Softw Syst Model (2008) 7:399–422
DOI 10.1007/s10270-008-0083-2

SPECIAL SECTION PAPER

Semantics of OCL specified with QVT

Slaviša Marković · Thomas Baar

Received: 19 March 2007 / Revised: 23 January 2008 / Accepted: 29 January 2008 / Published online: 11 March 2008

Abstract The Object Constraint Language (OCL) has been for many years formalized both in its syntax and semantics in the language standard. 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 this paper, 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, our semantics definition for OCL can be automatically transformed into a tool, which evaluates OCL expressions in a given context. 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 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.

Communicated by Prof. Oscar Nierstrasz.

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

S. Marković (✉) · T. Baar
École Polytechnique Fédérale de Lausanne (EPFL),
School of Computer and Communication Sciences,
1015 Lausanne, Switzerland
e-mail: slavisa.markovic@epfl.ch

T. Baar
e-mail: thomas.baar@epfl.ch

Keywords QVT · OCL Semantics · Graph-transformations · DSL

1 Introduction

The OCL has proved to be a very versatile constraint language that can be used for different purposes in different domains, e.g., for restricting metamodel instances [1], for defining UML profiles [2], for specifying business rules [3], for querying models [4,5] or databases [6].

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 [7]. Nowadays, a user can choose among many OCL parsers (e.g. OSLO [8], Eclipse Model Development Tool (MDT) for OCL [9], Dresden OCL Toolkit [10], OCTOPUS [11], USE [12], OCLE [13]), which strictly implement the abstract syntax of OCL defined in the OCL standard [14].

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 [14], 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.

[1] There is also an *informative semantics* given in Annex A of [14], which is formulated in a set-theoretical style and goes back to the dissertation of Richters [15].

In this paper we present an 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 will, to a great extent, reuse package *Values* 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 postconditions). 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 [16]. In order to improve readability, we use in this paper a visualization of QVT rules, which is inspired from graph-grammars. All QVT rules presented in this paper are also available in its textual form. The complete set of rules can be downloaded, together with all relevant metamodels, from [17]. Note that the QVT rules are executable on QVT-compliant engines, what is demonstrated by our OCL tool RoclET [17], which uses internally the QVT rules for the evaluation of OCL expressions. The QVT engine of our choice was Together Architect 2006 [18] which offers mature support for QVT editing and debugging. Please note, however, that our tool RoclET was intended to serve only as a reference implementation for the OCL semantics described in this paper. The goal of the RoclET development was not to build an optimized OCL tool for industrial applications. RoclET can nevertheless be used for teaching purposes. For example, the evaluation of about 80 invariants on ten objects takes less than 4 s.[2]

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 ([19,20]) 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 Domain-Specific Language (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., [21]) 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.

This paper is a revised and enhanced version of [22]. While [22] concentrates of the evaluation semantics for invariants, we have added to this paper also rules for the evaluation of pre-/postconditions. Furthermore, many rules were redesigned with the aim to make OCL's underlying semantic concepts more explicit and to make evaluation rules more reusable in other language definitions. We also added a section on tailoring the semantics of OCL towards the needs of a DSL.

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 basic evaluation steps are formalized by QVT rules in Sect. 3. The formalized QVT rules have to be consistent to each other,

[2] The exact results depend, of course, on the structure and length of evaluated invariants, the performance of the used computer, etc.

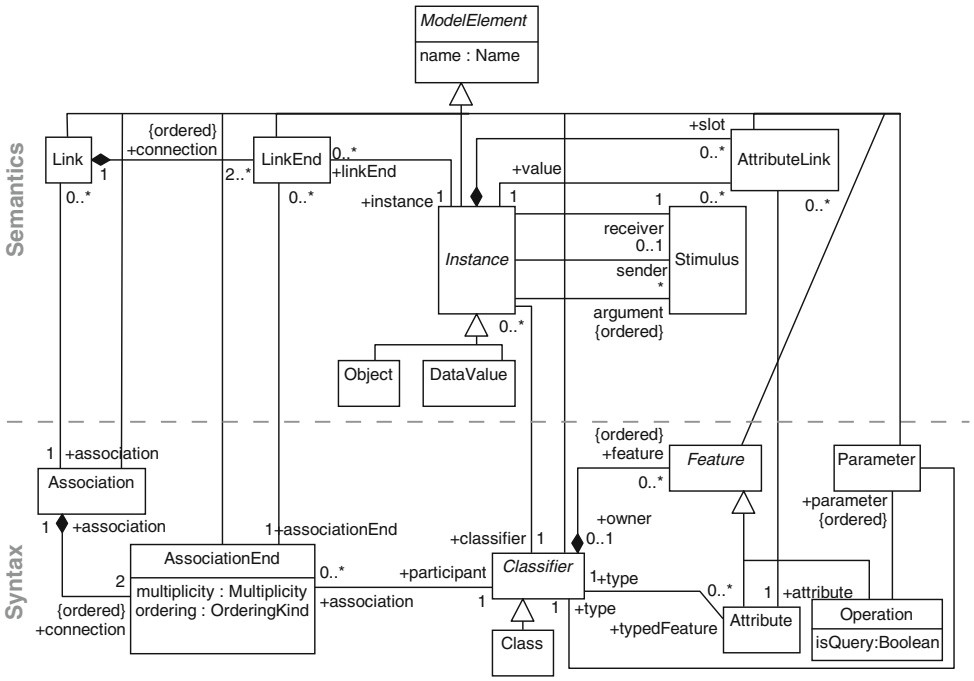


Fig. 1 Metamodel for class diagrams—syntax and semantics

at least to a certain degree. Achieving consistency is more likely, if the underlying semantic concepts are made more explicit. Section 4 proposes a list of semantic concepts and discusses their impact on evaluation rules. Section 5 shows the flexibility of our approach and presents a stepwise adaptation of OCL's semantics, so that the adapted version can be used as a constraint language for a given DSL. In Sect. 6, we compare our approach with existing approaches for formalizing the semantics of constraint languages. Section 7 draws conclusions.

2 A 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. We concentrate on the evaluation of an invariant constraint in a given state. Difficulties arising from the evaluation of pre-/postconditions are discussed in Sect. 4.

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 [14]. However, since our approach had the requirement to be integrated in the OCL tool RoclET, which currently does only support UML1.5 diagrams, we refer also in this paper to UML1.5 as the metamodel of the UML part, on which OCL constraints rely. 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.

2.2 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 package *Values*, which is part of the official OCL metamodel (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 *LoopExp* 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 data-object from its value. We have created two new metaclasses *StateTransition* and *ObjectMap* that are used in evaluations of pre- and postconditions. 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.

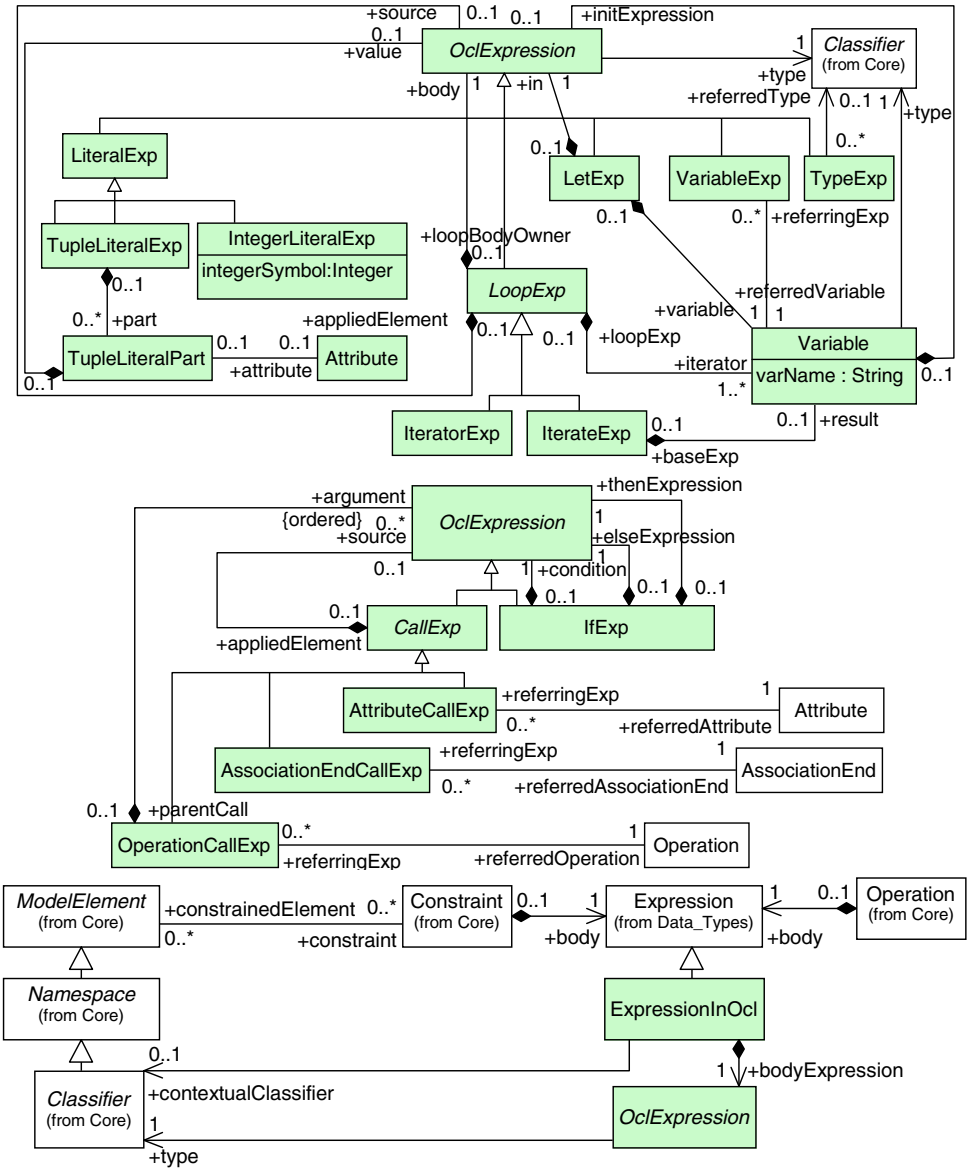


Fig. 2 Metamodel for OCL—syntax

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 or equal to 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. Steps (a)–(b) performs the evaluation of the leaf nodes. Depending on the results of these evaluations, steps (b)–(c) performs evaluation of nodes at the middle level. Finally, the last steps (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. 2.4.

