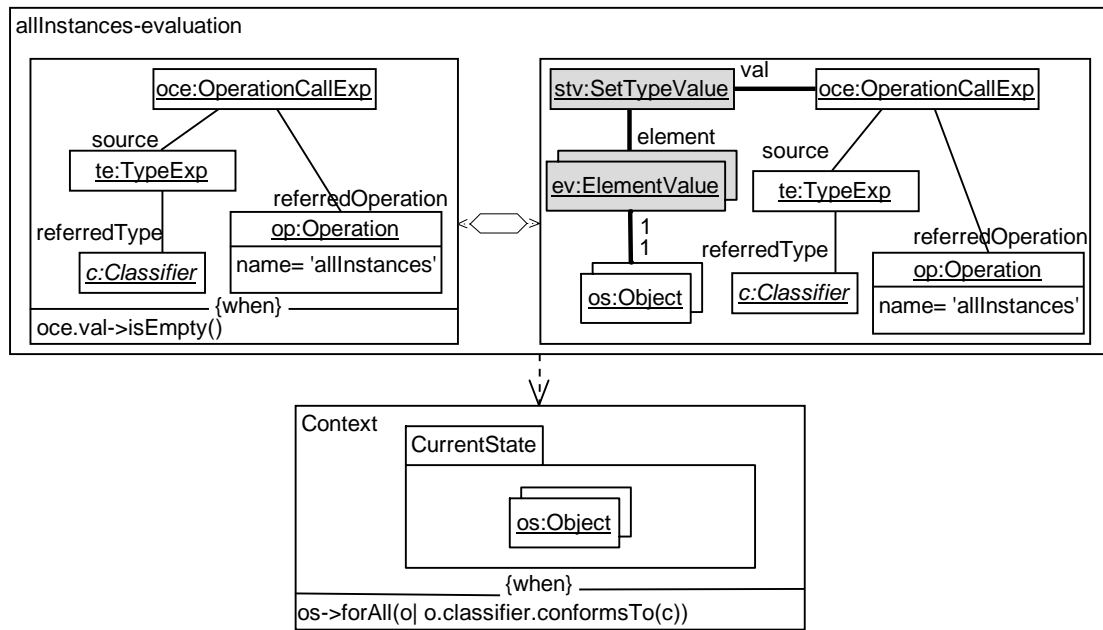
Figure 4.17: Evaluation rule for allInstances

Loop Expressions

Iterator expressions are those in OCL which have as the main operator one from *select*, *reject*, *forAll*, *iterate*, *exists*, *collect*, *any*, *one*, *collectNested*, *sourtedBy*, or *isUnique*. Since all these expressions can be expressed by macros based on *iterate*, it is sufficient to refer for their semantics just to the semantics of *iterate*.

In the Fig. 4.18 are shown evaluation rules that describe the semantics of *iterate*.

The rule *Iterate-Initialisation* makes a copy of evaluation of the source expression, and assigns it under the role *current* to `ie`. Furthermore, one *NameValueBinding* is created and assigned to the body expression. The name of the *NameValueBinding* is the same as the name of *result* variable, and its value is the same as the value of the *initExpression* for the *result* variable. For some technical reasons, the attribute *freshBinding* of `ie` is set to *false*.

The rule *Iterate-IteratorBinding* updates the binding on body expression `oe` for the iterator variable `v` with a new value `vp`. The element with the same value `vp` is chosen from the collection *current* and is removed afterwards from this collection. The attribute *freshBinding* is set to *true* and the binding for `oe` has changed.

The rule *Iterate-IntermediateEvaluation* updates the binding for the variable with the same name as the result variable of `ie` based on the new evaluation of `oe`. Furthermore, the value of attribute *freshBinding* is flipped and the evaluation of body expression `oe` is removed.
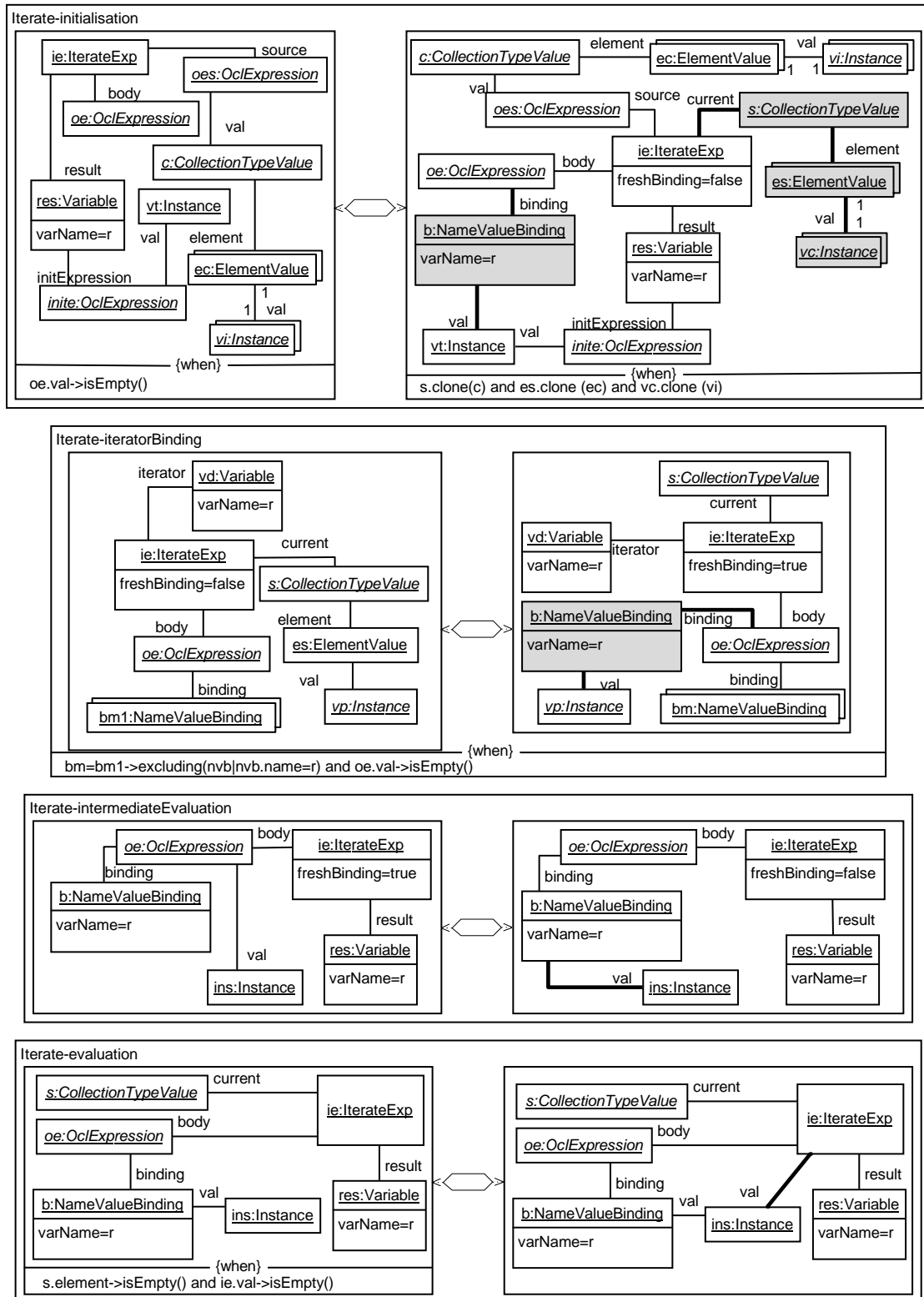
Figure 4.18: Iterate - evaluation rules

The final rule *Iterate-evaluation* covers the case when the collection `current` of `ie` is empty. In this case the value of `ie` is set to that value which is bound to the NameValueBinding with the same name as the result variable.

Variable Expressions

Figure 4.19 shows the evaluation rule for *VariableExp*. When this rule is applied, a new link is created between *VariableExp* and the value to which *NameValueBinding*, with the same name as *VariableDeclaration*, is connected.
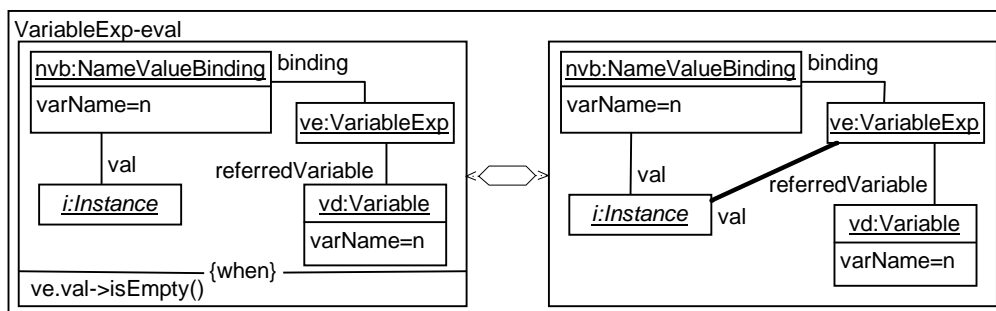
Figure 4.19: Variable Expression evaluation

Literal Expressions

In Fig. 4.20, the evaluation of *IntegerLiteralExp* is shown. By applying this rule, a new *IntegerValue* is created whose attribute *integerValue* has the same value as the attribute *integerSymbol* for expression `ie`. Note, that this type of expressions does not need variable bindings because their evaluation does not depend on the evaluation of any variable.

If-Expressions

Figure 4.21 shows the evaluation rule for an *if* expression. The result of the evaluation depends on the value to which condition expression `c` is already evaluated. As it is stated in the *when* clause of the rule, if the value of the condition is *true* then the result of the evaluation will be the value of the *thenExpression*, otherwise it will be value of the *elseExpression*. Please note that in this example we don't deal with evaluation to undefined and that this aspect of OCL will be discussed later.
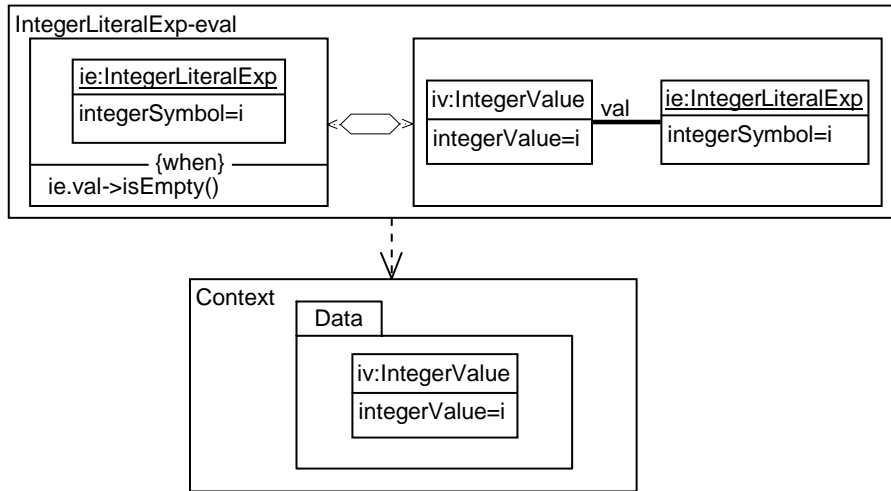
Figure 4.20: Integer Literal Expression evaluation
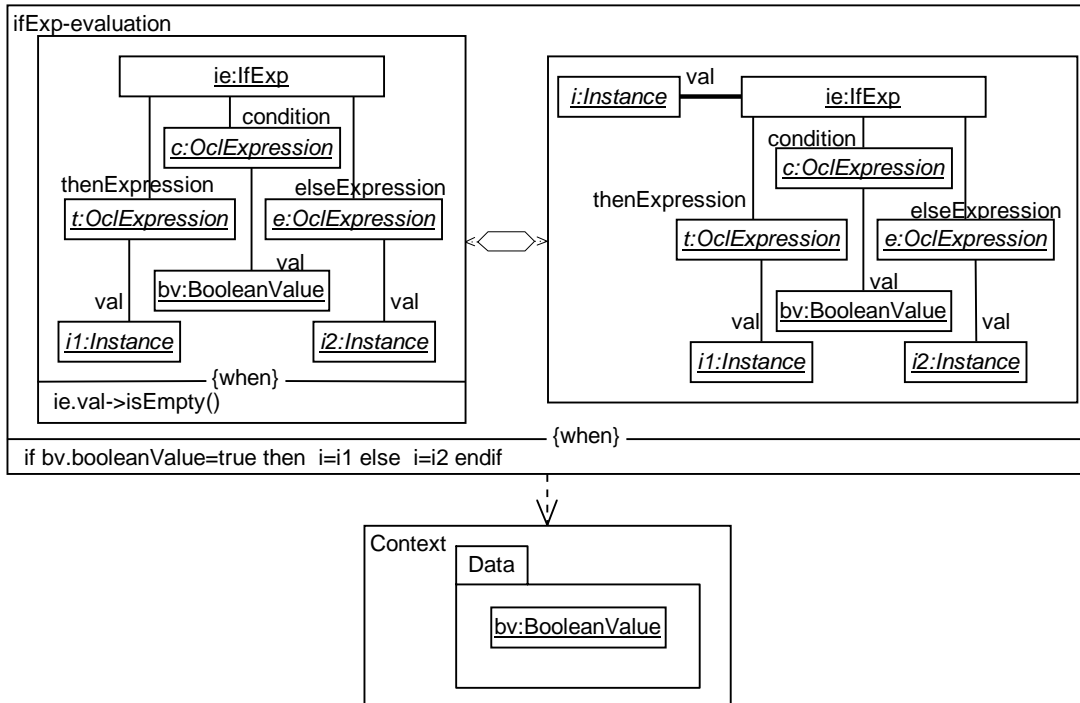
Figure 4.21: If Expression evaluation

Let-Expressions

The evaluation of *Let-Expressions* is a little bit different from the other rules because it changes *NameValueBinding* for its subexpressions (similarly to *LoopExpression*). The evaluation rules for *LetExp* are shown in Fig. 4.22. The first rule performs binding of the *Let-variable* to the value to which `initExpression` evaluates (by creating a new *NameValueBinding* instance), and then passes this *NameValueBinding* to the `in` part of the expression. The second part specifies that result of evaluation of an *LetExp* will be the same as evaluation of its `in` expression.
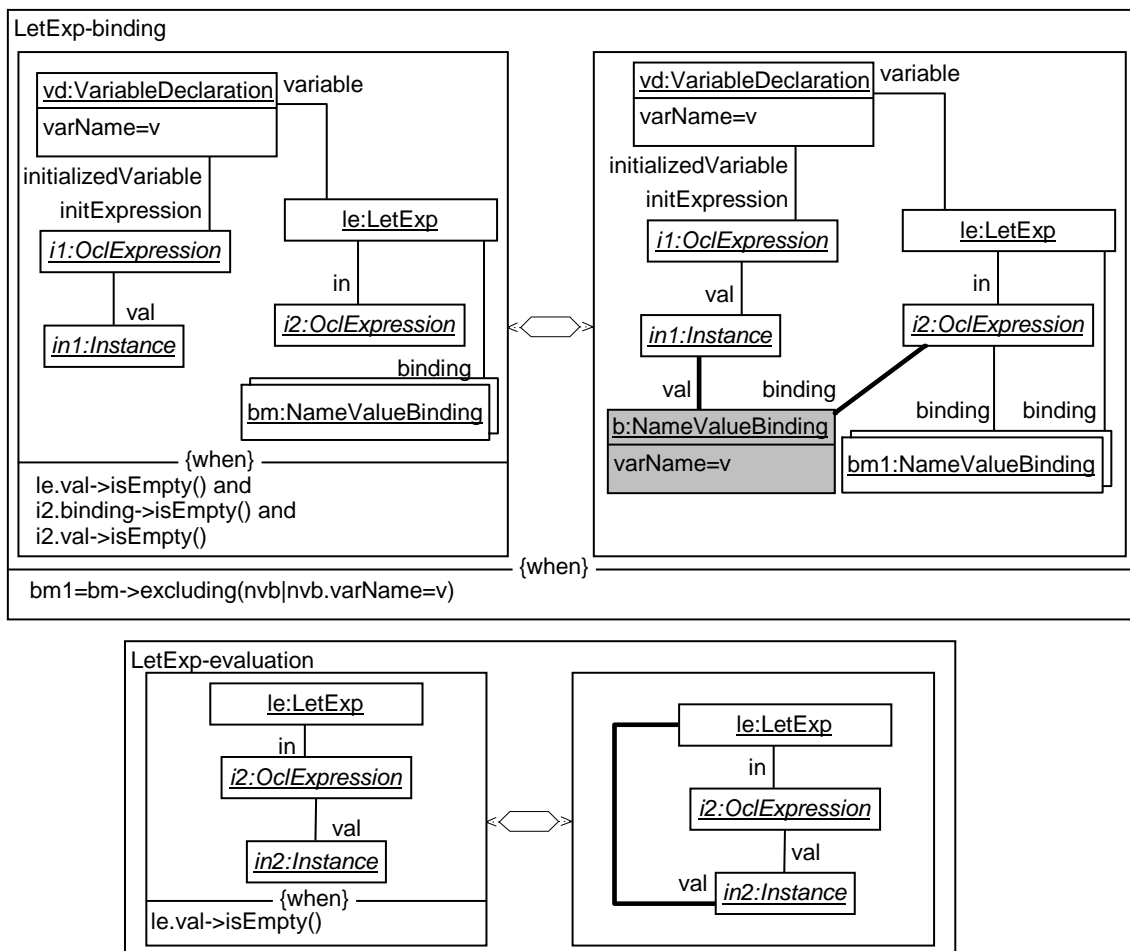
Figure 4.22: Let Expression: binding and evaluation

