On the Expressive Power of OCL*

Luis Mandel[1] and María Victoria Cengarle[2]

[1] Forschungsinstitut für Angewandte Software-Technologie (FAST e. V.)
Arabellastr. 17, D-81925 Munich, Germany
mandel@fast.de
[2] Ludwig-Maximilians-Universität München.
Oettingenstr. 67, D-80538 Munich, Germany
cengarle@informatik.uni-muenchen.de

Abstract. This paper examines the expressive power of OCL in terms of navigability and computability. First the expressive power of OCL is compared with the relational calculus; it is showed that OCL is not equivalent to the relational calculus. Then an algorithm computing the transitive closure of a binary relation –operation that cannot be encoded in the relational calculus– is expressed in OCL. Finally the equivalence of OCL with a Turing machine is pondered.

1 Introduction

The Object Constraint Language (OCL), developed within IBM and based on IBEL (Integrated Business Engineering Language, IBM) and on Syntropy [4], is part of the Unified Modeling Language (UML) from version 1.1 on. This extension has been designed to augment a class diagram with additional information which cannot be otherwise expressed by UML diagrams; previous versions of UML have only allowed the definition of constraints as annotations in an informal textual way. OCL allows the definition of integrity constraints at the user level, and it has also been used for the formalization of the metamodel of UML. The introduction of a constraint language is an important step towards the formalization of system specification. Constraints represent necessary conditions for a domain to constitute a model of the static aspects of the specified system. OCL is based on standard set theory and it was designed to specify invariants of classes and types in the class model, to specify type invariant of stereotypes, to describe pre- and postconditions on operations and methods, to describe guards; it is also suited to specify queries in the database sense. That is, OCL can be used to write expressions that evaluate to "true" or "false" and also to write expressions that once evaluated return the values respectively satisfying the constraint specified by those query expressions.

In [6] some weak points of OCL have been shown, for example the difference between data values and object instances is not clear. Normally data values are immutable whereas objects (or class instances) are mutable. The term

* This work was partially supported by the Bayerische Forschungsstiftung.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 854–874, 1999.
© Springer-Verlag Berlin Heidelberg 1999

value/object as well as the term subtype/subclass is not consistently used in the specification document of OCL [9]. In fact, the specification document is very informal and not even the examples given there are consistent with their English explanation. Moreover there it is said that an OCL expression can be part of a guard but neither examples nor explanation of how guards work are given. Some steps have been done in order to formalize it, for example in [10] a formal semantics of OCL has been proposed.

This paper aims at examining the completeness of OCL. We show that OCL is not powerful enough to denote any query expression of the relational calculus. However, by means of OCL it is possible to calculate the transitive closure of a relation. We also show that any primitive recursive function can be calculated by an OCL expression, and hint that OCL is not Turing complete.

The paper is organized as follows. Section 2 briefly introduces "OCL by examples." Section 3 studies the expressive power of OCL: Section 3.1 demonstrates that the expressiveness of OCL is *not* as powerful as the relational calculus, in Section 3.2 it is shown how the transitive closure of a relation can be computed in OCL, and in Section 3.3 the Turing incompleteness of OCL is shown. Finally in Section 4 some conclusions are drawn.

2 OCL Examples

This section briefly introduces the OCL language using the example of a class hierarchy of a diagram editor; see Fig. 1. The editor supports the notion of group of graphic elements. A document consists of pages, and a page consists of graphic elements. Graphic elements are either geometric figures or groups of at least two graphic elements; a graphic element can be member of at most one group. Graphic elements can be moved, rotated, etc. Geometric figures are either one dimensional (points and curves) or two dimensional (circles, ellipses, etc.). Two dimensional figures can be filled with a color.

The diagram of Fig. 1 can be enhanced with OCL constraints that further restrict the possible system states. OCL expressions are either of an OCL predefined type or of a class of the class model to which the expressions are attached. OCL expressions compute values without changing the system state. OCL uses dot notation for accessing the attributes of objects. The attribute `radius` of the class `Circle` is accessed by the expression `Circle.radius`. If all the instances of that class are restricted to have a positive radius, a constraint `Circle.radius > 0` can be written. The result type of this expression is `Boolean`, establishing an invariant for the class `Circle`. Alternatively, one can write

```
Circle
   self.radius > 0  -- two dashes precede a comment
```

where the name of the class underlined is the *context* of the constraint, and an occurrence of `self` in it refers to any instance of that class.

A basic data structure of OCL is `Set`. The expression `self.vertices` in the context `Polygon` computes the set of all the vertices of a polygon object and returns a value of type `Set(Point)`. A further data structure `Bag` which stands

