On the Expressive Power of the Object Constraint Language OCL*

Luis Mandel
Forschungsinstitut für Angewandte Software Technologie (FAST e. V.[†])
Arabellastr. 17
D-81925 München, Germany
Tel: +49 89 920047 39
Fax: +49 89 920047 18
mandel@fast.de

María Victoria Cengarle
Institut für Informatik[‡]
Ludwig-Maximilians-Universität München
Oettingenstr. 67
D-80538 München, Germany
Tel: +49 89 2178 2275
Fax: +49 89 2178 2152
cengarle@informatik.uni-muenchen.de

4th February 1999

Abstract

The Object Constraint Language (OCL) has been introduced by IBM for business modelling and adopted by UML as a mean to specify invariants of classes and types in a class model, to specify type invariant of stereotypes, to describe pre- and postconditions on operations and methods, to describe guards, and also as a navigation language. OCL is a language of typed expressions, where an expression can be universally and existentially quantified.

This paper aims at examining the expressive power of OCL in terms of navigability and computability. First the expressive power of OCL is compared with the relational calculus, and it is showed that OCL is not equivalent to the relational calculus. Then it is studied if the transitive closure of a binary relation is expressible in OCL, showing how it is coded. Finally the equivalence of OCL with a Turing machine is pondered.

*This work was partially supported by the Bayerische Forschungsstiftung.

1 Introduction

The Object Constraint Language (OCL), developed by IBEL (Integrated Business Engineering Language, IBM), 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 thought to specify invariants on classes and types in the class model, to specify type invariant for stereotypes, to describe pre- and postconditions on operations and methods, to describe guards, and 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 [GR97] 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 considered to be immutable whereas object instances are thought to be mutable. The term value/object as well as the term subtype/subclass were not used consistently in the specification document of OCL [RAT97]. 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 [RG98] a formal semantics of OCL has been proposed.

This paper aims at examining the completeness of OCL. We show that the expressiveness of 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 OCL is not Turing complete.

The paper is organized as follows. Section 2 briefly introduces "OCL by examples" and Section 3 gives the basic concepts of the relational calculus. Section 4 studies the expressive power of OCL, Section 4.1 demonstrates that the expressiveness of OCL is *not* as powerful as the relational calculus, in Section 4.2 it is shown how the transitive closure of a relation can be computed in OCL, and in Section 4.3 the Turing incompleteness of OCL is shown. Finally in Section 5 some conclusions are drawn.

2 OCL Examples

In this section the OCL language will be briefly presented using the example of Fig. 1. In this example the static class hierarchy of a diagram editor is shown. The editor supports the notion of group of graphic elements. A document consists of pages, and a page consists of graphic elements. A graphic element is either a geometric figure or a group of at least two graphic elements; a graphic element can be member of at most one group. Graphic elements can be moved, rotated, and flipped. Geometric figures are either one dimensional (points and curves) or two dimensional (circles, ellipses, and polygons). Two dimensional figures can be filled with a color.