Tuple Expressions

In Figure 4.23, the evaluation rule for *TupleExp* is shown. This rule consists of three parts. The first part creates a temporary *TupleValue* object that will become the result of evaluation once all *TupleLiteralParts* are traversed. The middle rule shows

the core semantics of *TupleExpression* evaluation. This rule will be executed as many times as there are *TupleLiteralParts* in the expression. Each time this rule is triggered, a new *AttributeLink* is created and attached to the temporary *TupleValue*. This newly created *AttributeLink* will point to one attribute from the tuple type, and to the value that *TupleLiteralPart* has. The third rule is used to create the final value of the *TupleExp*.
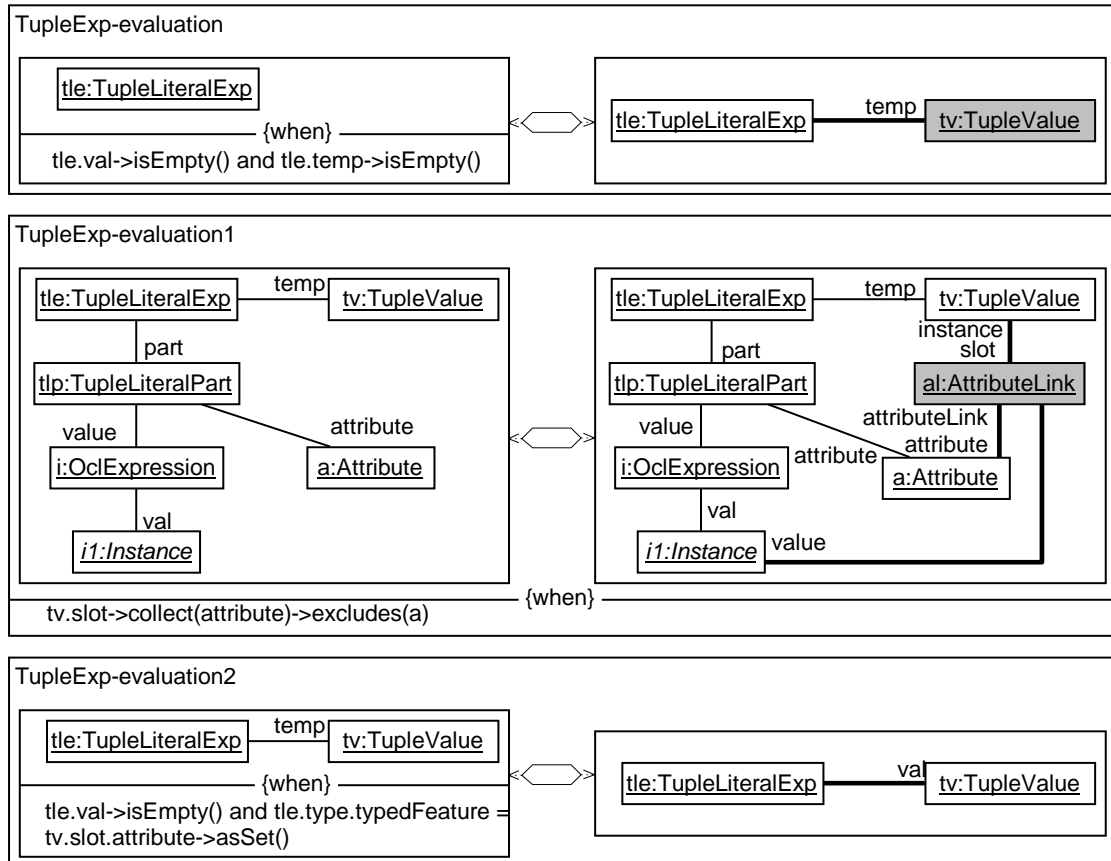
Figure 4.23: Tuple Expression evaluation

4.3.4 Syntactic Sugar

Many pre-defined OCL operations are defined as an abbreviation for more complex terms. For instance, the operation *exists* can be simulated by the operation *iterate*. More precisely, expressions of form

```
coll −>e x i s t s ( x |  body ( x ) )
```

can be rewritten to

```
coll −>i t e r a t e ( x ;  acc : Boolean=f a l s e |  acc  or  body ( x ) )
```

This rewriting step can also be expressed as a graph transformation rule what would make the rule for evaluating the pre-defined operation *exists* superfluous.
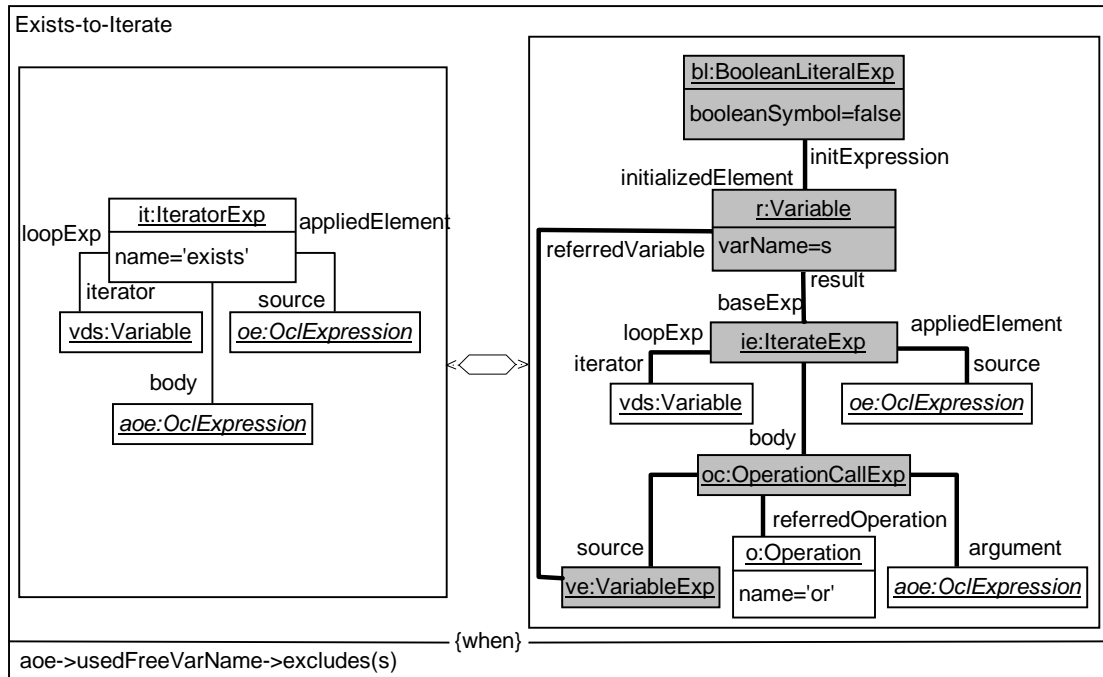
Figure 4.24: Transforming Exists expression to an iterate expression

Figure 4.24 shows a QVT rule that transforms one *exists* expression into corresponding *IterateExp*. RHS of the rule states that a new *IterateExp* is created, new *VariableDeclaration*, new *BooleanLiteralExp* with *booleanSymbol false*. The source of the expression and the iterator remain the same as for the *exists* operation. The body expression is modified and after the transformation it represents the disjunction of the previous body and the newly created variable expression that refers to the new *VariableDeclaration*. In the when-clause, we state an additional constraint that the *varName* s used in the newly created *VariableDeclaration* is not yet used as a name by any of the free variables in the body. Note that this constraint was not specified for the textual representation of the transformation and that would mean that body expression does not contain any free variable with the name `acc`.

4.4   Semantic Concepts in OCL

In the previous section, the most important evaluation rules for each of the possible kinds of OCL expressions were given. The rules basically describe the necessary evaluation steps in a given state, but they do not reflect yet the complete semantics of OCL. For example, nothing has been said yet on how an operation contract

consisting of pre-/postconditions is evaluated, how to handle the *@pre* construct in postconditions, under which circumstances an expression is undefined, etc. These are examples for *additional semantic concepts*, that are realized in OCL but which are most likely not realized in every other constraint language. Besides the syntactic dimension already explained in Sect. 4.3.3 for the categorization of rules, the additional semantic concepts form a second dimension for the rule categorization. We have identified the following list of semantic concepts, which must be taken into account when formulating the final version of evaluation rules (note that in Sect. 4.3.3 only the rudimentary version of evaluation rules has been shown).

- evaluation of operation contracts (pre-/postconditions)

- message sending

- evaluation to undefined (including strict evaluation with respect to undefined, with some exceptions)

- dynamic binding when invoking a query

- non-deterministic constructs (any(), asSequence())[3]

In the next subsections, we discuss the semantical concepts that have the most impact on the evaluation rules from Sect. 4.3.3.

4.4.1 Evaluation of Operation Contracts

The evaluation of an operation contract is defined with respect to a transition between two states.

StateTransition metaclass from our metamodel (see Fig. 4.3) is used to capture one transition from a pre- to a post-state. This transition is characteristic of one concrete operation execution with concrete values passed as operation parameters. In order to be able to evaluate one pre- or one post-condition we need all information about the state transition for which we want to perform the evaluation: operation that caused the transition, values of the operation parameters, pre-state, post-state, relationships between objects from pre- and post-state.

The evaluation of preconditions can be done analogously to the evaluation of invariants. The current state the evaluation rules referred to in the *Context* is in this case just the pre-state. In addition, the bindings for the operation arguments

---

[3]As argued in [8], non-deterministic constructs lead to semantical inconsistencies. They are not further discussed here.

have to be extracted from a *Stimulus* that belongs to the *State Transition* for which we perform the evaluation.

The evaluation of the postcondition is basically done in the post-state. The keyword *result* is evaluated according to the binding for the return parameter. The evaluation of *result* is fully analogous to the evaluation of variable expressions.

The evaluation of *@pre* is more complicated. It requires a switch between pre- and post-state, more precisely, we have to manage the different values for properties of each object in the pre- and post-state. Even more complicated, it might be the case that the set of objects itself has changed between pre- and post-state.

In the semantics of OCL described in [68, Annex A], the pre- and post-states are encoded as a set of functions (each function represents an attribute or a navigable association end) that work on a constant domain of objects. Furthermore, there is an extra function that keeps track which of the objects are created in the current state. This formalization has the advantage that the involved objects do not change their identity and thus is very easy to understand. Unfortunately, we were not able to apply this simple model to our semantics due to technical problems caused by the format of graph transformations. In our semantics, the objects in the pre- and post-state have different identities, but each object can be connected with one object from the opposite state via an instance of the *ObjectMap* metaclass. Please note, that for one object there can exist many *ObjectMaps* depending on the number of *StateTransitions* one object is involved in. A pair of related objects represents the same object when we would view a pre-/post-state pair as an evolvement over the same domain. If an object from the pre-state is not related with any object from the post-state, it means that this object was deleted during the state transition. Analogously, objects in the post-state without a counterpart in the pre-state were created.

Fig. 4.25 shows an example. The pre-state consists of two objects with identifiers `p1`, `p2` whose type is a class with name `Person`. The attribute links for the attribute named `age` refer to the value `dv1` and `dv2`, which reside in the package *Data*. In the post-state, the identifiers for objects and attribute links have completely changed. But since object `p1` and `p11` are related by an *ObjectMap* `om1`, we know that `p11` and `p1` represent the same object. Note, however, that the state of this object has changed since the attribute link for attribute named `age` doesn't refer any longer to the value `dv2` but to `dv3`. Since there are no other *ObjectMap*s we can conclude that during the state transition from the pre-state to the post-state, the object `p2` was deleted and object `p21` was created.

The *@pre*-Operator can now be realized as an extension to the already existing
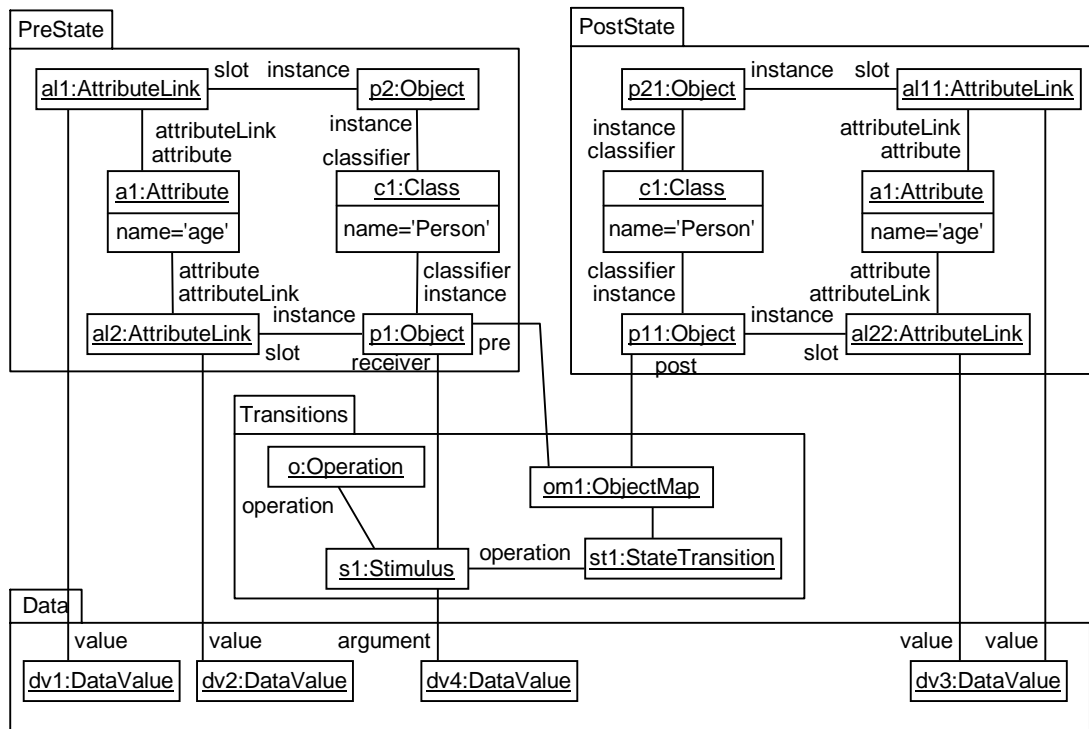
Figure 4.25: Relationship between pre- and poststate

core rules. Note that the official OCL syntax allows to attach *@pre* on every functor, but *@pre* is only meaningful when attached to Navigation Expressions or to an *allInstances*-expression. The most complicated case is the application to *AssociationEndCallExps*.

Figure 4.26 shows the extended evaluation rule for *AssociationEndCallExp* with an object-valued multiplicity (upper limit is 1). The current OCL metamodel encodes *@pre* expressions as operation call expressions of a predefined operation with name *@pre*. The source expression of this operation call expression is exactly that expression, to which the *@pre* operator is attached. The rule reads as follows: First, we wait for the situation in which the source expression of the association end call expression is evaluated (here, to o1). Note that the *Context* requires that o1 is an object from the post-state (what should be always the case). Then, the corresponding object of o1 in the pre-state is searched (o1pre) for which the original rule for evaluation of the association end call is applied (in the pre-state). The object representing the result of the association end call (o2pre) is then projected to the post-state (o2), what is then given back as the result of the evaluation. Note that we didn't specify so far the cases, in which o1 does not have a counterpart on the pre-state (i.e. the source expression oc evaluates to a newly created object) or that the result of the association end call in the pre-state (o2pre) does not have
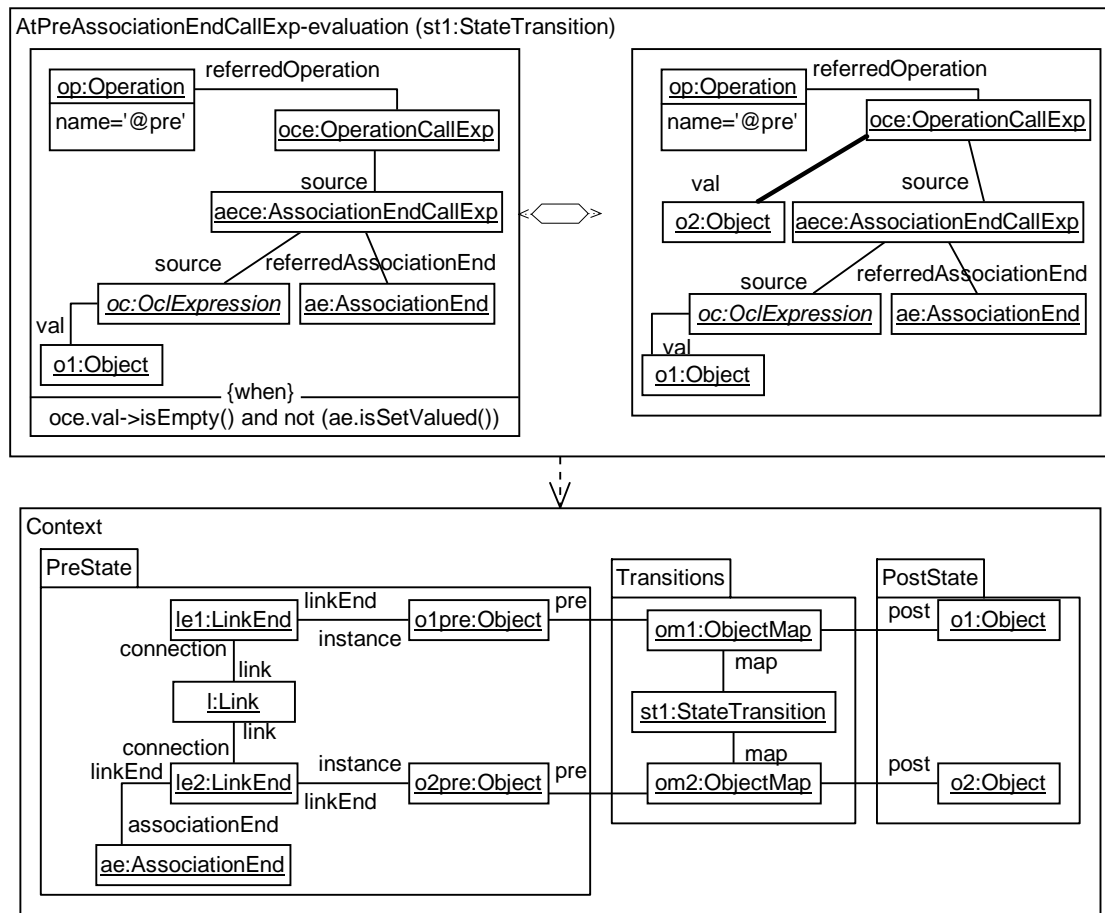
Figure 4.26: Evaluation of @pre attached to an object-valued association end call expression

a counterpart in the post-state (i.e. the object o2pre was deleted during the state transition). This question is answered in the next subsection. Another remark is that evaluation of *AssociationEndCallExp* as specified in Fig. 4.11 is not prevented, but result of the evaluation is ignored when performing evaluation as specified in Fig. 4.26.