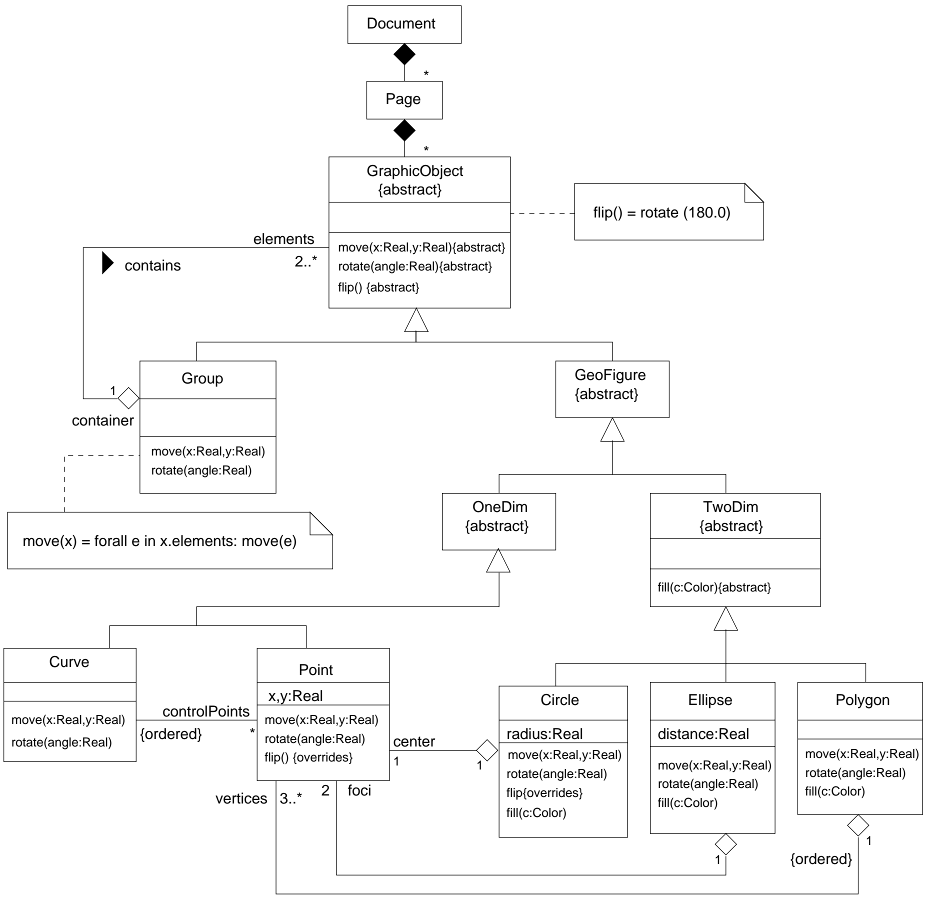

Fig. 1. A diagram editor

for a multiset, i.e. a set with possibly repeated elements. Besides sets and bags another data structure is `Sequence` as usual denoting ordered bags. All these data structures are parametric, one writes `Set(T)`, `Bag(T)`, and `Sequence(T)` for `T` a type, and sets, bags, and sequences are subtypes of the abstract parametric class `Collection`. In the context of `Group` the query `self.elements` returns a set whose cardinality is calculated by applying the *feature* `size` associated with collections:

```
Group
  self.elements->size
```

The result of this expression is of type `Integer` and is restricted to values greater than or equal to 2 by the multiplicity at the `elements` end of the aggregation `contains` between `Group` and `GraphicObject`.

OCL provides universal and existential quantification. A constraint of the `Polygon` class is that any two vertex points of a polygon must be in different positions. This is expressed using the feature `forAll` as follows:

```
Polygon
    self.vertices->forAll(p1, p2 : Point |
    (p1.x = p2.x and p1.y = p2.y) implies p1 = p2)
```

This expression has type `Boolean`.

OCL also allows the navigation through the information using queries. For instance, the bag containing the surface of each circle with center `p` can be computed using the feature `collect` as in the following expression (where the classname with small initial is used if the association end has no rolename):

```
p.circle->collect(3.14*radius)
```

whose result type is `Bag(Real)`.

`allInstances` is a feature associated with any type that returns the set of all instances of the given type. For example the set of polygons which are triangles is calculated as follows:

```
Polygon.allInstances->select(p : Polygon | p.vertices->size = 3)
```

A powerful feature of collections is `iterate`, by means of it many other ones can be implemented. `iterate` traverses a collection, performs a calculation with each of its elements, and stores the results in an accumulator, whose last value is the result of the `iterate` expression. For instance the sum of the surfaces of all the circles present in the model is calculated by the following expression:

```
Circle.allInstances->iterate(c : Circle ;        -- iteration variable
sum : Real = 0                        -- accumulator with initial value 0
|   sum + 3.14 * c.radius * c.radius)             -- new value of sum
--  ^^^ this occurrence of sum refers to its rvalue
```

A more involved example is the calculation of the set of all the polygons with at least one vertex in common with a given polygon `p0`:

```
Polygon.allInstances->iterate(p1 : Polygon ;
cv : Set(Polygon) = Set{}   -- accumulator, initially the empty set
|   if p1 <> p0 and
       (p1.vertices->intersection(p0.vertices))->notEmpty
    then cv->including(p1) else cv endif)
```

It is worth mentioning that OCL does not have the concept of tuple. Moreover collections (i.e. sets, sequences and bags), which could be used to simulate tuple functions, are flat, that is, in OCL there is no possibility to create a set of sets, for example (if a query returns a set of sets, then this result is flattened).

3 Completeness

This section examines the expressive power of OCL from three different viewpoints. First we ponder if the operations of the relational calculus can be formulated in OCL, second we try to compute the transitive closure of a relation (which is an operation that cannot be expressed by means of the relational calculus), and third we consider the Turing completeness of OCL.

3.1 Equivalence of OCL and Relational Calculus

In this section we show that OCL is not complete in the sense defined by Ullman (see [13]) i.e. it is not possible to express in OCL the five basic operations of the relational algebra. We also show that some of the derived expressions of the relational algebra are primitive of OCL or can be expressed using OCL. Given that OCL does not have the concept of tuple, some operations like for instance the projection trivially cannot be expressed in OCL.

There are three abstract query languages, namely the relational algebra, the tuple relational calculus, and the domain relational calculus. They are all equivalent in expressive power to the each other and were proposed by Codd [3] to represent the *minimun capability of any reasonable query language using the relational model*. Moreover, as stated in [13]:

> A language that can (at least) simulate tuple calculus, or equivalently, relational algebra or domain calculus, is said to be *complete*.

Query languages for the relational model break down into two broad classes:

algebraic languages: queries are expressed by applying operators to relations;
predicate calculus languages: a desired set of tuples is described by specifying a predicate the tuples must satisfy.

As shown in the previous section, on the one hand OCL expressions define the set of elements satisfying a constraint expression, and on the other some operators can be applied to sets of instances. Therefore OCL follows a mixed model. We now consider how to express the operations of the relational algebra in OCL.

Union is a primitive operation for collections in OCL and is expressed as:

```
set->union(set2 : Set(T)) : Set(T)
bag->union(bag2 : Bag(T)) : Bag(T)
sequence->union(sequence2 : Sequence(T)) : Sequence(T)
```

Both collections must be of the same type; only the union of sets and bags is allowed.

Difference is also a primitive operation for sets and is expressed as follows:

```
set - (set2 : Set(T)) : Set(T)
```

Difference is not defined for other subtypes of collection.

Cartesian product is not directly expressible in OCL.[1] Moreover, if an OCL expression computes a value of type T, then T either is a class present in the

[1] The only mention of Cartesian product in [9] is when introducing the extended variant of the forAll operation, namely the one with more than one iterator: collection->forAll(e1,e2 | <Boolean-expression-on-e1-and-e2>), which in fact is a forAll on the Cartesian product of the collection with itself. The result of an expression of this form is Boolean.