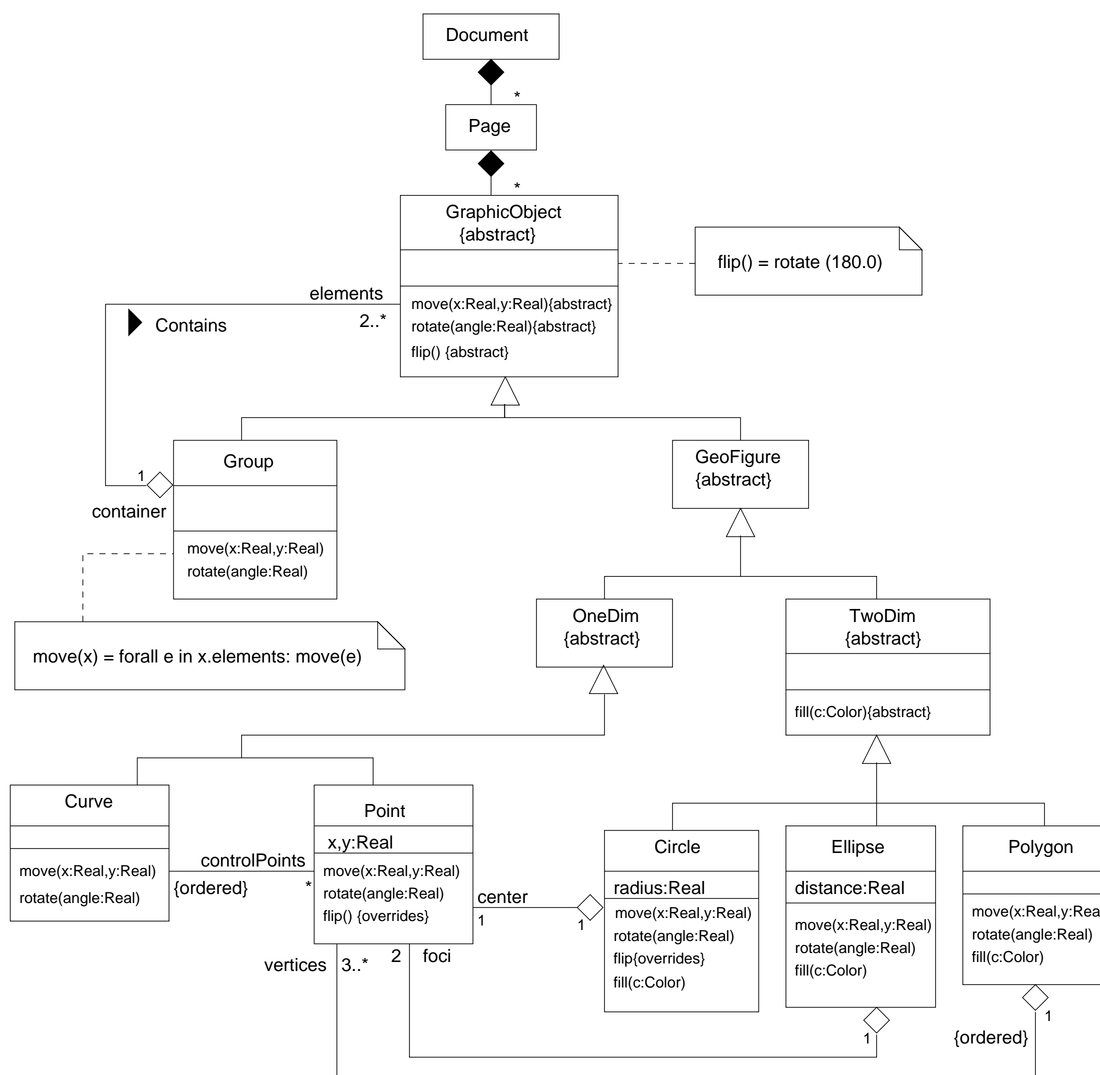

Figure 1: A diagram editor

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. An OCL expression computes a value 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`, and establishes an invariant for the class `Circle`. Alternatively, one can write the following expression:

```
Circle
  self.radius > 0
```

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

One of the basic data structures 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 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 number of members is calculated by applying the *feature* `size` associated with collections as follows:

```
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 |
  (p1.x = p2.x and p1.y = p2.y) implies p1 = p2)
```

This expression has type `Boolean`.

OCL has also the possibility to navigate through the information using queries. For instance, the bag containing the square of the distance of each vertex of a polygon to the origin can be computed using the feature `collect` as in the following expression:

```
Polygon
  self.vertices->collect(x*x+y*y)
```

whose result type is Bag(Real). (We do not calculate the distances since the type Real does not have a square root function associated, cf. Section 4.2.)

As another example, and in order to get all the points which are center of a circle whose radius is bigger than 2, we can write the following query:

```
Point.allInstances->select(p : Point | p.circle.radius > 2)
```

Notice that the above OCL expression lacks a context; there is no need to fix a context since self is not used. Also notice that the association between class Point and class Circle lacks a role name on the side of Circle, thus we use the class name (with lower case initial) instead of a role name in order to access the radius of a circle whose center point is a given point p. If an instance p of class Point is not center of any instance of class Circle, then it is not selected (and of course no error is raised by trying to access p.circle.radius).

