An OCL Semantics Specified with QVT*

Slaviša Marković and Thomas Baar

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

Abstract. Metamodeling became in the last decade a widely accepted tool to describe the (abstract) syntax of modeling languages in a concise, but yet precise way. For the description of the language's semantics, the situation is less satisfactory and formal semantics definitions are still seen as a challenge. In this paper, we propose an approach to specify the semantics of modeling languages in a graphical way. As an example, we describe the evaluation semantics of OCL by transformation rules written in the graphical formalism QVT. We believe that the graphical format of our OCL semantics has natural advantages with respect to understandability compared to existing formalizations of OCL's semantics. Our semantics can also be seen as a reference implementation of an OCL evaluator, because the transformation rules can be executed by any QVT compliant transformation engine.

1 Introduction

Modeling is an important activity in all engineering disciplines, including software development. While the general purpose modeling language UML has proven to be versatile enough for many different domains (see, e.g., chapter 1 of [1]), it has also been recognized that the structure and the behavior of the system under development can often be captured as well with a much simpler, *domain-specific* modeling language [2].

UML and DSLs have much in common. Their abstract syntax is usually defined by a metamodel and UML's core modeling concepts such as Class, Object, State, etc. can also be found, possibly under a different name, in many DSLs. If a DSL comprises a constraint language, i.e. a language to impose restrictions on the modeled system, then some core concepts of UML's constraint language OCL such as *model navigation*, *variable quantification* and *pre-defined functions* are likely to be used. In this paper, we present a new approach to define the semantics of constraint languages formally. We illustrate our approach on a rather complex example, the semantics of OCL, but since our technique is based on general techniques such as metamodeling and model transformation, the semantics of other constraint languages can be defined in a similar way.

* This work was supported by Swiss National Scientific Research Fund under the reference number 200020-109492/1.

Before sketching existing approaches to define the semantics of OCL it is worthwhile to reflect the purpose and semantics of UML diagrams that can also be used without OCL constraints. A diagrammatic UML model describes the structure and behavior of a system at a certain level of details. The structure of the system clarifies which *states* (in UML jargon also called *snapshots*) the system can have and the behavioral description imposes restrictions on system changes. The question on how a class diagram corresponds to the state space of the system it describes has particular relevance for our later considerations. This correspondence (or semantics) of class diagrams has been given in the literature in many different forms, e.g. by an informal description (see UML User Guide [1]), by a mapping of classes into a set-theoretic domain (see [3]), by a metamodel of the semantic domain. The metamodel for the semantic domain became in UML1.x a part of the UML language standard because it is the basis for *object diagrams*, which are used to visualize system states.

The purpose of an OCL constraint is to make the already existing diagrammatic UML model more precise. For instance, a constraint attached as an invariant to a class shrinks the statespace to those states of the system, in which the constraint is evaluated to *true*. A pair of OCL constraints $(preCond, postCond)$ attached as pre-/postcondition to an operation *op* means that the implementation of *op* can realize only those state transitions $(preState, postState)$, for which *postCond* is evaluated in *postState* to *true* whenever *preCond* is evaluated in *preS* to *true*. No matter for which purpose an OCL constraint is used (as an invariant, as a guard, within pre-/postcondition), the semantics of the constraint can always be reduced to the question, how the evaluation of a constraint in a given state is defined. In the literature, the evaluation function $eval$: Constraint x State $\rightarrow \{true, false, undefined\}$ is defined either mathematically by structural induction over Constraint (see official OCL semantics, appendix A in [4]) or by embedding OCL into another logic [5]. While these two approaches have basically succeeded in describing the evaluation of OCL constraints in a formal, non-ambiguous manner, they still have some disadvantages. One drawback is the gap between OCL's official syntax definition (which is given as a metamodel) and the OCL syntax, given by structural induction, that is assumed in the semantics definition. The main, very related drawback, however, is understandability. We made the experience that many of our students, who learned OCL in our course, were quite reluctant to deepen their knowledge on OCL by reading the official mathematical semantics, just because it is presented in a format they are not very familiar with (in set theory). If the purpose of the semantics is to inform the prospective OCL users about all the details of the language, then the semantics should be given in a format OCL users are familiar with.