4.4.2   Message Sending

Similarly to the evaluation of an operation contract, sending of a message is defined with respect to a transition between two states.

Besides the transitions from pre- to post- states, *StateTransition* metaclass from our metamodel (see Fig. 4.3) is used to capture all messages sent during an operation invocation (described with one pre-/post-condition). All sent messages are represented using metaclass *Stimulus* that relates an operation (invoked with the message), sender of the message, and all the message parameters.

An OCL message sending expression, depicted in its concrete syntax, looks like

```
dest^msg(a, b)
```

where `msg` is an operation related to a *Stimulus*, and `a` and `b` are instances representing arguments of a *Stimulus*.

The evaluation of *MessageCallExp* is performed in relation with one *StateTransition* as shown in Fig. 4.27.

The result of the evaluation will be `true` or `false` depending if specific *StateTransition* has a message sent, that has the same sender as the target from the *MessageCallExp*, referring the same operation, and having the same arguments as arguments for the *MessageCallExp*.

4.4.3   Evaluation to Undefined

The evaluation of OCL expressions to *undefined* is probably one of the most complicated semantic concepts in OCL and has raised many discussions. The value *undefined* has been often mixed in the literature with the null-value (known from Java). Furthermore, questions like *Can an* AttributeLink *refer to undefined in a state? Can a Set-expression be evaluated to undefined? Can a Set-value have elements that are undefined?* are not fully clarified by the official OCL semantics (cmp. also [23]).

First of all, we should note that the value undefined was added to the semantic domain for the sole purpose to indicate exceptional situations during the evaluation.
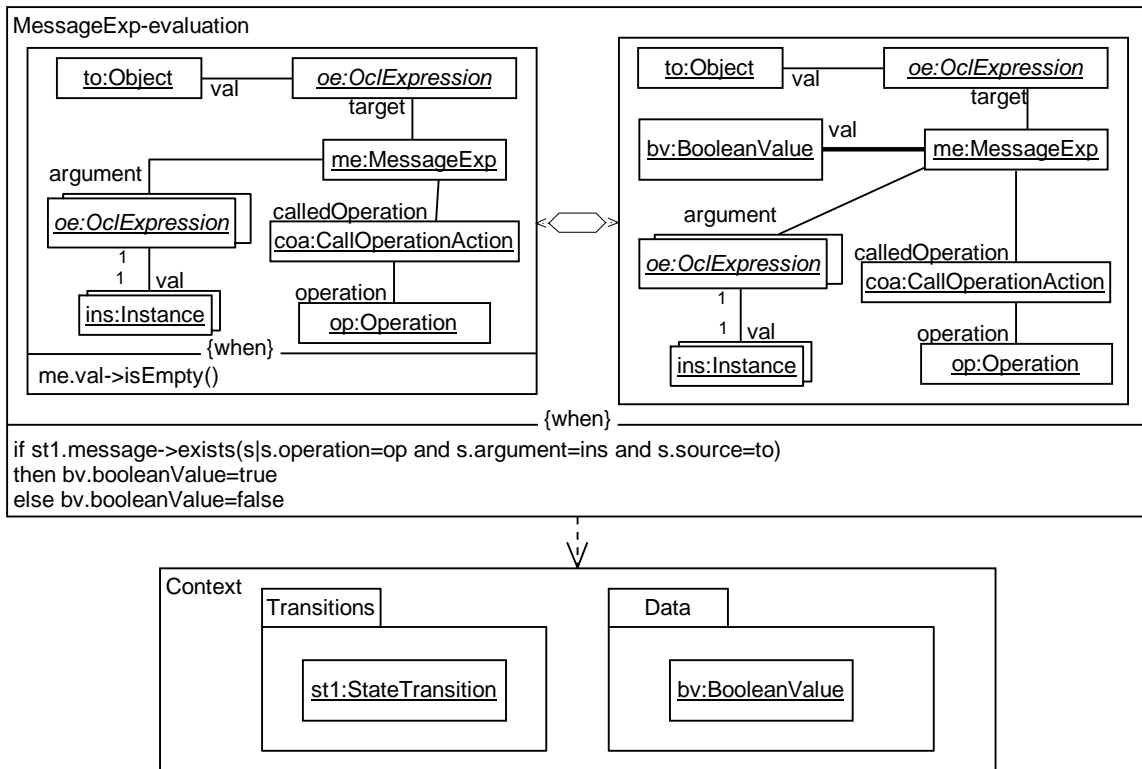
Figure 4.27: Message Expression evaluation

For instance, when an object-valued *AssociationEndCallExp* tries to navigate over
non-existing links or that a cast of an expression to a subclass fails. Thanks to
the pre-defined operation *oclIsUndefined()* is it possible to test if an expression is
currently evaluated to *undefined*; what – together with the exception from strict
evaluation for *and*, *or*, *implies*, *forAll* etc – is a powerful tool to write OCL con-
straints reflecting the intended semantics even in the presence of *undefined* values.

But when is actually an expression evaluated to *undefined*? Strictly speaking,
we had to add for each core evaluation rule a variant of this rule, that captures all
situations in which undefinedness would occur. Fortunately, we have designed our
evaluation rule in such a way, that this additional rule can be generated. Evaluation
to *undefined* is always needed in all cases, in which the pattern given in the Context
does not match with the current situation.

Let's have a look to the rule for *@pre* on association end call expressions (Fig. 4.26).
If for instance the object `o1` (evaluation of the source expression) was newly created
during the state transition so that the pre-post link to an object `o1pre` is missing,
then the whole *@pre*-Expression evaluates to undefined. Likewise, if the correspond-
ing object `o1pre` exists but does not have a link for association end `ae`. Another
reason could be that the link exists but the referred object `o2pre` was deleted dur-

ing the state change. In all these cases, the *@pre*-Expression should be evaluated to *undefined* and these cases have in common that the pattern given in the *Context* does not match.

4.4.4 Dynamic Binding

Dynamic (or late) binding is one of the key concepts in object-oriented programming languages but has been mostly ignored in the OCL literature. Dynamic binding becomes relevant for the evaluation of user-defined queries. Let's assume we have two classes `A` and `B`, the class `B` is a subclass of `A` and the operation `m()` is declared as query with return type `Integer` in `A`.

We have the following constraints:

context A::m( ): *Integer*
body : 5

context B::m( ): *Integer*
body : 7

Let *a* and *b* be expressions that evaluate to an `A` and a `B` object, respectively. The result of the evaluation of `a.m()` is clearly 5. The evaluation of `b.m()` depends on whether or not OCL supports dynamic binding.

The core rule for query evaluation shown in Fig. 4.15 does not realize dynamic binding so far because it doesn't take into account potential inheritance hierarchy in the model. Result of the second rule shown in the figure is value of any body expression (`oceb2`) regardless its context.

For the situation when different bodies can be attached to the same operation (as in our example with classes `A` and `B`) we have to define a strategy for choosing the right body. The most suitable strategy would be to search the inheritance tree and take the body expression defined for the classifier that is the least parent of the source classifier (in the case of `b.m()` that would be the second body constraint 7).

In order to transform the static-binding evaluation rules for queries shown in Fig. 4.15 to a dynamic-binding rule, we had to alter the when-clauses in the LHS of the second rule with the following constraint:

if bm. booleanValue $\diamond$ true **then** i . oclIsTypeOf (OclVoidValue)
else i=op. getRightBody (opce. source . val . oclAsType (Object)
          . classifier $\rightarrow$ any (true))
endif *and*
c. stereotype . name=' pre '

The *getRightBody* query (when multiple inheritance is not allowed) is defined as:

```
context Operation def: getRightBody(cl:Classifier):Instance
=
if self.body.oclAsType(ExpressionInOcl).contextualClassifier
        ->exists(cl) then
op.body->select(b|b.oclAsType(ExpressionInOcl)
                        .contextualClassifier->includes(cl))
    ->any(true).bodyExpression.val
else if cl.getDirectParent()->notEmpty() then
        self.getRightBody(cl.getDirectParent()->any(true))
    else getOclVoidValue()
    endif
endif
```

4.5   Tailoring OCL for DSLs

This section contains an example how our approach for defining the semantics of OCL can be applied for the definition of an OCL-based constraint language that is tailored to a domain specific language (DSL).

As a running example we will use a simple Relational Database Language for which we will define an extension of OCL. Two tables `Person` and `Dog` (see Fig. 4.28) will be used as an example, for which we develop domain-specific constraints. Each table has one primary key (`personID` for the `Person` table and `dogID` for the `Dog` table). In addition, column `ownerID` of table `Dog` has a foreign key relationship with the `personID` column of the `Person` table.

Person		
personID (PK)	name	age
1	John	23
2	Mark	17
3	Steve	45

Dog		
dogID (PK)	breed	ownerID (FK for personID)
1	Doberman	1
2	Bulldog	1
3	Poodle	2

Figure 4.28: An example of relational database

A simple metamodel for relational databases is shown in Fig. 4.29. This language is sufficient to specify the database from Fig. 4.28. Please note that, for the sake of simplicity, we have avoided to introduce database-specific types, but reuse already existing UML/MOF primitive types as types for table columns.
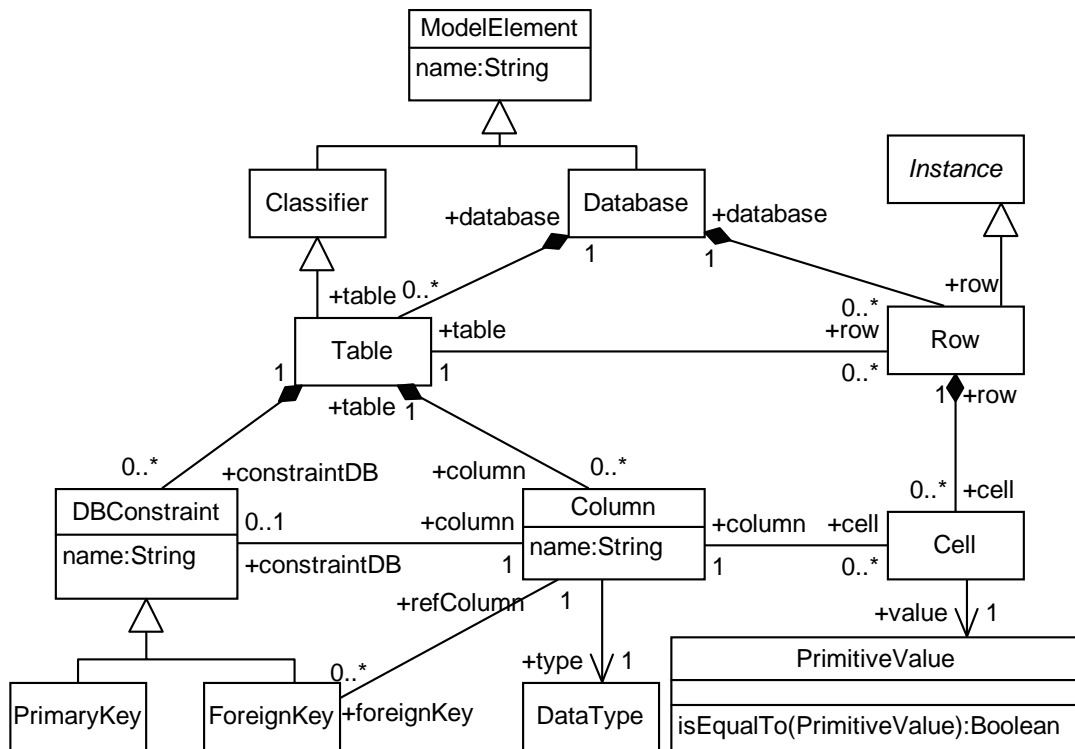
Figure 4.29: Relational database metamodel

When tailoring OCL as a constraint/query language for a domain specific language, it is necessary to introduce additional concepts to OCL in order to capture domain specific constructs. In our example, two constructs require an extension of the OCL metamodel: 1) navigation to a column 2) navigation to a column constrained with a foreign key. The first navigation is applied on a *Row* and has to return the value of the *Column* for this *Row* and the second one has to return a *Row* of the *Table* to which the *ForeignKey* refers.

An example for these two new navigation expressions is the following:

```
Dog.allInstances()−>select(d|d.breed='Doberman')
    −>forAll(dd|dd<=>ownerID.age>18)
```

This example expression uses three specificities of our relational database DSL: Ordinary navigation to columns `breed` and `age`, foreign key navigation to column `ownerID` (foreign key navigation is marked with $<=>$ in order to make it distinguishable from ordinary column navigation), and a call of *allInstances()* on a table.

Another way of expressing the same could be by using only ordinary column navigation and *allInstances()*, but this version is slightly longer:

```
Dog.allInstances()−>select(d|d.breed='Doberman')
    −>forAll(dd|Person.allInstances()
```

$$->\mathrm{any}(\mathrm{p\,|\,p.personID=dd.ownerID}).\mathrm{age}>18)$$

In order to incorporate ordinary and foreign key column navigation into the constraint language, the metamodel for OCL has to be altered. Figure 4.30 shows the part of the Domain Specific Query language that is different from the standard OCL.
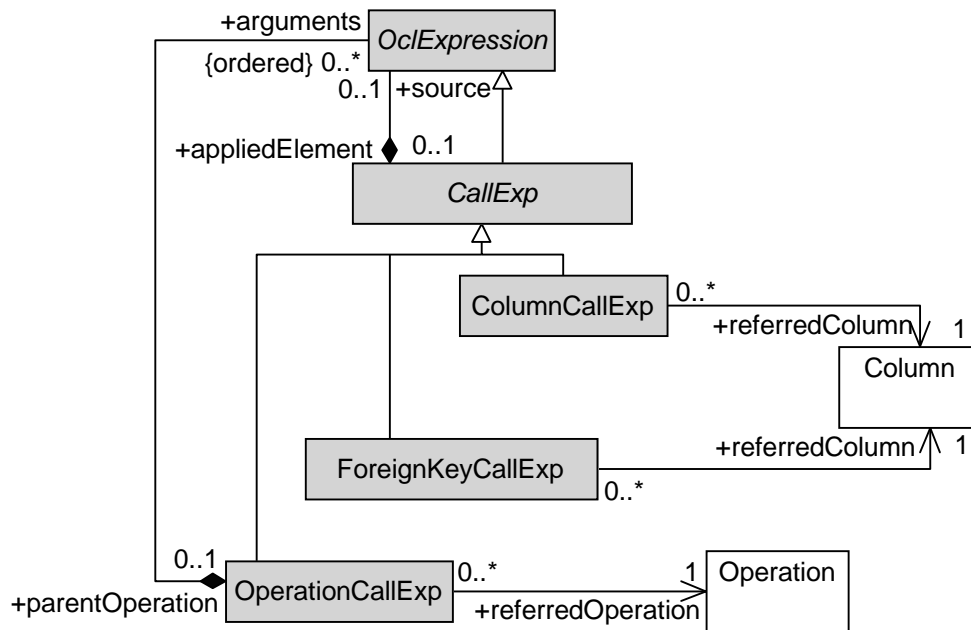
Figure 4.30: DSL navigation expressions

Fig. 4.31 shows the definition of the semantics of column call expressions in form of an evaluation rule. The result of evaluation of such an expression would be value of the *Cell* that belongs to the *Row* that is the source of the expression, and that is referred by the chosen *Column*.

The semantics of *ForeignKeyCallExp* is shown in Fig. 4.32. This rule specifies that the value of the *ForeignKeyCallExp* will be a *Row* r2 for which its primary key column has a *Cell* with the same value as the *Cell* of the source *Row* r for the foreign key column.

A mandatory construct that is needed when specifying the semantics of domain specific query languages and that cannot be reused from standard OCL is the operation call expression for the predefined operation *allInstances()*. This construct operates on model elements that do not exist in UML/MOF and therefore has to be explicitly defined as in Fig. 4.33.