The feature allInstances allows for a query of all the polygons which are triangles:

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

allInstances is a feature associated with any type that returns the set of all instances of the given type. In the above expression, from the set of Polygon instances we select those polygons p satisfying the property of having a set of vertices of cardinality 3.

If we wanted to define a class Triangle subclass of Polygon, we could write the above OCL expression replacing Polygon by Triangle and replacing select by forAll in order to ensure that all the instances of Triangle are real triangles:

```
Triangle.allInstances->forAll(p : Triangle | p.vertices->size = 3)
```

A powerful feature of collections is iterate, by means of which many of the above mentioned ones can be implemented. As its name already hints, iterate iterates over a collection, performs a calculation with each element of the collection, and stores the results in an accumulator. 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{}
|   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 The Relational Algebra

There are three abstract query languages, namely the relational algebra, the tuple relational calculus, and the domain relational calculus. Each of these abstract query languages is equivalent in expressive power to the others, and were proposed by Codd [Cod72] to represent the *minimun capability of any reasonable query language using the relational model.* Moreover, as stated in [Ull82]:

> 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: those languages where queries are expressed by applying specialized operators to relations;

predicate calculus languages: those languages where queries describe a desired set of tuples by specifying a predicate the tuples must satisfy.

As shown in the previous section queries in OCL are defined, on the one hand, as the set of elements satisfying a constraint expression, and on the other hand some special operators can be applied in order to interact with relations. Therefore OCL follows a mixed model.

Following the statement of above we present the operations of the relational algebra. In the following we assume that R and S are relations.

Union: The union of relations R and S denoted by $R \cup S$, is the set of tuples that are in R or in S.

Set difference: The difference of relations R and S, denoted $R - S$ is the set of tuples in R but not in S.

Cartesian product: Let R and S be relations of arity r and s, respectively. The Cartesian product $R \times S$ is the set of $(r+s)$-tuples whose first r components form a tuple in R and whose last s components form a tuple in S.

Projection: Let R a relation of arity r, then $\pi_{i_1,i_2,\ldots,i_m}(R)$ denotes the set of m-tuples result of the projection of R onto the components $i_1, i_2, \ldots, i_m$ $(1 \leq i_j \leq r, i_j \neq i_k$ if $j \neq k)$.

Selection: Let F be a constraint in some constraint language. Then $\sigma_F(R)$ is the set of tuples in R satisfying the constraint F.

There are some additional algebraic operators which, although non-primitive, are very useful and common for queries. All of them can be expressed using the five primitive operators of above.

Intersection: $R \cap S$ denote the set of tuples belonging to R and to S. It is a shorthand for $R - (R - S)$.

Quotient: Let R and S be relations of arity r and s respectively, where wlog $r > s$, and $S \neq \emptyset$. Then $R \div S$ is the set of $(r-s)$-tuples t that for all s-tuples u in S the tuple tu is in R. In other words if $Q = R \div S$ then $Q \times S = R$.

To express the quotient relation using the basic algebraic operators, let T the set of tuples $\pi_{1,2,\ldots,r-s}(R)$. Then $(T \times S) - R$ is the set of r-tuples that are not in R, but are formed by taking the first $r-s$ components of a tuple in R and following it by a tuple in S. Then let:

$$V = \pi_{1,2,\ldots,r-s}((T \times S) - R)$$

V is the set of $(r-s)$-tuples that are the first $r-s$ components of a tuple in R such that for some s-tuple u in S, tu is not in R. Hence $T - V$ is $R \div S$. Writing it in a single expression we get:

$$R \div S = \pi_{1,2,\ldots,r-s}(R) - \pi_{1,2,\ldots,r-s}((\pi_{1,2,\ldots,r-s}(R) \times S) - R)$$

Join: The θ-join of R and S on columns i and j is written $R \underset{i\theta j}{\bowtie} S$ where θ is a comparison operator ($=$, $<$, etc.). This operator is a shorthand for $\sigma_{i\theta r+j}(R \times S)$, if R is of arity r. That is, the θ-join of R and S is the set of those tuples of the Cartesian product of R and S such that the ith component of R stands in relation θ to the jth component of S. If θ is equality, the operation is called *equijoin*.