One technique how this can be achieved is metamodeling. Metamodels are already frequently used in abstract syntax definitions. Metamodels are very expressive and easy to understand for people who have a background in modeling (at least, these are our personal experiences we made with students). As mentioned above, metamodeling has already been applied to cover also the semantics

of class diagrams. Even more, the section 'Semantics Described using UML' in [4] presents already a metamodeling approach for the evaluation of OCL expressions. We took this approach as a starting point but added some important improvements. The most striking difference is how the evaluation process is modeled: In [4], evaluation is modeled by Evaluation-metaclasses whereas in our approach this is described by transformation rules written in QVT. We also changed the metamodel of the semantic domain significantly for many reasons; one was to have a better representation of predefined datatypes. Our approach has been implemented using the QVT engine provided by Together Architect for Eclipse.

To summarize, our semantics of OCL is specified with a metamodeling approach using MOF, OCL and QVT as a formalism at the metalevel. Since QVT depends also heavily on OCL, there is the natural question if our approach does not describe the OCL semantics in terms of OCL and thus has fallen into the trap of meta-circularity. We have avoided this trap because the semantics of the OCL used at the metalevel is given by an external mechanism, in our case by the semantics implemented by the QVT engine of Together Architect. The dependency of our semantics definition on a tool implementation might be seen as a drawback but for the purpose of our semantics – to help OCL users to deepen their knowledge on the peculiarities of OCL evaluation – this is not really an obstacle. Using a tool as an 'anchor' for our OCL semantics has also significant advantages such as automatic tool support (note that our OCL semantics is fully executable by QVT engines) and flexibility (users can easily adapt the OCL semantics to their needs).

The rest of the paper is organized as follows. In Sect. 2, we sketch our approach and show, by way of illustration, a concrete application scenario for our semantics. The steps the evaluator actually has to perform are formalized as graphical QVT rules in Sect. 3. Section 4 contains related work, while Sect. 5 draws some conclusion and points to problems, which we plan to address in the near future.

2 Our Metamodel Based Approach for OCL Evaluation

In this section we briefly review the technique and concepts our approach relies on and illustrate with a simple example the evaluation of OCL constraints.

2.1 Official Metamodels for UML/OCL

We base our semantics for OCL on the official metamodels for UML and OCL. We support the last finalized version of OCL 2.0 [4] but since this version still refers to UML1.5 [6] we were forced to support UML1.5 instead of UML2.0. Figures 1 and 2 show the parts of the UML and OCL metamodels that are relevant for this paper. Please note that Fig. 1 contains also in its upper part a metamodel of the semantic domain of class diagrams.