Another way of defining the semantics of OCL expressions on the instance level is by moving (transforming) an OCL expression to an equivalent expression that
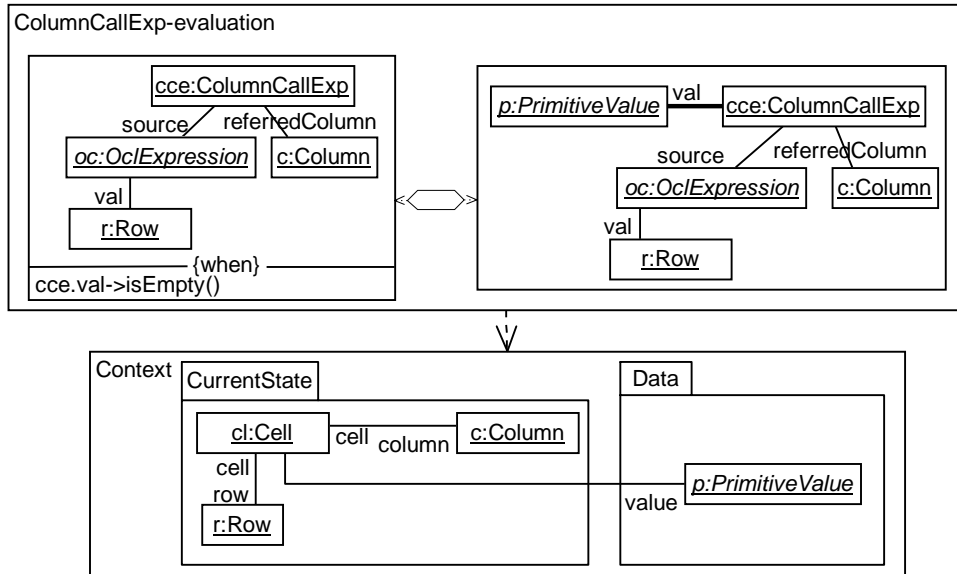
Figure 4.31: Semantics of column navigation specified with QVT
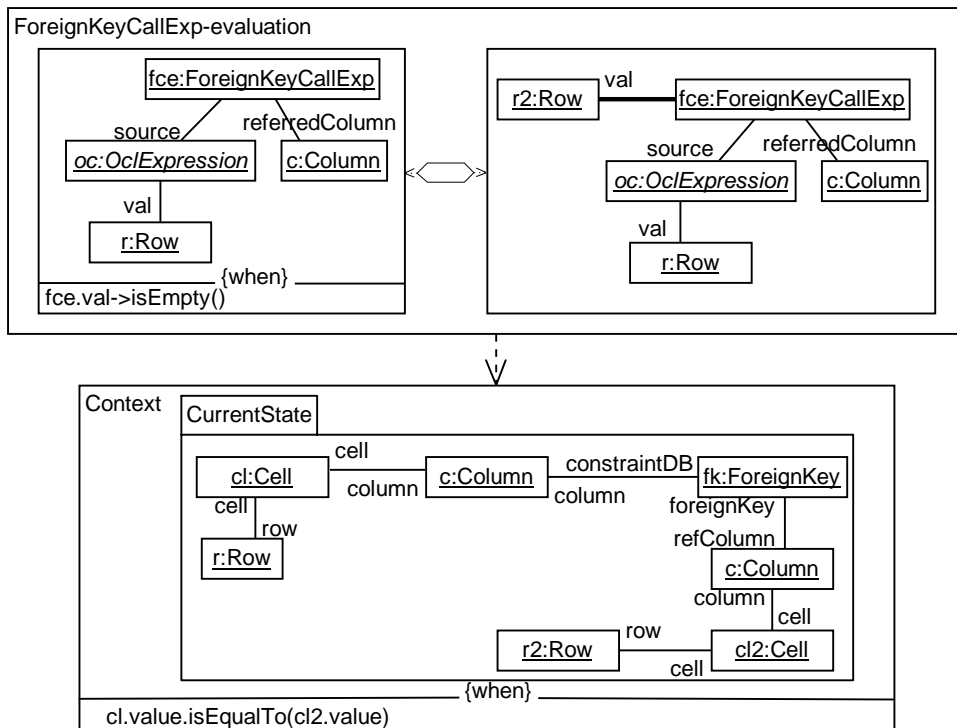
Figure 4.32: Semantics of foreign key navigation specified with QVT
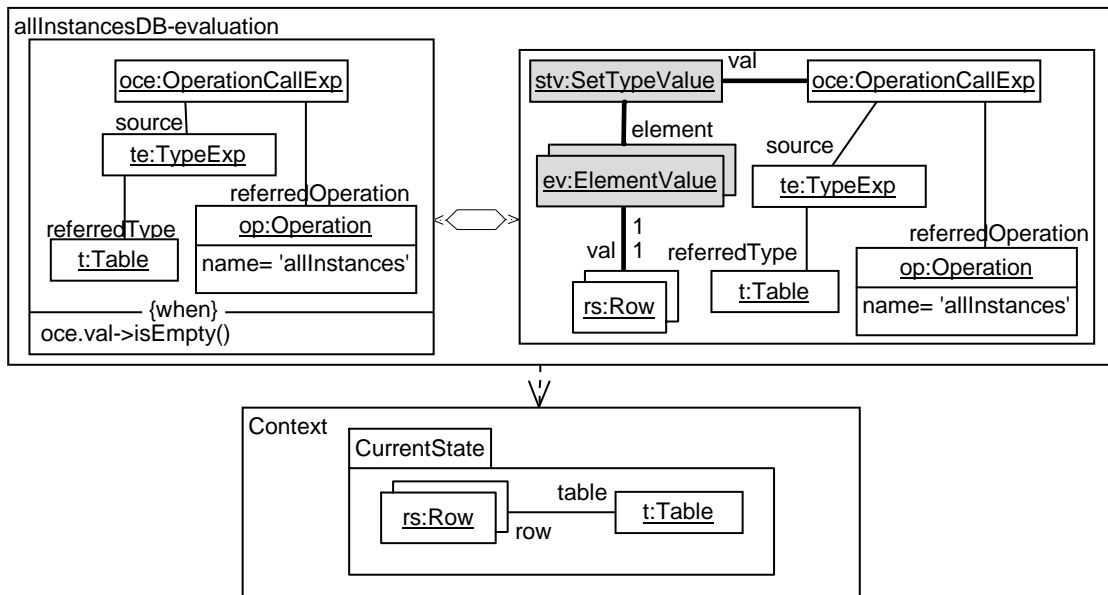
Figure 4.33: Semantics of allInstances Operation Call Expression for relational database

queries the corresponding metamodel. As an example, consider the following *ColumnCallExpression* specified using our concrete syntax:

```
exp.age
```

Please note that the source expression *exp* can be any expression of type *Table*. This short expression in the DSL-specific version of OCL can be emulated by the following expression, which exploits the metalevel. However, this expression is clearly much more complicated.

```
Column.allInstances()->select(col|col.name='age' and
col.table=exp.table).cell
        ->select(cc|cc.row=exp)
      ->any(true)
```

4.6   Related Work

The work described in this chapter combines techniques and results from different fields in computer science: logics, precise modeling with UML/OCL, model transformation, modeling language design. For this reason, we separated related work into three categories.

4.6.1 Approaches to Define the Semantics of OCL

There are numerous papers and some dissertations that propose a formal semantics for complete OCL or for a fragment of it, e.g., [72, 73, 71, 29, 33, 41, 48, 7, 30] and, recently, [22]. Many other papers have identified inconsistencies in the official OCL semantics and contributed in this form to a better understanding of OCL's concepts, e.g., [34, 40, 8, 3, 23].

In the subsections 4.6.1 and 4.6.1 we compare *the technique*, which we have been used for the semantics definition, with that of other approaches. We restrict ourselves to a comparison with the two semantics given in the OCL language standard.

Official OCL Semantics: Informative

Annex A of [68] presents a set-theoretical semantics for OCL, which goes back of the dissertation of Mark Richters [71]. This semantics has been marked in the OCL standard as *informative*.

The *semantic domain* of OCL is formalized by the notion of *system state* (a triple consisting of the set of objects, the set of attribute values for the objects, and the set association links connecting objects) and the *interpretation of basic types*. The notion of *system state* is defined on top of the notion of *object model*. What was formalized by Richters as *system state* is known in UML terminology as *object diagram*, an *object model* corresponds to a *class diagram*.

In our approach, the class and object diagrams are directly formalized by their metamodels and the interpretation of basic types is covered by the package *Values* of the OCL metamodel. All three metamodels, on which our approach relies, are part of the official language definition for UML/OCL. However, there is one important difference to Richters semantics: In Richter's approach, one object can be in multiple states, whereas in our approach, states are represented by object diagrams which can never contain objects with the same identity. We solved this problem by introducing `ObjectMap` objects (cmp. Sect. 4.2.1) whenever two different states are involved in the evaluation of OCL constraints (e.g., post-conditions). Note that a set of `ObjectMap` objects referring to a pre-state and a post-state can also encode the information which of the objects were created/deleted during the transition from pre- to post-state. In Richter's approach, the lifetime of an object is encoded by the function σ_{CLASS}.

The evaluation of OCL expressions is formalized in Richter's semantics by an *interpretation function* $\mathcal{I}$, which is defined separately for each type of OCL expression. The definitions for $\mathcal{I}$ are based on the above mentioned ingredients of the

semantics *object model, system state, interpretation of basic types*. In our approach, the interpretation function $\mathcal{I}$ is implicitly given by QVT rules, which are based on the metamodels for class diagrams, object diagrams, and on the *Values* package.

One of the most interesting details when comparing the formalization of expression evaluation is the handling of pre-defined functions. Following Richter, predefined functions like `=`, `union`, `concat`, etc., are interpreted by their mathematical counterparts, e.g. $\mathcal{I}(=_t)(v_1, v_2) = true$ if $v_1 = v_2$ and $v_1 \neq \bot$ and $v_2 \neq \bot$. Otherwise stated, the semantics of some operations of the object language (OCL) is reduced to the semantics of some operations of the meta language (mathematics). The same holds in our case, the semantics of operation '=' of the object language (OCL) is reduced to the semantics of the operation '=' in the metalanguage (QVT) (see Sect.4.3.3).

In both cases, it has to be assumed that the semantics of the metalanguage has been already defined *externally* (cmp. also [47]). In case of Richter's semantics, one could refer to textbooks introducing mathematics. In case of our semantics, we can refer to the implementation of QVT engines, which actually map QVT rules to statements in a programming language, e.g. Java.

Official OCL Semantics: Normative

The semantics described in [68], Sect. 10 *Semantics Described Using UML* is called *normative OCL semantics* and shares the same main goal as our approach: to have a semantics description of OCL, which is seamlessly integrated into the other artifacts (metamodels) of OCL's language definition. However, there are important differences.

The normative semantics defines a package *Values* to encode pre-defined data types and system states. We tried to align our approach as much as possible with this *Values* package (e.g. `NameValueBinding`), but some details differ. Most notable, as already mentioned in the comparison with Richters' semantics, our states never contain identical objects. The normative OCL semantics insists on keeping object identities across states, but this yields to a quite complicated encoding of attribute value and links, which have to be kept separated from objects (see metaclass `LocalSnapshot`). Moreover, the normative semantics encodes exactly one system trace (metaassociation `pred--succ` on `LocalSnapshot`), while in our approach state transitions are modeled explicitly by a new metaclass `StateTransition`.

The evaluation of OCL expressions is formalized in the normative semantics by so-called *evaluation classes*. For each metaclass from the metamodel of OCL's ab-

stract syntax, there is exactly one corresponding evaluation class, e.g. `AttributeCallExpEval`. Evaluation classes are complemented by a number of invariants, whose purpose is to specify the evaluation process. In many cases, the invariants can be mapped to exactly one QVT rule in our approach. For example, there is for each evaluation class one invariant specifying the propagation of the current binding of variables (called `Environment` in the normative semantics) to subexpressions, what corresponds to our variable binding propagation rules described in Sect. 4.2.3.

The normative semantics has been also the starting point for a semantics formalization given by Chiaradía and Pons in [31]. They alter the OCL semantics' metamodel by introducing visitor pattern in order to reduce the duplication of information in *AbstractSyntax* and *Evaluations* packages of OCL metamodel. Contrary to our approach, they use UML sequence diagrams to express the semantics of OCL expressions.

4.6.2 Approaches to Define Language Semantics by Model Transformations

The application of model transformations (or, more general, graph transformations) for the purpose of defining language semantics is not a new idea. However, we are only aware of one paper, which applies this technique for the definition of the semantics of OCL. Bottoni et al. propose in [21] a graphical notation of OCL constraints and, on top of this notation, some simplification rules for OCL constraints. These simplification rules specify implicitly the evaluation process of OCL expressions. However, the semantics of OCL is not developed as systematically as in our approach, only the simplification rules for *select* are shown. Since [21] was published at a time when OCL did not have an official metamodel, the simplification rules had to be based on another language definition of OCL.

For behaviorial languages, Engels et al. define in [38] a dynamic semantics in form of graph transformation rules, which are similar to our QVT rules. As an example, the semantics of UML statechart diagrams is presented.

In [90] Varró points out the abstraction gap between the "graphical" world of UML and mathematical models used to describe dynamic semantics. In order to fill this gap he uses graph transformation systems to describe visual operational semantics. Application of the approach is demonstrated by specifying semantics of UML statecharts.

Stärk et al. define in [80] a formal operational semantics for Java by rules of an

Abstract State Machine (ASM). The semantic domain of Java programs is fixed by defining the static structure of an appropriate ASM. The ASM encodes furthermore the Abstract Syntax Tree (AST) of Java programs. As shown by our motivating example in Sect. 4.2, there are no principal differences between an AST and an instance of the metamodel. Also, ASM and QVT rules are based on the same mechanisms (pattern matching and rewriting).

4.6.3　Other Related Work

An interesting classification of OCL language concepts was developed by Chiorean et al. in [32]. In this paper, OCL language constructs are classified according to their usage in different domains, such as *Transformations*, *Assertions*, and *Commands*. In our approach, we have concentrated on what is called *core OCL* in [32], but it would be definitely worthwhile to investigate the other domains as well.

Kolovos et al. define in [54] a navigation language for relational databases that is similar to our language defined in Sect. 4.5. They use the metalanguage *EOL* (which is based on OCL) to define the result of evaluation of new expressions like column navigation.

4.7　Conclusions

We have developed a metamodel-based, graphical definition of the semantics of OCL. Our semantics consists of a metamodel of the semantic domain (we have slightly adapted the existing metamodels from UML1.x), and a set of transformation rules written in an extension of QVT that specify formally the evaluation of an OCL constraint in a snapshot. To read our semantics, one does not need advanced skills in mathematics or even knowledge in formal logic; it is sufficient to have a basic understanding of metamodeling and QVT. The most important advantage, however, is the flexibility our approach offers to easily create an OCL dialect. Since the evaluation rules can directly be executed by any QVT compliant tool, it is now very easy to provide tool support for the new dialect of OCL.

In the Chapter 3 we have formalized a catalog of refactoring rules using the QVT formalism. Each of the rules is syntax preserving because after any application refactored model remains syntactically well-formed. This, so called, syntax preservation is just one of the properties of refactoring rules. Another property of refactoring rules is so called *semantics preservation*. We call a rule *semantics preserving* if in any given snapshot the evaluation of the original OCL constraint and the refactored

OCL constraint yields to the same result (in fact, this view is a simplified one since the snapshots are sometimes refactored as well). To argue on semantical correctness of refactoring rules, it has been very handy to have the OCL semantics specified in the same formalism as refactoring rules, in QVT. In the next chapter we define formal criterion for semantics preservation and, by using OCL semantics specified in this chapter, show that our refactoring rules specified in Chapter 3 are semantics preserving.

Chapter 5

Semantics Preservation of Refactoring Rules

In this chapter, we present a simple criterion and a proof technique for the semantic preservation of refactoring rules that are defined for UML class and object diagrams, and OCL constraints. Our approach is based on a novel formalization of the OCL semantics in form of graph transformation rules, given in the Chapter 4.

The content of this chapter was partially published in Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006 [9].

5.1   Introduction

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 [60]). This chapter concentrates on proving semantic preservation, while the syntactic preservation was discussed in Chapter 3.

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 [68, Annex A] but this semantics is clumsy when arguing on the semantic preservation of a graphically defined refactoring rule. For this reason, we have proposed in Chapter 4 a novel

formalization of OCL's semantics in form of graph transformation rules. In this chapter, we give a simple criterion for the semantic equivalence of two UML/OCL models and show how this criterion is met by the refactoring rules specified in Chapter 3.

The rest of the chapter is structured as follows. Section 5.2 defines a criterion for semantic preservation. Section 5.3 extends refactoring rule *MoveAttribute* (specified in Chapter 3) to support refactoring of UML object diagrams. The section closes with two, more complicated versions of *MoveAttribute* whose formalization requires the usage of semantic preconditions. Section 5.4 contains proofs of semantic preservation for refactoring rules presented in Chapter 3. Moreover, this section specifies necessary extensions of refactoring rules from Chapter 3 to cover UML object diagrams. Section 5.5 contains related work. Section 5.6 concludes the chapter.

5.2   A Correctness Criterion for Semantic Preservation

Semantic preservation, intuitively, means that source and target model express 'the same'. Established criterion for the refactoring of implementation code, where 'the same' usually means that the observable behavior of original and refactored program coincide, cannot be used for UML/OCL models, simply because the UML class and object diagrams, together with OCL constraints, model the static structure of the system.

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. 5.1(b) for illustration). In order to solve this problem, one has to bridge these structural differences of the model instances.

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

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.

5.3 Formalization of Semantic Preserving Refactoring Rules

Research on refactoring has focused so far on implementation code but, as it is shown in Chapter 3, many refactoring rules for (object-oriented) implementation languages can be adapted to UML class diagrams and OCL constraints.

Figure 5.1(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 chapter) 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
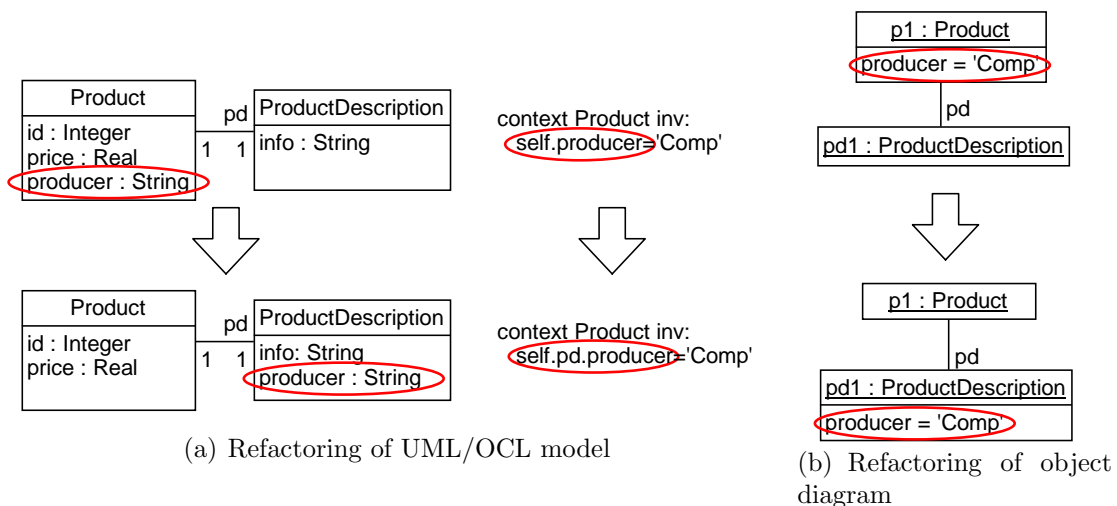
Figure 5.1: Application of *MoveAttribute* on an example

Refactoring of UML class diagrams and propagation of the refactoring to its OCL constraints (as shown in Fig. 5.1(a)) could potentially make corresponding UML object diagrams invalid. A solution for this problem is the propagation of UML class diagram refactorings to all corresponding object diagrams that represent

possible instantiations. On Fig. 5.1(b) is shown the necessary change of UML object diagram, once the corresponding UML class diagram is refactored. If `producer` is moved from class `Product` to `ProductDescription`, then all *AttributeLinks* and their values that correspond to `producer` have to be moved from instances of class `Product` to corresponding instances of class `ProductDescription`.

