An Integrated Metamodel for OCL Types

Thomas Baar[1] und Reiner Hähnle[2]

[1] Fakultät für Informatik, Universität Karlsruhe, D-76128 Karlsruhe
`baar@ira.uka.de`
[2] Dept. of Computing Science, Chalmers Univ. of Technology, S-41296 Gothenburg
`reiner@cs.chalmers.se`

Abstract The main objectives of OCL are to restrict UML models by additional constraints and to clarify the definition of the UML meta model. For certain applications, however, it is crucial *for the modeler* to have a flexible and precisely defined access mechanism to the meta level of UML models. In the present paper we sketch such a modeling scenario and we argue that the current definition of `OclType` is insufficient. We propose an alternative definition based on metamodeling the type system of OCL in such a way that it is fully integrated with the UML meta model. This also clarifies some ambiguous issues in the OCL language specification and makes the reflexion mechanisms in OCL explicit.

1 Introduction

The main objectives of OCL are to restrict UML models by additional constraints and to clarify the definition of the UML meta model. OCL is a formal language in the sense that it has a formal syntax given as an EBNF context-free grammar (see the current OCL language specification [4, Section 7])[1]).

The grammar alone, however, is not sufficient to check syntactical correctness of a given OCL expression. OCL is a typed language, so the conformance rules for the type system (informally described in Section 7.4) have to be met. In addition, the set of admissible identifiers in OCL expressions is not reflected in the grammar (they are simply referred to by `<name>`). OCL names are derived from two sources: most names for classes, data types, operations, etc., of an underlying UML model are valid OCL names (see Sections 7.3 and 7.5 for details). The second source is the library of predefined OCL types and their properties as stated in Section 7.8.

Most issues concerning syntax beyond the mere grammar are dealt with informally in [4]. We show in Section 2 of the present paper that this leads to a number of ambiguous or contradictory interpretations. Ultimately, the syntax of OCL should be defined at least as precise as the other parts of the UML with the help of a metamodel.

[1] Throughout the paper we refer to parts of [4]. It would be cumbersome to mention [4] each time we refer to this document—when it is obvious from the context we refer only to a section number or page of this document without citing it.

The advantages of the metamodeling approach for OCL are generally acknowledged [2], although the only suggestion for a full metamodel of OCL we are aware of is [5]. In Section 4 of the latter paper a metamodel for OCL's type system is suggested. This is an important contribution, but there are a number of drawbacks:

– the type metamodel in [5] does not reflect the changes made in OCL 1.3 [4]: according to [5], the type `OclAny` is the supertype of all types while [4, p. 7-29] states that "[t]he predefined OCL Collection types are not subtypes of OclAny";
– the type metamodel in [5] contains several metaclasses with the stereotype `<<singleton>>` resulting in a somewhat clumsy class structure;
– most importantly, there is no conceptual distinction between metamodeling of basic OCL types such as `Integer` and the special type `OclType`.

In the present paper we propose an OCL type metamodel that remedies these shortcomings. In particular, we treat `OclType` consequently as a metatype comprising all other types of OCL as instances, and which is not in subtype relation with any of them. As a byproduct, we obtain an extension of the current possibilities to access the meta level of OCL together with a systematic integration into the UML meta model core package (p. 2-13). A flexible and precisely defined access mechanism *for the modeler* to the meta level of UML is crucial for building tools that give modelers far-reaching possibilites to create and manipulate useful OCL constraints. This is required, for example, to integrate design patterns with OCL constraints [1].

The paper is organized as follows: In Section 2, we report some inconsistencies and ambiguities in the current definition of OCL's type system and we point out limitations of OCL regarding access of the metamodel level. The latter is critical in an application involving the precise definition of design patterns and briefly sketched in Section 2.2. Our proposal for a precise definition of OCL's type system including a metamodel is in Section 3. For (as far as we know) the first time, comprehensive well-formedness rules are given. Section 4 shows that the problems discussed in Section 2 are resolved in our metamodel. We wrap up in Section 5 and also state open issues we were not yet able to solve satisfactorily.

