A Metamodel for OCL

Mark Richters and Martin Gogolla

University of Bremen, FB 3, Computer Science Department
Postfach 330440, D-28334 Bremen, Germany
{mr|gogolla}@informatik.uni-bremen.de,
WWW home page: http://www.db.informatik.uni-bremen.de

Abstract. The Object Constraint Language (OCL) allows the extension of UML models with constraints in a formal way. While the UML itself is defined by following a metamodeling approach, there is currently no equivalent definition for the OCL. We propose a metamodel for OCL that fills this gap. The benefit of a metamodel for OCL is that it precisely defines the syntax of all OCL concepts like types, expressions, and values in an abstract way and by means of UML features. Thus, all legal OCL expressions can be systematically derived and instantiated from the metamodel. We also show that our metamodel smoothly integrates with the UML metamodel. The focus of this work lies on the syntax of OCL; the metamodel does not include a definition of the semantics of constraints.

1 Introduction

The UML provides a rich set of language constructs for describing structural, dynamic and functional aspects of a system [16]. For most parts of the language a graphical notation facilitates the construction of UML models. However, the graphical notation is inherently limited when specifying complex constraints. Often a textual constraint language is more appropriate for this task. The Object Constraint Language (OCL) has been designed to extend UML models with constraints in a concise and precise way [15, 21].

Since UML version 1.1, the OCL is an integral part of the UML standard. The application of OCL is not limited to user models, it is also used by the standard itself for specifying well-formedness rules in context of the UML semantics definition [17]. In [20, 10], the authors report on considerable improvements by using OCL instead of English text. Several items of the UML specification which previously allowed ambiguous interpretation could be identified and subsequently clarified or corrected.

The main reference for UML is the semantics document [17]. This document defines the language to be used for specifying well-formed UML models. Language constructs are defined by (1) their abstract syntax, (2) a set of well-formedness rules, and (3) an informal description of their intended semantics. The abstract syntax is defined as a metamodel in form of UML class diagrams. This style of presentation has been chosen for most UML parts but not for the

OCL. In order to achieve a uniform presentation for the complete UML *including* OCL, we therefore propose a definition of the OCL following the style of [17]. Our main contribution to achieve this goal is the definition of an OCL metamodel. The metamodel presented in this paper is not just a different kind of presentation of the OCL. We rather consider it a complementary work to [15]. The OCL document concentrates on the concrete syntax and application aspects of the Object Constraint Language. Our metamodel, on the other hand, tries to define the abstract syntax of OCL. Another benefit resulting from having a metamodel is that it is quite easy to derive an implementation. For example, we currently use the metamodel to build an OCL interpreter.

The UML metamodel is defined as one of the layers of a four-layer metamodeling architecture [17, p. 2-4]. In this architecture the UML metamodel is an instance of the meta-metamodel of the OMG Meta-Object Facility (MOF) [14]. For an integration of the OCL metamodel with this architecture there are basically two options. We can place the OCL metamodel at the same level (the M2 level) as the UML metamodel and use the MOF as the definition language (provided by the M3 level). The result is a general MOF-compliant metamodel of OCL. The second option is to lift the OCL metamodel to the M3 level. One reason is that OCL may obviously apply to other metamodels than just UML. Furthermore, the MOF also uses OCL for specifying constraints at the M3 level. Consequently it would make sense to define the OCL metamodel as part of the MOF. This way our work can be seen as a contribution to the meta-metamodel level of the standard OMG architecture. Note that there are no differences resulting from these two options with respect to the technical details of the OCL metamodel. In both cases the OCL metamodel is defined in terms of the MOF meta-metamodel. Since OCL cannot be used without a concrete modeling language, we have chosen the first option and placed the OCL metamodel at the same level as the UML metamodel in order to achieve a tight integration.

Related work in context of OCL can be summarized as follows. A general assessment and analysis of the Object Constraint Language has been presented in [9, 7]. In [13], the expressiveness of OCL has been investigated. Extensive applications of OCL can be found in the UML semantics document and, e.g. in [6], where it has been used to define additional constraints for equivalence and transformations rules between UML class diagrams. We consider the development of an OCL metamodel a step towards a formal model of the Object Constraint Language [8, 18]. This is part of a greater effort towards formalizing the Unified Modeling Language (e.g., see [1, 4, 12, 19, 11, 3, 2]).

The presentation of the OCL metamodel is structured in the following way. The metamodel is organized as a set of three UML packages containing a total of 56 classes. Each package describes a different aspect of the language, namely types, expressions, and values. For each package, we first present a class diagram defining the abstract syntax of a concept. Then we give an informal explanation of the diagram contents. Concrete examples are used to illustrate the application of the metamodel. Finally, a set of well-formedness rules specifies additional constraints on the metamodel. These rules are probably not complete, but they should give an idea of how OCL rules given in [15] can be translated and applied to the metamodel. Due to space limitations we do not give OCL expressions for the well-formedness rules but in principle this could be done with reasonable effort.