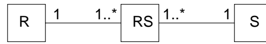

Fig. 2. Cartesian product

class diagram being navigated or is a primitive type of OCL; in other words, any operation that computes values of another type cannot be expressed in OCL. However, if it is indispensable to add to a class diagram constraints only expressible using the Cartesian product of classes `R` and `S`, then a further class `RS` should be included, associated with `R` and `S` as shown in Fig. 2, and the following OCL expression should be attached that guarantees that the set of instances of `RS` in fact always is the Cartesian product of `R` and `S`:

```
R.allInstances->union(S.allInstances)->forAll(r, s : oclAny
|   if r.oclType.name=s.oclType.name
    then true
    else RS.allInstances->exists( t : RS | t.r=r and t.s=s) endif)
```

We take the union of the set of instances of class `R` and the set of instances of class `S`, and test if every two elements of this set either are of the same type (given that we cannot compare types, we compare their names) or there is an instance of class `RS` such that it is associated to both of them. Note that `R.allInstances` is of type `Set(R)` and `S.allInstances` is of type `Set(S)`, and then the union of these two sets is of type `Set(oclAny)` where `oclAny` is the supertype of all types in the model. An alternative is to build a set of pairs `(r,s)` with `r` an instance of `R` and `s` an instance of `S`, where pairs are simulated by sequences of two elements. Unfortunately this is impossible since in OCL all collections of collections are automatically flattened. Still we can build a sequence `{r1,s1,r1,s2,...,r1,sN,r2,s1,...,rM,sN}` such that any subsequence `{rI,sJ}` (i.e. a subsequence of two elements beginning at an odd position) represents an element of the Cartesian product of `R` and `S`, such that conversely if `r` is an instance of `R` and `s` is an instance of `S` then there is a subsequence `{r,s}` of `rs` beginning at an odd position. This is achieved as follows:

```
R.allInstances->iterate(r : R ;
rs : Sequence(oclAny) = Sequence{}
|   S.allInstances->iterate(s : S ;
    rs1 : Sequence(oclAny) = rs
    |   rs1->append(r)->append(s)
)   )
```

`rs` is the sequence encoding the Cartesian product of `R` and `S`, which is of type `Sequence(oclAny)` since we do not know of another supertype of both `R` and `S`. The accumulator `rs1` is needed because the return value of the inner iteration is the last value of its accumulator, and then `rs` can be properly updated.

Projection is possible for just one attribute using the `collect` on collections:

```
collection->collect(attrName)
```

The result of such an expression is `Bag(T)` if `T` is the type of values for `attrName`; in order to eliminate duplicates, one can use the operation `asSet` of bags:

```
collection->collect(attrName)->asSet
```

and in this way a result of type `Set(T)` is obtained. Given that the Cartesian product is not expressible in OCL, the projection on more than one attribute cannot be expressed in OCL either.[2] If we need the projection of the set of instances of a class on more than one attribute, say `a1`, ..., `aN`, then –similarly to the alternative proposed for the Cartesian product– we can build the sequence of $n \times m$ values (where $n =$ `N` and m is the number of instances currently present in the class, say `R`, to be projected) as follows:

```
R.allInstances->iterate(r : R ;
proj : Sequence(oclAny) = Sequence{}
|   proj->append(r.a1)->...->append(r.aN))
```

Selection can be expressed by using the `select` operation on collections:

```
collection->select(Boolean-expr)
```

Some derived operations have also a representation in OCL.

Intersection is a primitive in OCL for sets and bags, and is written as follows:

```
set->intersection(set2 : Set(T)) : Set(T)
set->intersection(bag : Bag(T)) : Set(T)
bag->intersection(bag2 : Bag(T)) : Bag(T)
bag->intersection(set : Set(T)) : Set(T)
```

Join of two relations is the selection of those tuples of a Cartesian product whose i-th component is in the relation θ with the corresponding $(r+j)$-th component. Given that we cannot compute Cartesian products, we assume that there is a class `RS` whose set of instances invariantly is the Cartesian product of the sets of instances of classes `R` and `S` (cf. paragraph on Cartesian product above) and we compute the join of `R` and `S` w.r.t. attributes `a` and `b` in the relation `theta` (assuming that θ is expressible in OCL) as follows:

```
RS.allInstances->select( t : RS | t.r.a theta t.s.b )
```

If alternatively we have built the sequence `rs = {r1,s1,...}` (see above paragraph on Cartesian product), then the join can likewise be stored in a sequence as follows:

```
Sequence{1..(rs->size)/2}->iterate(i : Integer ;
join : Sequence(oclAny) = Sequence{}
|   if (rs->at(2*i-1)).a theta (rs->at(2*i)).b
    then join->append(rs->at(2*i-1))->append(rs->at(2*i))
    else join endif)
```

[2] Notice that in general it cannot be ensured that the projection of a set of instances of a class (i.e. a set of n-tuples) is of a type present in the model, thus an operation returning such a set is not expressible in OCL; cf. discussion above on Cartesian products in OCL.

Here `rs` is the sequence containing the Cartesian product of `R` and `S`, and `Sequence{1..(rs->size)/2}` is the sequence of integer numbers between 1 and the size of `rs` (which we know of even length) divided by two, whose elements are used to index the sequence `rs`. Each element in `rs` at an odd position is an element of `R` and each element in `rs` at an even position is an element of `S`; each element in `rs` at an odd position is paired with its following in `rs`. If their `a` resp. `b` attribute are in the relation `theta`, then both of them are stored in the result accumulator `join`. (Note that, given that we cannot store values in variables, in the above algorithm we should replace every one the four occurrences of `rs` by the algorithm computing it, besides the first which could also be replaced by `R.allInstances->size * S.allInstances->size`. This fact would represent a problem if two different computations of `rs` yield sequences in different order.)

Natural join is similarly calculated: if `a1`, ..., `aN` are all the attributes in both `R` and `S` with the same name, then

```
RS.allInstances->select( t : RS | t.r.a1 = t.s.a1 and
                                  ... t.r.aN = t.s.aN )
```

or alternatively

```
Sequence{1..(rs->size)/2}->iterate(i : Integer ;
join : Sequence(oclAny) = Sequence{} |
    if (rs->at(2*i-1)).a1 = (rs->at(2*i)).a1 and
       ... (rs->at(2*i-1)).aN = (rs->at(2*i)).aN
    then join->append(rs->at(2*i-1))->append(rs->at(2*i))
    else join endif)
```

The obvious conclusion is that OCL is incomplete. Completeness can be achieved by just including a concept of tuple functions (or creation of virtual classes) and a mechanism for creating instances of any type or class. These instances are of course not meant to be included to the current model of the class diagram but to allow navigation on a higher level of abstraction. The question is if in an object-oriented environment this is needed: one can argue that projection is not necessary since one can handle the whole object and use only the attrbutes of interest, and that if the Cartesian product is needed then the classes involved will be anyway connected in some form or another (e.g. by an association).