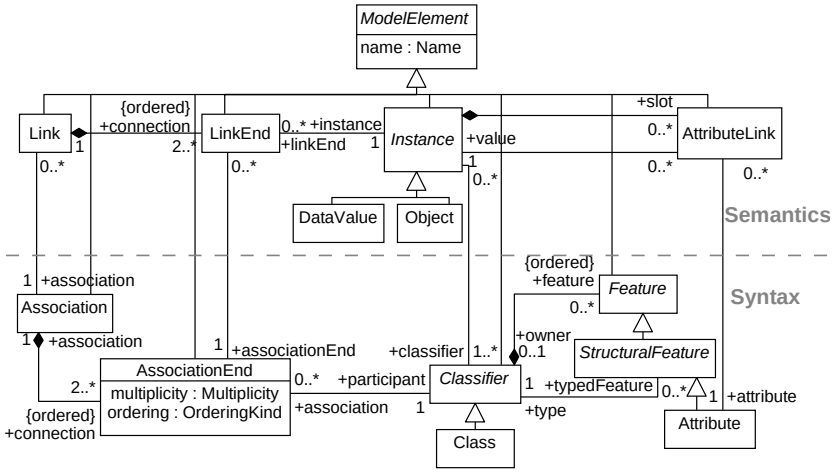

Fig. 1. Metamodel for Class Diagrams - Syntax and Semantics

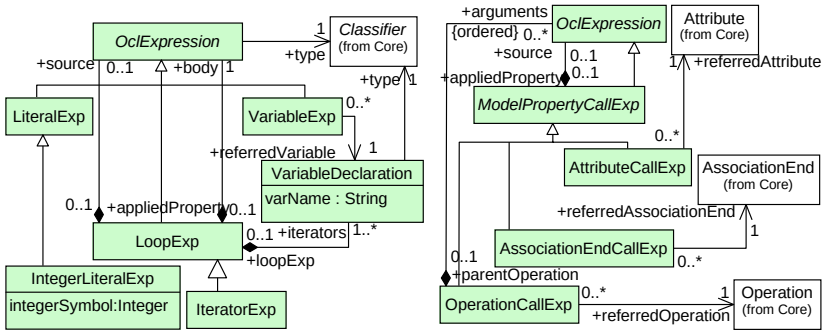

Fig. 2. Metamodel for OCL - Syntax

2.2 Changes in the OCL Metamodel

In order to realize our approach in a clear and readable way, we had to add some few metaassociations and -attributes to that part of the official metamodel of OCL that describes the semantic domain of OCL evaluations (see Fig. 3). The metaclass *OclExpression* has a new association to *Instance*, what represents the evaluation of the expression in a given object diagram. We revised slightly the concepts of bindings (association between *OclExpression* and *NameValueBinding*) and added to class *IteratorExp* two associations *current* and *intermediateResult*, and one attribute *freshBinding*. Furthermore, the classes *StringValue*, *IntegerValue*, etc. have now attributes *stringValue*, *integerValue*, etc. what makes it possible to clearly distinguish a datatype object from its value.

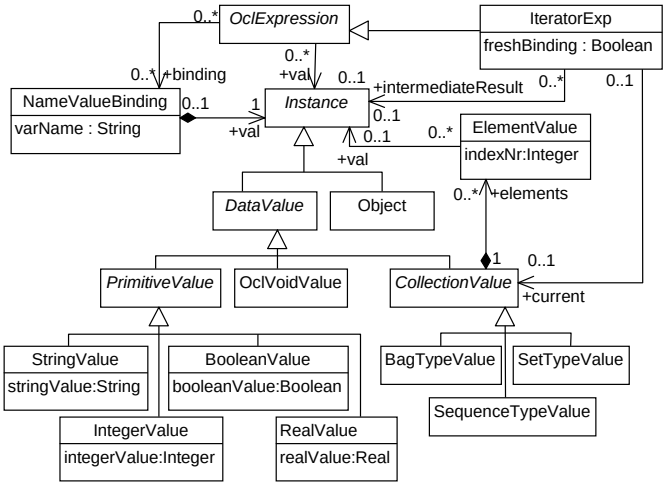

Fig. 3. Changed Metamodel for OCL - Semantics

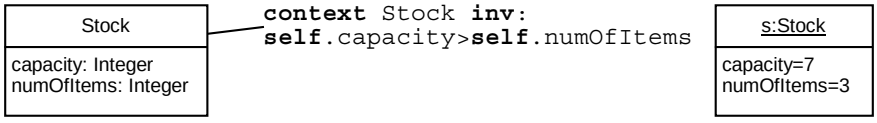

Fig. 4. Example - Class Diagram and Snapshot

2.3 Evaluation

We motivate our approach to define OCL's semantics with a small example. In Fig. 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 capacity of `Stock` and the current number of items it has, 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 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 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 on a given snapshot, we present in Fig. 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

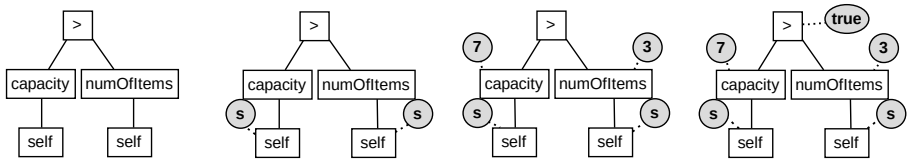

Fig. 5. Evaluation of OCL expressions seen as an AST: (a) Initial AST (b) Leaf nodes evaluated (c) Middle nodes evaluated (d) Complete AST evaluated

that in this example we were not concerned about concrete binding of the self variable. The problem of variable binding is discussed in Sect. 2.4.

The initial idea of our approach is that an OCL constraint can be analogously evaluated by annotating directly the OCL metamodel instance instead of the AST.