The rest of the paper is organized as follows. In Sect. 2 we illustrate the package structure of the OCL metamodel. Since OCL is primarily used for specifying constraints on UML models, there exists a strong relationship between both metamodels. A link between OCL and the UML core model will be established in Sect. 3. The *Types* package of our metamodel is presented in Sect. 4. The central part of the metamodel is the definition of *Expressions* which are discussed in Sect. 5. The result of evaluating expressions are *Values* which are not strictly part of the abstract syntax but are included in Sect. 6 for completeness. We close with a summary and draw some conclusions for future work.

2 Structure of the Metamodel

The OCL metamodel will be defined in a package called *OCL*. Fig. 1 shows how this new package is related to the existing UML packages *Core*, *Data Types*, and *Common Behavior* [17, p. 2-7].

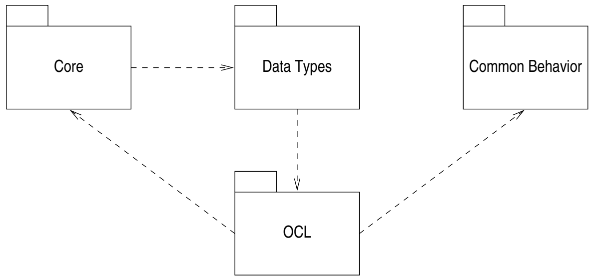

Fig. 1. Dependencies among UML packages and OCL

There are several dependencies among these packages. The *Core* package depends on *Data Types* where a general notion of expressions is defined. These expressions are used for modeling the concept of constraints in the *Core*. The *Data Types* package depends on OCL as soon as we use the Object Constraint Language for defining expressions. On the other hand, OCL depends on model elements (classes, attributes, etc.) of the *Core* package for building expressions. Finally, evaluating OCL expressions results in values or instances. Both concepts are defined in the *Common Behavior* package, hence we have another dependency from *OCL* to *Common Behavior*.

There are more UML packages which use expressions and which are not included in Fig. 1. For example, statecharts may contain boolean expressions for the specification of guard conditions. Of course, these packages will also indirectly (via *Data Types*) depend on (and benefit from) the *OCL* package when they use OCL for expressions.

The *OCL* package is further refined into three separate packages which model different aspects of OCL. Fig. 2 shows the packages *Expressions*, *Types*, and *Values* as part of the OCL metamodel.

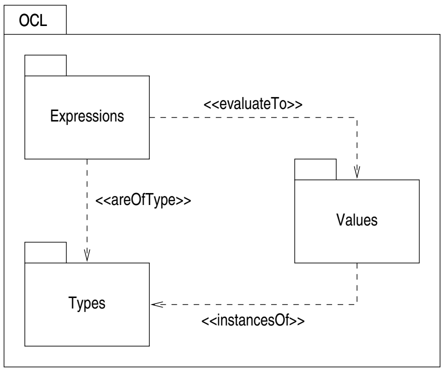

Fig. 2. Package Structure of the OCL Metamodel

We have used special stereotypes for the dependencies to emphasize their different roles. In OCL, every expression has a well-defined type. Instances of types are values which result from the evaluation of expressions. Each of these three packages will be discussed in its own section in the following. But first, we will have a closer look at the relationship between the UML core and OCL.

3 Constraints

Constraints can be used to specify restrictions on UML model elements. The concept of constraints is defined in the *Core* package of the UML metamodel. A constraint may be attached to all kinds of model elements. For example, a class invariant can be defined with a constraint by specifying a boolean expression that has to be true for all instances of the class. The UML does not prescribe the language or formalism for the boolean expression. In general, a constraint can be specified in natural language, OCL, or some other language.

The class diagram in Fig. 3 provides the link between the UML metamodel and the OCL metamodel. It shows partial views of the *Core* package and the *Data Types* package [17, p. 2-13 and p. 2-75]. Only classes and relationships that are important for our purposes are displayed. ***Constraints*** are ***ModelElements*** that specify restrictions on other model elements. A ***BooleanExpression*** forms the body of a ***Constraint***. ***BooleanExpressions*** are just a special kind of ***Expression*** having a boolean result type. An expression can be defined by using

OCL. In the UML *Data Types* package, ***Expression*** has attributes *language* and *body* where the body attribute keeps the textual representation of the constraint. We replace the *body* attribute by an optional component relationship to ***OclExpression***. It is now possible to use OCL for defining expressions, but we are not forced to do so. Other languages could be integrated similarly.

