A Model Theoretic Semantics of OCL

Peter H. Schmitt

Universität Karlsruhe
Institute for Logic, Complexity and Deduction Systems
D-76128 Karlsruhe, Germany
`i12www.ira.uka.de/~pschmitt`

Abstract. This paper proposes a model theoretic semantics of the Object Constraint Language (OCL), observing the OMG standard as close as possible

1 Introduction

The Unified Modeling Language, UML, has gained widespread acceptance as a standard for modelign object-oriented systems. The Object Constraint Language, OCL, is a part of UML used to add constraints to UML diagrams that can cannot be expressed in the visual models. The available descriptions, the OMG standard document [8] and the book [16], fall short of giving a rigorous semantic, and even syntactic description of the language. Deficiencies have been pointed out e.g. in [11, 4, 5, 7]. The purpose of this paper is to give a systematic definition of syntax and semantics of the OCL.

One way to provide OCL with a precise semantics is via a translation into a known, well understood specification language. This approach has been pursued e.g. in [6, 3]. As an additional advantage of this approach the translated expressions may be used as input to existing tools. The disadvantage is that those not familiar with the target language will gain nothing.

We describe here a semantics of OCL in an informal yet mathematically rigorous way, making use of naive set theory only. We believe that this is a common ground for all specification languages, like Z, Abstract Machine Notation, CASL, Isabelle or HOL, to name just a few. Formalizing our semantics in any of these will be straightforward complicated only by the requirement to get around the restrictions imposed by the chosen framework.

This research is supported by DFG within the KeY project, see the web page `http://i12www.ira.uka.de/~key`.

2 The UML Context

OCL expressions only make sense with respect to a given UML model. For the time being OCL expressions may only be attached to class diagrams. Figure 1 shows more precisely how syntax and semantics of OCL depend on their counterparts in UML. From a given UML class diagram we read off a set of *model types* $S_{\mathcal{D}}$. These will in Subsection 3.1 be extended to obtain the set $S_{\mathcal{D}}^{OCL}$ of all OCL types. The class diagram also provides a subtype ordering $<_{\mathcal{D}}$ on $S_{\mathcal{D}}$, which is in Definition 6 again extended to a subtype relation on all OCL types. The vocabulary that is used to built up OCL expressions also comes in two parts: the symbols in $F_{\mathcal{D}}$ that arise from the diagram $\mathcal{D}$, and pre-defined OCL operation symbols. OCL constraints occur as additions to

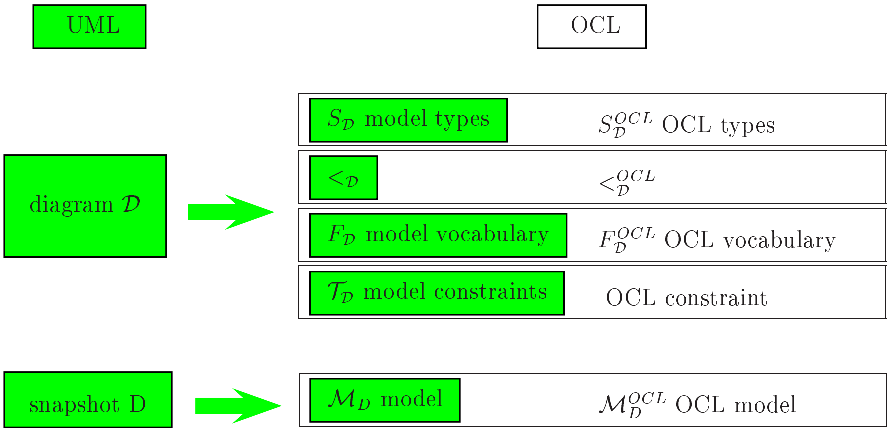

Fig. 1. Syntactic and semantic dependance of OCL on UML

UML diagrams in the form of invariants, preconditions and postconditions. But there is also information in $\mathcal{D}$ that goes beyond what is coded in $S_{\mathcal{D}}$, $<_{\mathcal{D}}$, and $F_{\mathcal{D}}$. A typical example are multiplicities of association ends. This information can easily be expressed by OCL expressions. For lack of space we do not pursue this here.

It is essential to distinguish between a class diagram $\mathcal{D}$ and a snapshot, or valid instance D of $\mathcal{D}$, see e.g. [13, pages 59–60] for a concise explanation. The class diagram completely determines the syntax of OCL-expressions over $\mathcal{D}$, while a snapshot D is needed to determine the meaning of an OCL expression. First a snapshot D gives rise to a many-sorted algebra $\mathcal{M}_D$ which will then be extended to an OCL-algebra $\mathcal{M}_D^{OCL}$ as depicted in the lower part of Figure 1. For the rest of the paper we assume that $\mathcal{D}$ is a fixed class diagram and D a valid instance of it.

2.1 The vocabulary of a UML diagram

The signature $\Sigma_{\mathcal{D}}$ consists of the set $S_{\mathcal{D}}$ of sorts, the set $F_{\mathcal{D}}$ of functions, and a subsort relation $<_{S,\mathcal{D}}$ on $S_{\mathcal{D}}$ as detailed in the following definitions.

Definition 1 **($S_{\mathcal{D}}, <_{\mathcal{D}}$).**

1. *The set $S_{\mathcal{D}}$ of model types consists of all class symbols in $\mathcal{D}$.*
2. *The subtype relation $S_1 <_{\mathcal{D}} S_2$ holds if and only if type S_1 is declared a subtype of S_2 in $\mathcal{D}$.*
3. *$\ll_{\mathcal{D}}$ denotes the transitive, reflexive closure of $<_{\mathcal{D}}$.*

In particular, $S_{\mathcal{D}}$ contains symbols for all classes with the stereotype
$\ll$ *enumeration* $\gg$ that happen to occur in $\mathcal{D}$.