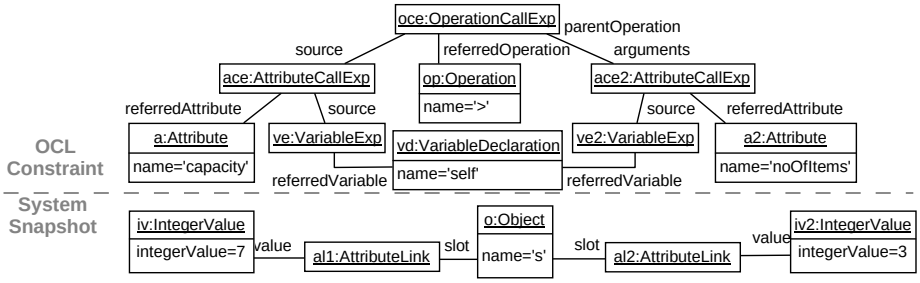

Fig. 6. OCL Constraint Before Evaluation

Figure 6 shows the instance of the OCL metamodel representing the invariant from Fig. 4. Here, we stipulate that all expressions have not been evaluated yet because for each expression the link *val* to metaclass *Instance* is missing.

The state of the metamodel instance after the last evaluation step has been finished is shown in Fig. 7. What has been added compared to the initial state (Fig. 6) is highlighted by thick lines. The evaluation of the top-expression (*OperationCallExp*) is a *BooleanValue* with *booleanValue* attribute set to `true`, the two *AttributeCallExpressions* are evaluated to two *IntegerValues* with values 7 and 3, and each *VariableExp* is evaluated to *Object* with name `s`.

2.4 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

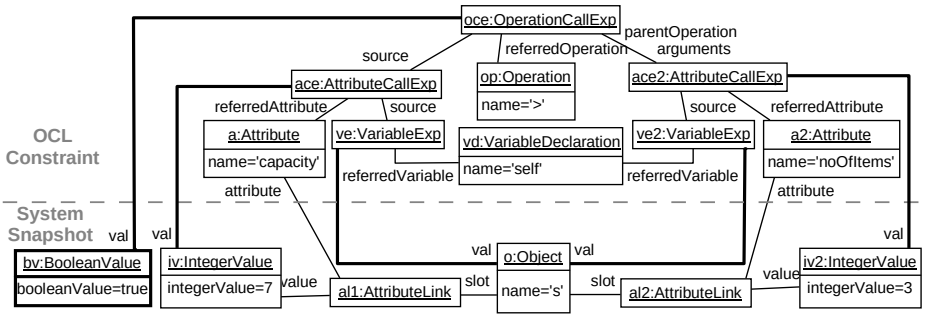

Fig. 7. OCL Constraint After Evaluation in a Given Snapshot

variables to current values. The binding of 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.

In the invariant of the `Stock` example we have used one free variable `self`. Although `self` is a predefined variable in OCL, it can be treated the same way as all other variables, which are introduced in Iterator Expressions. For example, the invariant

```
self.capacity > self.numOfItems
```

can be rewritten as

```
Stock.allInstances->forAll(self |
                self.capacity>self.numOfItems)
```

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. An example for adding new value-name bindings will be explained in more details in Sect. 3 where the evaluation rules for *forAll* expressions are explained.

Figure 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.

3 Evaluation Rules Formalized in QVT

The previous section has shown the main idea of our approach: we annotate all intermediate results of a constraint evaluation directly to the instance of the OCL metamodel. What has not been specified yet are the evaluation steps themselves,

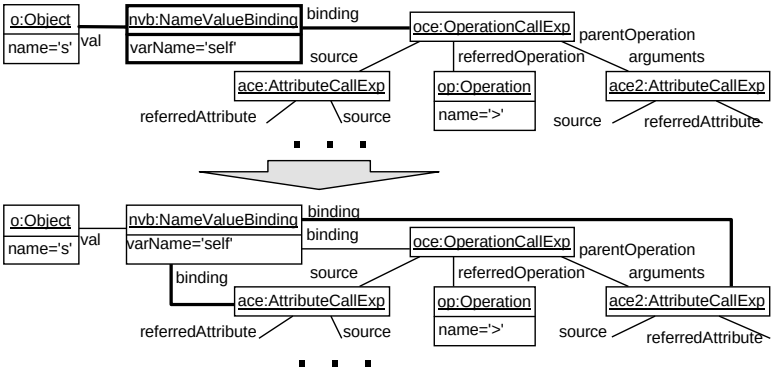

Fig. 8. Binding Passing

for example, that an *AttributeCallExp* is always evaluated to the attribute value on that object to which the source expression of *AttributeCallExp* evaluates.