5.3.1 Formalization of the simple form of *MoveAttribute*

In Chapter 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. One of the formalized rules is *MoveAttribute*. This refactoring is split into two graph transformation rules, where the second one (see Fig. 3.14), which describes changes on OCL, extends the first rule, which formalizes the changes on the UML class diagram (see Fig. 3.13).
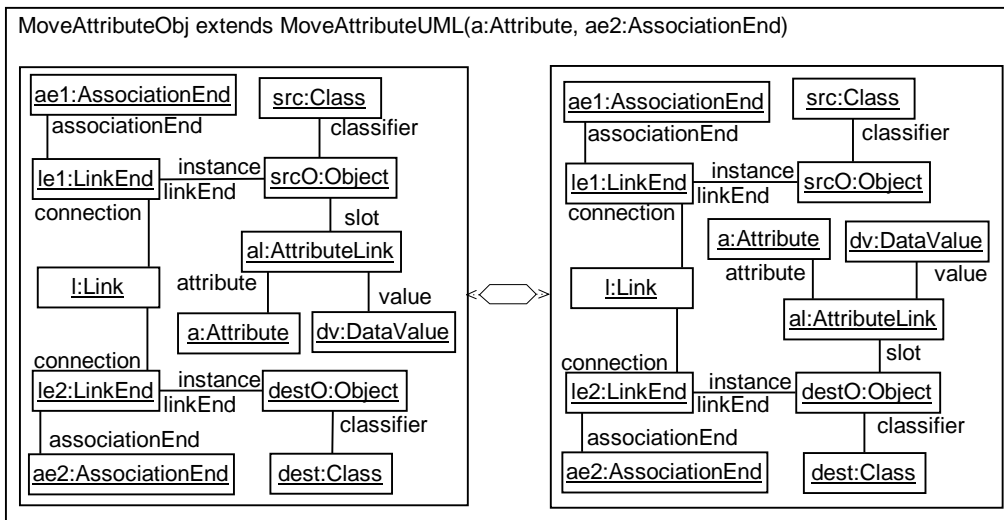
Figure 5.2: Influence of *MoveAttribute* on object diagrams

As an addition to the transformation rules for UML class diagram and attached OCL constraints, Fig. 5.2 shows transformation rule that specifies refactoring part for UML object diagrams. This rule is an extension of the class diagram refactoring part and specifies that when an attribute is moved from a source to a destination class, all attribute links that correspond to the moved attribute change their owner from source object to the destination one. The source and destination objects represent instance of the source and destination class, respectively.

5.3.2 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. 5.2). Before we present in Subsection 5.4.1 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. Please note that refactoring rules as specified in Chapter 3 do not take into account any possible instantiation of refactored class diagrams.

We discuss in the remainder of this subsection 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, and 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.

Multiplicities *–1

(a) Refactoring of UML/OCL model

(b) Refactoring of object diagram
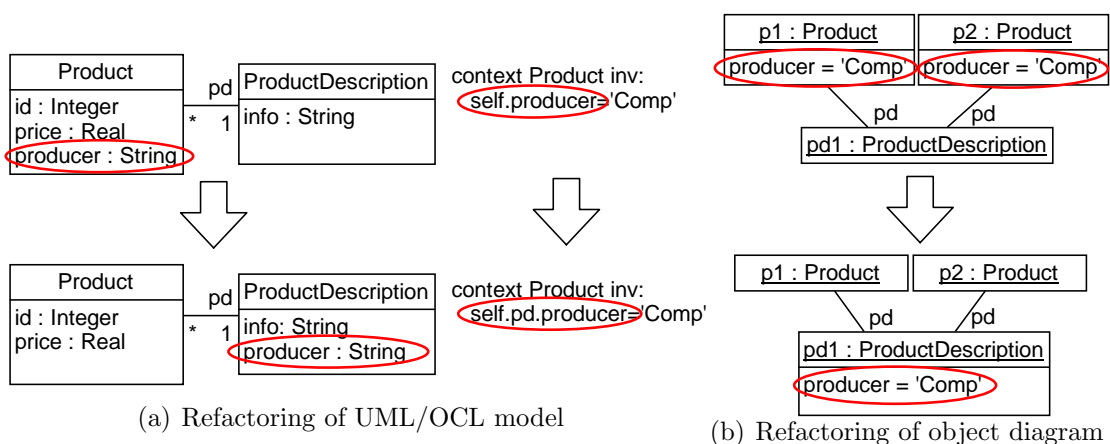
Figure 5.3: Example refactoring if connecting association has multiplicities *–1

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., ob-
jects of the source class), which are connected to the same destination object (in
Fig. 5.3, 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. 3.13 to
the version shown in Fig. 5.4.
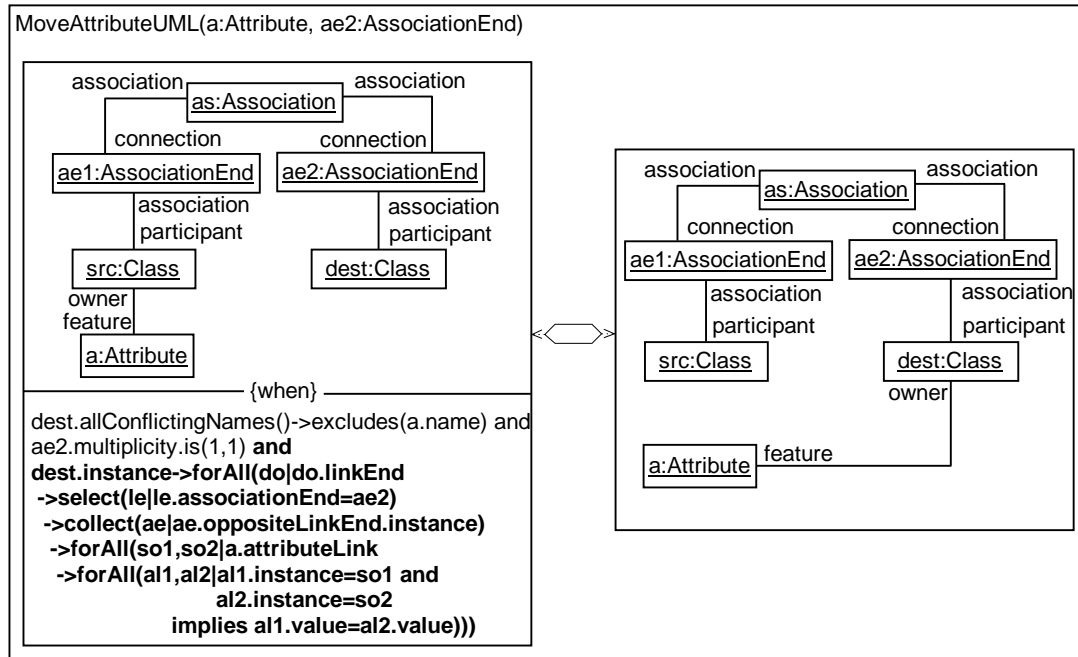
Figure 5.4: The new version of *MoveAttribute* refactoring rule for UML class dia-
grams

This *semantic precondition* seems, at a first glance, to be put at a wrong place.
Isn't a refactoring of UML/OCL models 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.

Figure 5.5: Object diagram part 1 of refactoring rule if association has multiplicities *–1
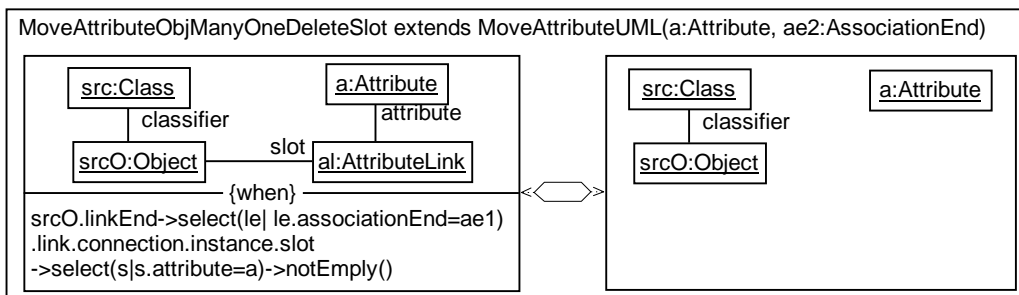
Figure 5.6: Object diagram part 2 of refactoring rule if association has multiplicities *–1

The object diagram part of the refactoring 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 shown in Fig. 5.5 and Fig. 5.6 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 shown in Fig. 5.7 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.

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
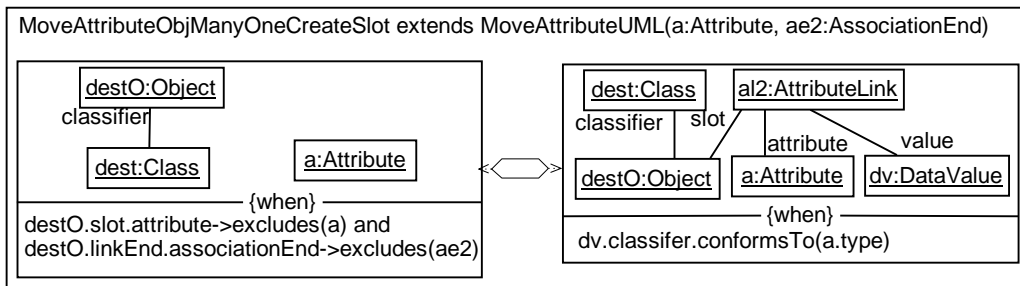
Figure 5.7: Object diagram part 3 of refactoring rule if association has multiplicities *–1

(a) Refactoring of UML/OCL model

(b) Refactoring of object diagram
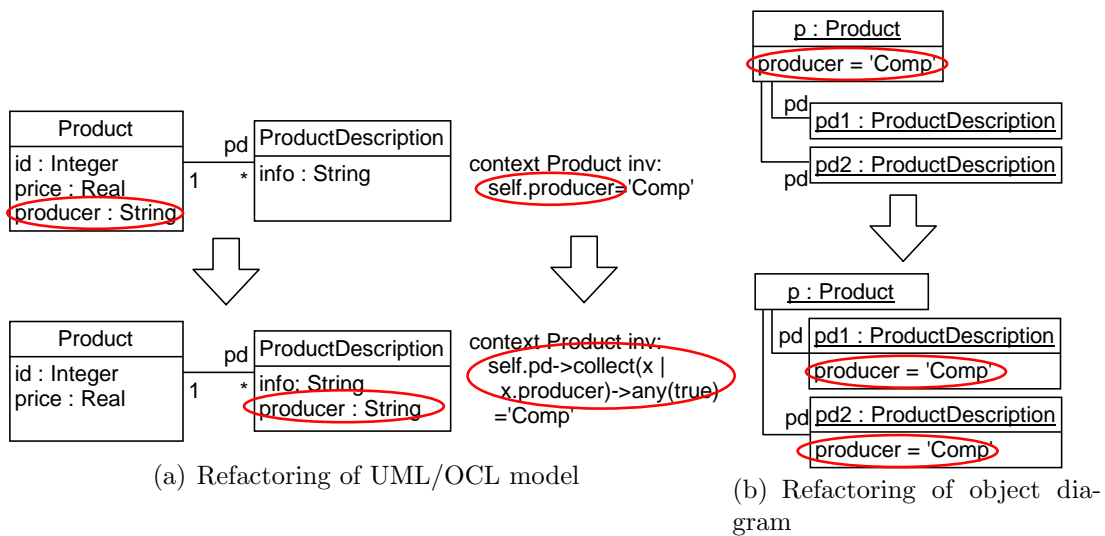
Figure 5.8: Example refactoring if connecting association has multiplicities 1–*

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. 5.8). 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 shown in Fig. 5.9.

The object diagram part of the refactoring rule is changed as shown by the two rules. The first rule shown in Fig. 5.10 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 shown in Fig. 5.11 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.

The third rule in Fig. 5.12 shows the OCL part of the refactoring rule. If the
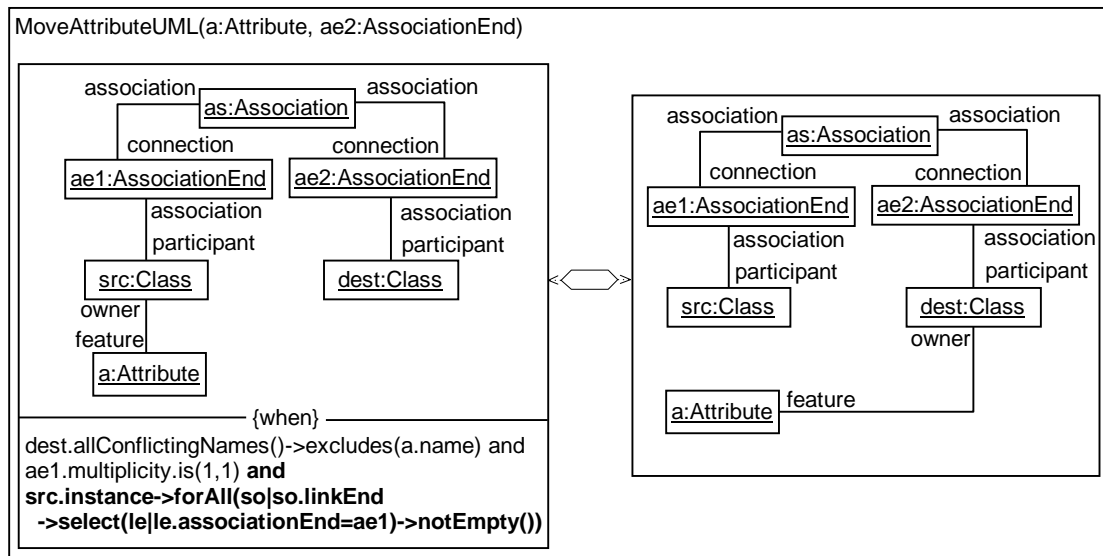
Figure 5.9: *MoveAttribute* refactoring rule for UML class diagrams when multiplicities are 1-*

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.

5.4 Proving Semantics Preservation of Refactoring Rules

5.4.1 *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 [68]. 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.
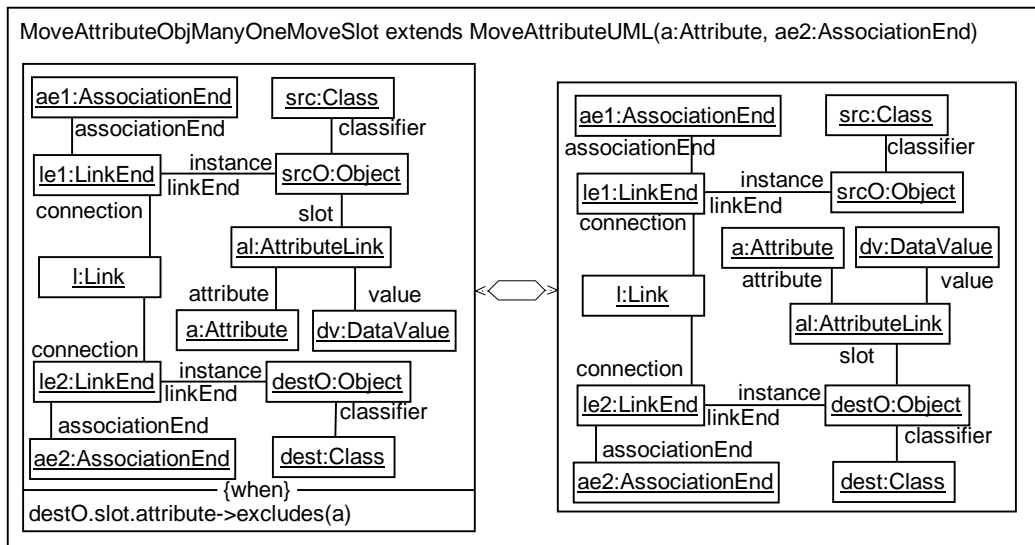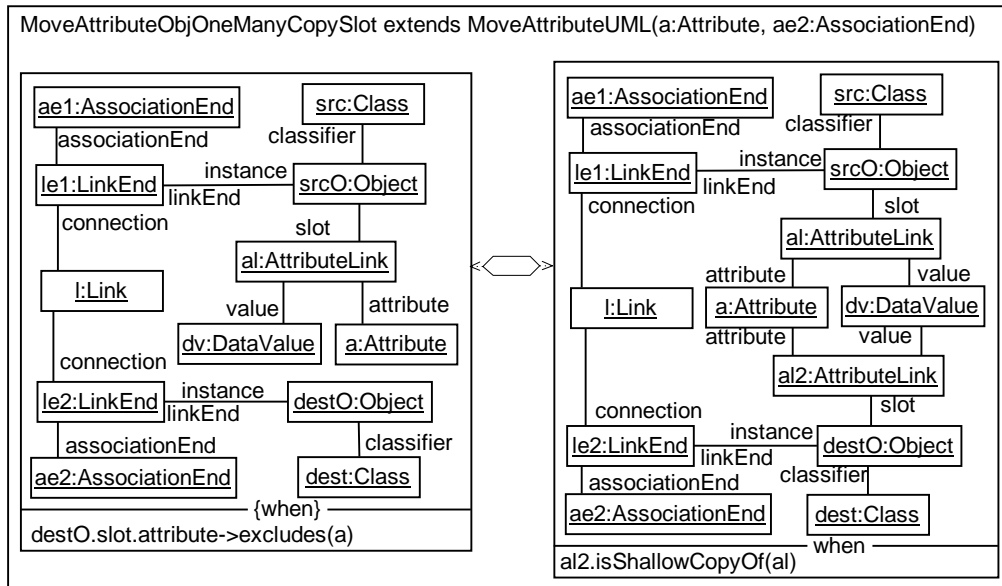
Figure 5.10: Object diagram part 1 of refactoring rule if connecting association has multiplicities 1–*
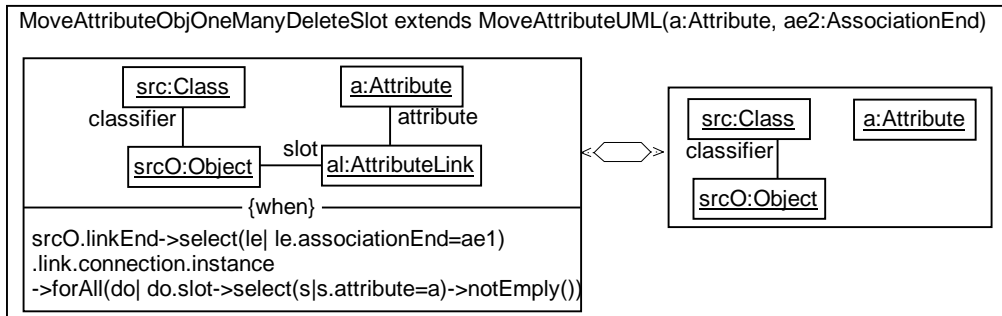
Figure 5.11: Object diagram part 2 of refactoring rule if connecting association has multiplicities 1–*

For this reason, in Chapter 4 we have proposed an alternative formalization of *eval* in form of graph transformation rules.