The functions in $F_{\mathcal{D}}$ arise from various sources in $\mathcal{D}$ as detailed in the following definition.

Definition 2 **($F_{\mathcal{D}}$).**

1. *For every association r in $\mathcal{D}$ and every two different association ends e_1, e_2 of r there is a function symbol $f_{r,e_1,e_2} \in F_D$.*
   If e_i is attached to the class S_i for $i = 1, 2$ then the function symbol receives the

corresponding signature: $f_{r,e_1,e_2} : S_1 \to Set(S_2)$.
In case the multiplicity at the end e_2 *is* 1 *the signature is:* $f_{r,e_1,e_2} : S_1 \to S_2$.
If the e_2*-end has stereotype* $\ll ordered \gg$ *then the signature is:*
$f_{r,e_1,e_2} : S_1 \to Sequence(S_2)$.

2. *For every attribute* a *of a class* S *in* $\mathcal{D}$ *there is a function symbol* $f_a \in F_{\mathcal{D}}$. *If* S_r *is the value type of* a *specified in* $\mathcal{D}$ *then* f_a *is given the signature* $f_a : S \to S_r$. *If* a *is a class attribute, (sometimes this is also called a static attribute), then* f_a *is a constant symbol of type* S_r.
3. *For every operation* c *of a class* S *with parameters of type* $S_1, \ldots, S_k$ *and result type* S' *there is a function symbol* $f_c \in F_{\mathcal{D}}$ *with signature* $f_c : S \times S_1 \times \ldots \times S_k \to S'$.
We will require that c *has no side effects, i.e.* c *satisfies the property* `isQuery()` *(see [9, p.2-25]).*
4. *For every association class* C *attached to an association* r*, where* r *associates the classes* S_1 *and* S_2 *there are symbols for the unary projection functions* pr_{S_1} *with signature* $C \to S_1$*, and* pr_{S_2} *with signature* $C \to S_2$.

We will restrict attention to binary relations. More than binary associations are rare anyway. The extension to associations with m association ends can be easily obtained by introducing m unary functions.

It is not clear if the e_2-end is allowed to be of multiplicity 1 **and** of stereotype $\ll ordered \gg$. If this case is possible we would suggest that the multiplicity takes precedence and the function have signature $f_{r,e_1,e_2} : S_1 \to S_2$ in clause 1 of Definition 2.

The explanations in [13, page 166ff] allow to attach a multiplicity different from 1 to attributes, including the exceptional case of multiplicity 0. This seems to have not been widely accepted. In [2] e.g. this possibility is not even mentioned. We will thus consider for each attribute a the associated function f_a to be total and single-valued.

From the point of view of abstract syntax the names of the symbols in $\Sigma_{\mathcal{D}}$ are irrelevant. But for all practical purposes it helps to stick to the following naming conventions:

Definition 3 (Naming Conventions).

1. *Sorts in* $S_{\mathcal{D}}$ *will be given the same name as the corresponding class. Sort names begin with an uppercase letter.*
2. *The function symbol* f_{r,e_1,e_2} *will be referred to by the role name of* r *at the association end* e_2. *If no role name is given, the name of the class attached to* e_2 *will be used. Function names start with a lowercase letter.*
3. *Function symbols arising from attributes or operations will carry the name of the attribute or operation. If attr is a static attribute of class* C *then the concrete syntax of the constant* f_{attr} *will be C.attr. This is in accordance with common usage, see e.g. [16, Section 3.5.3]. Here the dot is not used to indicate application of a property, but is simply part of a name.*
In Java it is possible to apply a static attribute of a class C *to instances of class* C. *If we wanted to allow the same behaviour on the OCL level we would have to introduce a static attribute attr in addition to the constant C.attr a unary function symbol* $f_{attr} : C \to S$*, where* S *is the value type of attr.*

4. *The projection functions pr_{S_1} and pr_{S_2} for an association class C are considered as implicit attributes of C and denoted by lowercase class names s_1 and s_2. In case we want to also include n-ary associations we would, of course, have n projection functions.*

2.2 Semantics of UML diagrams

For any snapshot D of the UML class diagram $\mathcal{D}$ we will now define an associated many-sorted algebra $\mathcal{M}_D = (M_D, I_D)$ of signature $\Sigma_{\mathcal{D}}$. For ease of reading we will write $\mathcal{M}$ instead of $\mathcal{M}_D$ and I instead of I_D when there is no danger of confusion.

$\mathcal{M}$ may be viewed as a second-order algebra since function values will not always be elements of the universe of $\mathcal{M}$, but sometimes sets of elements. $\mathcal{M}$ is only the first step towards the OCL model $\mathcal{M}_D^{OCL}$ to be given in Section 4.