In this section, we specify these evaluation steps formally in form of QVT rules. These rules are minimal in the sense that they do not capture any optimization for an efficient evaluation nor impose any restrictions on the evaluation ordering, unless they are really necessary.

3.1 QVT

QVT is a recent OMG standard for model transformations (see [8] for a detailed account on QVT's semantics), which are described by a set of *transformation rules*. For our application scenario of QVT rules, source and target model are always instances of the same metamodel; the metamodel for UML/OCL including the small changes we have proposed in Sect. 2. Each QVT rule consists of two patterns (LHS, RHS), which are (incomplete) instantiations of the UML/OCL metamodel. When a QVT rule is applied on a given source model, a LHS matching sub model of the source model is searched. Then, the target model is obtained by rewriting the matching sub model by a new sub model that is derived from RHS under the same matching. If more than one QVT rule match on a given source model, one of them is non-deterministically applied. The model transformation terminates as soon as none of the QVT rules is applicable on the current model.

3.2 A Catalog of Rules

To specify the evaluation process, we have to formalize for each non-abstract subclass of metaclass *OclExpression* one or more QVT rules. Due to space limit, only the most important rules can be presented in this subsection. In order to give a representative selection of our rules, we categorized them according to the kind of expression they target: *Navigation Expressions*, *OCL Predefined*

Operations, *Iterator Expressions*, and *Atomic Expressions*. For each category, we discuss one or two rules in detail. The main goal is to demonstrate that the evaluation of all kinds of OCL expressions can be formulated using QVT in an intuitive way.

Navigation Expressions OCL expressions of this category are 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. Before the source expression can be evaluated, the current binding of variables has to be passed from the parent expression to the sub expression. We show in Fig. 9 how the binding rule is defined for *AttributeCallExp*. When applying this rule, the binding of the parent object `ace` (represented by a link from `ace` to the multiobject `nvb` in LHS) is passed to subexpression `o` (a link from `o` to `nvb` is established in RHS). Analogous rules exist for all other kinds of OCL expressions which have subexpressions. For the (subclasses of) *LoopExp* (see below) one needs also additional rules for handling the binding because the subexpressions are evaluated under a different binding than the parent expression.

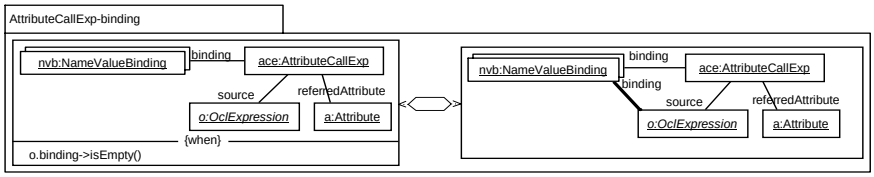

Fig. 9. Attribute Call Expression Bindings Passing

AttributeCallExp The semantics of *AttributeCallExp* is specified by the rule *AttributeCallExp-evaluation* given in Fig. 10. The evaluation of `ace` is datavalue `d`, which is also the value of 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`.

As the rule for *AttributeCallExp* shown in Fig. 10, all our QVT rules have two regions in the LHS and RHS patterns. The upper part of the patterns represents the expression that should be evaluated. The lower one specifies the system state on which the evaluation is performed. Since the evaluation of OCL rules does not have any side-effect on the system state, the lower parts of LHS and RHS will always coincide.