Natural join: The natural join, written $R \bowtie S$, is applicable only when both R and S have columns that are named by attributes. If R and S do not have an attribute in common, then $R \bowtie S = R \times S$. Otherwise, to compute $R \bowtie S$,

1. compute $R \times S$;

2. for each attribute A that names both a column in R and a column in S, select from $R \times S$ those tuples whose values agree in the columns $R.A$ and $S.A$;
3. for each tuple of above, project out the column $S.A$.

Formally then, if $A_1, A_2, \ldots, A_k$ are all the attribute names used for both R and S, $R \bowtie S$ is $\pi_{i_1,i_2,\ldots,i_m}\sigma_{R.A_1=S.A_1\wedge\ldots\wedge R.A_k=S.A_k}(R \times S)$, where $i_1, i_2, \ldots, i_m$ is the list of all components of $R \times S$, in order except the components $S.A_1, \ldots, S.A_k$.

4 Completeness

In the present section we examine the expressive power of OCL from three different viewpoints. First we ponder if it is complete in the sense defined by Ullman (see [Ull82]) i.e. 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.

4.1 Equivalence of OCL and relational calculus

In this section the completeness of OCL will be analyzed. It will be shown that it is *not* possible to express in OCL the five basic operations of the relational algebra. It will also be shown that some of the derived expressions of the relational algebra are primitive of OCL or can be expressed using OCL. It is worth mentioning that OCL does not have the concept of tuple and therefore some operations like for instance the projection trivially cannot be expressed in OCL.

Union is a primitive operation for collections (sequences, bags and sets) 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)
```

Notice that both collections have to be of the same type, i.e. the union of, for instance, sets and sequences is not defined. Only the union of bags and sets is allowed.

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

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

The evaluation of the expression of above results in the set of those elements that belong to `set` (assumed of type `Set(T)`) and are not in `set2`. The result is of type `Set(T)`. Difference is not defined for other types of collection, however,

```
bag - (bag2 : Bag(T)) : Bag(T)
```

can nevertheless be implemented iterating the operation `including`:

```
bag->iterate(
e : T;
acc : Bag(T) = Bag{}
|   if bag2->includes(e)
    then acc
    else acc->including(e)
    endif
)
```

Here `e` is the iteration variable (taking in each iteration the value of an element of `bag`) and `acc` is the accumulator initially empty. The resulting bag, which we call `result`, verifies the following condition:

```
result->forAll(e : T | result->count(e) = bag->count(e))    and
bag->forAll(e : T | results->includes(e) xor bag2->includes(e))
```

Cartesian product is not directly expressible in OCL.[1] Moreover, if `T` is a class or type that neither is present in the class diagram being navigated nor is primitive of OCL, then any operation that computes values of type `T` cannot be expressed in OCL. However, if it is indispensable to add to a class diagram constraints expressible only 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 the set of instances of `R` and the set of instances of `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

[1] The only mention of Cartesian product in [RAT97] 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`.

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. Notice that `R.allInstances` is of type `Set(R)` and `S.allInstances` is of type `Set(S)`, therefore the union of these two sets is of type `Set(oclAny)` where `oclAny` is the supertype of all types in the model.

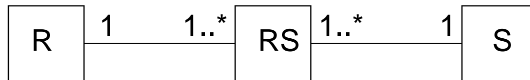

Figure 2: Cartesian product

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 within 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`, and 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 can be 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)
)   )
```

Here `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 in this way `rs` can be properly updated.

Projection is possible for just one attribute by means of the operation `collect` on collections:

```
collection->collect(attributeName)
```

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

```
collection->collect(attributeName)->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 in OCL by means of the `select` operation on collections:

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

Some derived operations have also a representation in OCL.

Intersection is also 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)
```

Quotient is not primitive of OCL and cannot be expressed using the primitives of OCL since it is based on Cartesian product and on projection.