3.2 Transitive Closure of a Relation

Data manipulation languages normally have capabilities beyond those of relational calculus, like *arithmetic* operations, *assignment* commands, and *aggregate* functions. Often algebraic expressions must involve some arithmetic operations like $a < b + c$; notice that e.g. $+$ does not appear in the relational algebra. The assignment of a computed relation to be the value of a relation name is undoubtedly useful, see discussion about Cartesian product or projection in the previous section. Furthermore some operations like sum, average, max, min are

also desirable as aggregate functions, that can be applied to columns of a relation to obtain a single quantity. For these reasons a (complete) query language with such capabilities is said to be *more than complete* [13, p. 175]. OCL includes the following arithmetic and aggregate functions:

type of operands	**operations**
Real	$=, +, -, *, /$, abs, floor, max, min, $<, >, <=, >=$
Integer	$=, +, -, *, /$, abs, div, mod, max, min
Boolean	=, or, xor, and, not, implies, if-then-else[3]
String	=, size, concat, toUpper, toLower, substring
Enumeration	$=, <>$

Notice that `Integer` is a subclass of `Real`, that is, for each formal parameter of type `Real` an actual parameter of type `Integer` can be used. OCL does not include assignment commands.

Some languages are *more than complete* even after eliminating the above mentioned functions. For example QBE (Query-by-Example, see [13]) allows the computation of the transitive closure of a relation. The transitive closure R^+ of a relation $R \subseteq A \times A$ is the least transitive relation containing R:

1. $xRy \Rightarrow xR^+y$;
2. $(x\ R^+\ y \wedge y\ R^+\ z) \Rightarrow xR^+z$;
3. if S satisfies (1) and (2), then $R^+ \subseteq S$.

The transitive closure of a relation cannot be expressed in relational algebra or relational calculus; see [2, 1]. The transitive closure of a relation is needed in our example of Fig. 1 if we want to ensure that no instance of `Group` is (recursively) an element of itself, since what we require is the non-reflexivity of the transitive closure of the relation `contains`. In the remainder of this section we study how to express this constraint in OCL.

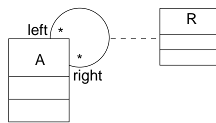

Fig. 3. An association class `R` from class `A` to class `A`

First of all what we need is a relation that, as R above, is a subset of the Cartesian product of a set A with itself. In the framework of UML class diagrams, the easiest is to have an association class (and not just an association name) as depicted in Fig. 3. We therefore lift the association name `contains` to an

[3] In the specification document [9] of OCL, the `if-then-else` operation is listed among the `Boolean` ones, of course just the first argument of an `if-then-else` operation has to be of type `Boolean` and its result value is of the (least) supertype of the types of the `then`-branch and of the `else`-branch.

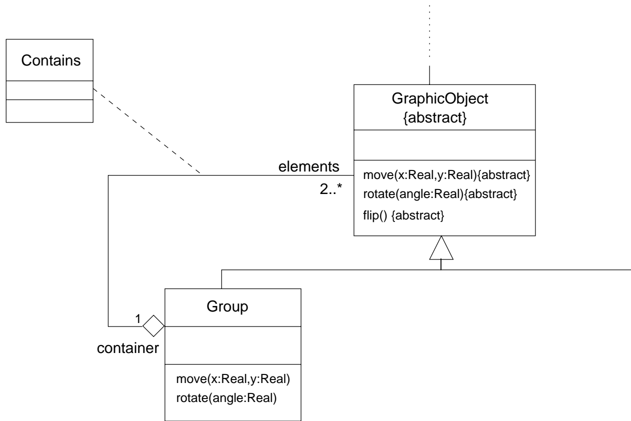

Fig. 4. The `Contains` association class

association class as pictured in Fig. 4. Notice that the relation `Contains` might include pairs of objects of different classes, like an instance of `Group` containing an instance of (a subclass of the abstract class) `GeoFigure`. The subset (which we call `r`) of instances of `Contains` that includes only the desired pairs, namely $\mathtt{r} = \mathtt{Contains} \cap (\mathtt{Group} \times \mathtt{Group})$, can be computed as follows:

```
-- algorithm computing r included in Group x Group:
Contains.allInstances->iterate(pair : Contains;
r : Set(Contains) = Set{}
|   if pair.elements.oclIsTypeOf(Group)
    then r->including(pair) else r endif)
```

Let us remark that, in the above query, every `pair` is just one pair of an instance of `Group` and a single instance of `GraphicObject`, which can be referred to by `container` resp. `elements`. The class `GraphicObject` is abstract, that means, any of its instances must be an instance of any of its concrete subclasses. In the iteration, therefore, we only include those pairs whose `GraphicObject`-component is of type `Group`. Now the Warshall's algorithm (see e.g. [5]) can be applied to `r` and in this way a new set `s` of instances of `Contains` is computed that is the transitive closure of `r`:

```
-- algorithm computing s, the transitive closure of r:
Group.allInstances->iterate(g3 : Group ;
s : Set(Contains) = r
|   Group.allInstances->iterate(g2 : Group ;
    s2 : Set(Contains) = s
    |   Group.allInstances->iterate(g1 : Group ;
        s1 : Set(Contains) = s2
        |   if s1->exists(c1,c2 : Contains |
                (c1.container=g1 and c1.elements=g2) or
                (c1.container=g1 and c1.elements=g3 and
```

```
                  c2.container=g3 and c2.elements=g2)  )
              then s1->including(c) else s1 endif
           -- where: c : Contains with c.container=g1 and c.elements=g2
    )   )   )
```

In the above algorithm the sets `s1` and `s2` had to be declared in order for `s` to be properly updated. The variant of `exists` with two iterators used in this algorithm is inexistent in OCL. However, given the equivalences $(\exists x)\varphi \equiv \neg(\forall x)\neg\varphi$ and $(\exists x)(\exists y)\varphi \equiv \neg(\forall x)\neg(\exists y)\varphi \equiv \neg(\forall x)\neg\neg(\forall y)\neg\varphi \equiv \neg(\forall x)(\forall y)\neg\varphi$,

```
collection->exists(e1,e2 | <Boolean-expr-on-e1-and-e2>)
```

is the abbreviation we use for

```
not collection->forAll(e1,e2 | not <Boolean-expr-on-e1-and-e2>).
```

The OCL constraint we are looking for is the non-reflexivity of the set `s`:

```
not s->exists(d : Contains | d.container=d.elements)
```