Definition 4 **($\mathcal{M}_D$).**

1. *For each sort symbol $S \in S_{\mathcal{D}}$ the domain $I(S)$ consists of all objects in the class S of the snapshot D plus one new element $\perp$ reserved to stand for the value of otherwise undefined terms. The universe M_D of $\mathcal{M}_D$ is the union of all type universes $I(S)$.*
2. *If S is a sort symbol arising from an association class attached to an association r between the classes S_1 and S_2, then $I(S)$ is the cartesian product of $I(S_1)$ and $I(S_2)$.*
3. *For sort symbols $S_1, S_2 \in S_{\mathcal{D}}$ with $S_1 \ll_{\mathcal{D}} S_2$ we stipulate $I(S_1) \subseteq I(S_2)$. For sort symbols $S_1, S_2 \in S_{\mathcal{D}}$ satisfying neither $S_1 \ll_{\mathcal{D}} S_2$ nor $S_2 \ll_{\mathcal{D}} S_1$ the sort universes $I(S_1)$, $I(S_2)$ are disjoint.*
4. *The function symbol $f_{r,e_1,e_2} \in F_{\mathcal{D}}$ with signature $S_1 \to Set(S_2)$ will be interpreted by the function $I(f_{r,e_1,e_2})$. For an arbitrary object $a \in I(S_1)$ $I(f_{r,e_1,e_2})(a)$ will be the set of all objects b in $I(S_2)$ that are in D linked to a via association r.*
5. *For function symbols $f_a : S \to S_r$ arising from an attribute a, the function value $I(f_a)(b)$ is the value of the attribute a of the object b of type S as given by the snapshot D.*
6. *For function symbols $f_c \in F_{\mathcal{D}}$ arising from query operations, the interpretation $I(f_c)$ is defined analogously to the previous clause. Query operations are not explicitly required to terminate. If operation c on the object b does not terminate then we set $I(f_c)(b) = \perp$.*
7. *The value of $I(f)$ for argument tuples containing one entry outside the required sort or one entry equal to $\perp$ equals $\perp$.*

Comments

1. If in clause 1 of the previous definition S is an abstract class, then S has no instances in D. In this case $I(S)$ is the set-theoretic union $I(S_1) \cup \ldots \cup I(S_k)$ where $S_1, \ldots, S_k$ are all immediate subclasses of S. This contradicts the position taken in [16, 4.3.1], which would imply $I(S) = \emptyset$ for an abstract class S. Our position is, however, in accordance with the semantics of an abstract class in [13, p. 117]:

   > An abstract class may not have direct instances. It may have indirect instances through its concrete descendents.

2. If a is in particular a static attribute of class C then $I(C.a)$ is interpreted as an element of $I(T)$, where T is the type of a. If also a unary function f_a was introduced for the static attribute (see Definition 3 item 3), then we require in addition $I(C.a) = I(f_a)(o)$ for every $o \in I(C)$.
3. The approach just outlined treats association classes as classes, i.e. as sets of objects. In the above semantics an association class is **not** an association. Note that this does not contradict the UML metamodel. It is stated in [9, pages 2-20/2-21] that

   > AssociationClass is a subclass of both Association and Class (i.e., each AssociationClass is both an Association and a Class); therefore, an AssociationClass has both AssociationEnds and Features.

   The meta-model describes how to build syntactically correct diagrams, and gives restrictions on which model elements may be combined in which way. It has no effect on the semantics of diagrams.

3 The Syntax of OCL

The grammar for OCL in [8] is hard to understand. It is our goal to give here a human-oriented definition. We make use of the papers [12] which contains a description of the OCL syntax at the level of the UML metamodel, and [11] which gives a first account of a semantics for OCL.

We will present the syntax of OCL in two steps. In the first part the OCL type system will be explained. The second part contains a human-oriented description of the OCL grammar.

3.1 The OCL type system

Syntax of Type Expressions

Definition 5 (Type expressions). *Let $\mathcal{D}$ be a fixed class diagram. The type expressions with respect to $\mathcal{D}$ are as follows:*

1. *Integer, Real, Boolean, String*
   are type expressions. These are refered to as the simple OCL type expressions.
2. *OclType, OclAny, OclExpression, OclState*
   are type expressions. We call these meta type expressions.
3. *Any $s \in S_{\mathcal{D}}$, i.e. any class occuring in $\mathcal{D}$ is a type expression.*
   Following [16] we call these model types.
4. *If T is a type expression that is not itself of the form $Collection(T')$, $Set(T')$, $Bag(T')$, or $Sequence(T')$, then*

   $Collection(T)$, $Set(T)$, $Bag(T)$, $Sequence(T)$

   are type expressions. The types denoted by these expressions are usually refered to as collection types.

We will use $\mathrm{TE}_{\mathcal{D}}$*, or simply* TE *if no confusion is possible, to refer to the set of type expressions with respect to $\mathcal{D}$. In accordance with [10, Section 7.8.1] we refer to the types from clauses 1 and 2 as* basic types.

Comments

1. We consider the sentence "Collection, Set, Bag and Sequence are basic types as well." on [10, Page 7-7] contradicting the above stipulation of basic types as a plunder that will be remedied in a future release.
2. Unlike [12], we adhere to the standard and do not allow nesting of collection type constructors, i.e. $Set(Set(Integer))$ is not a legal type expression.
3. The accepted issue #3143 of the UML RTF proposes to drop the special OCL syntax for enumerations and use instead UML classes with stereotype $\ll enumeration \gg$. This also makes the OCL type *Enumeration* superfluous, so we did omit it. Following [1] one might reintroduce it as the common supertype of all model types with stereotype $\ll enumeration \gg$. But this remains to be decided.
4. $Set(OclType)$, $Set(OclAny)$, $Set(OclExpression)$, $Set(OclState)$ are legal type expressions.