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

Figure 6 shows the instance of the OCL metamodel representing the invariant from Fig. 4. Here, we stipulate that all

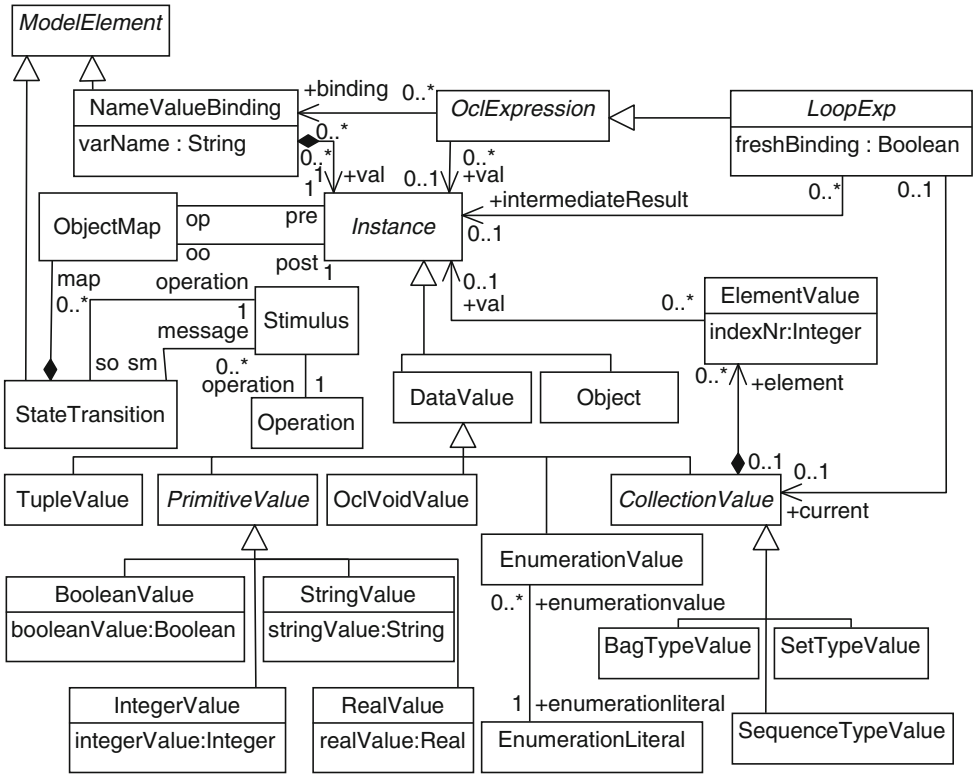


Fig. 3 Changed metamodel for OCL—semantics

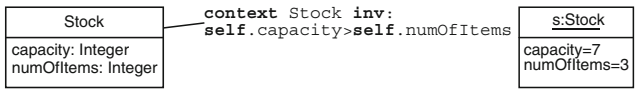


Fig. 4 Example—class diagram and snapshot

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 object `o`. For the sake of readability this information is omitted in Figs. 6 and 7.

The final state of the metamodel instance, i.e. 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 *AttributeCallExp*s are evaluated to two *IntegerValue*s 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 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, called *self*. Although *self* is a predefined variable in OCL, it can be treated the same way as all other variables, which are introduced in *LoopExp*. For example, the invariant

```
context Stock inv:
 self.capacity > self.numOfItems
```

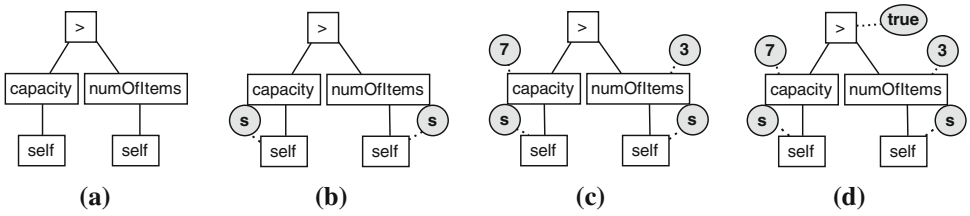


Fig. 5 Evaluation of OCL expressions seen as an AST: **a** initial AST, **b** leaf nodes evaluated, **c** middle nodes evaluated, **d** complete AST evaluated

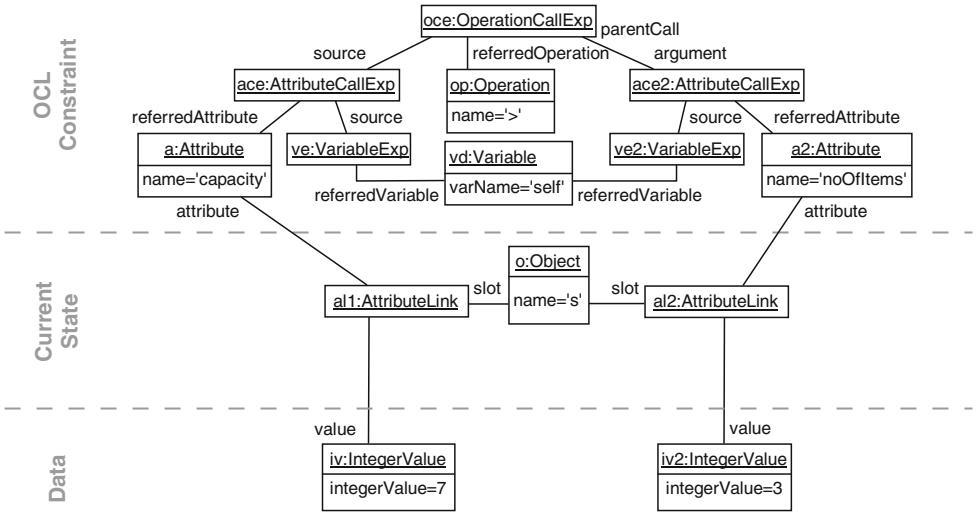


Fig. 6 OCL Constraint before evaluation

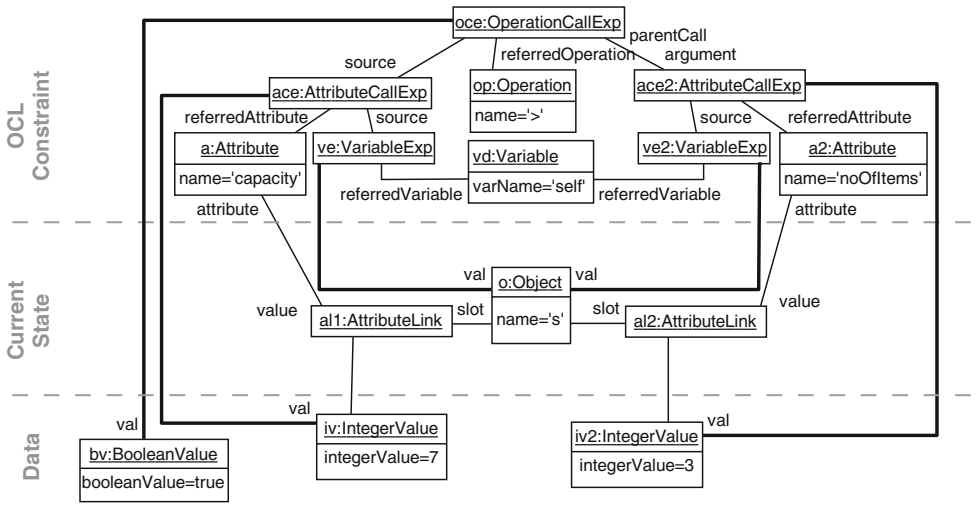


Fig. 7 OCL Constraint after evaluation in a given snapshot

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 change them. An example for adding new value-name bindings will be presented in more details in Sect. 3 where the evaluation rules for *iterate* and *let* 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 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 graph-transformation rules are generally readable and understandable nicely, 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

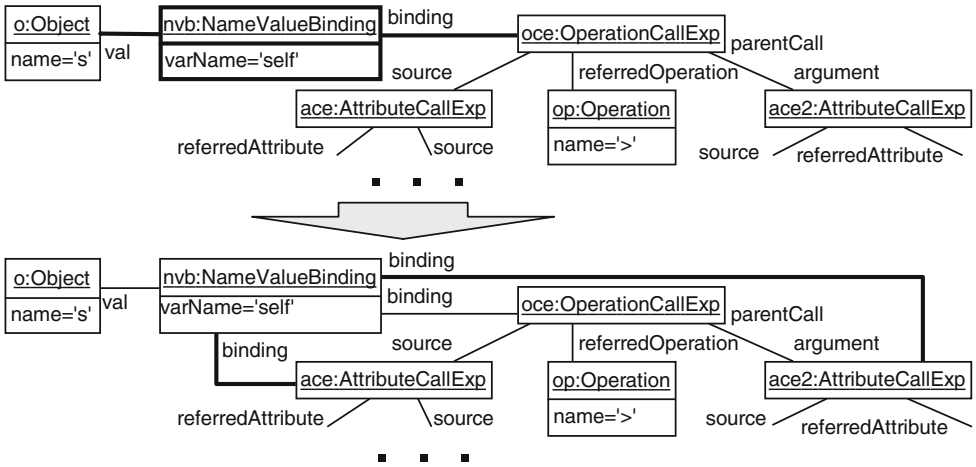


Fig. 8 Binding passing

of evaluation rules for certain types of expressions and will explain in the next Sect. 4 how these core rules have to be extended/adapted in order to reflect all semantic concepts of OCL.

3.1 Model transformation rules

For the specification of evaluation rules we use the formalism of model transformations, more precisely a graphical syntax of QVT (Query/View/Transformation) rules [16].

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.