The desired OCL constraint is therefore the following:

```
not
Group.allInstances->iterate(g3 : Group ;
s : Set(Contains) = Contains.allInstances->iterate(pair : Contains;
                    r : Set(Contains) = Set{}
                    |   if pair.elements.oclIsTypeOf(Group)
                        then r->including(pair) else r endif
                    )
|   Group.allInstances->iterate(g2 : Group ;
    s2 : Set(Contains) = s
    |   Group.allInstances->iterate(g1 : Group ;
        s1 : Set(Contains) = s2
        |   if s1->exists(c1,c2 : Contains |
                 (c1.container=g1 and c1.elements=g2) or
                 (c1.container=g1 and c1.elements=g3 and
                  c2.container=g3 and c2.elements=g2)  )
              then s1->including(c) else s1 endif
           -- where: c : Contains with c.container=g1 and c.elements=g2
    )   )   )->exists(d : Contains | d.container=d.elements)
```

Here we discover a problem: the instance `c` of `Contains` with `c.container=g1` and `c.elements=g2`, necessary when computing `s` for recording partial paths, cannot be created. This instance is not meant to be added to the actual state of the model, the same as many sets and such that can be calculated using OCL are not meant to be added to the actual state of the model. Furthermore, and differently to what was remarked in Section 3.1 about Cartesian product and projection, we are not trying to generate an instance of an unknown type but of a type present in the class diagram. An alternative would be to manipulate, instead of a set `s` of instances of `Contains`, a set `s'` of pairs of instances of `Group` (represented by a sequence of length two) such that, if the sequence `{g1,g2}`

belongs to s', then there is a path in r from g1 to g2. Unfortunately this is impossible since within OCL all collections of collections are automaticaly flattened (it is not specified what type has the result of flattening a set of sequences).

The third idea that comes to mind, in order to overcome nested collections, is to manipulate a sequence t with an even number of elements such that if g1 and g2 belong to t, g1 is at an odd position i of t, and g2 is at position i+1, then there is a path in r from g1 to g2. This seems to be a satisfactory solution, the problematic sentence s1->including(c) could then be replaced by t1->append(g1)->append(g2) (also in this case we need auxiliary variables t1 and t2). Now, the initial value of t is not r but

```
t : Sequence(Group)
= r->iterate(pair : Contains ;
  res : Sequence(Group) = Sequence{}
  |   res->append(pair.container)->append(pair.elements))
```

Also the algorithm computing the transitive closure of r has to be accordingly adapted wherever s, s1 or s2 are used. There is just one more place where one of these variables is used, namely when the existence is asked of two pairs c1 and c2 in s1 that satisfy certain property. Now we have to look for the existence of two subsequences of length two both beginning at an odd position and satisfying the same property. We access the subsequences using an index that points to their position in t. We replace the test

```
s1->exists(c1,c2 : Contains |
(c1.container=g1 and c1.elements=g2) or
(c1.container=g1 and c1.elements=g3 and
 c2.container=g3 and c2.elements=g2)  )
```

by the following

```
Sequence{1..(t1->size)/2}->exists(i,j : Integer |
(t1->at(2*i-1) = g1 and t1->at(2*i) = g2) or
(t1->at(2*i-1) = g1 and t1->at(2*i) = g3 and
 t1->at(2*j-1) = g3 and t1->at(2*j) = g2)       )
```

where Sequence{1..(t1->size)/2} is the sequence of integer numbers between 1 and the size of t1 (which we know of even length) divided by two, and the elements of this sequence are used to index the sequence t1.

Finally in a similar way we can test the non-reflexivity of t:

```
not Sequence{1..(t->size)/2}->exists(i : Integer
|   t->at(2*i-1) = t->at(2*i)                   )
```

There is still a last hurdle to surmount. t is just a mnemonic we have used, that has to be replaced by the algorithm that computes it, since we cannot use a variable. t occurs three times in the above test of non-reflexivity, and we would like to calculate it only once. Moreover, if we replace each one of the three occurrences of t in the test above, then we cannot be sure that each instance of the sequence is equal to another one, and in particular when testing if t->at(2*i) = t->at(2*i+1) it is absolutely necessary that both t's at each

side of the equation are equal. The only constructs using new names are the operations iterating on collections, and of them only `iterate` allows for a variable of an arbitrary type (namely the accumulator) and the assignment of an arbitrary value to this variable. But the last value of the accumulator is the return value of an `iterate`, and we need a `Boolean`. We can therefore just *assume* that the different occurrences of the algorithm computing `t` return always the same sequence. The resulting algorithm is as follows:

```
not Sequence{1..(
    -- t
    Group.allInstances->iterate(g3 : Group ;
    t : Sequence(Group) = -- r
        Contains.allInstances->iterate(pair : Contains;
        r : Set(Contains) = Set{}
        |   if pair.elements.oclIsTypeOf(Group)
            then r->including(pair) else r endif
        )->iterate(pair : Contains ;
        res : Sequence(Group) = Sequence{}
        |   res->append(pair.container)->append(pair.elements)
        )
    |   Group.allInstances->iterate(g2 : Group ;
        t2 : Sequence(Group) = t
        |   Group.allInstances->iterate(g1 : Group ;
            t1 : Sequence(Group) = t2
            |   if Sequence{1..(t1->size)/2}->exists(i,j : Integer |
                   (t1->at(2*i-1) = g1 and t1->at(2*i) = g2) or
                   (t1->at(2*i-1) = g1 and t1->at(2*i) = g3 and
                    t1->at(2*j-1) = g3 and t1->at(2*j) = g2)       )
                then t1->append(g1)->append(g2) else t1 endif
    )   )   )
    ->size)/2}->exists(i : Integer
|   -- t
    Group.allInstances->iterate(g3 : Group ;
    t : Sequence(Group) = -- r
        Contains.allInstances->iterate(pair : Contains;
        r : Set(Contains) = Set{}
        |   if pair.elements.oclIsTypeOf(Group)
            then r->including(pair) else r endif
        )->iterate(pair : Contains ;
        res : Sequence(Group) = Sequence{}
        |   res->append(pair.container)->append(pair.elements)
        )
    |   Group.allInstances->iterate(g2 : Group ;
        t2 : Sequence(Group) = t
        |   Group.allInstances->iterate(g1 : Group ;
            t1 : Sequence(Group) = t2
            |   if Sequence{1..(t1->size)/2}->exists(i,j : Integer |
                   (t1->at(2*i-1) = g1 and t1->at(2*i) = g2) or
                   (t1->at(2*i-1) = g1 and t1->at(2*i) = g3 and
                    t1->at(2*j-1) = g3 and t1->at(2*j) = g2)       )
```

```
                then t1->append(g1)->append(g2) else t1 endif
        )   )   )
        ->at(2*i-1) =
        -- t
        Group.allInstances->iterate(g3 : Group ;
        t : Sequence(Group) = -- r
            Contains.allInstances->iterate(pair : Contains;
            r : Set(Contains) = Set{}
            |   if pair.elements.oclIsTypeOf(Group)
                then r->including(pair) else r endif
            )->iterate(pair : Contains ;
            res : Sequence(Group) = Sequence{}
            |   res->append(pair.container)->append(pair.elements)
            )
        |   Group.allInstances->iterate(g2 : Group ;
            t2 : Sequence(Group) = t
            |   Group.allInstances->iterate(g1 : Group ;
                t1 : Sequence(Group) = t2
                |   if Sequence{1..(t1->size)/2}->exists(i,j : Integer |
                       (t1->at(2*i-1) = g1 and t1->at(2*i) = g2) or
                       (t1->at(2*i-1) = g1 and t1->at(2*i) = g3 and
                        t1->at(2*j-1) = g3 and t1->at(2*j) = g2)       )
                    then t1->append(g1)->append(g2) else t1 endif
        )   )   )
        ->at(2*i)
    )
```