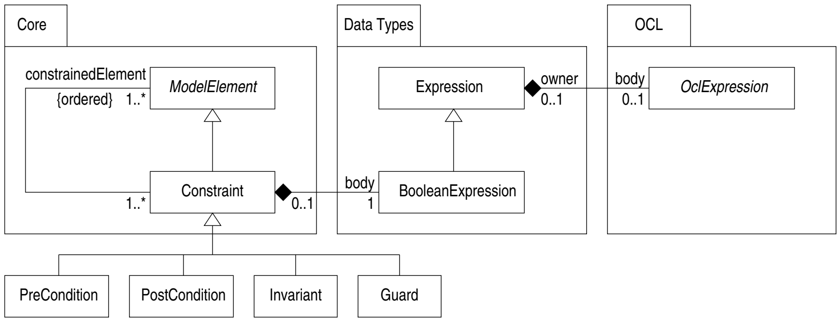

Fig. 3. Integration of OCL Expressions into the UML Core and Data Types Packages

The abstract class ***OclExpression*** defines the set of all legal expressions in OCL. It is the top-level element of the OCL metamodel and will be further refined in Sect. 5. All kinds of expressions are specializations of ***OclExpression***.

Constraints can appear in different contexts. They may be used to specify pre- and postconditions, invariant, and guards. Therefore, in Fig. 3, we have specialized the ***Constraints*** class into corresponding subclasses ***PreCondition***, ***PostCondition***, ***Invariant***, and ***Guard***. The distinction is necessary because some OCL constructs are allowed only in certain contexts, e.g. the `@pre` modifier makes sense only in postconditions. Alternatively, instead of modeling the different kinds of constraints as subclasses, we could also model this distinction as an enumeration in the ***Constraint*** class itself. However, our goal was to avoid changes to the original UML metamodel where possible.

The following well-formedness rules refer to some concepts that will be introduced in later sections.

Well-formedness rules:

1. When a ***BooleanExpression*** is used in a ***Constraint*** and it is defined by an ***OclExpression***, the ***Type*** of the ***OclExpression*** must be an instance of ***BooleanType***.
2. ***PreConditions*** and ***PostConditions*** may be attached only to ***Operations*** or ***Methods*** (both are part of the *Core* package).

4 Types

OCL is a typed language. Each expression has a type which is either explicitly declared or can be statically derived. Evaluation of the expression yields a value of this type. Therefore, before we can define expressions, we have to provide a model for the concept of type. A metamodel for OCL types is shown in Fig. 4. Note that instances of the classes in the metamodel are the types themselves (e.g. *Integer*) not instances of the domain they represent.

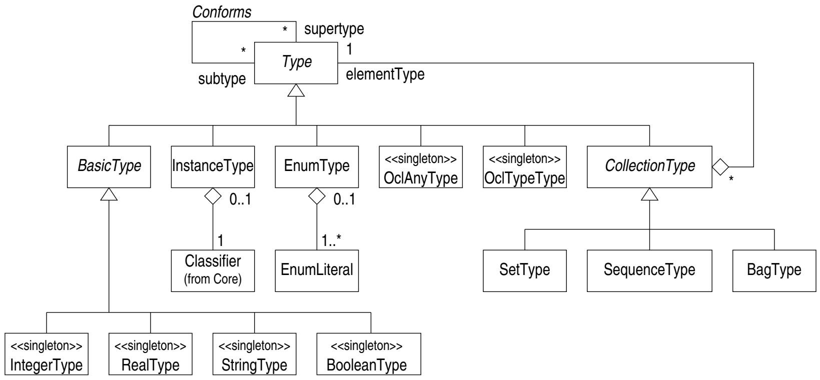

Fig. 4. Metamodel for OCL Types

All available types in OCL are modeled as specializations of the abstract class ***Type***. Concrete types are classified in the metamodel as follows.

- basic types: ***IntegerType***, ***RealType***, ***StringType***, and ***BooleanType***
- model types: ***InstanceType***
- enumeration types: ***EnumType***
- special types: ***OclAnyType*** and ***OclTypeType***[1]
- collection types: ***SetType***, ***SequenceType***, and ***BagType***

The first group models the basic types *Integer*, *Real*, *String*, and *Boolean*. We have marked the corresponding elements in the metamodel with a stereotype ≪`singleton`≫ to indicate that there is exactly one instance for each of these classes, e.g. the one and only instance of ***IntegerType*** is the type *Integer*.

InstanceTypes are used to refer to ***Classifiers*** defined by users in a UML class model, e.g. *Person*, *Company*, etc. The meaning of special types is explained in [15]. ***EnumTypes*** are defined by a list of distinct literals, e.g. *Color:{red, green, blue}*. ***CollectionTypes*** are parameterized by an element

[1] The strange names are a result of our naming scheme: For each OCL type we create a corresponding metamodel element by attaching the word *Type* to its name.

type, e.g. *Set(Integer)*. Those familiar with design patterns [5] will recognize the application of the composite pattern for modeling collection types. In our metamodel, there is no limit on the depth of nesting of collection types. Note that OCL tries to avoid complex collections by a process called “automatic flattening”. However, complex values may nevertheless result from the evaluation of some expressions (e.g. by navigating more than one association with multiplicities greater than one). Thus, *before* “flattening” a complex value we need to be able to specify its precise and complete (possibly nested) type.