Based on this formalization we can state the following theorem:

Theorem 5.4.1 (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
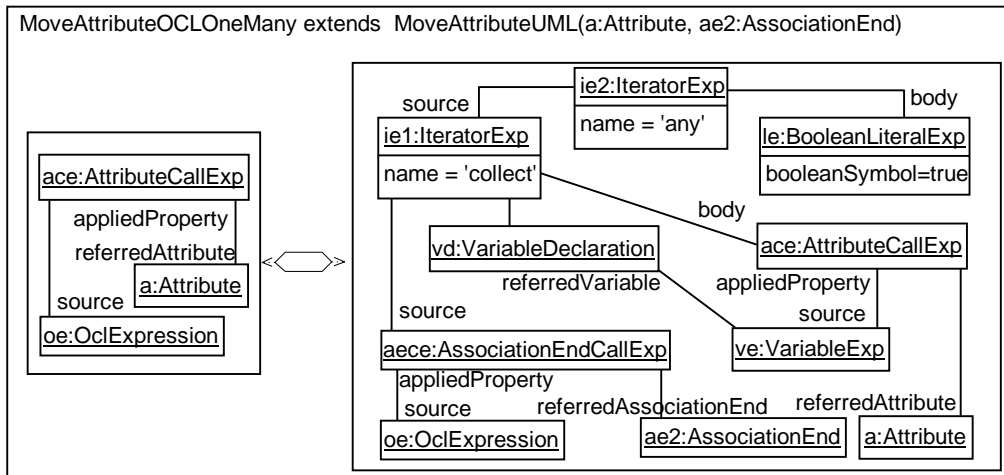
Figure 5.12: OCL part of refactoring rule if connecting association has multiplicities 1–*

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* (see Fig. 4.10), $oe.a$ 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* (see Fig. 4.11), 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.

5.4.2 *MoveAssociationEnd* is Semantic Preserving

In order to be able to reason about semantics preservation of *MoveAssociationEnd* we must extend the rule specified in Chapter 3 so that it includes refactoring of object diagrams. The extension is shown in the Fig. 5.13. Note that this extension covers only the case when an association end is moved over an 1-1 association. The cases when association end is moved over association with different multiplicities than 1-1 are analogous to the *MoveAttribute* rule and will be omitted here.

The upper part of Fig. 5.13 extends the refactoring rule for UML class diagrams (shown in Fig. 3.19) and specifies that whenever an associationEnd is moved from a source to a destination class, every corresponding linkEnd is moved from an object that instantiate the source class to an object that instantiates the destination class.

The lower part of Fig. 5.13 does not have a counterpart in *MoveAttribute* refactoring rule. It specifies that if the owner of the link, whose linkEnd is moved, is an

object of the source class then the new owner of the link becomes an object of the destination class.
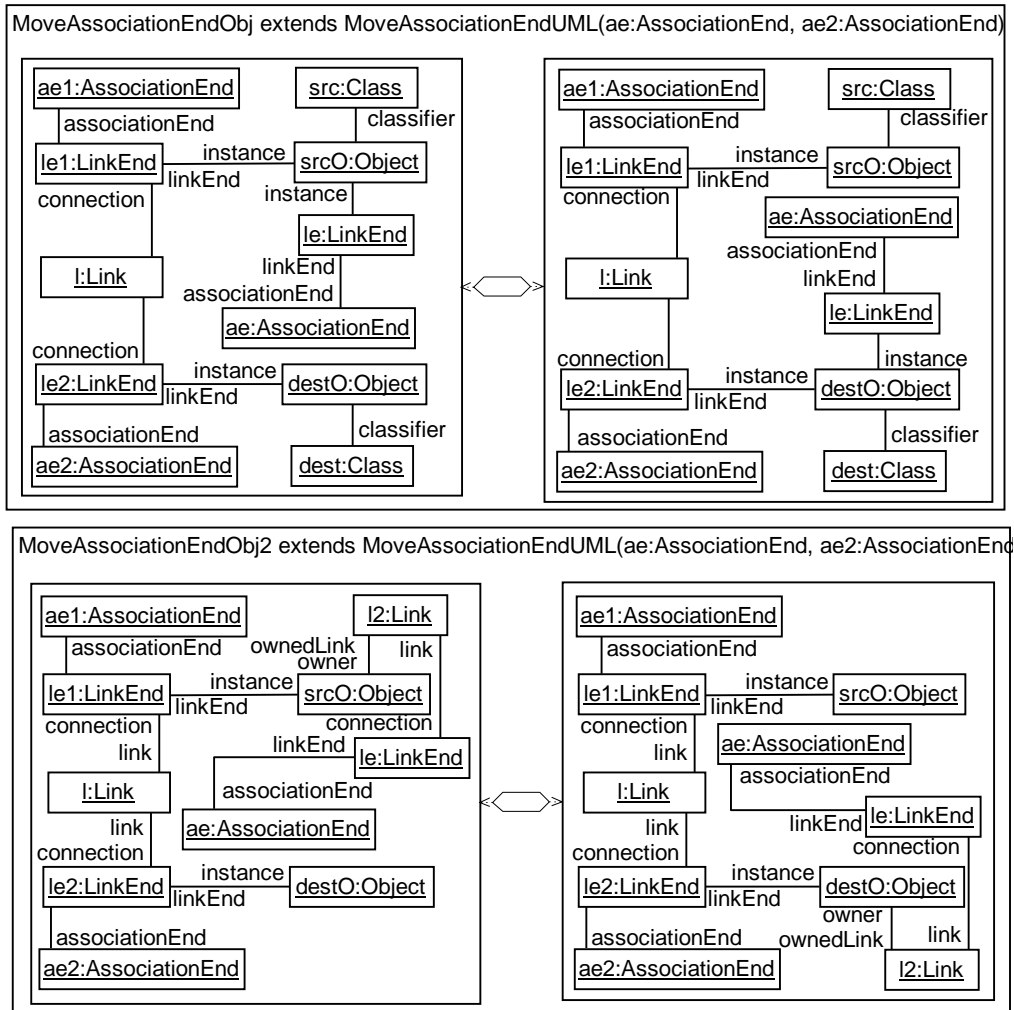
Figure 5.13: Influence of *MoveAssociationEnd* on object diagrams

The proof that *MoveAssociationEnd* refactoring is semantic preserving is almost identical to the proof for the *MoveAttribute* and will be omitted here.

5.4.3   Semantic Preservation of the *PushDown* Rules

In Chapter 3, when performing *PushDown* refactoring rules, it was only checked if attached OCL expressions have expressions that conform to the superclass from which we want to push an element down in the hierarchy. The rules themselves didn't have any impact on OCL attachments. When taking into account possible object diagrams, the same reasoning can be applied. The refactoring will not be executed if any of the object diagrams contains an instance of the superclass referring

to the element that is to be pushed down.

When taking object diagrams into account, the refactoring rule shown in Fig. 3.7 has to be rewritten as shown in Fig. 5.14. Note that the difference between the previous version of the rule and this one is marked using bold face.
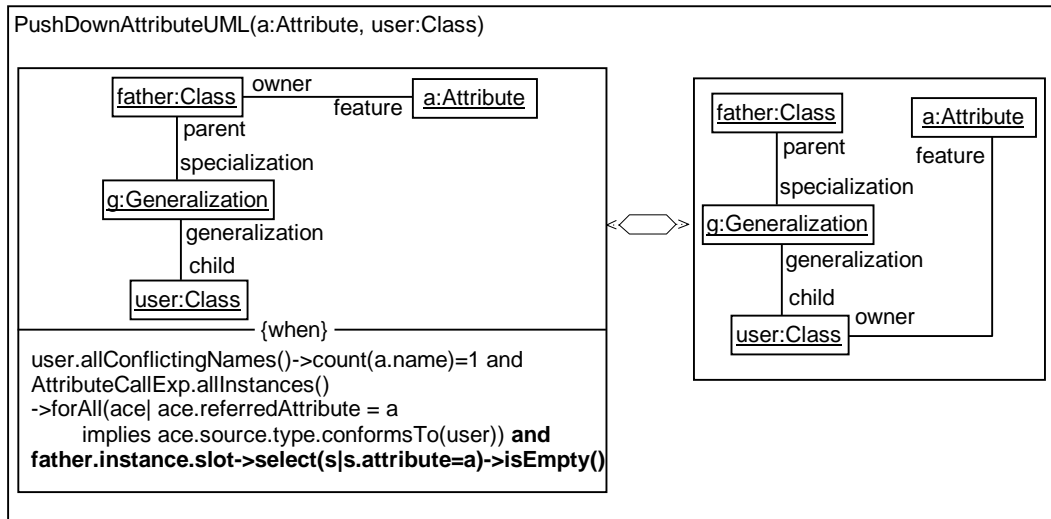
Figure 5.14: Modified version of *PushDownAttribute* refactoring rule

As long as the refactoring rule, when the application condition is satisfied, does not influence neither OCL constraints, nor object diagrams we can conclude that the rule is semantic preserving because whenever $constr_o = constr_r$ and $od_o = od_r$ hold, also $eval(constr_o, od_o) = eval(constr_r, od_r)$ holds.

5.4.4   Semantic Preservation of the *Rename* Rules

Renaming any of the elements from UML class diagram influences the textual notation of all OCL expressions that access that element as well as the object diagram concrete syntax representation. However, instances of OCL part and object diagram part of the metamodel are not altered. Therefore there are no refactoring rules for OCL and object diagrams.

The refactoring rule *RenameAttribute* shown in Fig. 3.2 renames an attribute a that can be accessed by *AttributeCallExp* expressions and is related to slots of objects. The evaluation rule *EvalAttributeCallExp* shown in Fig. 4.10 is influenced with this refactoring because the upper and the lower part of the evaluation rule refer to the attribute a whose name is changed. Regardless of this change, in Fig. 4.10 we can see that the result of the evaluation, *DataValue* d is not influenced by the change, and that evaluation of *AttributeCallExp* remains the same, i.e. $eval(constr_o, od_o) = eval(constr_r, od_r)$.

Similar reasoning can be applied in case of other types of *Rename* refactoring, *RenameAssociationEnd*, *RenameClass*, and *RenameOperation*.

5.4.5   Semantic Preservation of the *Extract* Rules

ExtractClass and *ExtractSuperclass* rules just introduce a new class to the model, without any interference with already existing ones like existing OCL expressions or object diagrams, as assured with condition clauses in Fig. 3.10 and Fig. 3.11. Therefore all OCL annotations and all object diagrams remain the same before and after refactorings.

This drives us to conclusion that $eval(constr_o, od_o) = eval(constr_r, od_r)$ because $constr_o = constr_r$ and $od_o = od_r$.

5.4.6   Semantic Preservation of the *PullUp* Rules

As it is shown in chapter 3, *PullUp* refactoring rules don't influence attached OCL constraints but only widen the application of the attribute/associationEnd that is moved to the superclass.

When a *PullUp* refactoring is performed the full descriptor of objects (see Section 2.5.4.4 of [64]), that are instances of the source class, remain the same. In other words there is no influence on object diagrams.

From this we can conclude that, similarly to *Extract* rules, $eval(constr_o, od_o) = eval(constr_r, od_r)$ because $constr_o = constr_r$ and $od_o = od_r$.

5.5   Related Work

In his seminal work [69], Opdyke gives a catalog of refactoring rules for C++ programs. Opdyke 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 [61]).

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' is 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. [49].

In [91], Wachsmuth presents an approach for automatic metamodel evolution and gives two model preservation criteria. The first criterion, so called *semantics-preservation* is based on relations between two metamodels, source and destination one. The second criterion, *instance-preservation*, is used when reasoning about possible relations between two models whose corresponding metamodels are non-transformed, and transformed one. Similarly to our approach, this work distinguishes two levels of transformations: 1) model transformations (so called *metamodel adaptations*), and 2) instance transformations (so called *model co-adaptations*) that are dependent on the first level. Contrary to our work, their preservation criteria are applied on not only refactoring transformations but on arbitrary model element constructions and destructions as well.

Contrary to some authors, we allow object diagrams also to be transformed when applying a refactoring rule. We believe that our definition of semantic correctness gives more freedom in performing refactorings and allows a wider spectrum of refactoring rules to be applied on a UML class diagram.

5.6 Conclusions

While the Model Driven Architecture (MDA) initiative of the OMG ([63]) 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 criterion 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 chapter, we defined and motivated 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 refactoring rules specified in Chapter 3 preserve the semantics

according to our criterion. Our proofs refer 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 specified in Chapter 4.

Chapter 6

Conclusions

This chapter provides concluding remarks. It also indicates some of directions in which research on model refactoring, and model synchronization could be pursued.

6.1 Summary

The central element of this thesis was refactoring of UML/OCL models. In Chapter 3 we have presented a catalog of refactoring rules for UML class diagrams annotated with OCL constraints. We have specified all refactoring rules using a QVT inspired formalism, in a clear and readable manner. For each refactoring rule that can be applied on UML class diagrams we have investigated and formalized any potential impact on attached OCL constraints. For every UML refactoring rule that has impact on OCL constraints we have specified, using the same notation, how the constraint has to be altered in order to preserve its syntax. For one refactoring rule we have shown how syntax preservation can be proven. This was achieved using the KeY tool.

Refactorings are just one subset of possible model transformations applicable in the case of UML/OCL diagrams. With QVT transformation rules it is possible to specify any other structural change like arbitrary element creation or deletion, though it wouldn't be possible to prove semantics preservation property for arbitrary model transformation rules.

Semantics issues are discussed in Chapter 4. In this chapter we have specified the semantics of OCL using the same formalization as for defining refactoring rules. We have defined the semantics of OCL as a set of evaluation rules given in the form of QVT transformations. The usage of QVT rules led not only to OCL semantics that is easy to read and understand, but also to a formalization of the semantics that

is directly executable by model transformation engines. Moreover, in this chapter we have identified and classified OCL expressions into two groups: 1) Core OCL expressions, and 2) Advanced semantic concepts. On the example of relational database we have shown how OCL can be tailored to support various DSL's and how it is possible to specify semantics of these OCL variations.

In order to prove that our refactoring rules are semantics preserving, in Chapter 5 we have defined a simple criterion for semantics preservation. Our criterion is based on results of evaluation of OCL expressions, and thanks to semantics defined in Chapter 4, it was easily applicable to refactoring rules specified in Chapter 3. Although some authors advocate that it is hard and time-consuming task to prove semantics preservation of model refactorings [52], in the case of UML/OCL models we have successfully applied our criterion and proved that our refactoring rules are semantics preserving.

The work presented in this thesis is implemented in our ROCLET tool. The tool is built as an Eclipse plug-in and consists of a GUI that supports creating UML class and object diagrams, an OCL parser and pretty-printer, and model transformation rules that perform the manipulation of created models. UML models created in the tool can be easily refactored by applying rules presented in Chapter 3 while all OCL constraints are updated accordingly. The tool implements the evaluation rules shown in Chapter 4 which allows us to easily check semantic preservation property of the refactoring rules.

6.2   Future Work

Although the work presented in this thesis concentrates mostly on refactoring UML class diagrams and propagating necessary changes to object diagrams and OCL constraints, it would be possible to apply the same technique to different UML models. One example would be the refactoring of UML class diagrams and automatic update of corresponding sequence, collaboration, or any other type of UML diagrams.

Another research area would be the application of QVT transformation rules for synchronizing different software artifacts including various models and programming code. In this thesis we were mostly concentrated on unidirectional change propagation, but the same technique is applicable to model synchronization problems and round-trip engineering [78].

Another branch of future activities would be the description of the semantics of programming languages with graphical QVT rules. The ultimate goal would be to demonstrate that also the description of the semantics of a programming

language can be given in an easily understandable, intuitive format. This might finally contribute to a new style of language definitions where the semantics of the language can be formally defined as easy and straightforwardly as it is today already the case with the syntax of languages.

Tool Support

In this chapter, we describe the architecture and the functionality of our own OCL tool called ROCLET. Besides standard features of OCL tools such as editing of class and object diagrams and parsing of OCL assertions (invariants, pre-/post-conditions), our tool supports also the evaluation of OCL constraints in a given system snapshot (object diagram), and the refactoring of UML/OCL models. ROCLET is deployed in form of an Eclipse plugin.