Definition 6 (Direct Subtypes). *For type expressions $T_1, T_2 \in \mathrm{TE}_{\mathcal{D}}$ the subtype relation $T_1 <^{OCL}_{\mathcal{D}} T_2$ is the least relation satisfying the following conditions:*

1. *If T_1, T_2 are model types and T_1 is a subtype of T_2 in the UML model, i.e. $T_1 <_{\mathcal{D}} T_2$, then $T_1 <^{OCL}_{\mathcal{D}} T_2$.*
2. *$Integer \ <^{OCL}_{\mathcal{D}}\ Real$.*
3. *For all type expressions T, not denoting a collection type,*
   (a) *$Set(T) <^{OCL}_{\mathcal{D}} Collection(T)$*
   (b) *$Bag(T) <^{OCL}_{\mathcal{D}} Collection(T)$*
   (c) *$Sequence(T) <^{OCL}_{\mathcal{D}} Collection(T)$*
4. *If T is a model type or a basic type different from $OCLAny$, then $T <^{OCL}_{\mathcal{D}}\ OCLAny$.*
5. *If $T_1 \ <^{OCL}_{\mathcal{D}}\ T_2$ and C is any of the type constructors Collection, Set, Bag, Sequence, then $C(T_1) <^{OCL}_{\mathcal{D}} C(T_2)$.*

Definition 7 (Type conformance). *The transitive, reflexive closure of the subtype relation $<^{OCL}_{\mathcal{D}}$ is denoted by $\ll^{OCL}_{\mathcal{D}}$. If $T_1 \ll^{OCL}_{\mathcal{D}} T_2$ holds, we say that T_1 conforms to T_2.*

Semantics of Types The constructions explained in this subsection are performed with respect to a fixed snapshot D of a UML class diagram $\mathcal{D}$. As described in Section 2 we associate with D a many-sorted structure $\mathcal{M}_D = (M_D, I)$ consisting of the universe M_D and the interpretation function I. Let $S_{\mathcal{D}}$ denote the model types in $\mathcal{D}$, see Definition 1. Then every $C \in S_{\mathcal{D}}$ is interpreted via I as a subset of the universe M_D, i.e. $I(C) \subset M_D$.

The objective is to define the structure $\mathcal{M}^{OCL}_D = (M^{OCL}_D, I^{OCL})$ extending $\mathcal{M}_D$ such that for every OCL expression E the interpretation $I^{OCL}(E)$ is defined in $\mathcal{M}^{OCL}_D$. In this subsection we define $\mathcal{M}^{OCL}_D$ and the interpretation I^{OCL} on all type expressions. The definition of I^{OCL} will then be continued in Section 4.

When no confusion is possible we will supress the superscript OCL.

The universe of $\mathcal{M}^{OCL}_D$ is the union of all sort universes. Sort universes correspond to certain types. With the type $Integer$, for example, there is a sort universe $I(Integer)$ that consists of all integers plus the additional symbol $\perp$. For each model sort C there is the sort universe $I(C)$, as already explained in Definition 4, Clause 1. Remember that the special symbol $\perp$ for undefinied values is also an element of $I(C)$. In addition there are sort universes for $Set(C)$, $Bag(C)$ and $Sequence(C)$ and the universe for $Collection(C)$ will be the disjoint union of them. The sort universes for $Set(C)$, $Bag(C)$ and $Sequence(C)$ consist of abstract objects o, representing subsets, bags (multisets) or sequences of the sort universe of C respectively. The sort universe for $OclType$ will consist of objects that are in one-to-one correspondance with the set consisting of model types and all basic OCL types, except $OclType$.

For example, for the pre-defined type $Integer$ there will be an object $o_{Integer}$ in $M(OclType)$. In [10, Subsection 8.8.1.1] we read the somewhat mysterious sentence "All types defined in a UML model, or pre-defined within OCL, have a type." In our setting this now makes perfect sense: For every type S there is an object o_S in $M(OclType)$. We will refer to o_S as the *type object* of S.

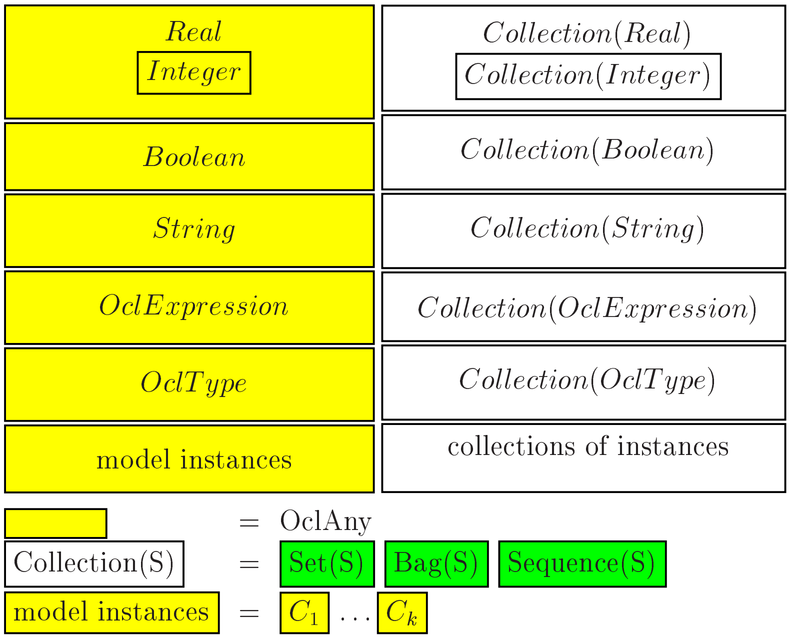

Fig. 2. Sort universes of $\mathcal{M}_D^{OCL}$

Figure 2 shows the sort universes of $\mathcal{M}_D^{OCL}$.

Definition 8. *1. For the OCL type expressions $S \in \{Integer, Real, Boolean, String\}$ the obvious universes $M'(S)$ are the integers, reals, booleans and all strings over a fixed alphabet A, respectively.*
$I(S) = M(S) = M'(S) \cup \{\perp\}$
Here $\perp$ is a new element denoting an undefined value. The sets $M'(S)$ are considered disjoint with the exception of $M'(Integer)$ being a subset of $M'(Real)$.