The specialization of ***Type*** in the metamodel is used for classifying types by common properties. It is, however, not used for modeling subtype relationships which are defined by OCL type conformance rules. Rather, we use an association ***Conforms*** to model the subtype relation on types. This also allows us to express the type conformance rules for parameterized collection types (see the set of well-formedness rules later in this section). As an example, consider the subtype relationship between the *Integer* and *Real* type: *Integer* is a subtype of *Real* such that an *Integer* value can be used anywhere where a *Real* value is expected. In the metamodel both types are generalized into a ***BasicType***. The following constraint utilizes the ***Conforms*** association to enforce the requirement that *Real* is a supertype of *Integer*.

```
IntegerType
  self.supertype->includes(t | t.oclIsTypeOf(RealType))
```

Example. We apply the metamodel to the OCL type *Set(Person)*. Figure 5 shows the representation of this type as a UML object diagram. The diagram is an instantiation of the *Types* metamodel. Person is a class defined in a user model. The class induces a corresponding instance type that can be used in OCL. This instance type is used to parameterize the set type.

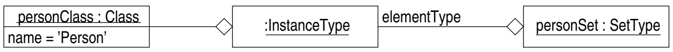

Fig. 5. Object Diagram for the Collection Type *Set(Person)*

In Fig. 6 we continue the example and add another type *Set(Employee)*. In this example, class Person is a generalization of class Employee. We have also included in this diagram links of the ***Conforms*** association indicating type conformance. Since Person is a generalization of Employee, the instance types have a similar relationship in OCL. Following from the type conformance rules for collections (see [15, p. 7-20] and items 6 and 7 of the well-formedness rules below), *Set(Person)* is also a supertype of *Set(Employee)*.

Most of the following well-formedness rules can be found as textual descriptions in the OCL documents [15, 21]. With the metamodel, they now can (in principle) be formalized as OCL constraints.

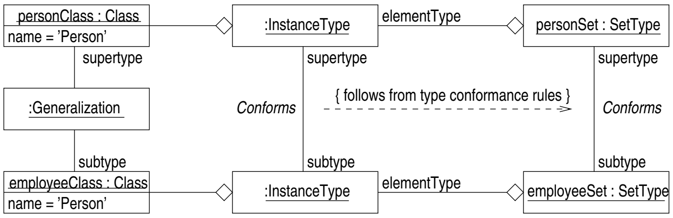

Fig. 6. Type Conformance between Types *Set(Person)* and *Set(Employee)*

Well-formedness rules:

1. ***OclAnyType*** is the supertype of all ***Types***.
2. ***IntegerType*** conforms to ***RealType***.
3. All ***EnumerationLiterals*** of an ***Enumeration*** are distinct.
4. ***Types*** are equal if they have the same textual representation.
5. Type conformance (represented by the association ***Conforms***) is transitive, reflexive and anti-symmetric.
6. ***CollectionTypes*** *C1(T1)* and *C2(T2)* conform to each other when their element types *T1* and *T2* do conform.
7. An ***InstanceType*** *I1* conforms to a type *I2* when their associated ***classifiers*** *C1* and *C2* have a generalization relationship, i.e. *C2* is a supertype of *C1*.

5 Expressions

In this section, we define the metamodel for OCL expressions. Perhaps surprisingly, it is quite difficult to say what actually constitutes an expression in OCL. In [15], the term is used informally in a number of places and different contexts. We suggest the following classification based on careful reading and analysis of the description and examples given in [15]. Our definition of expressions has been introduced more formally in our work on developing a formal syntax and semantics for OCL (see [18]). An OCL expression is defined to be one of the following.

1. The `self` keyword.
2. The `result` keyword in a postcondition.
3. A variable.
4. The application of an operation. We distinguish between
   - predefined operations: e.g. `+`, `-`, `*`, `<`, `>`, `size`, `max`, ...
   - attribute operations: e.g. `self.age`
   - operations defined by a classifier: e.g. `p.income(...)`
   - navigation by role names: e.g. `self.employees`
   - constants: e.g. `25`, `'aString'`, ...

5. The `iterate` construct. This also subsumes `select`, `reject`, `collect`, `exists`, and `forAll` since these may be rewritten as `iterate` expressions by simple syntactic transformations. We will refer to all these functions as *query* expressions in the sequel.

Fig. 7 shows part of the metamodel for OCL expressions. Due to the size of the class model we have decomposed it into two diagrams. The structure of operations (***OperationExp***) will be refined later in this section.