A.1   Introduction

The Unified Modeling Language (UML) is today the most popular object-oriented modeling language for software systems. UML is in the first place a graphical notation what makes software models easily accessible by humans. UML diagrams can give a good overview on the modeled software system, but there is a lack of expressive power once the details of the software system have to be captured as well. A prevailing practice to resolve this problem is to add comments to UML diagrams and to clarify the intended meaning using natural language. Such informal comments, however, do not alter the formal meaning of the model and are ignored by tools when processing the model, e.g. in order to generate code. Another disadvantage is, that reading informal comments can become easily a hard and also ambiguous task once the comments are a little bit more complex.

The Object Constraint Language (OCL), see [68] for both an introduction and the language specification, is a textual language with formal syntax and semantics. OCL constraints capture a wide range of details that software developers wish to express in precise software models. The main application scenario are UML class diagrams. Here, OCL constraints can express conditions that should be obeyed

in each system state (invariants) and contracts for system operations (pre-/post-conditions).

Most of the current OCL tools - USE [89], Octopus [86], Dresden OCL Toolkit [83] and OCLE [85] being the most influential ones - were developed in academia. Whereas almost each tool offers, besides parsing facilities for OCL, a functionality to generate implementation stubs out of UML/OCL models, relatively little effort has been made so far to analyze the OCL constraints themselves, to provide functionalities for automatic constraint simplification, for refactoring, for analyzing of which impact a (small) change in a snapshot on the validity of a given OCL invariant has. ROCLET aims at providing facilities for a painless authoring, processing and analysis of OCL constraints. The main functionalities of ROCLET are:

- Parsing and type analysis

- Refactoring of UML class diagrams including necessary changes on attached OCL constraints (see Chapter 3)

- Evaluation in a given object diagram (applying the technique described in Chapter 4)

A.2   Architecture of ROCLET

We have chosen a 3-layer architecture for ROCLET (comp. Fig.A.1): presentation layer, application layer and data layer.

The presentation layer consists of editors and views for user interaction. Due to a lack of high quality diagram editors we have decided to implement our own editors for class and object diagrams whereas the editor for OCL constraints is (currently) based on the work of [84].

The presentation layer has direct access to the data layer where the edited UML/OCL model is stored in a repository as a formal instance of the UML/OCL metamodel.

ROCLET's functionalities are implemented in the application layer, mainly in form of transformation rules written in QVT. These transformations work on the repository and usually alter it directly.
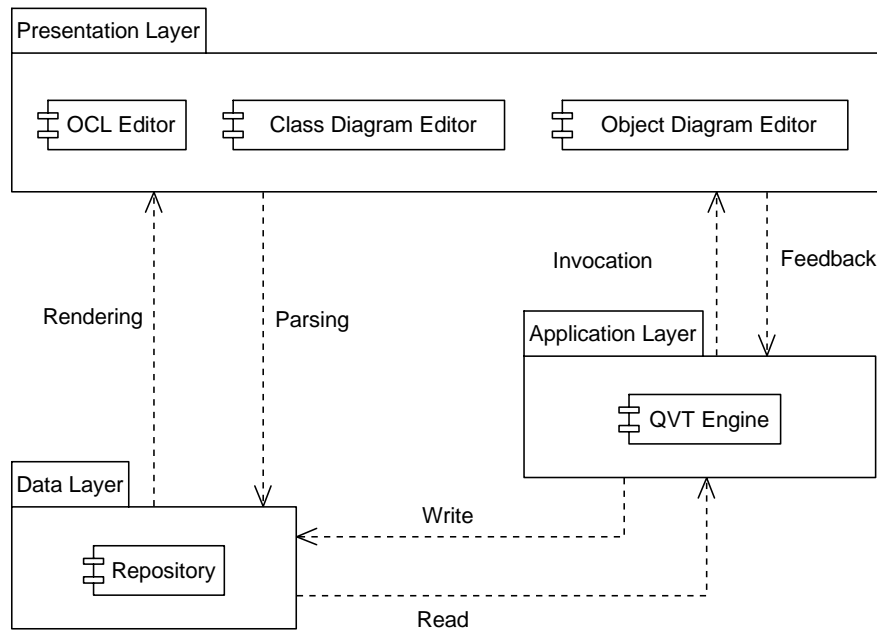
Figure A.1: ROCLET Architecture

A.3   Implementation of Refactoring Rules in QVT

We implemented all rules of our refactoring catalog using the QVT implementation of Together Architect 2006 for Eclipse [20]. 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 [88].

A.3.1   Overview

Together Architect 2006 for Eclipse implements a large body of the QVT standard[1]. The implemented version of the transformation rules looks at the first glance quite different from what was specified in graphical form in Sect. 3.2. 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]A list of missing features not implemented yet is shipped with Together.

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 rules we specified in this thesis, 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 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
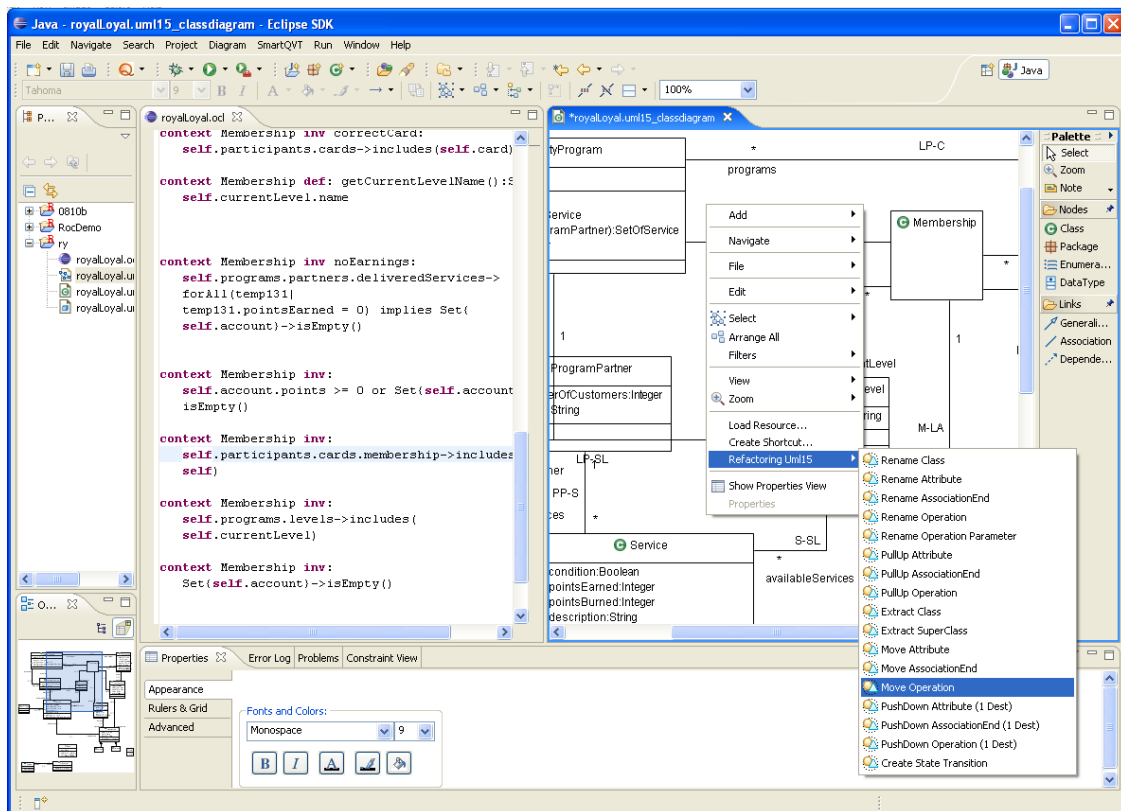
Figure A.2: Application of refactoring rule on a concrete UML/OCL model

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 A.3 shows the application of a QVT transformation on a concrete UML/OCL source model in our tool ROCLET[2], into which the implementation of refactoring rules has been integrated.

A.3.2 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.

---

[2]ROCLET is available from www.roclet.org

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 'roclet';

mapping main(in model: roclet::Model): roclet::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;
    }
}
```

A.3.3 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:roclet::Package,
                   in src:roclet::Class,
```

```
                in newName: String ,
                in role1 : String ,
                in role2 : String ): OclVoid{
  if findUMLMatch( src , newName, role1 )
  then applyRHSUML( src . namespace , src , newName, role1 , role2 )
  else true
  endif;

  undefined
}

query findUMLMatch( in src : roclet :: 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( roclet :: Classifier ))
  then nsp . oclAsType( roclet :: Classifier )
        . allConflictingNames()−>excludes(newName)
  else nsp . ownedElement . name−>excludes(newName)
  endif
}
query whentest2( in src : roclet :: Class ,
              in role1 : String ): Boolean{
  src . allConflictingNames()−>excludes( role1 )
}
```

A.3.4 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 trans-formation rule. The rewrite step uses extensively the new facilities integrated into Imperative OCL in order to manipulate data structures.

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

A.4 Implementation of Evaluation Rules in QVT

All evaluation rules presented in Chapter 4 are implemented in ROCLET. The implementation was straightforward just like in the case of refactoring rules.

A.4.1 Overview

Unlike implementation of refactoring rules described in Sect. A.3 which are made in a "single" traversal of UML/OCL model, the implementation algorithm for evaluation rules works in two passes.

The first pass performs binding of free variables to all OCL expressions (and subexpressions) and is a necessary precondition for performing evaluation of the expressions.

The second pass executes the evaluation rules as specified in Chapter 4.

Both of these passes will be explained in more details in following subsections.

Figure A.3 shows an application of evaluation as implemented in our tool ROCLET [88].

A.4.2 Invocation

All OCL constraints attached to a UML class diagram are specified in the context of a specific *Classifier* from the diagram. Therefore, when performing evaluation of OCL constraints, two parameters have to be provided: 1) Constraint that is to be evaluated; 2) Object for which we perform the evaluation, and that is an instance of the contextual *Classifier* for the given constraint.

As the first step when performing evaluation of OCL expressions, the object passed as a parameter is bound to *self* variable, i.e. a new binding is made. The second step is performing free variables binding (starting from the "top" expression, i.e. the body expression of the constraint passed as a parameter), and then, as the third step, actual evaluation rules are invoked (again starting from the "top" expression).

mapping Evaluate(
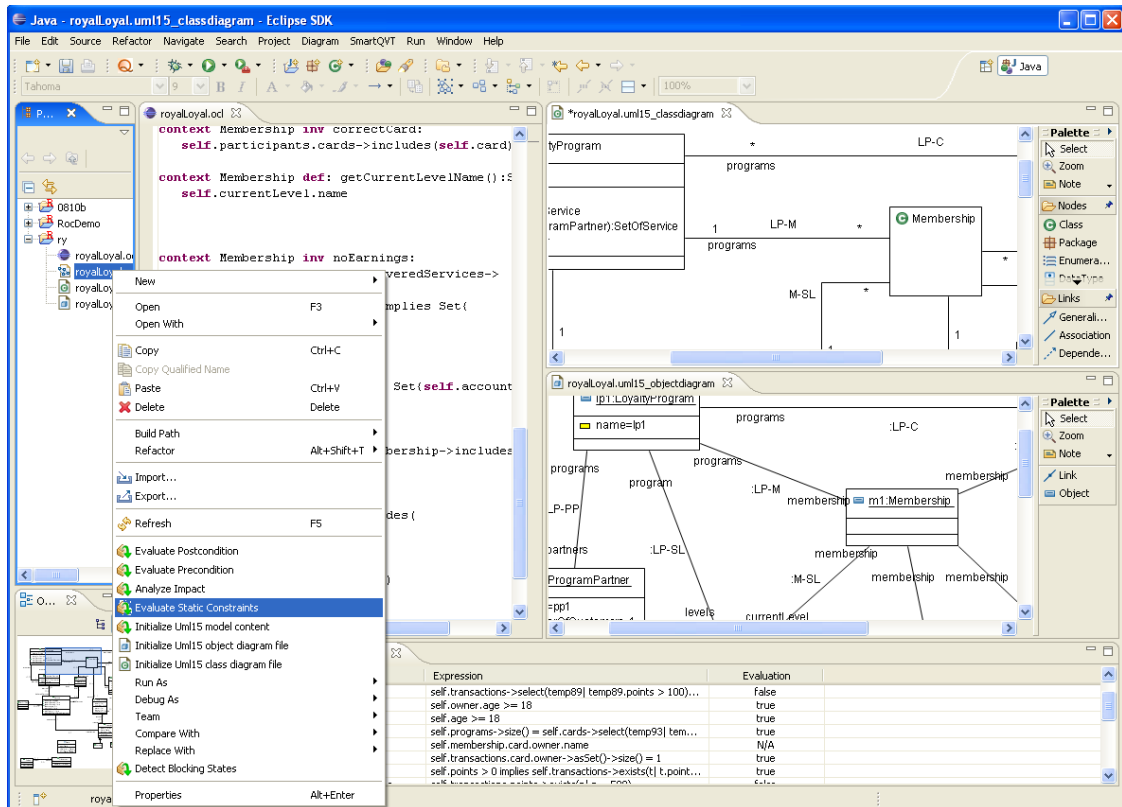
Figure A.3: Application of evaluation of a concrete UML/OCL model

```
        inout expInOcl: roclet :: ExpressionInOcl ,
        in obj: roclet :: Object ): roclet :: Object {
init {
    var empty: OrderedSet ( roclet :: NameValueBinding):= OrderedSet { };
    var dummy:=empty−>append (
        object roclet :: NameValueBinding{varName:= ' s e l f ' ;
            namespace:=expInOcl . bodyExpression . type . namespace ;
            val:=obj;})−>asOrderedSet ( );

    var d1:=BindOCLExpression ( expInOcl . bodyExpression , dummy );

    var d2:=EvaluateOCLExpression ( expInOcl . bodyExpression );
    result:=undefined ;
        }
}
```

A.4.3 Variable Bindings

Binding of variables to an expression depends on the type of that expression (i.e. *AttributeCallExp, IfExp, LoopExp...*). Therefore, we need a "switch" statement that will invoke appropriate binding depending on the expression type.

```
mapping BindOCLExpression(
        inout oclexp:roclet::OclExpression,
        in nvb:OrderedSet(roclet::NameValueBinding))
                                    :roclet::OperationCallExp{
init {
 var dummy1:OclAny:=
 if oclexp.oclIsKindOf(roclet::IntegerLiteralExp)
   then BindIntegerLiteralExp(
            oclexp.oclAsType(roclet::IntegerLiteralExp), nvb)
   else if oclexp.oclIsKindOf(roclet::IfExp)
   then BindIfExp(
            oclexp.oclAsType(roclet::IfExp), nvb)
   else if oclexp.oclIsKindOf(roclet::AttributeCallExp)
   then BindAttributeCallExp(
            oclexp.oclAsType(roclet::AttributeCallExp), nvb)


            ...


}
```

All bindings to an expression are performed in two steps: 1) Adding bindings passed as a parameter to the expression's collection of bindings; 2) Passing expression's bindings (potentially altered) to the expression's subexpressions (if any).

```
mapping BindIntegerLiteralExp(
        inout oclexp:roclet::IntegerLiteralExp,
        in nvb:OrderedSet(roclet::NameValueBinding))
                                    :roclet::IntegerLiteralExp{
init {
    oclexp.binding:=nvb;
    result:=oclexp;}
}


mapping BindIfExp (
        inout oclexp:roclet::IfExp,
        in nvb:OrderedSet(roclet::NameValueBinding))
```

```
                       : roclet :: IfExp {
init {
oclexp . binding:=nvb ;
var d:=BindOCLExpression ( oclexp . condition , oclexp . binding );
var d2:=BindOCLExpression ( oclexp . thenExpression , oclexp . binding );
var d3:=BindOCLExpression ( oclexp . elseExpression , oclexp . binding );
result:=oclexp ;}
}


mapping BindAttributeCallExp (
        inout oclexp : roclet :: AttributeCallExp ,
        in nvb : OrderedSet ( roclet :: NameValueBinding ))
                : roclet :: AttributeCallExp {
init {
   oclexp . binding:=nvb ;
   var d:=BindOCLExpression ( oclexp . source , oclexp . binding );
   result:=oclexp ;}
}
```

A.4.4 Evaluations

Like variable binding, evaluation of each OCL expression depends on the expression's type. Similarly to the case of variable bindings, a "switch" statement is needed, to invoke corresponding evaluation rules.

```
mapping EvaluateOCLExpression (
        inout oclexp : roclet :: OclExpression )
                : roclet :: OperationCallExp {
init {
 var dummy1 : OclAny:=
     if oclexp . oclIsKindOf ( roclet :: IntegerLiteralExp )
     then EvaluateIntegerLiteralExp (
             oclexp . oclAsType ( roclet :: IntegerLiteralExp ))
      else if oclexp . oclIsKindOf ( roclet :: IfExp )
      then EvaluateIfExp (
             oclexp . oclAsType ( roclet :: IfExp ))
      else if oclexp . oclIsKindOf ( roclet :: AttributeCallExp )
      then EvaluateAttributeCallExp (
             oclexp . oclAsType ( roclet :: AttributeCallExp ))
```

```
        ...

}
```

Each of the rules first evaluates subexpressions of the expression passed as a parameter, and then performs evaluation of the expression itself.

```
mapping EvaluateIntegerLiteralExp (
        inout oclexp : roclet :: IntegerLiteralExp )
                : roclet :: IntegerLiteralExp {
init {
    oclexp . val:= object roclet :: IntegerValue {
                        namespace:=oclexp . type . namespace
                        integerValue:=oclexp . integerSymbol );

    result:=oclexp ;}
}



mapping EvaluateIfExp (inout oclexp : roclet :: IfExp ): roclet :: IfExp {
init {
 var d1:=EvaluateOCLExpression ( oclexp . condition );
 var d2:= if ( oclexp . condition . val
            . oclAsType ( roclet :: BooleanValue ). booleanValue=true )
        then EvaluateOCLExpression ( oclexp . thenExpression )
        else EvaluateOCLExpression ( oclexp . elseExpression )
        endif ;

 oclexp . val:= if ( oclexp . condition . val
            . oclAsType ( roclet :: BooleanValue ). booleanValue=true )
        then oclexp . thenExpression . val
        else oclexp . elseExpression . val
        endif ;
 result:=oclexp ;}
}



mapping EvaluateAttributeCallExp (
    inout ace : roclet :: AttributeCallExp ): roclet :: AttributeCallExp {

init {
```

```
    var d1:= EvaluateOCLExpression(ace.source);
    ace.val:= if ace.source.val
              .oclIsTypeOf(roclet::OclVoidValue) then
                  object roclet::OclVoidValue{
                          namespace:=ace.type.namespace;}
             else
                 ace.referredAttribute.attributeLink
                         −>select(a|a.instance=ace.source.val)
                                 −>any(true).value
             endif;

    result:=ace; }
}
```

A.5 Conclusions

The encoding of the graphical refactoring rules as given in Sect. 3.2 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 imple-
mented 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 el-
ements 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.