2. – *$I(OclType)$ = a set of objects in one-to-one correspondance with all basic or model types with the exception of $OclType$ itself.*
– *$I(OclAny) = \bigcup\{I(S) \mid S$ a basic or model type$\}$*
– *$I(OclExpression) = \mathrm{TE}_{\mathcal{D}}$*

– *$I(OclState)$ is not treated here.*
3. *for $S \in S_{\mathcal{D}}$ $I^{OCL}(S) = I(S) \cup \{\bot\}$, see Definition 4, Clause 1.*
4. – *$I(Set(T)) =$ set of all subsets of $I(T)$,*
   – *$I(Bag(T)) =$ set of all multisets of elements from $I(T)$,*
   – *$I(Sequence(T)) =$ set of all sequences of elements from $I(T)$,*
   – *$I(Collection(T)) = I(Set(T)) \cup I(Bag(T)) \cup I(Sequence(T))$*

The universe of the structure $\mathcal{M}_D^{OCL}$ is now taken to be the union of all $I(S)$.

Comments

1. While the standard [8] is precise on the meaning of *OclAny* the intended meaning of *OclType* is less clear. The definition adopted here seems a reasonable extrapolation. A more fundamental change which raises *OclType* to a class in the metamodel is proposed in [1].
2. We did not introduce type objects for collection types. The standard seems not to exclude this. But consider the following difficulty: Let o be the type object for type $Set(Integer)$ what is $o.allInstances$ supposed to be?
3. It would probably not harm to have a type object for *OclType* itself, but it will on the other hand not be of much help and certainly hard to swallow.

3.2 The Syntax of OCL Constraints

A constraints starts with a *header* fixing the context in which it is to be understood. Headers come in two forms, one for the classifier context, and one for the operator context. We start with the classifier context:

context (c **:**)? typeName **inv** expressionName? **:** OclExpression

The trailing question mark ? indicates optional elements; OCL keywords are set in **boldface**. 'typeName' could for example be the name of a class in the fixed UML diagram. In general we allow all type expressions that are not collection expressions as context type. It is possible to introduce a name for easy referencing of expressions. The optional parameter c will act very much like a variable of the type given by typename in the following OCL expression. Variable is here to be understood in the way it is used in formal logic. A header may define more than one expression:

context (c **:**)? typeName
  inv expressionName$_1$? **:** OclExpression$_1$
  ...
  ...
  inv expressionName$_n$? **:** OclExpression$_n$

Constraints for an operator context look like this:

context (c **:**)? typeName **::**opName**(**p_1**:** type$_1$**;** ... **;**p_k**:** type$_k$ **):**rtype
  {**pre** ,**post** } expressionName? **:** OclExpression

Here opName is meant to be the name of an operator defined on the given type. The list of parameters $p_1 \dots p_k$ may be empty and the return type, rtype, may be missing or both. As above, an operator constraint may contain more than one expression.

In the headers just shown OCL expressions have to be of type Boolean. Also the stereotype **inv** can only appear in a classifier context while the stereotypes **pre** and **post** can only show up in operator contexts.

We have added the optional parameter(c **:**)? also in the operator context to set it on equal footing with classifier contexts, though we have seen no example of this in the literature.

The definition of OCL expressions to be given presently depends on a fixed class diagram $\mathcal{D}$. More precisely, $\mathcal{D}$ uniquely determines the set $S_{\mathcal{D}}$ of model types and the set $F_{\mathcal{D}}$ of functions together with their sorting signature. Model types and type expressions for OCL have been introduced in Subsection 3.1 already.

Definition 9 (OCL function symbols). *The set $F_{\mathcal{D}}^{OCL}$ of function symbols admitted in forming OCL expressions for diagram $\mathcal{D}$ is*

1. *$F_{\mathcal{D}}$ (see Definition 2) plus*
2. *For any basic or model type S there is a constant symbol S in F with type$(S) =$ OclType,*
3. *All properties of the pre-defined OCL types as detailed in the standard [8, Section 7.8] plus*
4. *The constant symbol* result*. This is only needed for expressions within the context of an operator opname. The type of* result *will then be the return type of opname.*

Definition 10 (OCL expressions). *The set OCLExp of OCL expressions is the smallest set satisfying the following recursive conditions. At the same time we define for every OCL expression e its unique type type(e).*

1. *For every model type $t \in S_{\mathcal{D}}$ there is an unlimited number of variables v_t^i. Each variable v_t^i is in OCLExp with type$(v_t^i) = t$. The parameters of an operator constraint are special instances of variables of the type specified in their declaration.*
2. **self** *is a special variable, where type(***self***) is given by context information.*
3. **result** *is in OCLExp.*
   This is only allowed if the expression occurs in the context of an operator m in the stereotype **post***. Then type(***result***) = return type of m.*
4. *There are constant symbols for integers, reals, and strings, for example 15, 7.88, 'Peter'. The precise syntax for these constants is given by the OCL grammar, where they are called* literals*. In addition there are constants for the two Boolean values, $true$ and $false$.*
5. *If f is a function symbol in F with argument types $t_1, \dots, t_k$ and result type t_r and $e_1, \dots, e_k$ are OCL expressions with type$(e_i) \ll_{\mathcal{D}}^{OCL} t_i$ for all $1 \leq i \leq k$ (see Definition 7) then*
   $f(e_1, \dots, e_k) \in OCLExp$ with type$(f(e_1, \dots, e_k)) = t_r$.
6. *If f is a function symbol in F with argument types $t_1, \dots, t_k$ and result type t_r where f is not the name of an operation and $e_1, \dots, e_k$ are OCL expressions with type$(e_i) \ll_{\mathcal{D}}^{OCL} t_i$ for all $1 \leq i \leq k$ (see Definition 7) then*