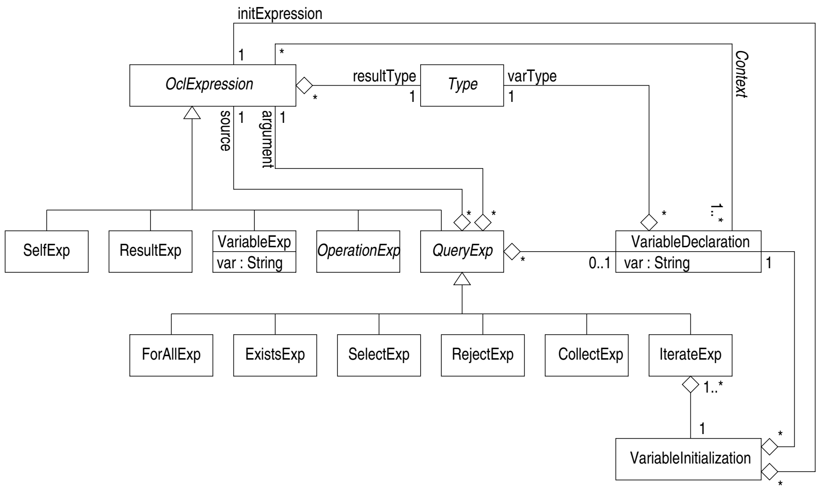

Fig. 7. Metamodel for OCL Expressions

All ***OclExpressions*** have a well-defined result ***Type***. The structure of ***Types*** has been shown in Fig. 4. ***OclExpressions*** are specialized into five fundamental expression classes according to the definition given above. A `self` expression is an instance of ***SelfExp***, a `result` expression is an instance of ***ResultExp***, a variable an instance of ***VariableExp***, etc.

Every OCL expression needs a context for evaluation. The context is given by the name of an instance type, e.g. *Person*. All occurrences of `self` expressions are then bound to an instance of this type during evaluation. Furthermore, variables can be explicitly declared in a context, e.g. `p:Person`. A context for a pre- or postcondition includes the signature of the operation to be constrained. Parameters of this operation introduce variable bindings that can be used just as any other variable in an expression. Variable bindings are represented by instances of ***VariableDeclaration***. Each declaration associates a variable name with a ***Type***.

In the metamodel, the association ***Context*** connects each expression with a non-empty set of ***VariableDeclarations***. There must be exactly one declaration with an empty *var* attribute. This anonymous declaration will be used for binding `self` expressions.

The predefined query expressions `iterate`, `select`, `reject`, `collect`, `exists`, and `forAll` differ from ordinary operations because they may contain language constructs for declaring and – in case of `iterate` – initializing variables that have special meaning in context of their argument expression. We use the simple example model in Fig. 8 to illustrate this.

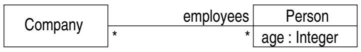

Fig. 8. Example UML Model

The following expression selects from the set of all persons working for a given company those being older than 45.

```
Company
  self.employees->select(p : Person | p.age > 45)
```

The query uses a select expression where a variable `p` of type *Person* is declared as part of the expression. This variable will be bound implicitly to each element of the source collection (`self.employees`) as the argument expression (`p.age > 45`) is evaluated for each of the collection's elements. In the metamodel, the expression can be represented as a ***SelectExp***. We repeat the expression below and mark different components of the expression with braces. Labels refer to the names used in Fig. 7.

$$\underbrace{\underbrace{\texttt{self.employees}}_{\text{source}}\texttt{->select(}\;\underbrace{\texttt{p : Person}}_{\text{VariableDeclaration}}\;\mid\;\underbrace{\texttt{p.age > 45}}_{\text{argument}}\texttt{)}}_{\text{SelectExp}}$$

The metamodel class ***QueryExp*** represents common properties of query expressions. They all have a source expression resulting in a collection which forms the input of the query, and they have an argument expression which is evaluated for each of the source collection's elements. Furthermore, they optionally may have a ***VariableDeclaration*** introducing an identifier which may be referred to as a variable expression (***VariableExp***) in the argument expression.

The more general `iterate` construct is a query expression (an ***IterateExp*** in the metamodel) which additionally has a mandatory variable initializer. A ***VariableInitialization*** is split into a declaration part and an initializing expression. The following example illustrates the possible structure of `iterate` expressions.

$$\underbrace{\underbrace{\texttt{Sequence\{1..5\}}}_{\text{source}}\texttt{->iterate(}\ \underbrace{\texttt{i : Integer}}_{\text{VariableDeclaration}}\ \texttt{;}\ \underbrace{\underbrace{\texttt{acc : Integer}}_{\text{VariableDeclaration}}\texttt{=}\ \underbrace{\texttt{1}}_{\text{initExpression}}}_{\text{VariableInitialization}}\ \texttt{|}\ \underbrace{\texttt{acc * i}}_{\text{argument}}\texttt{)}}_{\text{IterateExp}}$$

A large group of expressions can be classified as applications of operations. The metamodel element ***OperationExp*** in Fig. 7 is an abstract class representing all kinds of operations in OCL. A refinement of ***OperationExp*** is shown in Fig. 9.