All the evaluation rules were implemented on the similar manner like the refac-
toring rules. The only difference is that contrary to refactoring rules for which the
LHS match is searched depending on the passed parameters, for the evaluation rules
the complete OCL model is traversed starting from the top expression to the subex-
pressions. Along the traversal path all evaluation rules are accordingly applied.

All refactoring and evaluation rules presented in this thesis are implemented in
QVT textual syntax using Together Architect 2006. There would be no obstacle to
use any other tool for model transformations, like Fujaba [44] or ATL [5] to produce
the same results.

A unique feature of our tool is its adaptability to specific needs a user might have. Since OCL is basically a very versatile language and applicable in many different domains, there are frequent requests for domain-specific changes of OCL's semantics. It is relatively easy for the user to adapt ROCLET to a new OCL dialect (assuming that a parser for the new OCL dialect exists). The only thing to be done is to modify some of the QVT rules that implement ROCLET's functionalities. In order to do this, however, the user must have installed Together Architect for Eclipse [20], which implements the QVT engine on which ROCLET is based.

Bibliography

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, 2005.

[2] David H. Akehurst and Behzad Bordbar. On querying UML data models with OCL. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*, pages 91–103. Springer, 2001.

[3] David H. Akehurst, Gareth Howells, and Klaus D. McDonald-Maier. Supporting OCL as part of a family of languages. In Thomas Baar, editor, *Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, October 4, 2005*, Technical Report LGL-REPORT-2005-001, pages 30–37. EPFL, 2005.

[4] Dave Astels. Refactoring with UML. In *International Conference eXtreme Programming and Flexible Processes in Software Engineering*, pages 67–70, 2002.

[5] ATL team. ATL project. http://www.eclipse.org/m2m/atl/, 2007.

[6] Thomas Baar. The definition of transitive closure with OCL - limitations and applications. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers*, volume 2890 of *Lecture Notes in Computer Science*, pages 358–365. Springer, 2003.

[7] Thomas Baar. *Über die Semantikbeschreibung OCL-artiger Sprachen.* PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 2003. ISBN 3-8325-0433-8, Logos Verlag, Berlin, In German.

[8] Thomas Baar. Non-deterministic constructs in OCL – what does any() mean. In Andreas Prinz, Rick Reed, and Jeanne Reed, editors, *SDL 2005: Model Driven, 12th International SDL Forum, Grimstad, Norway, June 20-23, 2005, Proceedings*, volume 3530 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2005.

[9] Thomas Baar and Slaviša Marković. A graphical approach to prove the semantic preservation of UML/OCL refactoring rules. In Irina Virbitskaite and Andrei Voronkov, editors, *Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27-30, 2006. Revised Papers*, volume 4378 of *Lecture Notes in Computer Science*, pages 70–83. Springer, 2007.

[10] Thomas Baar and Jon Whittle. On the usage of concrete syntax in model transformation rules. In Irina Virbitskaite and Andrei Voronkov, editors, *Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27-30, 2006. Revised Papers*, volume 4378 of *Lecture Notes in Computer Science*, pages 84–97. Springer, 2007.

[11] Hanna Bauerdick, Martin Gogolla, and Fabian Gutsche. Detecting OCL traps in the UML 2.0 superstructure: An experience report. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *UML 2004 - The Unified Modeling Language: Modeling Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004. Proceedings*, volume 3273 of *Lecture Notes in Computer Science*, pages 188–196. Springer, 2004.

[12] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.

[13] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[14] Kirsten Berkenkötter. OCL-based validation of a railway domain profile. In Thomas Kühne, editor, *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, volume 4364 of *Lecture Notes in Computer Science*, pages 159–168. Springer, 2007.

[15] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. EMF model refactoring based on graph transformation concepts. In Tom Mens Jean-Marie Favre, Reiko Heckel, editor, *Third Workshop on Software Evolution through Transformations: Embracing the Change (SeTra 2006), volume 3, Natal, Brazil, September 2006, Proceedings*, volume 3 of *Electronic Communications of the EASST*, 2006.

[16] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical definition of in-place transformations in the Eclipse Modeling Framework. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2006.

[17] Marko Boger, Thorsten Sturm, and Per Fragemann. Refactoring browser for UML. In *International Conference eXtreme Programming and Flexible Processes in Software Engineering*, pages 77–81, 2002.

[18] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings Publishing Co., second edition, 1993.

[19] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, second edition, 2005.

[20] Borland. Together technologies. www.borland.com/together/, 2007.

[21] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Consistency checking and visualization of OCL constraints. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, volume 1939 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2000.

[22] Achim D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications.* Ph.D. thesis, ETH Zurich, 2007. ETH Dissertation No. 17097.

[23] Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. Semantic issues of OCL: Past, present, and future. In Birgith Demuth, Dan Chiorean, Martin Gogolla, and Jos Warmer, editors, *OCL for (Meta-)Models in Multiple Application Domains*, pages 213–228, Dresden, 2006. University Dresden. Available as Technical Report, University Dresden, number TUD-FI06-04-Sept. 2006.

[24] Achim D. Brucker and Burkhart Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[25] Jordi Cabot and Ernest Teniente. Computing the relevant instances that may violate an OCL constraint. In Oscar Pastor and João Falcão e Cunha, editors, *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, volume 3520 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2005.

[26] Jordi Cabot and Ernest Teniente. Incremental evaluation of OCL constraints. In Eric Dubois and Klaus Pohl, editors, *Advanced Information Systems Engineering, 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg, June 5-9, 2006, Proceedings*, volume 4001 of *Lecture Notes in Computer Science*, pages 81–95. Springer, 2006.

[27] Jordi Cabot and Ernest Teniente. Transforming OCL constraints: a context change approach. In Hisham Haddad, editor, *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pages 1196–1201. ACM, 2006.

[28] Eric Cariou, Raphaël Marvie, Lionel Seinturier, and Laurence Duchien. OCL for the specification of model transformation contracts. In Octavian Patrascoiu, editor, *OCL and Model Driven Engineering, UML 2004 Conference Workshop, October 12, 2004, Lisbon, Portugal*, pages 69–83. University of Kent, 2004.

[29] María Victoria Cengarle and Alexander Knapp. A formal semantics for OCL 1.4. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*, pages 118–133. Springer, 2001.

[30] María Victoria Cengarle and Alexander Knapp. OCL 1.4/5 vs. 2.0 expressions formal semantics and expressiveness. *Software and Systems Modeling*, 3(1):9–30, 2004.

[31] Juan Martín Chiaradía and Claudia Pons. Improving the OCL semantics definition by applying dynamic meta modeling and design patterns. In Birgith Demuth, Dan Chiorean, Martin Gogolla, and Jos Warmer, editors, *OCL for (Meta-)Models in Multiple Application Domains*, pages 229–239, Dresden, 2006. University Dresden. Available as Technical Report, University Dresden, number TUD-FI06-04-Sept. 2006.

[32] Dan Chiorean, Maria Bortes, and Dyan Corutiu. Proposals for a widespread use of OCL. In Thomas Baar, editor, *Tool Support for OCL and Related Formalisms - Needs and Trends, MoDELS'05 Conference Workshop, Montego Bay, Jamaica, October 4, 2005, Proceedings*, Technical Report LGL-REPORT-2005-001, pages 68–82. EPFL, 2005.

[33] Tony Clark, Andy Evans, and Stuart Kent. Engineering modelling languages: A precise meta-modelling approach. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering. 5th International Conference, FASE 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 2002, Proceedings*, volume 2306 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2002.

[34] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Cameron Wills. The amsterdam manifesto on OCL. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 115–149. Springer, 2002.

[35] Alexandre Correa and Cláudia Werner. Applying refactoring techniques to UML/OCL. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *UML 2004 - The Unified Modeling Language: Modeling Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004. Proceedings*, volume 3273 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2004.

[36] Birgit Demuth and Heinrich Hußmann. Using UML/OCL constraints for relational database design. In Robert B. France and Bernhard Rumpe, editors,

UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings, volume 1723 of *Lecture Notes in Computer Science*, pages 598–613. Springer, 1999.

[37] Birgit Demuth, Heinrich Hußmann, and Sten Loecher. OCL as a specification language for business rules in database applications. In *UML'01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 104–117. Springer, 2001.

[38] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2000.

[39] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Theory and Application of Graph Transformations, 6th International Workshop, TAGT'98, Paderborn, Germany, November 16-20, 1998, Selected Papers*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 1998.

[40] Stephan Flake. Ocltype - a type or metatype? *Electronic Notes in Theoretical Computer Science*, 102:63–75, 2004.

[41] Stephan Flake and Wolfgang Mueller. Formal Semantics of Static and Temporal State-Oriented OCL-Constraints. *Software and Systems Modeling*, 2(3):164.186, 2003.

[42] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[43] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, third edition, 2003.

[44] Fujaba team. Fujaba homepage. http://www.fujaba.de, 2007.

[45] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[46] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, volume 2863 of *Lecture Notes in Computer Science*, pages 144–158. Springer, 2003.

[47] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Software*, 37(10):64–72, October 2004.

[48] Rolf Hennicker, Alexander Knapp, and Hubert Baumeister. Semantics of OCL operation specifications. *Electronic Notes in Theoretical Computer Science, Proceedings of OCL 2.0 Workshop at UML'03*, 102:111–132, 2004.

[49] C. A. R. Hoare. Proof of correctness of data representation. In Friedrich L. Bauer and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 183–193. Springer, 1975.

[50] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[51] Joshua Kerievsky. *Refactoring to Patterns.* Addison-Wesley, 2004.

[52] Anneke Kleppe and Jos Warmer. Do MDA transformations preserve meaning? An investigation into preserving semantics. In Andy Evans, Paul Sammut, and James S. Willans, editors, *Metamodelling for MDA. First International Workshop, York, UK, November 2003*, pages 13–22, 2003.

[53] Anneke Kleppe and Jos Warmer. *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison-Wesley, second edition, 2003.

[54] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Towards using OCL for instance-level queries in domain specific languages. In Birgith Demuth, Dan Chiorean, Martin Gogolla, and Jos Warmer, editors, *OCL for (Meta-)Models in Multiple Application Domains*, pages 26–37, Dresden, 2006. University Dresden. Available as Technical Report, University Dresden, number TUD-FI06-04-Sept. 2006.

[55] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley, third edition, 2004.

[56] Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. In Lionel C. Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, volume 3713 of *Lecture Notes in Computer Science*, pages 280–294. Springer, 2005.

[57] Slaviša Marković and Thomas Baar. An OCL semantics specified with QVT. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*, pages 660–674. Springer, October 2006.

[58] Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. *Software and Systems Modeling*, 7(1):25–47, 2008.

[59] Slaviša Marković and Thomas Baar. Semantics of OCL specified with QVT. *Software and Systems Modeling*, 2008. (Accepted for publication).

[60] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.

[61] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution*, 17(4):247–276, 2005.

[62] Mel O'Cinneide. *Automated Application of Design Patterns: a Refactoring Approach.* PhD thesis, University of Dublin, Trinity College, 2001.

[63] OMG. MDA guide version 1.0.1. OMG Document omg/03-06-01, June 2003.

[64] OMG. UML 1.5 Specification. OMG Document formal/03-03-01, March 2003.

[65] OMG. UML 2.0 Infrastructure Specification. OMG Document ptc/03-09-15, Sep 2003.

[66] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. OMG Document ptc/05-11-01, Nov 2005.

[67] OMG. Meta Object Facility (MOF) Core Specification – OMG Available Specification, version 2.0. OMG Document formal/06-01-01, Jan 2006.

[68] OMG. Object Constraint Language – OMG Available Specification, version 2.0. OMG Document formal/06-05-01, May 2006.

[69] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[70] Ivan Porres. Model refactorings as rule-based update transformations. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, volume 2863 of *Lecture Notes in Computer Science*, pages 159–174. Springer, 2003.

[71] Mark Richters. *A precise approach to validating UML models and OCL constraints*. PhD thesis, Bremer Institut für Sichere Systeme, Universität Bremen, Logos-Verlag, Berlin, 2001.

[72] Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In Tok Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Conceptual Modeling - ER '98, 17th International Conference on Conceptual Modeling, Singapore, November 16-19, 1998, Proceedings*, volume 1507 of *Lecture Notes in Computer Science*, pages 449–464. Springer, 1998.

[73] Mark Richters and Martin Gogolla. A metamodel for OCL. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 1999.

[74] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[75] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, second edition, 2005.

[76] Bernhard Rumpe. *Agile Modellierung mit UML*. Springer, 2005. In German.

[77] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[78] Shane Sendall and Jochen Malte Küster. Taming model round-trip engineering. In *Workshop on Best Practices for Model-Driven Software Development MDSD 2004 (part of 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications), Vancouver, Canada, October 25, 2004, Proceedings*, 2004.

[79] Shane Sendall and Alfred Strohmeier. UML based fusion analysis applied to a bank case study. In Robert B. France and Bernhard Rumpe, editors, *UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, volume 1723 of *Lecture Notes in Computer Science*, pages 278–291. Springer, 1999.

[80] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation.* Springer, 2001.

[81] 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 Klaus R. Dittrich, Andreas Geppert, and Moira C. Norrie, editors, *Advanced Information Systems Engineering, 13th International Conference, CAiSE 2001, Interlaken, Switzerland, June 4-8, 2001, Proceedings*, volume 2068 of *Lecture Notes in Computer Science*, pages 433–447. Springer, 2001.

[82] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 2001.

[83] Dresden OCL Team. Dresden OCL Toolkit. http://dresden-ocl.sourceforge.net/, 2007.

[84] MDT-OCL Team. Eclipse MDT - OCL project. http://www.eclipse.org/modeling/mdt/?project=ocl, 2007.

[85] OCLE Team. OCLE – Object Constraint Language Environment. http://lci.cs.ubbcluj.ro/ocle/index.htm, 2007.

[86] Octopus Team. Octopus – OCL Tool for Precise UML Specifications. http://octopus.sourceforge.net/, 2007.

[87] Oslo Team. Oslo project. http://oslo-project.berlios.de/, 2007.

[88] RoclET Team. RoclET project. http://www.roclet.org/, 2007.

[89] USE Team. USE – a UML-based Specification Environment. http://www.db.informatik.uni-bremen.de/projects/USE/, 2007.

[90] Dániel Varró. A formal semantics of UML Statecharts by model transition systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, volume 2505 of *Lecture Notes in Computer Science*, pages 378–392. Springer, October 7–12 2002.

[91] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer, 2007.

Curriculum Vitae

Slaviša Marković

Education

$2003 - 2008$	**Ph.D. Candidate**, Software Engineering Laboratory (LGL), School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland.
2003	**Doctoral School in Computer Sciences**, School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland.
2002	**Dipl.Ing. in Computer Science**, School of Business Administration, University of Belgrade, Serbia.

Personal

Born in Arandjelovac, Serbia, on February 1, 1975. Nationality: Serbian.

Publications

[1] Slaviša Marković, and Thomas Baar, "Refactoring OCL annotated UML class diagrams", Software and Systems Modeling, 7(1):25-47, 2008.

[2] Slaviša Marković, and Thomas Baar, "Semantics of OCL Specified with QVT", Software and Systems Modeling, 2008. (Accepted for publication)

[3] Thomas Baar, Slaviša Marković, Frédéric Fondement and Alfred Strohmeier, "Definition and Correct Refinement of Operation Specifications", In B. Meyer, A. Schiper, J. Kohlas, editors, Dependable Systems: Software, Computing, Networks, volume 4028 of Lecture Notes in Computer Science, pages 127-144, Springer 2006.

[4] Slaviša Marković, and Thomas Baar, "An OCL Semantics Specified with QVT", In O. Nierstrasz, J. Whittle, D. Harel, G. Reggio, editors, Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings, volume 4199 of Lecture Notes in Computer Science, pages 661-676. Springer, 2006.

[5] Slaviša Marković, and Thomas Baar, "Refactoring OCL annotated UML class diagrams", In Lionel C. Briand and Clay Williams, editors, Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings, volume 3713 of Lecture Notes in Computer Science, pages 280-294. Springer, 2005.

[6] Thomas Baar, and Slaviša Marković, "A graphical approach to prove the semantic preservation of UML/OCL refactoring rules". In Irina Virbitskaite and Andrei Voronkov, editors, Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27-30, 2006. Revised Papers, volume 4378 of Lecture Notes in Computer Science, pages 70-83. Springer, 2007.

[7] Cédric Jeanneret, Leander Eyer, Slaviša Marković, and Thomas Baar, "RoclET: Refactoring OCL expressions by transformations", Software and Systems Engineering and their Applications, 19th International Conference, ICSSEA 2006, Paris, France, December 5-7, 2006.

[8] Slaviša Marković, and Thomas Baar, "Synchronizing Refactored UML Class Diagrams and OCL Constraints", 1st Workshop on Refactoring Tools, ECOOP 07 Conference Workshop, July 31, 2007, Berlin, Germany, Danny Dig, Michael Cebulla (Eds.), TU Berlin Technical Report, ISSN 1436-9915, 2007, pp. 15-17.

[9] Slaviša Marković, "Composition of UML Described Refactoring Rules", OCL and Model Driven Engineering, UML 2004 Conference Workshop, October 12, 2004, Lisbon, Portugal, Octavian Patrascoiu (Ed.), University of Kent, 2004, pp. 45-59.