Although the computation capabilities for computing the transitive closure of a binary relation are present in OCL, here again a concept of tuple functions would have made the above algorithm considerably simpler. At this point we want to remark the notion of *relational completeness* as formulated in [8, p. 94]:

> In language implementations, the following two operations are needed to assure relational completeness:
> (a) The ability to represent assignments, that is, the ability to create new relations to store the results of relational algebra operations that are also relations. [...]
> (b) The ability to compute transitive closures which enables recursion and/or nesting of relational algebra operations to express expressions of arbitrary complexity. [...]

Also Codd (see [3]) asserted the need for more than complete languages, providing tuple and aggregate functions.

3.3 Turing Completeness

This section addresses the Turing completeness of OCL. We show that primitive recursive functions are expressible in OCL and hint why general recursive

functions cannot be expressed in OCL. In order to do so, we show that LOOP-programs can and WHILE-programs cannot be written in OCL (see [11]).

The syntax of LOOP-programs is as follows:

$$P ::= X \texttt{ <- } X + C \mid X \texttt{ <- } X - C \mid \texttt{LOOP } X \texttt{ DO } P \texttt{ END} \mid P \texttt{ ; } P$$
$$X ::= x_0 \mid x_1 \mid x_2 \mid \ldots \qquad (\textit{variables})$$
$$C ::= 0 \mid 1 \mid 2 \mid \ldots \qquad (\textit{constants})$$

The semantics of LOOP-programs is straightforward. The value assignment x_i `<-` $x_j + c$ is interpreted as usual, that is, the new value of the variable x_i is the value of x_j plus c. The value assignment x_i `<-` $x_j - c$ is the non-negative subtraction, that is, if $c > x_j$ then the new value of x_i is 0 otherwise the value of x_j minus c. A LOOP-program of the form P_1 `;` P_2 is interpreted as the execution of P_1 and afterwards the execution of P_2. Finally a LOOP-program of the form `LOOP` x_i `DO` P `END` is interpreted as follows: the program P is executed n times, where n is the value of the variable x_i *at the beginning* (i.e. the change of the value of x_i within P does not affect the number of repetitions). Given a LOOP-program that computes a k-ary function f, it is assumed that the input values $n_1, \ldots, n_k$ are initially stored in the variables $x_1, \ldots, x_k$, that any other variable has initial value 0, and that the result $f(n_1, \ldots, n_k)$ is stored in the variable x_0 after execution of the program.

LOOP-programs are WHILE-programs, and additionally if P is a WHILE-program then `WHILE` $x_i \neq 0$ `DO` P `END` is a WHILE-program. The semantics of the new construct is the following: the program P is repeatedly executed as long as the value of x_i is different from 0. (The `LOOP` construct becomes superfluous, `LOOP` x `DO` P `END` can be simulated by `WHILE` $y \neq 0$ `DO` P `;` $y := y - 1$ `END`.)

Every LOOP-program can be computed by an OCL expression. Indeed, given a LOOP-program P computing a k-ary function and using auxiliary variables $x_{k+1}, \ldots, x_r$, we write an OCL expression that manipulates an array $\texttt{vals}_1$ of $r+1$ values (representing the values of the variables $x_0, \ldots, x_r$) and returns the first value of this array after executing the translation of P:

```
Sequence{1..1}->iterate(it : Integer ; -- iterator, will be ignored
vals₁ : Sequence(Integer) = {0,n₁,...,nₖ, 0,...,0 }
                                          (r−k) times
|    trans(P,1)
)->first
```

The return value of the above expression is the first value of the sequence $\texttt{vals}_1$ after (iterating one time) the execution of $\textit{trans}(P, 1)$. Initially $\texttt{vals}_1$ stores the value n_i for the variable x_i $(i = 1, \ldots, k)$ and 0 otherwise. In general, the return value of $\textit{trans}(P, n)$ with $n \in \mathbb{N}$ is stored in $\texttt{vals}_n$. The OCL expression $\textit{trans}(P, n)$ is defined by induction on P as follows:

– The translation $\textit{trans}(P, n)$ of a program P of the form x_i `<-` $x_j + c$ depends on i and j and is defined by:
  - $\textit{trans}(x_i \texttt{ <- } x_i + c, n) =$
    ```
    valsₙ->iterate(val : Integer ;
    newvals : Sequence(Integer) = Sequence{}
    ```

```
|   if newvals->size = i
    then newvals->append(val+c)
    else newvals->append(val) endif)
```

- if $i > j$, then
  $trans(x_i$ `<-` $x_j + c, n) =$
  ```
  vals_n->iterate(val : Integer ;
  newvals : Sequence(Integer) = Sequence{}
  |   if newvals->size = i
      then newvals->append(newvals->at(j+1)+c)
      else newvals->append(val) endif)
  ```
- if $i < j$, then
  $trans(x_i$ `<-` $x_j + c, n) =$
  ```
  vals_n->iterate(val : Integer ;
  newvals : Sequence(Integer) = Sequence{}
  |   if newvals->size = j
      then (newvals->subSequence(1,i))          -- (1)
           ->append(val+c)
           ->union(newvals->subSequence(i+2,j)) -- (2)
           ->append(val)
      else newvals->append(val) endif)[4]
  ```