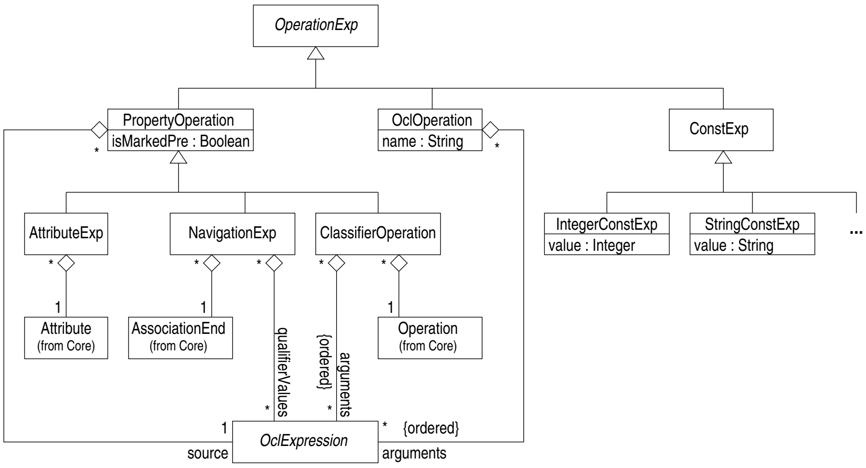

Fig. 9. Metamodel for OCL Operations

We first refine ***OperationExp*** into ***PropertyOperations*** which are operations referring to state dependent properties of objects. We use the term property as it is introduced in [15] for attributes, association ends, and operations/methods without side effects. All ***PropertyOperations*** have at least one argument: an ***OclExpression*** determining the source object. Furthermore, it is possible in postconditions to refer to a previous value of a property by postfixing it with the `@pre` keyword. The boolean attribute *isMarkedPre* indicates the reference to a previous value in a ***PropertyOperation***.

An ***AttributeExp*** like `self.age` is a function with one argument: an ***OclExpression*** resulting in an instance which owns the attribute. For the expression `self.age` the owner of the age attribute is given by an instance of ***SelfExp***. The name of the operation is the name of the ***Attribute*** itself.

A ***NavigationExp*** utilizes associations originating from a classifier. An associated classifier is specified by its role name. The role name is part of an

AssociationEnd in the *Core* package. When using qualified attributes on an ***AssociationEnd***, a ***NavigationExp*** may also specify ***qualifierValues***.

ClassifierOperations refer to ***Operations*** which are defined as classifier features in a user's application model, e.g.:

```
Employee::getSalary(d : Date) : Real
```

They may be used in OCL expressions, if they do not have any side effects, i.e. the *isQuery* attribute of ***BehavioralFeature*** in the *Core* package is true [17, p. 2-25]. ***ClassifierOperations*** may require any number of argument expressions.

Predefined operations on values are modeled by the element ***OclOperation***. These are characterized by an operation name and an argument list. Examples for predefined operations are: `+`, `-`, `*`, `<`, `>`, `size`, `max`, etc.

Finally, a further group of operations are those that produce constant values. We have modeled these for each of the basic types. For example, an ***IntegerConstExp*** simply contains an attribute *value* specifying the value of the constant to be created. The ellipsis in the diagram indicates the existence of further classes for the basic types *Real* and *Boolean*. Constant collection values like `Set{1,2}` can be modeled as ***OclOperations***, e.g. a function $\text{makeSet}(e_1 : T, \ldots, e_n : T) : \text{Set}(T)$ creates a set with n elements of type T. These functions are not visible to the user since they are only used for making the meaning of the literal notation explicit in the metamodel.

Example. We apply the metamodel for OCL expressions to the following query.

```
Company
  self.employees->select(p : Person | p.age > 45)
```

Figure 10 shows the abstract syntax of the above expression as an instantiation of the metamodel presented in Fig. 7 and Fig. 9.

The object diagram basically shows an abstract syntax tree. The root of this tree is the `select` expression. It has three child branches. First, the source of the `select` operation is a collection resulting from the navigation expression `self.employees`. The second branch models the variable declaration `p:Person`. Finally, the third branch represents the expression `p.age > 45`. We do not show result types of expressions and the context information (*Company*) in Fig. 10 in order to keep the diagram readable. However, adding this information would be straightforward.