$f@pre(e_1, \dots, e_k) \in OCLExp$ with $type(f(e_1, \dots, e_k)) = t_r$.
OCL expressions containing @pre may only occur under the **post** *stereotype.*

7. *If e_1, e_2 are OCL expression of the same type, then $e_1 = e_2$, $e_1 <> e_2$ are OCL expressions of type Boolean.*
8. *If e is an OCL expression of type Boolean, e_1, e_2 are expressions of type t_1 and t_2 respectively and either $t_1 \ll t_2$ or $t_1 \gg t_2$ then*
   if e **then** e_1 **else** e_2
   is an OCL expression.
   $$type(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2) = \begin{cases} t_2 \text{ if } t_1 \ll t_2 \\ t_1 \text{ if } t_1 \gg t_2 \end{cases}$$

If need arises we will write more precisely $OCLExp_{\mathcal{D}}$ instead of $OCLExp$.

Comments

1. We allow the shorthand notation for **collect** as stated in the standard [10, Section 7.6.2] and also in [16, Subsection 3.6.11].
   Assume e is an OCL expression with type $Set(T)$, and $attr$ is an attribute of type T. Then $e.attr$ is a shorthand for $e \to \textbf{collect}(c \mid c.attr)$. The same applies to functions arising from associations. Consider as an example the UML class diagram in Figure 3.

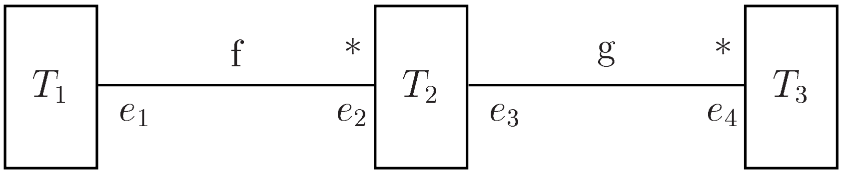

Fig. 3. Composition of navigation

   The OCL expression $\textbf{self}.e_2$ in the context T_1 has type $Set(T_2)$. The expression $\textbf{self}.e_2.e_4$ in the same context is short for $\textbf{self}.e_2 \to \textbf{collect}(c \mid c.e_4)$ and has type $Bag(T_3)$.
2. We have in clause 4 of the previous definition blurred the distinction between syntax and semantics. Adhering strictly to this distinction we should have introduced a constant symbol c_a for every integer a with the semantic interpretation $I(c_a) = a$. This seemed too much trouble for too little reason.
3. In Definition 10 we have used the functional notation as concrete syntax. This does not imply that we consider it superior to the concrete navigation syntax used in the OCL standard. We will in fact feel free to use both. Some remarks on the comparision of OCL notation to the notation used in the modeling language Alloy may be found in [15].

function syntax	navigation syntax
$g(f(c))$	$c.f.g$
$oclIsKindOf(self, Class)$	$self.oclIsKindOf(Class)$
$forAll(c, x, e)$	$c \to forAll(x \mid e)$

Definition 11 (Free variables). *For an OCL expression e we denote by* $\mathrm{FV}(e)$ *the set of free variables in e. The expression* **self** *is counted as a free variable.*

We trust that the notion of free variable, and of its counterpart bound variable, are intuitively clear. Here is an example:

$\mathrm{FV}(\textbf{iterate}(e, v_t : t, v_{t_1} : t_1 = e_1 \mid f)) = \mathrm{FV}(e) \cup (\mathrm{FV}(f) \setminus \{v_t, v_{t_1}\})$

Definition 12 (OCL expressions in context).

1. *An OCL expression e may occur in the context*

   context *(c* **:***)? typeName* **inv** *expressionName?* **:** *e*

   only if
   (a) *the postfix @pre and* **result** *do not occur in e*
   (b) $\mathrm{FV}(e) = \{\textbf{self}\}$ *if the parameter c is not specified. If c is specified, then we insist that* **self** *does not occur in e and* $\mathrm{FV}(e) = \{c\}$.
   (c) $type(e) = Boolean$.
2. *An OCL expression e may occur in the context*

   context *(c* **:***)? typeName*
       :: *opname***(***p_1***:** *type$_1$***;** *...* **;***p_k***:** *type$_k$* **):***rtype*
     {**pre** , **post** } *expressionName?* **:** *e*

   only if
   (a) $\mathrm{FV}(e) = \{p_1, \ldots, p_k\} \cup \{\textbf{self}\}$ *if c is not specified, and*
   $\mathrm{FV}(e) = \{p_1, \ldots, p_k\} \cup \{c\}$ *if c is specified.*
   (b) *in the case of the* **pre** *stereotype, @pre and* **result** *do not occur in e.*
   (c) $type(e) = Boolean$.

Comments and examples

1. OCL constraints have so far in the literature been considered with respect to a fixed context class, say S. Inspecting Definition 10 reveals that this information is only needed in determining the type of **self**. The same observation applies to [11, Definition 5]. Replacing **self** by the optional parameter $c : S$ specified in the context header we arrive at a definition of *OCLExp* not depending on context information.
   The notion of context is helpful, so we introduce it in Definition 12, thus arriving at a clear separation of concerns.
   The approach adopted here also solves another problem that has been kept under the rug in previous publications. Assume that e is an OCL expression in context c with $type(e) = \textbf{Sequence}(t)$ and we form the new expression

   $$e \to \textbf{iterate}(v_t : t, v_{t_1} : t_1 = e_1 \mid f)$$

   What should be the context of f? In most published examples it is not c, sometimes it is t, sometimes it is t_1.
   The above definitions at least give a clear answer, which we hope will also prove useful and in keeping with the spirit of OCL.