While in the conference version of this paper [22] we have stuck to the official syntax of QVT rules, we take now the freedom to introduce some additional shorthand notations, which will help to improve conciseness and readability of evaluation rules. One source for complexity of the rules given in [22] is the rules' LHS containing two subpatterns; one for the structure to look for in the OCL syntax tree (e.g. *AttributeCallExp*) and one for the structure in the state, in which the constraint is evaluated. The RHS has again two patterns; one for the updated structure of the OCL syntax tree and one for the structure in the state. Since the evaluation of OCL expressions does not have side-effects on the state in which the expression is evaluated, the state-subpattern of LHS must be the same as the subpattern of RHS.

In order to avoid the redundancy of having the same subpatterns in LHS and RHS, our evaluation rules contain besides LHS and RHS a third part called *Context*, that specify the structures in the input, which must be available when applying the rule but which are not changed (see Fig. 9 for a comparison of the old and the new form of evaluation rules). 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 rules.

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 10 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). 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.

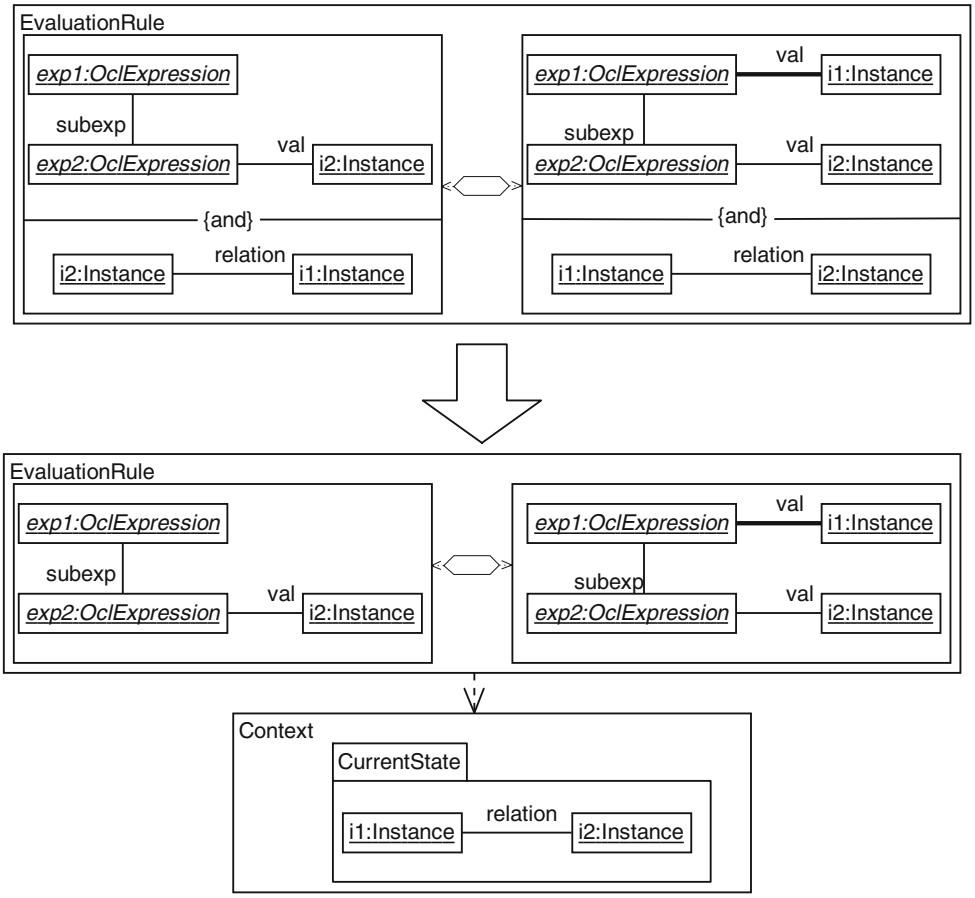


Fig. 9 Format of evaluation rules used in [22] and in this paper

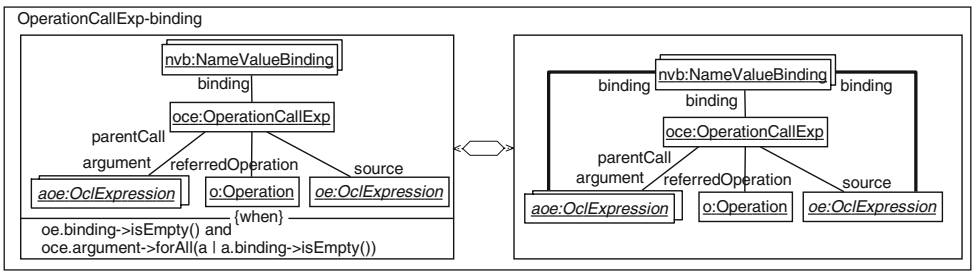


Fig. 10 Binding of an expression

3.3 A catalog of core rules

Each OCL expression is an instance of the metaclass *OclExpression* in the OCL metamodel; more precisely—since *OclExpression* is an abstract metaclass—an instance of one of the non-abstract subclasses of *OclExpression*. For each of these non-abstract metaclasses, the semantics definition must have at least one evaluation rule.

The semantics of a constraint language such as OCL can be split along this syntactic dimension (in Sect. 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*,

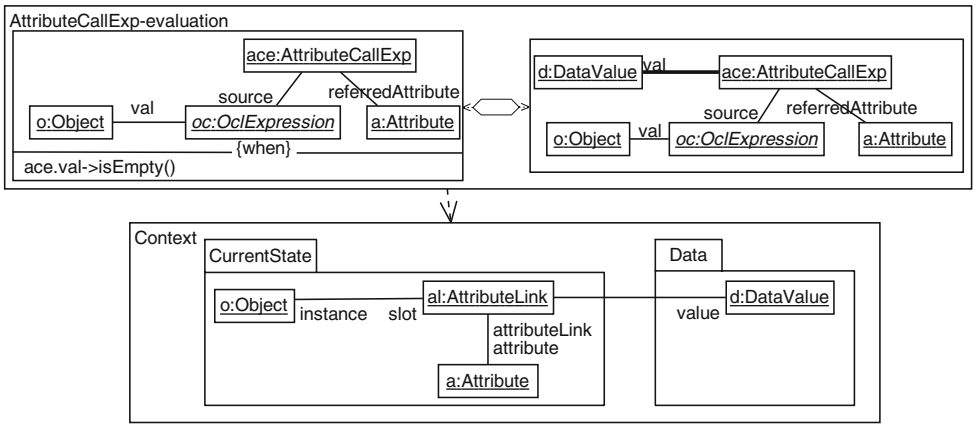


Fig. 11 Attribute call expression evaluation

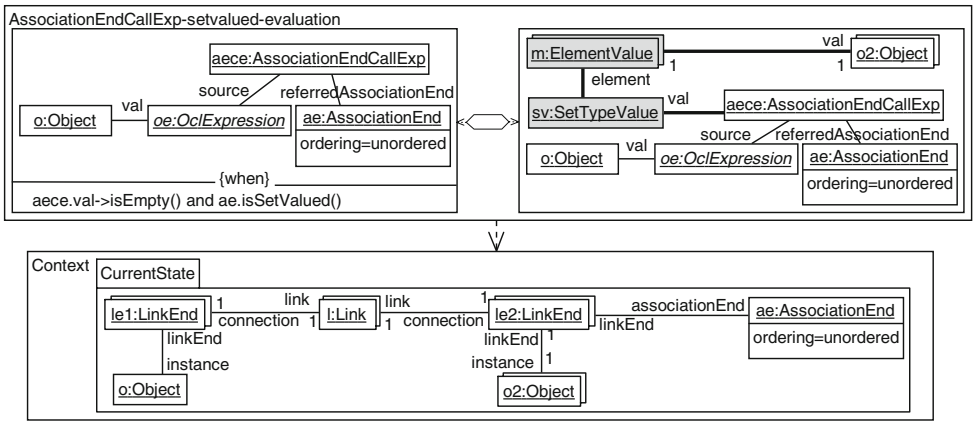


Fig. 12 Association end call expression evaluation that results in set of objects

If-Expressions, *Message Expressions*[3], *Let-Expressions*, *State Expressions*,[4] *Tuple Expressions*. For the class of *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.

3.3.1 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. 11. The evaluation of `ace` is data value `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`.

AssociationEndCallExp We discuss here only the case of navigating over 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. 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`. The rule shown in Fig. 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. 12 where each element of multiobject `l` must be connected only to one element from multiobject `le2`, and vice

[3] *Message Expressions* can occur only in postconditions and are ignored here.

[4] 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.

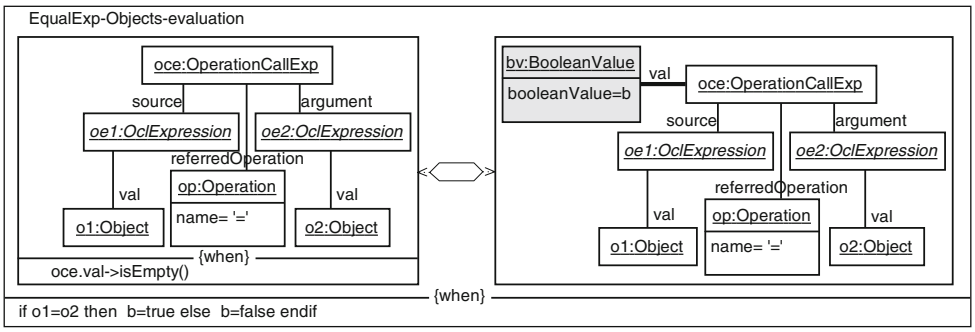


Fig. 13 Equal operation evaluation for objects

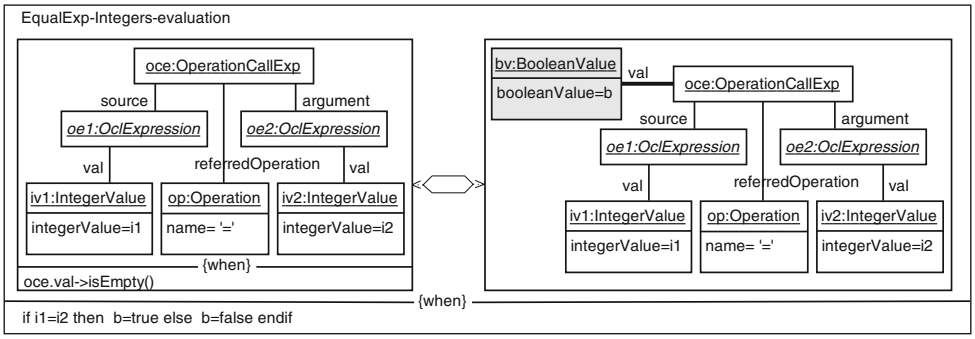


Fig. 14 Equal operation evaluation for integers

versa. By using 1–1 multiplicities, we indicate a non-standard semantics of links between two multiobjects.