AssociationEndCallExp We discuss here only the case of navigating to an unordered association end with multiplicity greater than 1 (the case of multiplicities equal to 1 is very similar to *AttributeCallExp*). The rule shown in Fig. 11

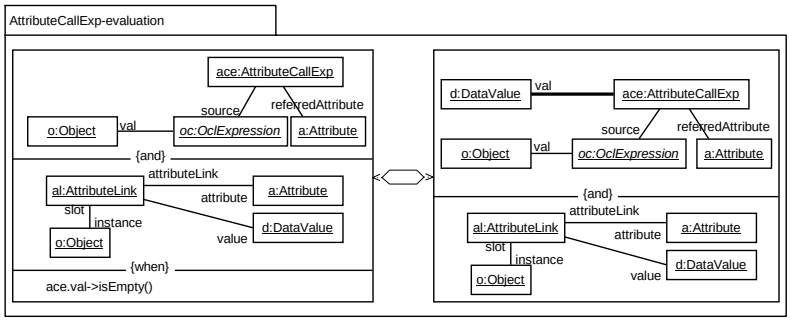

Fig. 10. Attribute Call Expression Evaluation

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`. The rule shown in Fig. 11 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. 11 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.

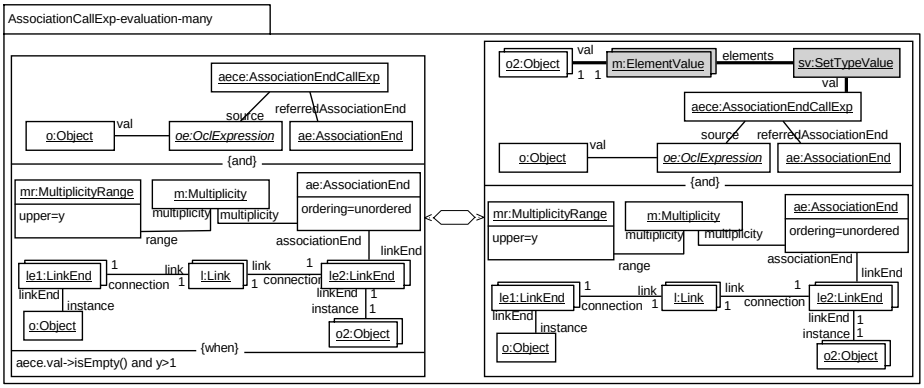

Fig. 11. Association End Call Expression Evaluation that Results in Set of Objects

OCL 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 the chapter on the OCL library in [4]. 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. 12, 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.

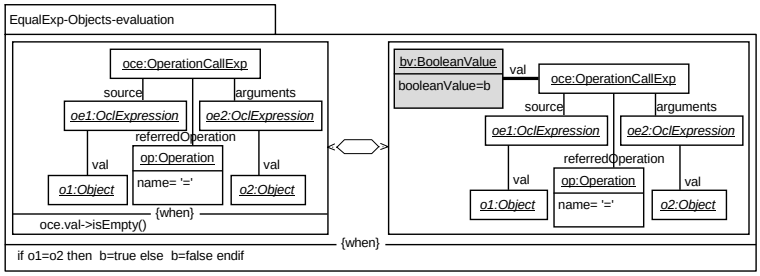

Fig. 12. Equal Operation Evaluation for Objects

If `oe1` and `oe2` evaluate to *IntegerValue*, the second QVT rule shown in Fig. 13 is applicable and the result of evaluation will be an instance of *BooleanValue* with attribute *booleanValue* set to *true* if *integerValue* of `iv1` is equal to *integerValue* of `i2`, and to *false* otherwise.