2. OCL allows as a shortcut to omit **self**. We assume in Definition 12 that this, and all similar shortcuts, have been removed.
3. Here are some examples for OCL expressions according to clause 5.
   If $f = +$ and $type(e_1) = type(e_2) = Integer$ then $e_1 + e_2$ is an expression of type $Integer$.
   There are two ways to derive the expression $e_1 + e_2$ within the OCL grammar in the standard [9]: first as a built-in operation on the type $Integer$ and second via the grammer rule for `additiveExpression`. This shows that one could simplify the grammar.
   If T is a class in $\mathcal{D}$, e an expression of type T and f an attribute for the class T with value type T_1 then $f(e)$ is in $OCLExp$ with $type(f(e)) = T_1$. In concrete OCL syntax this expression would be written as $e.f$.
   Let $f = f_{r,b_1,b_2}$ (see Definition 2) be a function symbol associated with the association ends b_1 and b_2 of an association r in $\mathcal{D}$. Assume that b_i is attached to class t_i and e is an OCL expression with $type(e) = T_1$. Then $f(e)$ is an OCL expression with $type(f(e)) = T_2$ if b_2 has multiplicity 1 and $type(f(e)) = Set(t_2)$ otherwise. In concrete OCL syntax this expression would be written as $e.b_2$ or $e.t_2$.
   If e is an expression with $type(e) = Set(T)$ then $size(e)$ is an OCL expression of type $Integer$, because $size$ is a built-in operator. In concrete syntax: $e \to size$.
4. If the type of e_1, e_2 in clause 7 is neither a meta type nor a collection type, then equality and inequality are built-in operations of the type $OCLAny$. Because of the restriction of the subtype relation in Definition 6 this does not cover equality between collection and meta types. Thus we have added clause 7 here.
5. Let e be an OCL expression of type **Sequence**(T), let v_T, v_{T_1} be variables of type T and T_1 respectively, e_1 and f expressions of type T_1, then

   $$\mathbf{iterate}(e, v_T : T, v_{T_1} : T_1 = e_1 \mid f)$$

   is an OCL expression of type T_1. **iterate** is a built-in operation for all collection types. It is most natural for sequences but also applicable on sets and bags. Its concrete syntax is $e \to \mathbf{iterate}(v_T : T, v_{T_1} : T_1 = e_1 \mid f)$.
6. The usual **let** construct for introducing abbreviations may be freely used in OCL expressions. In the formal treatment we assume that all abbreviations have been resolved.
7. We do not exclude, that within an expression e occuring as a **pre** or **post** condition in the context of an operator *opname*, the function symbol f_{opname} attached to *opname* does occur.

4 Semantics of OCL

We will now continue the definition of $\mathcal{M}_D^{OCL} = (M_D^{OCL}, I^{OCL})$ begun in Subsection 3.1. Remember that we assume a fixed snapshot D of a UML class diagram $\mathcal{D}$. From $\mathcal{D}$ we read off the set of sort and function symbols, $S_\mathcal{D}$ and $F_\mathcal{D}$, while D may be described by the many-sorted structure $\mathcal{M}_D = (M_D, I)$. We will again take the liberty to omit the superscript OCL, when no confusion can arise. In Subsection 3.1 $I(e)$ was defined for all OCL type expressions. The OCL expression e may contain free variables. The meaning of e then depends on the values assigned to these free variables. This is the job of the function β: for every variable v in e of type S we require $\beta(v)$ to be an element in $I(S)$ and we will explain how to calculate recursively the value of $I_\beta(e)$. The initial cases (Clause 1 to 4) in Definition 10 are easy to deal with and likewise the induction steps for clauses 7 and 8 are obvious. We concentrate here on clause 5 and postpone clause 6 for later.

Definition 13 (Semantics of model properties).

1. *Let f_a in $F_\mathcal{D}$ be the function symbol attached to an attribute a of the class T (see Definition 4). Let further e be an OCL expression with $type(e) \ll_\mathcal{D}^{OCL} T$. Then $I_\beta(f(e)) = I(f_a)(I_\beta(e))$.*
2. *Let $f = f_{r,e_1,e_2}$ be a function symbol in $F_\mathcal{D}$ attached to the association r and the association ends e_1, e_2 with argument type T and e an OCL expressions with $type(e) \ll_\mathcal{D}^{OCL} T$. Then $I_\beta(f(e)) = I(f_{r,e_1,e_2})(I_\beta(e))$.*

Next we should give the semantics for all pre-defined operations of OCL. We restrict ourselves to a few typical cases.

Definition 14 (Semantics of model properties).