3.3.2 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 the chapter on the OCL library in [14]. 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. 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.

If `oe1` and `oe2` evaluate to *IntegerValue*, the second QVT rule shown in Fig. 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.

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 by a 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 clause body under the current argument binding.

Figure 15 shows evaluation rules for user-defined queries. The first rule creates a set of *NameValueBinding*s 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 the evaluation of the query in such a way, that if the *precondition* does not evaluate to *true* then the result of the evaluation will be *undefined*, otherwise the result will be the result of evaluating *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 lead in some but not all cases to infinite loops during the evaluation. Brucker et al. propose in [23] that recursive query definitions should be checked by the user for unfounded recursions. In principle, such a check is possible but it requires 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

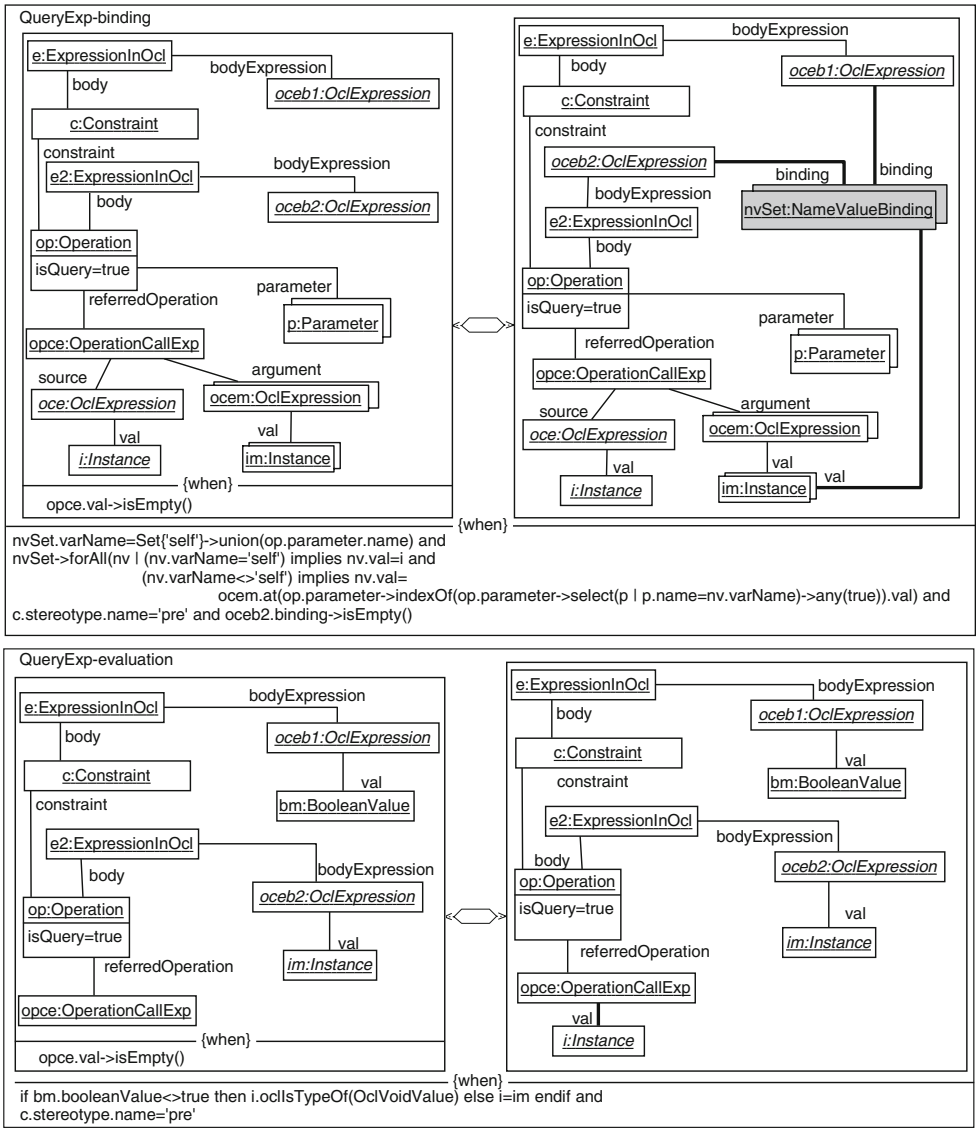


Fig. 15 Evaluation of an expression referring to a query

whole expression is evaluated to *undefined*. Because we treat the evaluation to *undefined* in the next Sect. 4 in a general manner, we skip the rule for *oclAsType* here. The rules for *oclIsTypeOf* and *oclIsKindOf* are very similar; Fig. 16 shows the 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. 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.

3.3.3 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*, *sortedBy*, 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 Fig. 18 are shown evaluation rules that describe the semantics of *iterate*.

The rule *Iterate-Initialisation* makes a copy of the evaluation of the source expression, and assigns this copy 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 variable *result* and its value is the same as the value of the *initExpression* for *result*. For some technical reasons, 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

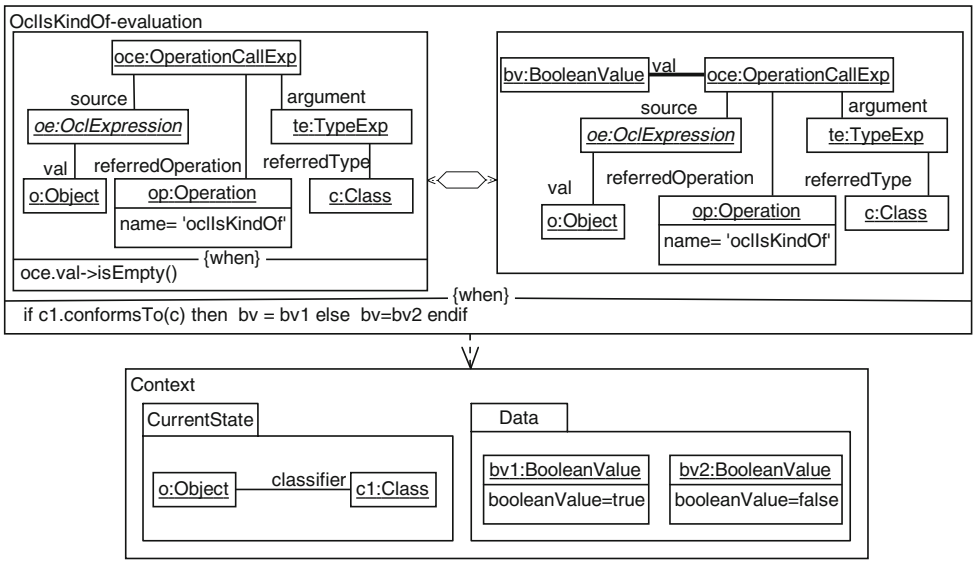


Fig. 16 Evaluation rule for oclIsKindOf

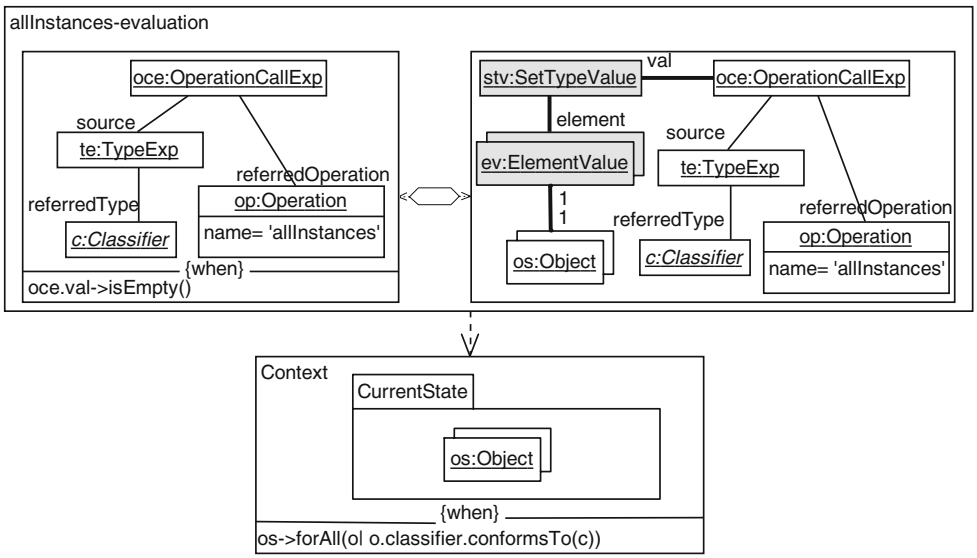


Fig. 17 Evaluation rule for allInstances

value `vp`. The element with the same value `vp` is chosen from 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.

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.

3.3.4 Variable expressions

Figure 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.