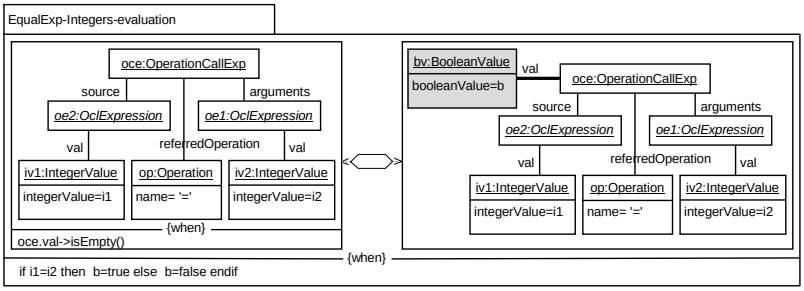

Fig. 13. Equal Operation Evaluation for Integers

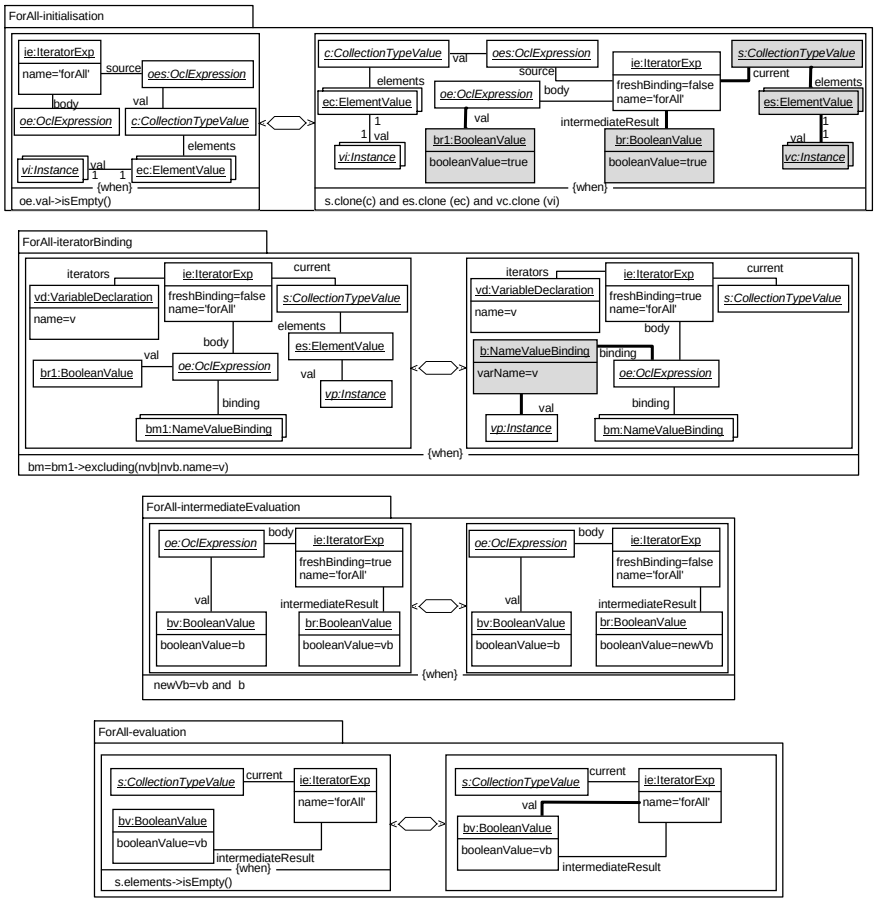

Fig. 14. ForAll - Evaluation Rules

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

We show here nevertheless a semantics for `forAll`, that is independent from the semantics of `iterate`. The rules describing the semantics of `forAll` are, compared with `iterate`, easier to understand, but contain already all mechanisms needed to describe `iterate`(see Fig. 14).

The rule *ForAll-Initialisation* makes a copy of evaluation of the source expression, and assigns it under the role *current* to `ie`. Furthermore, the role *intermediateResult* is initialized with *true* and, for some technical reasons, the attribute *freshBinding* of `ie` is set to *false* and the evaluation of body expression `oe` is also initialized with *true*.

The rule *ForAll-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 evaluation of body expression `oe` is removed (note that the binding for `oe` has changed and the old evaluation of `oe` became obsolete).

The rule *ForAll-IntermediateEvaluation* updates the *intermediateResult* of `ie` based on the new evaluation of `oe`. Furthermore, the value of attribute *freshBinding* is flipped.

The final rule *ForAll-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 *intermediateResult* currently has.

Atomic Expressions This category consists of expressions such as *LiteralExp* and *VariableExp* that do not have any subexpressions. As an example we present rules for these two cases. In Fig. 15, the evaluation of *IntegerLiteralExp* is shown. By applying this rule, a new *IntegerValue* is created that refers to the same integer as attribute *integerSymbol* in `ie`. Note, that this type of expressions does not need variable bindings because their evaluation does not depend on the evaluation of any variable. Figure 16 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.