1. *Consider the expression $e \equiv e_1 \to collect(c \mid e_2)$ (in functional notation $collect(e_1, c, e_2)$).*
   (a) *$type(e_2)$ is not a collection type.*
   In case $type(e_1) = Set(S)$ or $Bag(S)$:
   $$I_\beta(e) = \text{ the bag of elements } I_{\beta_c^a}(e_2) \text{ for all } a \in I_\beta(e_1)$$
   In case $type(e_1) = Sequence(S)$:
   $$I_\beta(e) = \langle I_{\beta_{c_1}^a}(e_2), \ldots, I_{\beta_{c_k}^a}(e_2)\rangle \text{ with } I_\beta(e_1) = \langle c_1, \ldots, c_k\rangle$$
   (b) *$type(e_2) = Set(T)$ or $Bag(T)$*
   $$I_\beta(e) = \bigcup\{I_{\beta_c^a}(e_2) \mid a \in I_\beta(e_1)\}$$
   In any case the union $\bigcup$ will result in a bag, i.e. multiple occurences will not be eliminated. The information what kind of a collection type e_1 did possess is lost, even in the case $type(e_1) = Sequence(S)$.
   (c) *$type(e_2) = Sequence(T)$*
   If $type(e_1) = Set(S)$ or $Bag(S)$ the definition of the previous item applies where $I_{\beta_c^a}(e_2)$ is treated as the bag of elements occuring in the sequence.
   If $type(e_1) = Sequence(S)$ then $I_\beta(e)$ is the concatenation of the sequences $I_{\beta_{c_i}^a}(e_2)$, with $1 \le i \le k$ and $I_\beta(e_1) = \langle c_1, \ldots, c_k\rangle$.

In these definitions we have used $\beta_c^a(x) = \begin{cases} \beta(x) & \text{if } x \neq c \\ a & \text{if } x = c \end{cases}$

Definition 15. *Let v be the only free variable that may occur in e, and β_m the variable assignment with $\beta_m(v) = m$.*

The constraint **context** *t* **inv** : *e*
is true in $\mathcal{M}_D^{OCL} = (M_D, I)$ if for every $m \in I(T)$: $I_{\beta_m}(e) =$ **true**.

The meaning of constraints on methods is defined with respect to a pair of snapshots D_1, D_2 for the same diagram $\mathcal{D}$. These lead to a pair of structures of the form: $(\mathcal{M}^{OCL}_{D_1}, \mathcal{M}^{OCL}_{D_2}) = ((M_{D_1}, I_1),(M_{D_2}, I_2))$. We will further assume that $M_{D_1} \subseteq M_{D_2}$. This amounts to the stipulation that instances once created cannot be removed.

Definition 16. *The constraint*

> **context** *(c :) typeName :: opname*$(p_1$*: type*$_1$; ... ;p_k*: type*$_k$ **)**:*rtype*
> **pre** : e_1
> **post** : e_2

is true in $(\mathcal{M}^{OCL}_{D_1}, \mathcal{M}^{OCL}_{D_2})$ *if for every* β

$$I_{1,\beta}(e_1) = \mathbf{true} \textit{ implies } I_{2,\beta}(e_2) = \mathbf{true}$$

Here β *ranges over all functions from the set* $\{c, p_1, \ldots, p_k\}$ *of variables into the universe* M_{D_1} *satisfying the typing restrictions.*

It is in Definition 14 clause 2 and 3 that flattening takes place and the creation of sets of sets or sequences of sequences is avoided.

The standard [8, Subsection 7.5.5] allows to access association ends of multiplicity [0..1] both as sets and as elements. There is no clear way in our setting to mimick this overloading. We even doubt its practicality.

5 Future Research

We are presently working on a translation of OCL into Dynamic Logic, which is the input language of the theorem prover in the KeYproject. Special care has to be taken in the translation of the @*pre* suffix. It is quite easy to use partial functions in a semantics description, as we have done here. A deduction calculus for reasoning about partial functions in the OCL framework still has to be decided on.

5.1 Acknowledgements

Many thanks are due to Wolfgang Ahrendt, Bernhard Beckert, Joao Marcos, and Reiner Hähnle, for their generous help in preparing this paper.

References

1. T. Baar and R. Hähnle. An integrated metamodel for OCL types. In R. France, B. Rumpe, and J. Whittle, editors, *Proc. OOPSLA 2000 Workshop Refactoring the UML: In Search of the Core, Minneapolis/MI, USA*, Oct. 2000.
2. M. Fowler and K. Scott. *UML Destilled. Applying the Standard Object Modeling Language.* Addison-Wesley, 1997.
3. M. Gogolla and M. Richters. On constraints and queries in UML. In Schader and Korthaus [14], pages 109–121.
4. A. Hamie. A formal semantics for checking and analysing UML models. In L. Andrade, A. Moreira, A. Deshpande, and S. Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.
5. A. Hamie, J. Howse, and S. Kent. Navigation expressions in object-oriented modelling. In *FASE'98.*

6. A. Hamie, J. Howse, and S. Kent. Interpreting the object constraint language. In *Proceedings of Asia Pacific Conference in Software Engineering*. IEEE Press, July 1998.
7. L. Mandel and M. V. Cengarle. On the expressive power of OCL. In *FM'99 - Formal Methods. World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999. Proceedings, Volume I*, volume 1708 of *LNCS*, pages 854–874. Springer, 1999.
8. OMG. Object constraint language specification, version 1.3. chapter 7 in [9]. OMG Document ad970808, September 1999.
9. OMG. OMG unified modeling language spezification, version 1.3. OMG Document, June 1999.
10. OMG. OMG unified modeling language spezification, version 1.3. OMG Document, March 2000.
11. M. Richters and M. Gogolla. On formalizing the UML object constraint language OCL. In T. W. Ling, S. Ram, and M. L. Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, pages 449–464. Springer, Berlin, LNCS 1507, 1998.
12. M. Richters and M. Gogolla. A metamodel for OCL. volume 1723 of *LNCS*, pages 156–171. Springer, 1999.
13. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
14. M. Schader and A. Korthaus, editors. *The Unified Modeling Language: technical aspects and applications*. Physica-Verlag, 1998.
15. M. Vaziri and D. Jackson. Some shortcomings of ocl, the object constraint language of uml. response to object management group's request for information on uml 2.0. Technical report, MIT Laboratory for Computer science, December 1999.
16. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.