– The translation $trans(P, n)$ of a program P of the form x_i `<-` $x_j - c$ depends not only on i and j but also on the values of x_j and c, and is defined by:
  - $trans(x_i$ `<-` $x_i - c, n) =$
    ```
    vals_n->iterate(val : Integer ;
    newvals : Sequence(Integer) = Sequence{}
    |   if newvals->size = i
        then newvals->append(if val<c then 0 else val-c endif)
        else newvals->append(val) endif)
    ```
  - if $i > j$, then
    $trans(x_i$ `<-` $x_j - c, n) =$
    ```
    vals_n->iterate(val : Integer ;
    newvals : Sequence(Integer) = Sequence{}
    |   if newvals->size = i
        then newvals->append(if newvals->at(j+1)<c then 0
                             else newvals->at(j+1)-c endif)
        else newvals->append(val) endif)
    ```
  - if $i < j$, then
    $trans(x_i$ `<-` $x_j - c, n) =$
    ```
    vals_n->iterate(val : Integer ;
    newvals : Sequence(Integer) = Sequence{}
    |   if newvals->size = j
        then (newvals->subSequence(1,i))          -- (1)
             ->append(if val<c then 0 else val-c endif)
    ```

[4] Notice that $i < j$ implies $i + 1 \leq j$, therefore it might be incorrect to speak of the subsequence of `newvals` starting at $i + 2$ and ending at j, see statement above commented with `(2)`. The same w.r.t. the sentence commented with `(1)` if $i = 0$. [9] does specify the result of extracting a subsequence whose upper position number is less than its lower position number, we therefore assume that in such a case the subsequence is empty.

```
              ->union(newvals->subSequence(i+2,j)) -- (2)
              ->append(val)
         else newvals->append(val) endif)
```
[5]

– $trans$(`LOOP` x_i `DO` P `END`$, n) =$
```
Sequence{1..vals_n->at(i+1)}->iterate(it : Integer ; -- will be ignored
vals_{n+1} : Sequence(Integer) = vals_n
|    trans(P, n+1))
```
– $trans(P_1\ ;\ P_2, n) =$
```
Sequence{1..2}->iterate(step : Integer ;
vals_{n+1} : Sequence(Integer) = vals_n
|    if step = 1 then trans(P_1, n+1) else trans(P_2, n+1) endif)
```

The number n accompanying the definition of *trans* allows for the definition of new variables that do not shadow previously (in the outer block) defined ones.

So for instance the function $f(n, m) = n + m$ is computed by the program `X0 <- X1 + X2` and can be encoded as the following LOOP-program `P`

```
X0 <- X1 + 0 ;     -- P1
LOOP X2 DO         -- P2
    X0 <- X0 + 1   -- P21
END
```

which is translated to OCL by successively calculating *trans* as follows:

1.
```
Sequence{1..1}->iterate(it : Integer ;
vals1 : Sequence(Integer) = {0,n,m} | trans(P,1))->first
```
2.
```
Sequence{1..1}->iterate(it : Integer ;
vals1 : Sequence(Integer) = {0,n,m}
|  Sequence{1..2}->iterate(step : Integer ;
   vals2 : Sequence(Integer) = vals1
   |  if step = 1 then trans(P1,2) else trans(P2,2) endif) )->first
```
3.
```
Sequence{1..1}->iterate(it : Integer ;
vals1 : Sequence(Integer) = {0,n,m}
|  Sequence{1..2}->iterate(step : Integer ;
   vals2 : Sequence(Integer) = vals1
   |  if step = 1
      then vals2->iterate(val : Integer ;
           newvals : Sequence(Integer) = Sequence{}
           |  if newvals->size = 1
              then (newvals->subSequence(1,0))->append(val+0)
                   ->union(newvals->subSequence(0+2,1))->append(val)
              else newvals->append(val)
              endif)
      else Sequence{1..vals2->at(2+1)}->iterate(it : Integer ;
           vals3 : Sequence(Integer) = vals2
           |  trans(P21,3))
      endif)
)->first
```

[5] Here again we assume that the subsequence `(1)` of `newvals` starting at 1 and ending at i is empty if $i = 0$, and that the subsequence `(2)` of `newvals` starting at $i+2$ and ending at j is empty if $i + 1 = j$.

4.
```
Sequence{1..1}->iterate(it : Integer ;
vals1 : Sequence(Integer) = {0,n,m}
|  Sequence{1..2}->iterate(step : Integer ;
   vals2 : Sequence(Integer) = vals1
   |  if step = 1
      then vals2->iterate(val : Integer ;
           newvals : Sequence(Integer) = Sequence{}
           |  if newvals->size = 1
              then (newvals->subSequence(1,0))->append(val+0)
                   ->union(newvals->subSequence(0+2,1))->append(val)
              else newvals->append(val)
              endif)
      else Sequence{1..vals2->at(2+1)}->iterate(it : Integer ;
           vals3 : Sequence(Integer) = vals2
           |  vals3->iterate(val : Integer ;
              newvals : Sequence(Integer) = Sequence{}
              |  if newvals->size = 0
                 then newvals->append(val+1)
                 else newvals->append(val) endif)
           )
      endif)
)->first
```

We now prove that the OCL expression defined in terms of a LOOP-program computes the same function.

Proposition 1. *Let P be a LOOP-program used in a context where only variables among $x_0, \ldots, x_r$ are used. For any $m_0, \ldots, m_r, m'_0, \ldots, m'_r \in \mathbb{N}$, for any $n \in \mathbb{N}$, if the variable x_i changes its value from m_i to m'_i $(i = 0, \ldots, r)$ after the execution of P, then $\mathtt{vals}_n$ changes its value from $\{m_0, \ldots, m_r\}$ to $\{m'_0, \ldots, m'_r\}$ after the execution of trans(P, n).*

Proof. The thesis is proved by induction on the structure of P.