3.3.5 Literal expressions

In Fig. 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

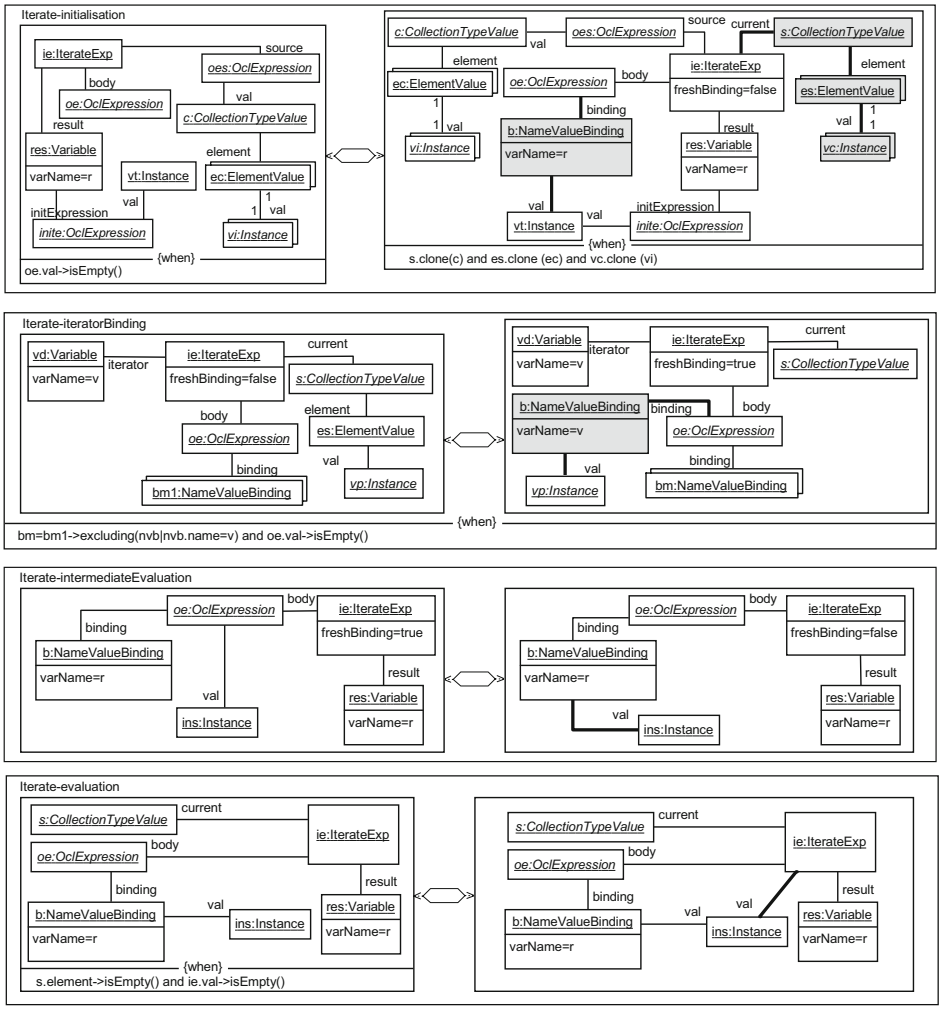


Fig. 18 Iterate—evaluation rules

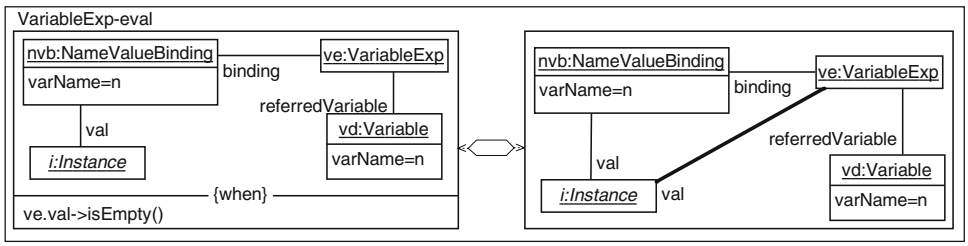


Fig. 19 Variable expression evaluation

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.

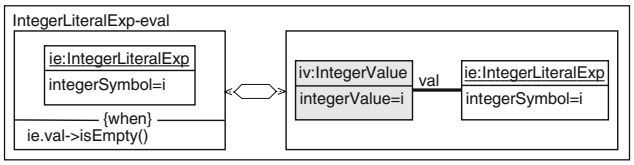


Fig. 20 Integer literal expression evaluation

3.3.6 If-expressions

Figure 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 do not deal with evaluation to *undefined* and that this aspect of OCL will be discussed later.

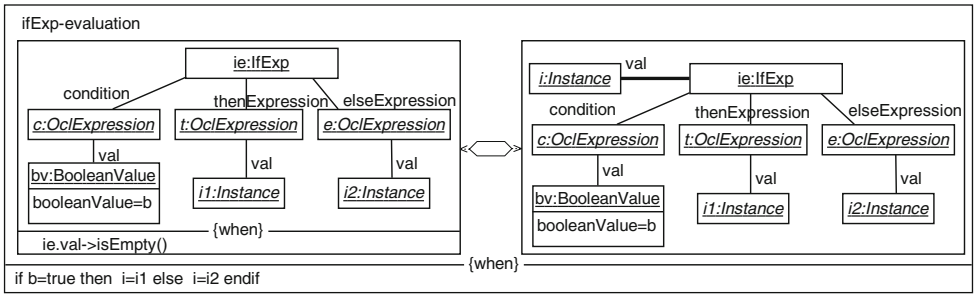


Fig. 21 If-expression evaluation

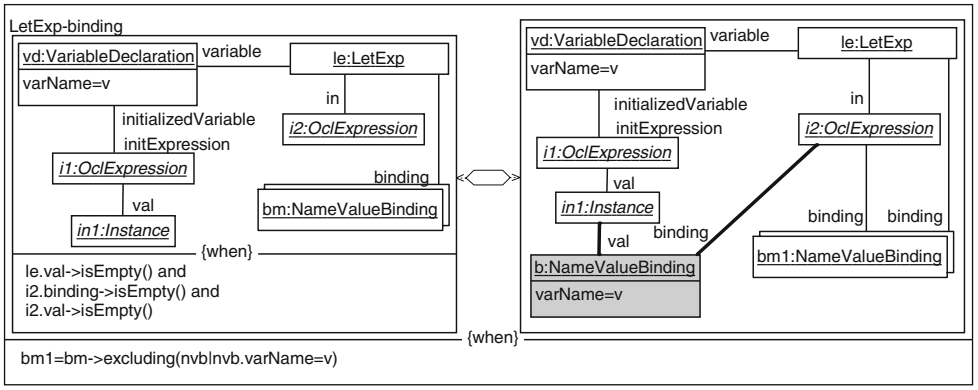
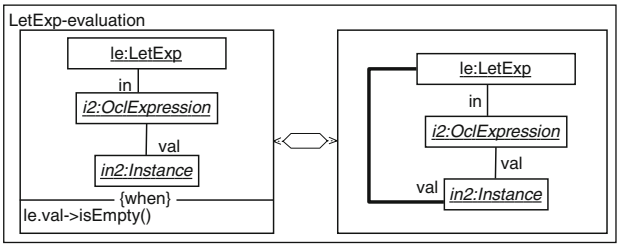


Fig. 22 Let expression: binding and evaluation



3.3.7 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 *Loop Expressions*). The evaluation rules for *LetExp* are shown in Fig. 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.

3.3.8 Tuple expressions

In Fig. 23, the evaluation rule for *TupleLiteralExp* 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 *TupleLiteralExp* 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 *TupleLiteralExp*.

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 operation *iterate*. More precisely, as described in [14], expressions of form

```
coll->exists(x| body(x))
```