2 Shortcomings of the OCL language specification

The quality of the OCL semantics documents improved considerably in recent years, but the current version [4, Chapter 7] still contains a number of contradictions and ambiguities.

2.1 Inconsistencies and Ambiguities in OCL Semantics

What are Basic Types? The term *basic type* is used informally in [4]. A clarification is crucial to understand the role of the type `OclType`. On p. 7-28 we read: "The basic types used are Integer, Real, String, and Boolean. They are supplemented with OclExpression, OclType, and OclAny." This would imply that there is no conceptual difference between `OclType` and, say, `Integer`. The role of `OclType` is then explained: "All types [...] have a type. This type is an instance of [...] OclType." The conclusion is that the type `OclType` is an instance of itself. While this is not contradictory in itself, we find it unintuitive and unnecessary. Recall that `OclAny` was already modified in OCL 1.3 for similar reasons [2].

On p. 7-7 we read: "Collection, Set, Bag and Sequence are basic types as well" contradicting the statement "[...] OclAny is the supertype of [...] the basic predefined OCL type[s]. The predefined OCL Collection types are not subtypes of OclAny." on p. 7-29.

Are Nested Collection Types Permitted? On p. 7-20 flattening of nested collections is explained and from this nested collections clearly are imaginable. On p. 7-37ff the properties of type `Collection(T)` are defined: "A real collection type is created by substituting a type for the T." In particular, `T` can be a collection type, hence nested collection types are not excluded.

Consider declaration `collection->includes(object: OclAny): Boolean` on p. 7-37, where the parameter of `includes` must be of type `OclAny`. On p. 7-38, on the other hand, the declaration of `includesAll` features a parameter `c2` of type `Collection(T)` with postcondition

(1) `result = c2->forAll(elem | collection->includes(elem))` .

Now, if `c2` is of type `Collection(T)` then `elem` must be of type `T` and, because `elem` is used as argument of `includes` in (1), `T` must be a subtype of `OclAny`. Therefore, `T` *cannot* be a collection type (p. 7-29) and nested collection would cause a type error.

OclExpression. "Each OCL expression itself is an object in the context of OCL. The type of the expression is OclExpression" (p. 7-31). In [5] `OclExpression` does not occur in the metamodel of the OCL type system, instead, it serves as the root metaclass of another part of the metamodel, where as well the query properties `forAll`, `collect`, etc., are handled. Hence, unlike `OCLType`, the type `OclExpression` constitutes not a metatype, but an analogous type to `String`. It is useful to imagine instances of `OclExpression` to be surrounded by quotation marks similar to instances of `String`.

Enumerations. On p. 7-26 we read: "The OCL type Enumeration represents the enumerations defined in an UML model." This suggests that the instances of `Enumeration` are classifiers with stereotype <<enumeration>> in an UML model. We know from Section 2.5.2.13 that "[d]ata types include primitive built-in types as well as definable enumeration types." So `OclType.allInstances` yields all types in a model including all enumerations; `Enumeration.allInstances` yields all enumerations. In contrast to this, on p. 7-8: "The type of an enumeration attribute is Enumeration." If in the UML model an enumeration `SeasonKind` is defined and used in an attribute `season: SeasonKind`, then we know (Section 2.5.2.5) that `season` has type `SeasonKind`. If `season` has type `Enumeration`, too, then `Enumeration` must be supertype of each enumeration, not its metatype.

2.2 Pattern extension – a scenario for using OCL's meta level

The purpose of `OclType` is that it "allows the modeler limited access to the meta-level of the model" (p. 7-28). As far as we know, this possibility is rarely used so far. Here we sketch the precise definition of design patterns within UML (see [1] for a detailed explanation) as one application, where flexible and precisely defined access to the meta level is crucial.

Design patterns [3] are widely acknowledged as one of the most useful tools available to software designers. Typically, a software design pattern is described in a form structured into several slots: *Motivation*, introducing the problem solved by the pattern; *Structure*, presenting a skeleton solution, usually with a class diagram; *Forces*, trading off advantages and disadvantages of its application; *Implementation*, where implementation variants are discussed, etc. Most slots are described informally using natural language. In [1] we show that some of the informally specified parts of a design pattern can be formalized in OCL. Probably the simplest example is the formalization of the semantics of the *Singleton* pattern [3], expressed in OCL as: `Singleton.allInstances->size<=1`.