- (P is x_i `<-` $x_j \pm c$)
  Trivial.
- (P is `LOOP` x_j `DO` P' `END`)
  $trans(P,k) =$ `Sequence{1..vals`$_k$`->at(`j`+1)}->iterate(i : Integer ;`
  `vals`$_{k+1}$ `: Sequence(Integer) = vals`$_k$
  `|` $trans(P', k + 1)$)
  By IH, for any $m_1, \ldots, m_r, m'_1, \ldots, m'_r \in \mathbb{N}$, for any $n \in \mathbb{N}$, if the variable x_i changes its value from m_i to m'_i $(i = 1, \ldots, r)$ after the execution of P', then $\mathtt{vals}_n$ changes its value from $\{m_0, \ldots, m_r\}$ to $\{m'_0, \ldots, m'_r\}$ after the execution of $trans(P', n)$, in particular for $n = k + 1$.
  Therefore, if $\mathtt{vals}_k = \{l_0, \ldots, l_r\}$, then $\mathtt{vals}_k$`->at(`j`+1)` $= l_j$, $\mathtt{vals}_{k+1}$ is initialized by $trans(P, k)$ with the value of $\mathtt{vals}_k$, and if after l_j successive executions of P' the variable x_i changes its value from l_i to l'_i, then after

l_j successive executions of $trans(P', k+1)$ `vals`$_{k+1}$ changes its value from `{`l_0`,...,`l_r`}` to `{`l'_0`,...,`l'_r`}`.
Given that the last value of `vals`$_{k+1}$ is the return value of $trans(P, k)$, the thesis holds.

– (P is P_1 `;` P_2)

$trans(P, k) =$ `Sequence{1..2}->iterate(step : Integer ;`
`vals`$_{k+1}$ `: Sequence(Integer) = vals`$_k$
`|   if step = 1`
`then` $trans(P_1, k+1)$ `else` $trans(P_2, k+1)$ `endif)`

This case is also trivial by IH.

□

Hence, every LOOP function is also computable using an OCL expression.

Consider now the `WHILE` construct of WHILE-programs. The iterating construct `iterate` runs through a collection (randomly ordered if not a sequence) from its beginning to its end. Thus, an `iterate` terminates if, and only if, the collection is finite. Notice that, on the one hand and according to [9, p. 13], there are three ways of getting a collection:

1. by a literal, e.g. `Set{1,2,5,3}`, or
2. by a navigation, e.g. `Polygon self.vertices`, or
3. by operations on collections, e.g. `set->union(set2)`.

The first two possibilities return a finite collection, and finite collections are closed under the operations mentioned as third possibility.[6] But, on the other hand, the feature `allInstances` associated with the type `oclType` of types returns a set; see [9, p. 20]. That is, by writing `Integer.allInstances` (or even `Real.allInstances`) we could also obtain an infinite collection.

In any way, an `iterate` either performs a previously determined number of iterations or does not terminate, since there is no possibility of interrupting an `iterate` (like e.g. the `break` command of C). In other words, the `WHILE` construct cannot be encoded in OCL, and thus semidecidable problems in general cannot be solved using OCL.

Therefore, given that the class of primitive recursive functions coincides with the class of LOOP-computable functions and that the class of μ-recursive functions coincides with the class of WHILE-computable functions (see [11]), OCL allows only for the definition of primitive recursive functions (or totally undefined functions if `Integer.allInstances` is a valid OCL expression).

4 Conclusions

OCL brought to UML 1.1 two advantages: At metalevel it has been used for the definition of the UML metamodel and at user level it can be used to describe additional constraints about the objects in the model, constraints that cannot be described in a graphic way. OCL can also be used as a navigation language. We

[6] This is also true for the negation given the (implicit) closed world assumption.

have shown that OCL is not as expressive as the relational calculus and therefore it is incomplete as query language in the database sense. On the other hand we have shown that in OCL the transitive closure of a relation, operation that cannot be expressed in relational calculus, can be computed by an OCL expression (assuming some kind of determinism when constructing twice a sequence); the resulting code is somehow tricky and neither intuitive nor easy to read. Both relational completeness and an easier to read OCL expression computing the transitive closure of a binary relation can be achieved by just adding a concept of tupling. Finally we demonstrated that OCL can compute any primitive recursive function and hinted that not any recursive function can be computed by an OCL expression; in other words, OCL is not equivalent to a Turing machine.

Since we first wrote this article, a book [14] on OCL was published that completes the original OCL specification with many and detailed examples; unfortunately, some questions like e.g. the result type of flattening collections in general is still missing. Due to the ambiguities, some inconsistencies and the lack of formality of the OCL specification some authors have suggested to replace it by other well-founded language such as EER (see [6]) or CASL (see [12] and also [7]). It is expected that new revisions of UML will also bring to the community a new revised version of OCL or, may be better, a new approach to facilitate navigation and specification of model properties in a formal way.

References

[1] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Sixth ACM Symposium on Principles of Programming Languages (POPL79, proceedings)*, pages 110–117, 1979.

[2] P. Atzeni and V. D. Antonellis. *Relational database theory.* The Benjamin/-Cummings Publishing Company, Inc., 1993.

[3] E. F. Codd. Relational Completeness of Data Base Sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65–98. Prentice Hall, Englewood Cliffs, New Jersey, 1972.

[4] S. Cook and J. Daniels. *Designing Object Systems—Object Oriented Modeling with Syntropy.* Prentice Hall, 1994.

[5] J. L. Gersting. *Mathematical Structures for Computer Science.* Computer Science Press, 3rd edition, 1993.

[6] M. Gogolla and M. Richters. On Constraints and Queries in UML. In M. Schader and A. Korthaus, editors, *Proc. UML'97 Workshop 'The Unified Modeling Language - Technical Aspects and Applications'*, pages 109–121. Physica-Verlag, Heidelberg, 1997.

[7] P. D. Mosses. CoFI: The common framework initiative for algebraic specification and development. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Proceedings of the Seventh Joint Conference on Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, number 1214 in LNCS, Lille, France, Apr. 1997. Springer.

[8] E. Ozkarahan. *Database Machines and Database Management.* Prentice Hall, 1986.

[9] RATIONAL Software Corporation. *Object Constraint Language Specification*, Sept. 1997. Version 1.1. Available at `www.rational.com/uml/`.

[10] M. Richters and M. Gogolla. On Formalizing the UML Object Constraint Language OCL. In T.-W. Ling, editor, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*. Springer, Berlin, LNCS, 1998.
[11] U. Schöning. *Theoretische Informatik – kurzgefaßt*. Spektrum Akademischer Verlag, 2nd edition, 1995.
[12] The CoFI Task Group on Language Design. CASL: The common algebraic specification language: Summary. Available at `www.brics.dk/Projects/CoFI/-Documents/CASL/Summary/`, 1998.
[13] J. D. Ullman. *Principles of Database Systems*. Computer Software Engineering Series. Computer Science Press, 1982.
[14] J. B. Warmer and A. B. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.