Well-formedness rules:

1. A ***ResultExp*** may only be used in a ***PostCondition***.
2. A modifier `@pre` is only allowed in a ***PostCondition***.
3. The variable name used in a ***VariableExp*** must be part of a ***VariableDeclaration*** in an enclosing expression (or a parameter if the expression is attached to an operation as a pre- or postcondition). With other words, a variable must be declared before it may be used in an expression.
4. The source expression of a ***QueryExp*** must have a collection type.

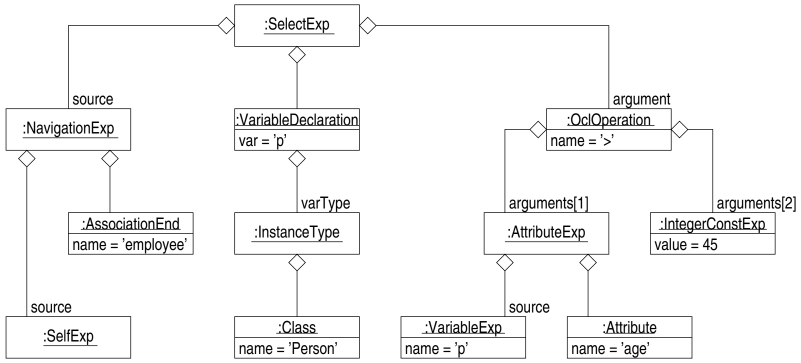

Fig. 10. Example Instantiation of the Expressions Metamodel

6 Values

In the previous sections, we have defined a complete metamodel for the abstract syntax of OCL. Another important aspect is the interpretation of OCL expressions. The result of evaluating an expression is a value. Since our focus is on meta-modeling issues, we do not consider the semantics of expressions here. A complete semantics would have to define a mapping from expressions to values [18]. In the following, we will concentrate on the structure of values. A metamodel for values suitable for representing results from evaluating OCL expressions is shown in Fig. 11.

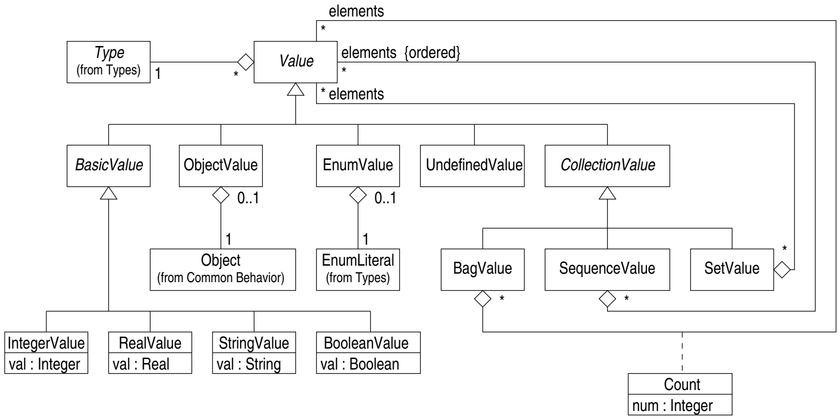

Fig. 11. Metamodel for Values

The general structure of values is very similar to the metamodel for types in Fig. 4 since each kind of value represents the domain of a corresponding type. ***Value*** is the abstract base class of all kinds of values. Each value has a ***Type*** defining its properties. Values of basic types are ***IntegerValue***, ***RealValue***, ***StringValue***, and ***BooleanValue***. An ***ObjectValue*** is just a placeholder for an ***Object*** which is defined as part of the UML *Common Behavior* package. An ***EnumValue*** is exactly one of the ***EnumLiterals*** used to define the corresponding type. Since the result of an OCL expression may be undefined (e.g. following from a division by zero), a special ***UndefinedValue*** is required for these cases.

We also need structures for the collection types *Set*, *Sequence*, and *Bag*. Values of these types may include other values as elements. For ***SetValue*** we can use an ordinary aggregation, a ***SequenceValue*** requires the `ordered` constraint on its elements, and a ***BagValue*** associates a number (***Count***) with each of its elements.

Example. We apply the metamodel to the value resulting from the OCL expression `Set{1,2}`. Figure 12 shows the set value containing the two integer values. Each value has a link to its defining type on the left side. In this example, the hierarchical structure of values reflects the hierarchical structure of composite types.

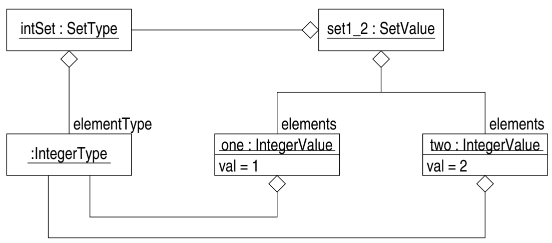

Fig. 12. Object Diagram for the value `Set{1,2}`

7 Conclusion

In this paper, we have presented a metamodel for the Object Constraint Language OCL. We also have demonstrated the application of the metamodel to concrete OCL expressions. The metamodel has been defined as a set of class diagrams together with well-formedness rules. This style of presentation closely follows the style of the UML semantics document. The metamodel could therefore be easily integrated with the existing UML specification providing a uniform style of presentation for all parts of the Unified Modeling Language.