Let us make this example slightly more complex: assume that the class `StorageElement` has to ensure that there exist at any time at most `maxInstance` instances. The instances of `StorageElement` are kept in a `Storage` with a fixed number of cells. Each attribute defined in `StorageElement` occupies one storage cell. Furthermore, it is required that `StorageElement` has an attribute `name` whose value is distinct for each instance. The class diagram

StorageElement
name: String

is supplemented by the following constraints:

```
StorageElement.allInstances->size <= maxInstance
StorageElement.allInstances->isUnique(s | s.name)
```

These constraints use the `allInstances` operation of type `OclType`. They have, therefore, access to the meta level but their effect is to restrict the instances of `StorageElement` and not the class itself.

Another kind of constraint is used to restrict the applicability of design patterns. When adapting a design pattern to a concrete problem the modeler usually extends and modifies the skeleton solution in the slot *Structure*, for example, by adding new features to a class, by adding new subclasses, etc.

If a pattern is intended to be adapted only in a restricted manner, then its author can often formalize such requirements in OCL. In that case, an OCL constraint has the effect to restrict the language of UML diagrams, which can be used by the modeler when applying the pattern.

In our example, we want a pattern to ensure that the capacity of the storage is not exceeded. Therefore, not only the number of instances but also the number of attributes of `StorageElement` must be limited—it may not exceed `limit = capacity.div(maxInstance)`. The constraint

```
StorageElement.attributes->size <= limit
```

ensures that the modeler does not add too many attributes to `StorageElement` when applying the design pattern. The constraint implements in effect a well-formedness rule for the UML classifier `StorageElement`. This example may seem contrived, but it is easy to imagine that company-wide or project-wide regulations, say, naming rules for features, are implemented in this way.

2.3 Limitations of OCL

Sometimes it is necessary to ensure that a class has no superclasses. In OCL a constraint like `StorageElement.supertypes = Set{OclAny}` does the job. If, however, we want to express constraints on subclasses of `StorageElement`, we run into problems, because `OclType` has a property `supertypes`, but not `subtypes`. For example, the following constraint, restricting the number of attributes in subclasses of `StorageElement`, is illegal:

```
StorageElement.subtypes-> forAll(s | s.attributes->size <=
limit)
```

A further limitation of the current standard is that the properties `attributes`, `operations`, etc., of `OclType` return a set of strings and no structural information. So it is not possible, for example, to distinguish between static attributes (`ownerScope` has value `#classifier`) or normal ones (`ownerScope` has value `#instance`). In our example, such a distinction would permit a tighter constraint, assuming that static attributes do not occupy a storage cell for each instance of `StorageElement`. In Section 4 we will see how these limitations can be overcome by our judicuous definition of `OclType`.

3 Precise Definition of OCL Types

3.1 Metamodel for OCL's Typesystem

Metamodeling is a technique to precisely define complex syntactical issues and is used in the specification of the diagrammatical parts of the UML, but not for OCL in [4]. The paper [5] suggests a metamodel for OCL as well and discusses its integration into the UML metamodel.

Our alternative suggestion, displayed in Figure 1, encompasses merely the type system of OCL, but it could be extended to a metamodel of full OCL. As mentioned above, we handle `OclType` as a proper metatype, which, as we shall see, renders its properties declared on p. 7-28f obsolete.

The metaclass `Classifier` on the left constitutes the connection with the UML core package backbone metamodel and UML extension mechanisms [4, pp. 2-13, 2-70]. This will be heavily used in the examples below.

The root class of our metamodel is `OclType` and represents the OCL type `OclType`. It has an association `conform` with roles `subType` and `superType` that models the conformance relation among OCL types (Sections 7.4.4, 7.5.14 in [4]). `OclType` has an attribute `name` to identify its instances.