can be rewritten to

```
coll->iterate(x; acc:Boolean=false| 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 24 shows a QVT rule, which transforms one *exists*-expression into corresponding *IterateExp*. RHS of the rule

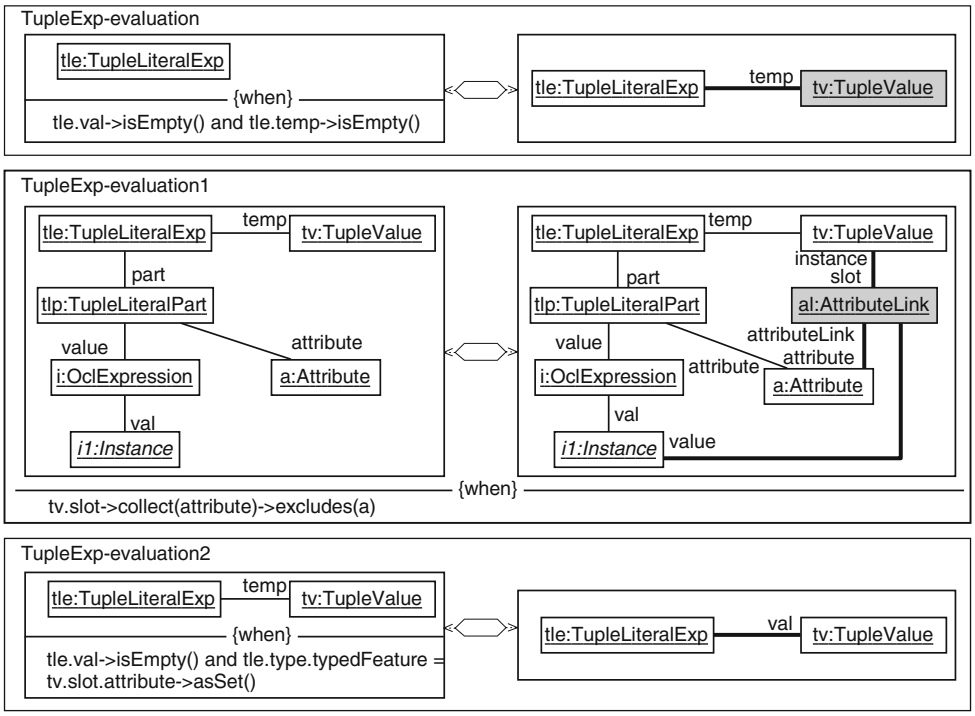


Fig. 23 Tuple expression evaluation

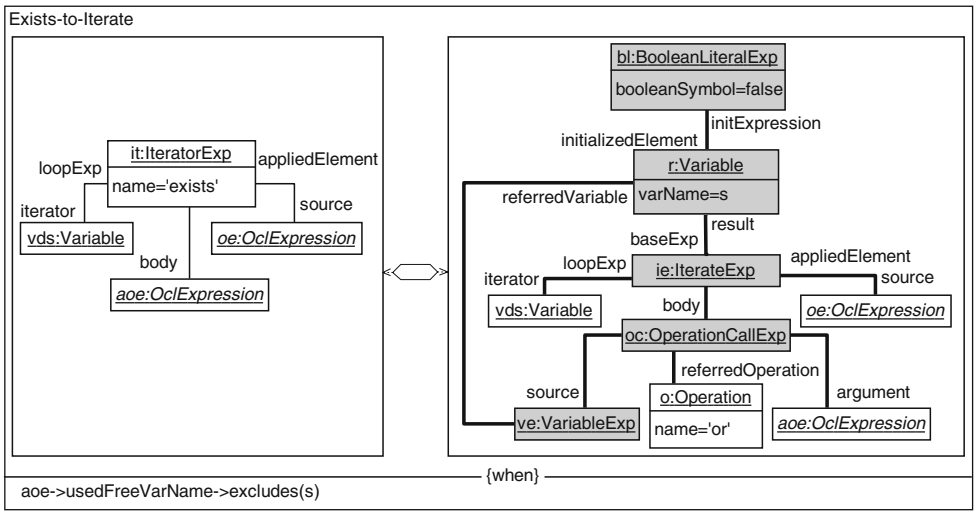


Fig. 24 Transforming exists expression to an iterate expression

states that a new *IterateExp* is created, together with a new *VariableDeclaration* and a new *BooleanLiteralExp* with *booleanSymbol* set to *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 *varName* `s` used in the newly created *VariableDeclaration* is not yet used as a name by any of the free variables in the body.

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*,

which are supported by OCL but which are most likely not supported by every other constraint language. Besides the syntactic dimension already explained in Sect. 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. 3.3 only the rudimentary version of evaluation rules has been shown):

- evaluation of operation contracts (pre-/postconditions)
- evaluation to *undefined* (including strict evaluation with respect to *undefined*, with some exceptions)
- dynamic binding when invoking a query
- non-deterministic constructs (*any*(), *asSequence*())[5]

In the next subsections, we discuss the semantical concepts that have the most impact on the evaluation rules from Sect. 3.3.

4.1 Evaluation of operation contracts

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

Metaclass *StateTransition* from our metamodel (see Fig. 3) is used to capture one transition from a pre- to a post-state. This transition represents one concrete operation execution with concrete values passed as operation parameters. In order to be able to evaluate one pre- or one postcondition, we need all information about the state transition for which we want to perform the evaluation: operation that caused the transition, values of 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 to which the *Context* of the evaluation rule refers to is in this case just the pre-state. In addition, the bindings for the operation arguments have to be extracted from a *Stimulus* that belongs to the *StateTransition* 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 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.

[5] Non-deterministic constructs lead to semantical inconsistencies as one of the authors argues in [24]. They are not further discussed here.

In the semantics of OCL described in [14, 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. The 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 metaclass *ObjectMap*. 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, this 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.

Figure 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 no other *ObjectMap*s exist, 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 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 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

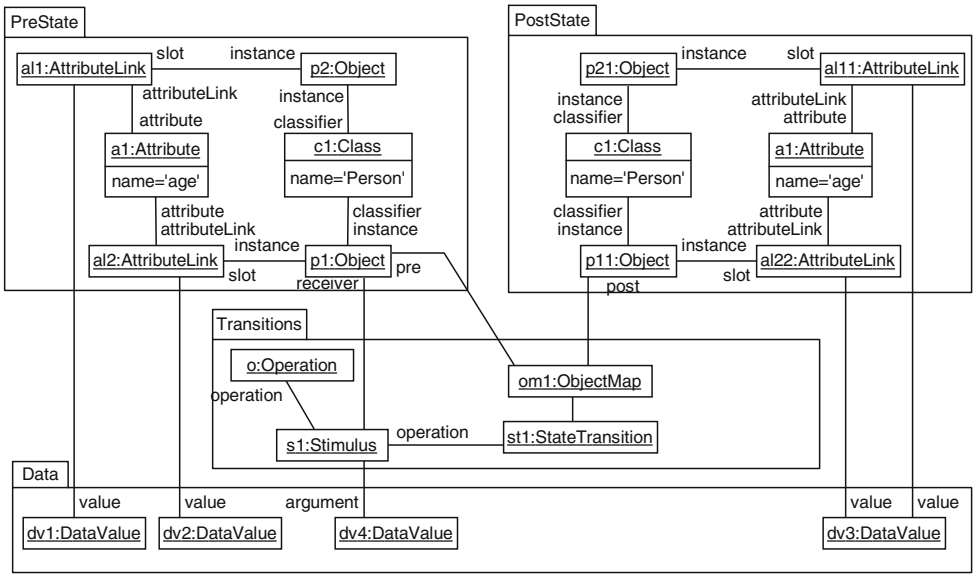


Fig. 25 Relationship between pre- and post-state

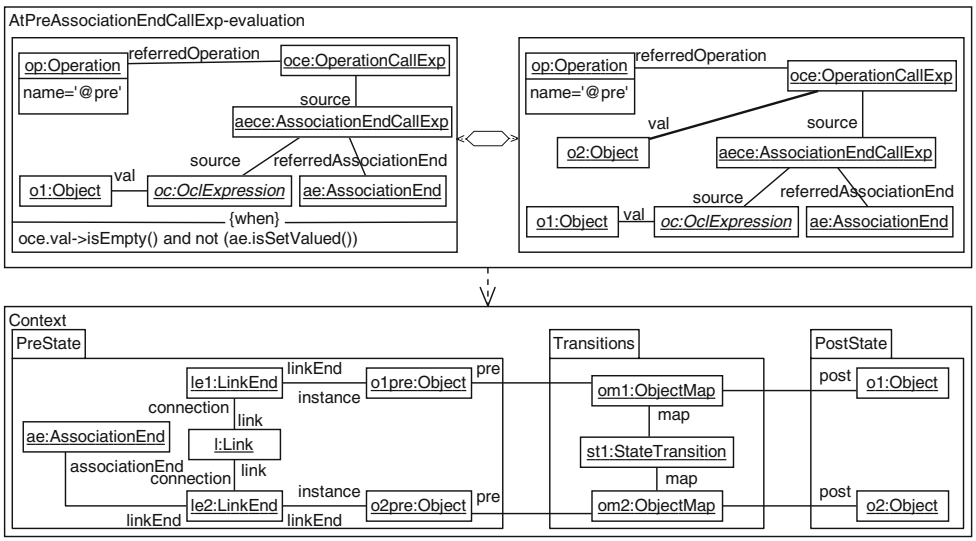


Fig. 26 Evaluation of @pre attached to an object-valued association end call expression

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 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.

4.2 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 value *null*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 value *undefined* was added to the semantic domain for the sole purpose to indicate exceptional situations during the 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*(), it is possible to test if an expression actually evaluates to *undefined*; what—together with the exception from strict evaluation for *and*, *or*, *implies*, *forAll* etc.—is a powerful tool to write OCL constraints reflecting the semantics intended by the user.

But when is an expression actually 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 the evaluation results to *undefined*. 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 the current situation.

In order to illustrate the idea, we have a look to the rule for *@pre* applied on association end call expressions (Fig. 26). If, for example, 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 corresponding object `o1pre` exists but does not have a link for association end `ae`. Another reason could be that the link exist but the referred object `o2pre` is deleted during the state change. In all these cases, the *@pre*-expression should be evaluated to *undefined* and all these cases have in common that the pattern given in the *Context* does not match.

4.3 Dynamic binding

Dynamic binding (also called late binding) is one of the key concepts in object-oriented programming languages but has been mostly ignored in the OCL literature so far. Dynamic binding becomes relevant for the evaluation of user-defined queries. Consider 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`. Please note that besides by using the body expressions, queries can be defined using `def` expressions. Moreover, assume to 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 to 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. 15 does not realize dynamic binding so far because it does not take into account the potential inheritance hierarchy in the model. The result of the second rule shown in Fig. 15 is the value of any possible body expression (`oceb2`) regardless its context.

For this 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()`, this would be the second body defined with expression 7).

In order to transform the static-binding evaluation rules for queries shown in Fig. 15 into 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<>true then i.oclIsTypeOf(OclVoid
        Value)
else i=op.getRightBody(opce.source.val.oclAsType
        (Object).classifier->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).contextual
Classifier->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
```

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, which 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. 27) will be used as an illustrating example, for which we develop domain-specific constraints. Each table has one primary key (`personID` for table `Person` and `dogID` for table `Dog`). In addition, column `ownerID` of table `Dog` has a foreign key relationship with column `personID` of table `Person`.

A simple metamodel for relational databases is shown in Fig. 28. This language is sufficient to specify the database from Fig. 27. Please note that, for the sake of simplicity, we have avoided to introduce database-specific types, but reuse

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

Fig. 27 An example of a relational database

already existing UML/MOF primitive types as types for table columns.

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.

As an example for these two new navigation expressions consider the following constraint:

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

This example constraint 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 much longer:

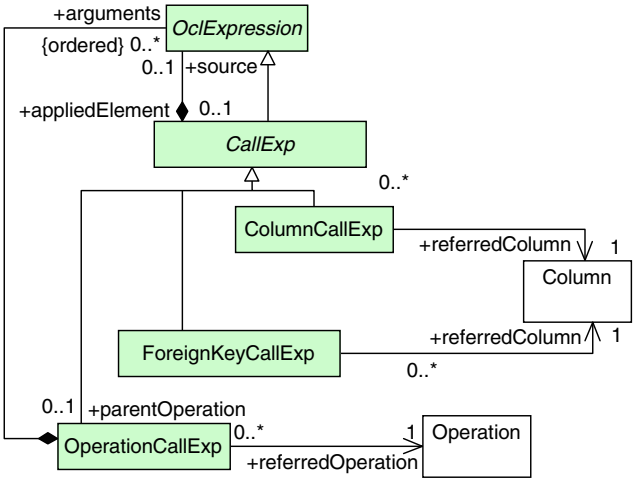


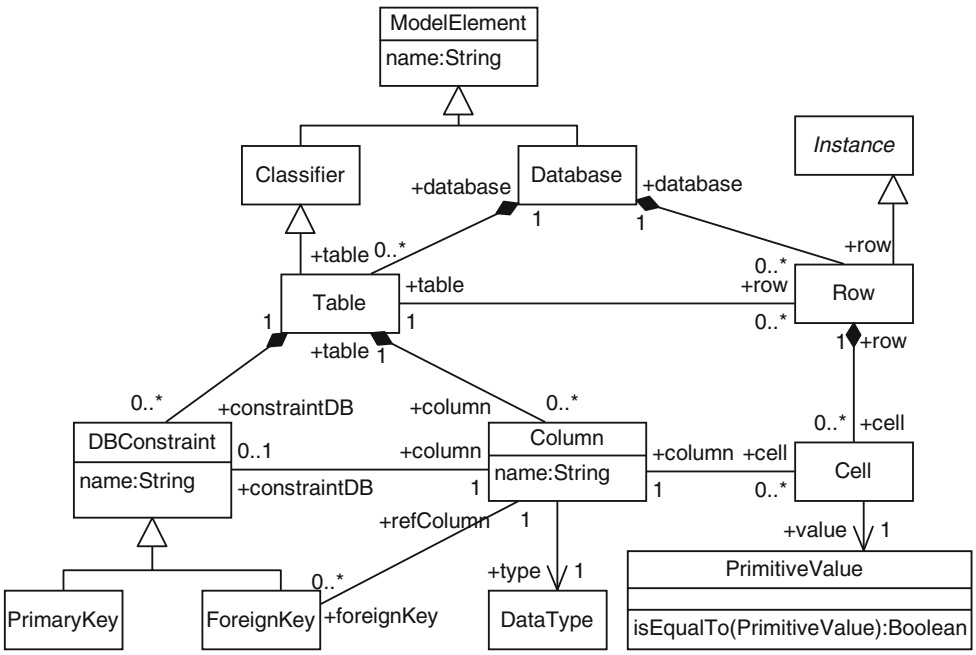
Fig. 29 DSL Navigation expressions

```
Dog.allInstances()->select(d|d.breed='Doberman')
->forAll(dd|Person.allInstances()
->any(p|p.personID=dd.ownerID).age>18)
```

In order to incorporate ordinary and foreign key column navigation into the constraint language, the metamodel for OCL had to be altered. Figure 29 shows the part of the Domain Specific Query language that is different from standard OCL.

Figure 30 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 the value of the *Cell* that belongs to the *Row* that is the source of the expression, and that is referred by the chosen *Column*.

Fig. 28 Relational database metamodel



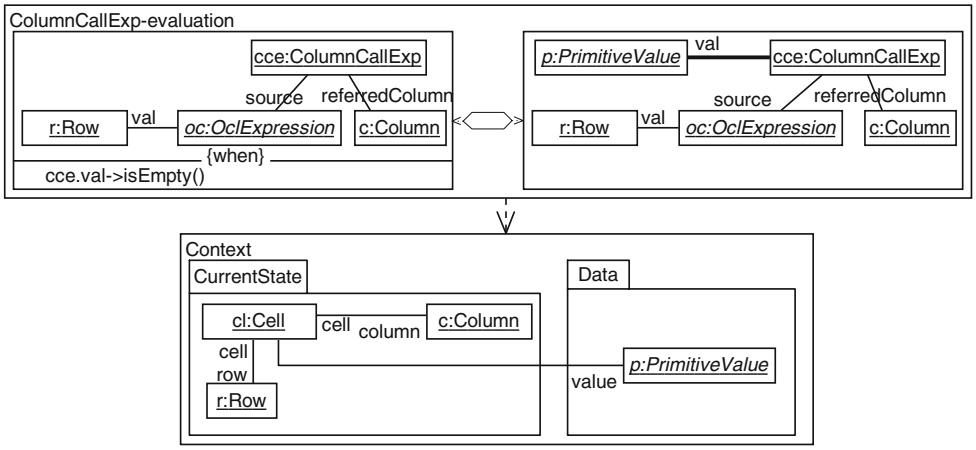


Fig. 30 Semantics of column navigation specified with QVT

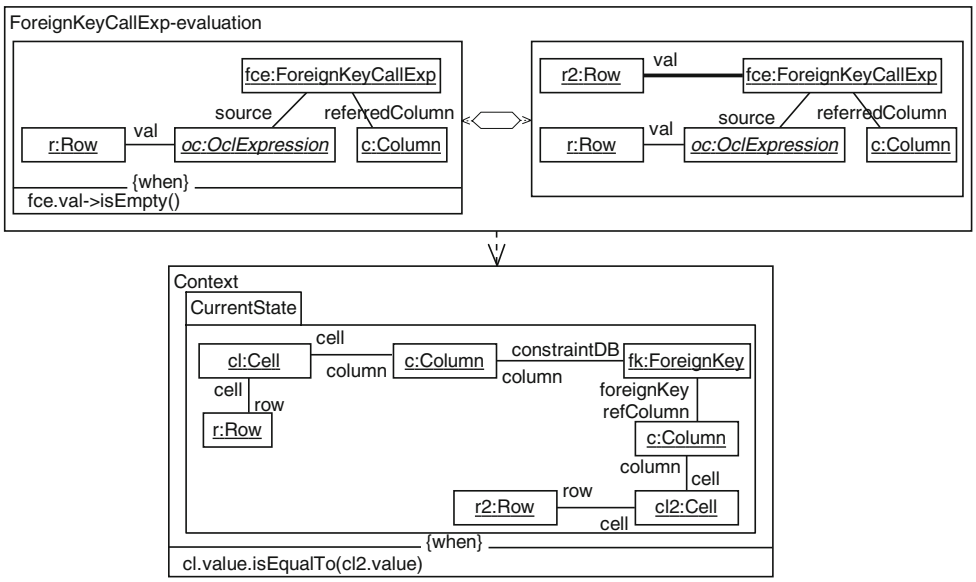


Fig. 31 Semantics of foreign key navigation specified with QVT

The semantics of *ForeignKeyCallExp* is shown in Fig. 31. 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. 32.