The metamodel delivers a more precise and detailed view of the OCL. As a consequence from using only well-known modeling concepts of UML which are compliant with the Meta Object Facility (MOF), the metamodel can easily be read by everybody familiar with UML. The decomposition into packages for types, expressions, and values emphasizes the basic structure of OCL. Some issues that remained unclear from the original documentation had to be resolved and solutions for these issues have been integrated into the metamodel. For example, we have presented a concrete approach for connecting the OCL with the UML core concepts. Furthermore, we have given an interpretation for several important OCL concepts, e.g. the context of an expression, undefined values, and result values of expressions.

Finally, we have to note that, although the metamodel provides a precise description of the abstract syntax of OCL expressions, a complete description of OCL still requires a semantics definition. In our framework, a mapping from expressions to values would be necessary. A proposal for this has been made in [18]. It would be interesting to see, whether the UML modeling features are sufficient to express the OCL semantics within the metamodel.

Acknowledgments

We thank the anonymous referees for their helpful comments.

References

[1] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a formalization of the unified modeling language. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 344–366, Jyväskylä, Finland, 9–13 June 1997. Springer-Verlag.

[2] Jean-Michel Bruel. Transforming UML models to formal specifications. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[3] Andy Evans. Making UML precise. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[4] Robert France, Andy Evans, and Kevin Lano. The UML as a formal modeling notation. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, pages 75–81. Technische Universität München, TUM-I9737, 1997.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[6] Martin Gogolla and Mark Richters. Equivalence rules for UML class diagrams. In Pierre-Alain Muller and Jean Bézivin, editors, *Proceedings of UML'98 International Workshop, Mulhouse, France, June 3 - 4, 1998*, pages 87–96. ESSAIM, Mulhouse, France, 1998.

[7] Martin Gogolla and Mark Richters. On constraints and queries in UML. In Martin Schader and Axel Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications*, pages 109–121. Physica-Verlag, Heidelberg, 1998.

[8] Ali Hamie. A formal semantics for checking and analysing UML models. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[9] Ali Hamie, Franco Civello, John Howse, Stuart Kent, and Richard Mitchell. Reflections on the object constraint language. In Pierre-Alain Muller and Jean Bézivin, editors, *Proceedings of UML'98 International Workshop, Mulhouse, France, June 3 - 4, 1998*, pages 137–145. ESSAIM, Mulhouse, France, 1998.

[10] Anneke Kleppe, Jos Warmer, and Steve Cook. Informal formality? the object constraint language and its application in the UML metamodel. In Pierre-Alain Muller and Jean Bézivin, editors, *Proceedings of UML'98 International Workshop, Mulhouse, France, June 3 - 4, 1998*, pages 127–136. ESSAIM, Mulhouse, France, 1998.

[11] Kevin Lano. Defining semantics for rigorous development in UML. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[12] Kevin Lano and Juan Bicarregui. Formalising the UML in structured temporal theories. In Haim Kilov and Bernhard Rumpe, editors, *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 105–121. Technische Universität München, TUM-I9813, 1998.

[13] Luis Mandel and María Victoria Cengarle. On the expressive power of the object constraint language OCL. Technical report, Forschungsinstitut für angewandte Software-Technologie (FAST e.V.), 1999.

[14] OMG, editor. *Meta Object Facility (MOF) Specification, Version 1.3 RTF, 2 July 1999.* Object Management Group, Inc., Framingham, Mass., Internet: http://www.omg.org, 1999.

[15] OMG. Object Constraint Language Specification. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [16], chapter 7.

[16] OMG, editor. *OMG Unified Modeling Language Specification, Version 1.3, June 1999.* Object Management Group, Inc., Framingham, Mass., Internet: http://www.omg.org, 1999.

[17] OMG. UML Semantics. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [16], chapter 2.

[18] Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In Tok Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, pages 449–464. Springer, Berlin, LNCS 1507, 1998.

[19] Bernhard Rumpe. A note on semantics (with an emphasis on UML). In Haim Kilov and Bernhard Rumpe, editors, *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 177–197. Technische Universität München, TUM-I9813, 1998.

[20] Jos Warmer, John Hogg, Steve Cook, and Bran Selic. Experience with formal specification of CMM and UML. In Haim Kilov and Bernhard Rumpe, editors, *Proceedings ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques*, pages 167–171. Technische Universität München, TUM-I9725, 1997.

[21] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.