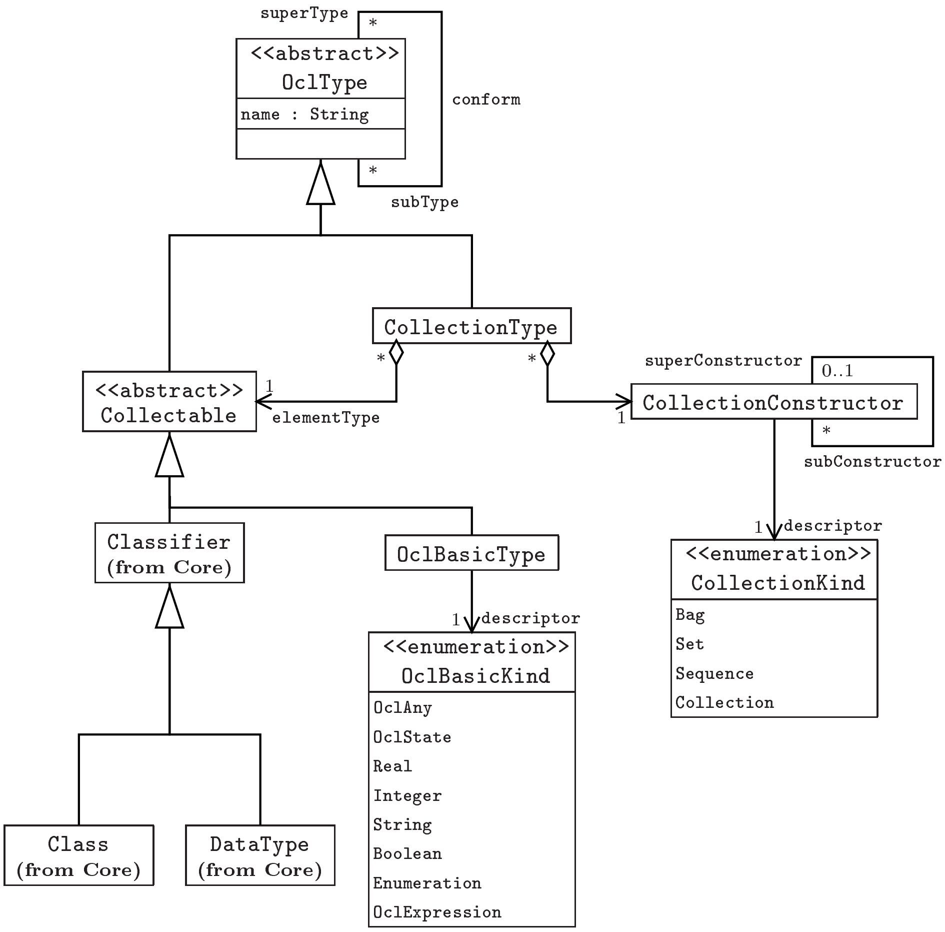


Figure1. Metamodel of OCL's type system

The instances of `OclType` are all types in OCL's type system. This is different from the current specification of OCL. In our metamodel, neither is `OclType` an instance of itself, nor is it a subtype of `OclAny`. The `subType` association is only applicable to *instances* of `OclType` but not to the class itself.

The metaclass `CollectionType` is a subclass of `OclType` and represents the collection types in OCL's type system. Even though in [4] no terminological distinction is made between, for example, `Set` and `Set(T)`, we think that such a distinction is crucial. Therefore, the auxiliary metaclass `CollectionConstructor` representing `Collection`, `Set`, `Bag`, `Sequence` is associated to `CollectionType`. There is a `sub-/superConstructor` hierarchy over `CollectionConstructor`, which is important to define the conformance relation on instances of `OclType` (see Section 3.2 below).

`CollectionConstructor` cannot directly be made into an enumeration to represent the finite set {`Collection`, `Set`, `Bag`, `Sequence`}, because UML enumerations cannot be part of inheritance hierarchies or undirected associations [4, Section 2.5.3.12[2]]. The solution is to introduce a separate enumeration metaclass `CollectionKind` whose client is `CollectionConstructor` (in conformance with Section 2.5.3.3[1]). A well-formedness rule (see Section 3.2 below) ensures isomorphy between the literals of `CollectionKind` and the instances of `CollectionConstructor`. The same technique is applied to `OclBasicType`.