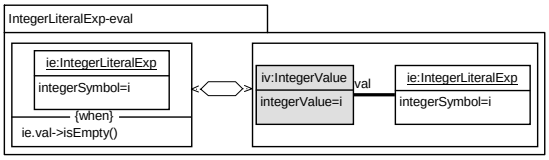

Fig. 15. Integer Literal Expression Evaluation

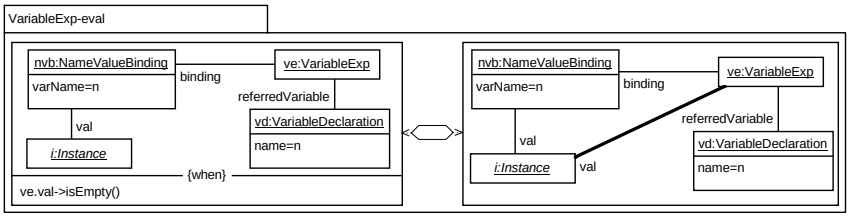

Fig. 16. Variable Expression Evaluation

4 Related Work

The only paper we are aware of that shares similar interests in applying a graph-transformation based approach in order to deal with OCL constraints is [9]. In this paper, a graphical visualization of OCL constraints is proposed. On top of this notation, simplification rules for OCL constraints are proposed, that implicitly also define a semantics for OCL. However, the semantics of OCL is not developed as systematically as in our approach, only the simplification rules for `select` are shown. Since [9] was published at a time where OCL did not have an official metamodel, the graph-transformation rules had to be based on another language definition.

For a different kind of languages, behaviorial languages, Engels et al. define in [10] their 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.

Stärk et al. define in [11] a formal semantics of Java. Even if they use a completely different notation to specify an operational semantics, we see nevertheless a lot of striking similarities. Stärk et al. map the state space of a Java program to an Abstract State Machine (ASM) and describe possible state changes by a set of ASM rules that manipulate the Abstract Syntax Tree of a program. As shown in our motivating example, 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).

5 Conclusions and Future Work

We developed a metamodel-based, graphical definition of the semantics of OCL. Our semantics consists of a metamodel of the semantic domain (we slightly adapted existing metamodels from UML1.x) and a set of transformation rules written in 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 adapt the semantics of OCL to domain-specific needs. Since the evaluation rules can directly be executed by any QVT compliant tool, it is now very easy to provide tool support for a new dialect of OCL. This is an important step forward to the OMG's vision to treat OCL as a family of languages.

We are currently investigating how an OCL semantics given in form of QVT rules can be used to argue on the semantical correctness of refactoring rules for UML/OCL, which we have defined as well in form of QVT rules. A refactoring rule describes small changes on UML class diagrams with attached OCL constraints. A rule is considered to be *syntactically correct* if in all applicable situations the refactored UML/OCL model is syntactically well-formed. We call a rule *semantically correct* 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. A more detailed description together with a complete argumentation on the semantical correctness of the MoveAttribute refactoring rule can be found in [12].

Another branch of future activities is the description of the semantics of programming languages with graphical QVT rules. Our ultimate goal is 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 straightforward as it is today already the case with the syntax of languages.

References

1. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, second edition, 2005.
2. Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.
3. 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.
4. OMG. UML 2.0 OCL Specification – OMG Final Adopted Specification. OMG Document ptc/03-10-14, Oct 2003.
5. Achim D. Brucker and Burkhart Wolff. A proposal for a formal OCL semantics in Isabelle/HOL. In Victor Carreño, César Muñoz, and Sofiène Tashar, editors, *TPHOLs*, volume 2410 of *LNCS*, pages 99–114. Springer, 2002.
6. OMG. UML 1.5 Specification. OMG Document formal/03-03-01, March 2003.
7. OMG. UML 2.0 Infrastructure Specification. OMG Document ptc/03-09-15, Sep 2003.
8. OMG. Meta object facility (MOF) 2.0 Query/View/Transformation Specification. OMG Document ptc/05-11-01, Nov 2005.
9. Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Consistency checking and visualization of OCL constraints. In *UML 2000*, volume 1939 of *LNCS*, pages 294–308. Springer, 2000.
10. 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 *UML 2000*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.
11. Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer, 2001.
12. Thomas Baar and Slaviša Marković. A graphical approach to prove the semantic preservation of UML/OCL refactoring rules. In *Proceedings, Sixth International Andrei Ershov Memorial Conference , Perspectives of System Informatics (PSI), Novosibirsk, Russia*, LNCS. Springer, July 2006. To appear.