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

```
exp.age
```

Please note that the source expression *exp* can be any expression of type *Table*. This very concise expression written in the DSL-specific version of OCL could be emulated by the following plain OCL 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)
```

6 Related work

The work described in this paper 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 separate related work into three categories.

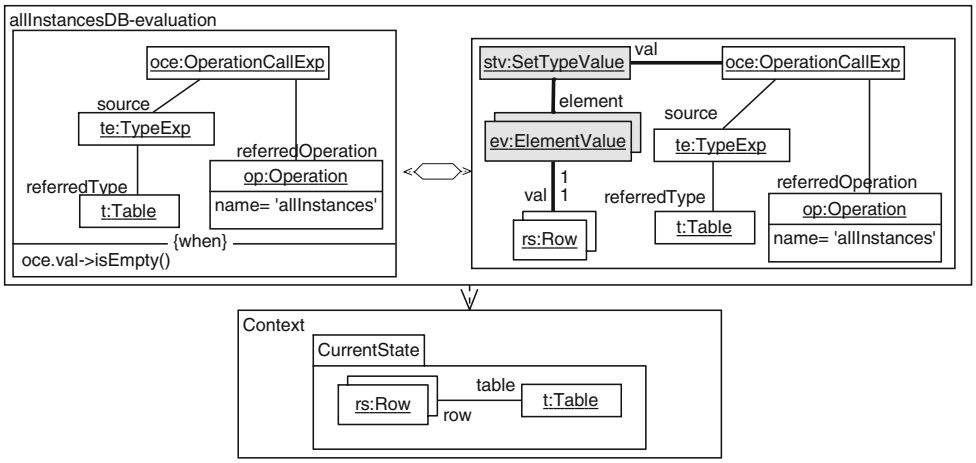


Fig. 32 Semantics of allInstances operation call expression for relational database

6.1 Approaches to define the semantics of OCL

There are numerous papers and even some dissertations that propose a formal semantics for complete OCL or for a fragment of it, e.g., [15,21,25–31] and, recently, [20]. 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., [23,24,32–34].

Though we hope to have addressed in our semantics many of the issues raised in previous papers, there is no guarantee we can give, that our semantics has resolved all problems (a discussion on this would deserve another paper). What is more relevant for the current paper is to compare *the technique*, which has 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.

6.1.1 Official OCL semantics: informative

Annex A of [14] presents a set-theoretical semantics for OCL, which goes back of the dissertation of Mark Richters [15]. 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 of 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 Richter's 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 share objects with the same identity. We solved this problem by introducing `ObjectMap` objects (cmp. Sect. 2.2) 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 package *Values*.

One of the most interesting details when comparing the formalization of expression evaluation is the handling of predefined functions. Following Richter, pre-defined 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 $v_1 \neq \perp$ and $v_2 \neq \perp$. 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! For example, the semantics of operation '=' of the object language (OCL) is reduced to the semantics of the operation '=' in the metalanguage (QVT) (see Sect. 3.3.2).

In both cases, it has to be assumed that the semantics of the metalanguage has been already defined *externally* (cmp. also [35]). 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.

6.1.2 *Official OCL semantics: normative*

The semantics described in [14, Sect. 10] *Semantics Described Using UML* is called *normative OCL semantics* and shares the same main goal as our approach: to give 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 package *Values* to encode pre-defined data types and system states. We tried to align our approach as much as possible with this package *Values* (e.g., `NameValueBinding`), but some details differ. Most notable, as already mentioned in the comparison with Richter's semantics, our states never contain identical objects. The normative OCL semantics insists on keeping the identities of objects across states, but this yields to a quite complicate encoding of attribute values 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 abstract syntax, there is exactly one corresponding evaluation class, e.g. `AttributeCallExpEval`. Evaluation classes are complimented by a number of invariants, whose purpose is to specify the evaluation process. In many cases, each of these 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 sub-expressions, what corresponds to our variable binding propagation rules described in Sect. 2.4.

The normative semantics has been also the starting point for a semantics formalization given by Chiaradía and Pons [36]. They alter the OCL semantics' metamodel by introducing the 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.

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 [37] 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 [37] was published at a time where 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. defined 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 [39] 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. An application of this approach is demonstrated by specifying semantics of UML statecharts.

Stärk et al. define in [40] 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. 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).

6.3 Other related work

An interesting classification of OCL language concepts was developed by Chiorean et al. [41]. 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 [41], but it would be definitely worthwhile to investigate the other domains as well.

Kolovos et al. define in [42] a navigation language for relational databases that is similar to our language defined in Sect. 5. They use the metalanguage *EOL* (which is based on OCL) to define the result of evaluation of new expressions like column navigation.

7 Conclusions and future work

We have developed a metamodel-based, graphical definition of the semantics of OCL. Our semantics consists of a meta-

model 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, which specify formally the evaluation of an OCL constraint in a given 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 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, i.e. in QVT. A more detailed description together with a complete argumentation on the semantical correctness of the *MoveAttribute* refactoring rule can be found in [43].

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, in which the semantics of a language can be formally defined as easy and straightforwardly as today's syntax definitions of modeling languages.

References

1. OMG. UML 2.0 Infrastructure Specification. OMG Document ptc/03-09-15, September 2003
2. Berkenkötter, K.: OCL-based validation of a railway domain profile. In: Kühne, T. (ed.) Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Genova, Italy, October 1–6, 2006, Reports and Revised Selected Papers, LNCS, vol. 4364, pp. 159–168. Springer, Heidelberg (2007)
3. Demuth, B., Hußmann, H., Loecher, S.: 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, Lecture Notes in Computer Science, vol. 2185, pp. 104–117. Springer, Heidelberg (2001)
4. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: Octavian Patrascoiu, editor, OCL and Model Driven Engineering, UML 2004 Conference Workshop, 12 October 2004, Lisbon, Portugal, pp. 69–83. University of Kent, Kent (2004)
5. Akehurst, D.H., Bordbar, B.: On querying UML data models with OCL. In: Gogolla, M., Kobryn, C. (eds.) UML 2001—The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, 1–5 October 2001, Proceedings, Lecture Notes in Computer Science, vol. 2185, pp. 91–103. Springer, Heidelberg (2001)
6. Demuth, B., Hußmann, H.: Using UML/OCL constraints for relational database design. In: France, R.B., Rumpe, B. (eds.) UML'99: The Unified Modeling Language—Beyond the Standard, Second International Conference, Fort Collins, CO, USA, 28–30 October 1999, Proceedings, Lecture Notes in Computer Science, vol. 1723. Springer, Heidelberg (1999)
7. Bauerdick, H., Gogolla, M., Gutsche, F.: Detecting OCL traps in the UML 2.0 superstructure: an experience report. In: Baar, T., Strohmeier, A., Moreira, A.M.D., Mellor, S.J. (eds.) UML 2004—The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference, Lisbon, Portugal, 11–15 October 2004. Proceedings, Lecture Notes in Computer Science, vol. 3273, pp. 188–196. Springer, Heidelberg (2004)
8. Oslo, T.: Oslo project (2007). http://oslo-project.berlios.de/
9. MDT-OCL Team: Eclipse MDT—OCL project (2007). http://www.eclipse.org/modeling/mdt/?project=ocl
10. Dresden OCL Team: Dresden OCL Toolkit (2007). http://dresden-ocl.sourceforge.net/
11. OCTOPUS Team: OCTOPUS—OCl TOol for Precise Uml Specifications (2007). http://octopus.sourceforge.net/
12. USE Team: USE—a UML-based Specification Environment (2007). http://www.db.informatik.uni-bremen.de/projects/USE/
13. OCLE Team: OCLE–Object Constraint Language Environment (2007). http://lci.cs.ubbcluj.ro/ocle/index.htm
14. OMG: Object Constraint Language—OMG Available Specification, version 2.0. OMG Document formal/06-05-01, May (2006)
15. Richters, M.: A precise approach to validating UML models and OCL constraints. PhD thesis, Bremer Institut für Sichere Systeme, Universität Bremen, Logos-Verlag, Berlin (2001)
16. OMG: Meta object facility (MOF) 2.0 Query/View/Transformation Specification. OMG Document ptc/05-11-01, November (2005)
17. RoclET Team. RoclET project (2007). http://www.roclet.org/
18. Borland. Together technologies (2007). http://www.borland.com/together/
19. Brucker, A.D., Wolff, B.: The HOL-OCL book. Technical Report 525, ETH Zurich (2006)
20. Brucker, A.D.: An Interactive Proof Environment for Object-oriented Specifications. PhD thesis, ETH Zurich (2007). ETH Dissertation No. 17097
21. Clark, T., Evans, A., Kent, S.: Engineering modelling languages: a precise meta-modelling approach. In: Kutsche, R.-D., Weber, H. (eds.) 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, LNCS, vol. 2306, pp. 159–173. Springer, Heidelberg (2002)
22. Marković, S., Baar, T.: An OCL semantics specified with QVT. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Proceedings, MoDELS/UML 2006, Genova, Italy, 1–6 October 2006, LNCS, vol. 4199, pp. 660–674. Springer, Heidelberg (2006)

23. Brucker, A.D., Doser, J., Wolff, B.: Semantic issues of OCL: Past, present, and future. In: Demuth, B., Chiorean, D., Gogolla, M., Warmer, J. (eds.) OCL for (Meta-)Models in Multiple Application Domains, pp. 213–228. University Dresden, Dresden (2006) (Available as Technical Report, University Dresden, number TUD-FI06-04-September 2006)
24. Baar, T.: Non-deterministic constructs in OCL—what does any() mean. In: Prinz, A., Reed, R., Reed, J. (eds.) Proceedings of 12th SDL Forum, Grimstad, Norway, June 2005, LNCS, vol 3530, pp. 32–46. Springer, Heidelberg (2005)
25. Richters, M., Gogolla, M.: On formalizing the UML object constraint language OCL. In: Ling, T.W., Ram, S., Lee, M.L. (eds.) Proceedings of 17th International Conference in Conceptual Modeling (ER'98), LNCS 1507, pp. 449–464. Springer, Berlin, (1998)
26. Richters, M., Gogolla, M.: A metamodel for OCL. In: France, R., Rumpe, B. (eds.) UML'99—The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, 28–30 October 1999, Proceedings, LNCS, vol. 1723, pp. 156–171. Springer, Heidelberg (1999)
27. Cengarle, M.V., Knapp, A.: A formal semantics for OCL 1.4. In: Gogolla, M., Kobryn, C. (eds.) UML, Lecture Notes in Computer Science, vol. 2185, pp. 118–133. Springer, Heidelberg (2001)
28. Stephan, F., Wolfgang, M.: Formal semantics of static and temporal state-oriented OCL-constraints. J. Softw. Syst. Model. (SoSym) **2**(3), 164–186 (2003)
29. Hennicker, R., Knapp, A., Baumeister, H.: Semantics of OCL operation specifications. Electronic Notes in Theoretical Computer Science. Proceedings of OCL 2.0 Workshop at UML'03 **102**, 111–132 (2004)
30. Baar, T.: Über die Semantikbeschreibung OCL-artiger Sprachen. PhD thesis, Fakultät für Informatik, Universität Karlsruhe (in German). ISBN 3-8325-0433-8, Logos, Verlag, Berlin (2003)
31. Cengarle, M.V., Knapp, A.: OCL 1.4/5 vs. 2.0 expressions formal semantics and expressiveness. Softw. Syst. Model. **3**(1), 9–30 (2004)
32. Cook, S., Kleppe, A., Mitchell, R., Rumpe, B., Warmer, J., Wills, A.C.: The amsterdam manifesto on OCL. In: Clark, T., Warmer, J. (eds.) Object Modeling with the OCL: The Rationale behind the Object Constraint Language, pp. 115–149. Springer, Heidelberg (2002)
33. Flake, S.: Ocltype—a type or metatype? Electr. Notes Theor. Comput. Sci. **102**, 63–75 (2004)
34. Akehurst, D.H., Howells, G., McDonald-Maier, K.D.: Supporting OCL as part of a family of languages. In: Baar, T. (ed.), Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms—Needs and Trends, Montego Bay, Jamaica, 4 October 2005, Technical Report LGL-REPORT-2005–001, pp. 30–37. EPFL (2005)
35. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? IEEE Comput. Softw. **37**(10):64–72 (2004)
36. Chiaradía, J.M., Pons, C.: Improving the OCL semantics definition by applying dynamic meta modeling and design patterns. In: Demuth, B., Chiorean, D., Gogolla, M., Warmer, J. (eds.) OCL for (Meta-)Models in Multiple Application Domains, pp. 229–239. University Dresden, Dresden (Available as Technical Report, University Dresden, number TUD-FI06-04-September) (2006)
37. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: Consistency checking and visualization of OCL constraints. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000—The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, 2–6 October 2000, Proceedings, LNCS, vol. 1939, pp. 294–308. Springer, Heidelberg (2000)
38. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000—The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, 2–6 October 2000, Proceedings, LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000)
39. Varró, D.: A formal semantics of UML Statecharts by model transition systems. In: Corradini, A., Ehrig, H., Kreowski, H., Rozenberg, G. (eds.) Proceedings of ICGT 2002: 1st International Conference on Graph Transformation, LNCS, vol. 2505, pp. 378–392. Springer, Heidelberg (2002)
40. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine—Definition, Verification, Validation. Springer, Heidelberg (2001)
41. Chiorean, D., Bortes, M., Corutiu, D.: Proposals for a widespread use of OCL. In: Baar, T. (ed.). 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, pp. 68–82. EPFL (2005)
42. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Towards using OCL for instance-level queries in domain specific languages. In: Demuth, B., Chiorean, D., Gogolla, M., Warmer, J. (eds.) OCL for (Meta-) Models in Multiple Application Domains, pp. 26–37. University Dresden. Dresden (available as Technical Report, University Dresden, number TUD-FI06-04-September) (2006)
43. Baar, T., Marković, S.: A graphical approach to prove the semantic preservation of UML/OCL refactoring rules. In: Virbitskaite, I., Voronkov, A. (eds.) Proceedings, Sixth International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI 2006), Akademgorodok near Novosibirsk, Russia, LNCS, vol. 4378, pp. 70–83. Springer, Heidelberg (2007)

Author's Biography

Slaviša Marković graduated from the University of Belgrade and currently is a PhD student and research assistant at the Software Engineering Laboratory (LGL), Swiss Federal Institute of Technology in Lausanne (EPFL). His research interests include model transformations, model refactorings, and semantics of constraint languages.

Thomas Baar is Senior Researcher and Lecturer for software engineering at the École Polytechnique Fédérale de Lausanne (EPFL). His research interests include quality-oriented software processes, (semi-)formal specification techniques, and automatic verification of system implementations. Dr. Baar holds a diploma degree in computer science from Humboldt-University Berlin and a doctoral degree from University Karlsruhe. He is a member of the ACM.