`CollectionType` has an association `elementType` to `Collectable` giving the parameter of a collection type. As a consequence, nested collection types do not occur in our metamodel. Each instance of `Collectable` must be either an instance of `Classifier` (the bridge to the core package of UML's metamodel) or an instance of `OclBasicType`. The latter represents all predefined OCL types, such as `OclAny`, `Real`, `Boolean`.

In contrast to [4], the properties `name`, `subType`, and `superType` of the metaclass `OclType` are defined already in the metamodel, not on the MOF model level. This yields, in our opinion, a more systematic approach to defining the properties of `OclType` than the one in Section 7.8.1.1, which seems rather *ad hoc* (`supertypes` is present, but not `subtypes`). The properties `attributes`, `associationEnds`, `operations` are available in our metamodel as well (by inheritance from the UML core metamodel) for those instances of the metaclass `Classifier` that need them. The OCL 1.3 expression `type.attributes` (and similarily, the other properties) would be defined in our metamodel via navigation, which has the advantage that `type` can be ensured to be a classifier:

```
type.oclAsType(Classifier).feature->
  select(f | f.oclIsKindOf(Attribute))->collect(a | a.name.body)
```

3.2 Well-Formedness Rules

We increase the precision of the available specifications of OCL's type system by supplying constraints (well-formedness rules) for our metamodel. Attribute `name` of `OclType` is defined as follows:

```
context OclType inv:
self.name =
  if self.oclIsKindOf(CollectionType)
```

```
then self.collectionConstructor.descriptor.name.body.
       concat('(').concat(self.elementType.name).concat(')')
else if self.oclIsKindOf(OclBasicType)
     then self.descriptor.name.body
     else -- self is a Classifier
       self.oclAsType(ModelElement).name.body
     endif
endif
```

We define `CollectionConstructor` as an isomorphic copy of the enumeration `CollectionKind`:

```
CollectionConstructor.allInstances->size =
  CollectionKind.allInstances->size and
CollectionConstructor.allInstances->isUnique(descriptor)
```

A similar constraint is needed for `OclBasicType` and `OclBasicKind`.

We define the value of `subConstructor` (and, by invertability, the value of `superConstructor`) with an auxiliary operation `directSubConstructor`; then `subConstructor` is defined as the reflexive closure of `directSubConstructor`.

```
context CollectionConstructor inv:
let directSubConstructor =
     if self.descriptor = #Collection
     then Set{#Bag, #Set, #Sequence}
     else Set{}
     endif in
  subConstructor = directSubConstructor->including(self)
```

Similarly, we define the association `subType` by the help of an auxiliary operation `directSubType`. Then `subType` is the reflexive and transitive closure of `directSubType`. We view `Enumeration` as a supertype of `Boolean` and, in general, of each datatype with stereotype `<<enumeration>>`. This is compatible with Sections 2.5.2.13, 2.7.2.

```
context OclType inv:
let directSubType =
     if self.oclIsKindOf(CollectionType)
     then CollectionType.allInstances->select(c |
            self.collectionConstructor.subConstructor->
               includes(c.collectionConstructor)
            and
            self.elementType.directSubType->
               includes(c.elementType))
     else if self.name = 'OclAny'
         then Collectable.allInstances->
                 reject(c | c.name = 'OclAny')
         else if self.name = 'Real'
               then Collectable.allInstances->
```

```
                select(c | c.name = 'Integer')
         else if self.name = 'Enumeration'
             then Collectable.allInstances->select(c |
                     c.name = 'Boolean' or
                     (c.oclIsKindOf(DataType) and
                      c.stereotype->notEmpty and
                      c.stereotye.name = 'enumeration'))
             else if self.oclIsKindOf(Classifier)
                  then self.specialization->collect(child)
                  else Set{}
                  endif
             endif
        endif
   endif
endif in
subType = directSubType->including(self)->
   union(self.directSubType->collect(d | d.subType)->asSet)
```

The definition ensures that `directSubType` and `subType` are acyclic because the association `specialization` between `Generalization`, `GeneralizableElement` is acyclic (see Section 2.5.3.18[3]).

4 Resolved Problems

Our metamodel resolves (among other issues) the ambiguities und contradictions uncovered in Section 2.1: `OclType` is a metatype, nested collections are excluded, `OclExpression` is regarded as a basic type, and so is `Enumeration`, which contains all enumeration literals of an underlying UML class diagram.

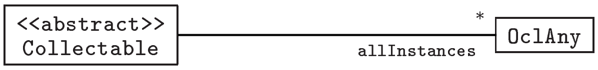
The main advantage, however, is to overcome the limitations in formulating meta level constraints. Now we can, for example, formulate those constraints on the class `StorageElement` that caused problems in Section 2.3:

```
-- Number of instance attributes does not exceed capacity
StorageElement.feature->select(f | f.ownerScope = #instance)->
   select(f | f.oclIsKindOf(Attribute))->size <= limit
-- Number of all attributes in subclasses does not exceed capacity
StorageElement.specialization->forAll(s | s.feature->
   select(f | f.oclIsKindOf(Attribute))->size <= limit)
```

5 Future Work and Conclusion

In general, the properties of OCL types, such as `size` of `Collection`, are not definable within the OCL metamodel. Due to its metaclass character, `OclType` is an exception: in Section 3.1 we defined all properties of `OclType` (pp. 7-28f), with the exception of `allInstances`, by virtue of the connection with the UML core metamodel. Unfortunately, `allInstances` causes problems: The property `allInstances` is declared as `type.allInstances:Set(type)` on p. 7-29, where `type` represents an instance of `OclType`. This allows to construct arbitrarily nested sets. Worse yet, the result type of `allInstances` depends on the argument type and cannot be statically determined.

One way to proceed from here is to treat `allInstances` analogously to `subType`, that is, model it as an association `allInstances` from `OclType` to a suitable metaclass (discussed in a moment). Then `allInstances` is applicable to *instances of* `OclType`, but not to `OclType` itself (in contrast to [4]): the expression `OclType.allInstances` becomes syntactically incorrect. A drawback of this proposal is that nested set types can still be constructed (take `Set(Boolean).allInstances`). Nested collection types are avoided by attaching `allInstances` not the metaclass `OclType` but to the metaclass `Collectable`:



The supplier of `allInstances` is the instance `OclAny` of `OclType`, which now plays a second role in the OCL metamodel as the metaclass representing instances of `OclAny`. As a consequence, the result type of `type.allInstances` is `Set(OclAny)`, not `Set(type)`, but this can be easily ensured by the constraint:

```
context Collectable inv:
self.allInstances->forAll(i | i.oclIsKindOf(self))
```

In conclusion, we are aware that our proposal of a metamodel of OCL types might be perceived problematic by some, because we decided to deviate from the current OCL standard in some points, notably, in the treatment of `OclType` as a metaclass. Moreover, a formal treatment of `allInstances` could only be had for the price of letting `OclAny` appear on different modeling levels. On the other hand, we feel that these are not quirks of our particular model, but consequences of the reflexion mechanism in OCL and, therefore, difficult (if not impossible) to avoid in a precise OCL specification. As our approach clearly improves the precision of available OCL type specifications and includes a clearly defined and flexible interface to metamodeling, we think that it deserves discussion.

References

[1] T. Baar, R. Hähnle, T. Sattler, and P. H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In G. Snelting, editor, *Softwaretechnik-Trends*, Informatik Aktuell. Springer-Verlag, 2000.

[2] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The Amsterdam manifesto on OCL. Technical Report TUM-I9925, Institut für Informatik, Technische Universität München, Dec. 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] Object Modeling Group. *Unified Modelling Language Specification, version 1.3*, Mar. 2000. OMG document formal/00-03-01.

[5] M. Richters and M. Gogolla. A metamodel for OCL. In R. France and B. Rumpe, editors, *Proc. 2nd International Confernece on the Unified Modeling Language (UML)*, volume 1723 of *LNCS*, pages 372–383. Springer-Verlag, 1999.