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 (see Section 3). 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:

[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.

```
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
)
```

Here `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, and the elements of this sequence 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`. (Notice 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. In order to achieve completeness OCL should just include 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.

4.2 More than Complete

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* [Ull82, 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 arithmetic and aggregate functions. Such languages allow, for example QBE (Query-by-Example, see [Ull82]), 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 [AA93, AU79]. 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. In the remainder of this section we will study how to express this constraint in OCL.

In order to write an OCL constraint expressing that any instance of `Group` is not an element of itself, we need a relation that, as R above, is a subset of the Cartesian product

[3]In the specification document [RAT97] 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.

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

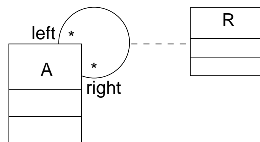

Figure 3: An association class `R` from class `A` to class `A`

the association name `Contains` to an association class as pictured in Fig. 4. Notice that

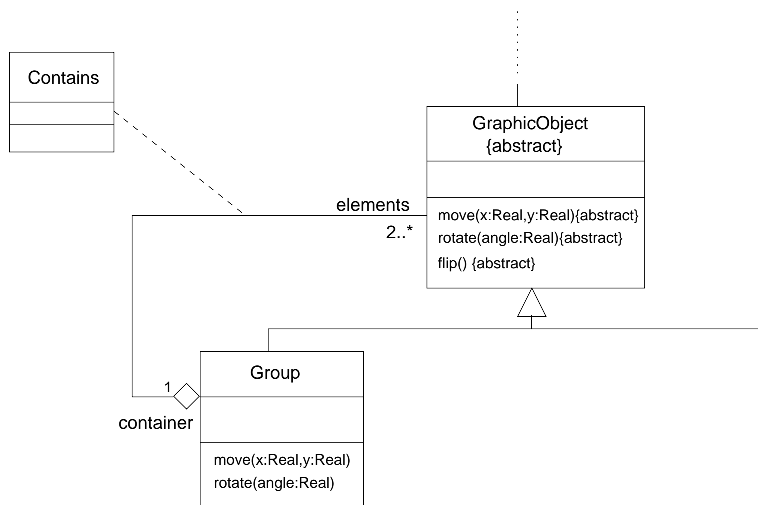

Figure 4: The `Contains` association class

the relation `Contains` might contain 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 contains only the desired pairs, namely an instance of `Group` that is an element of an instance of `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. [Ger93]) 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 4.1 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 (although 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 (or s1 or s2) is 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 are not looking for the existence of two elements in t1 but 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 in principle 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 [Ozk86, 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 [Cod72]) asserted the need for more than complete languages, providing tuple and aggregate functions.

4.3 Turing Completeness

This section addresses the Turing completeness of OCL. It is relatively easy to show that the language is *not* complete, moreover, that the functions expressible in OCL are all primitive recursive. In order to prove so, we show that LOOP-programs can and WHILE-programs cannot be written in OCL (see [Sch95]).

The syntax of LOOP-programs is as follows:

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

The semantics of LOOP-programs is straightforward. 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. 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).

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.
(Obviously the `LOOP` construct becomes superfluous, `LOOP` x `DO` P `END` can be simulated by `WHILE` $y \neq 0$ `DO` $y := y - 1$; P `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(
i : 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 *trans*$(P, 1)$. Initially $\texttt{vals}_1$ stores the value n_i for the variable x_i ($i = 1, \ldots, k$) and 0 otherwise. The return value of *trans*$(P, 1)$ is stored in $\texttt{vals}_1$. The function *trans*(P, n) with $n \in \mathbb{N}$ is a function that manipulates the sequence $\texttt{vals}_n$ and is defined by induction on the structure of P as follows:

- The translation *trans*(P, n) of a program P of the form x_i `<-` $x_j + c$ depends on i and j 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(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
)
```
(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 $i = 0$. [RAT97] 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.)

– 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 similarly 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)
    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)
         ->union(newvals->subSequence(i+2,j)) -- (2)
         ->append(val)
    else newvals->append(val)
    endif
)
```

  (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 $i + 1 = j$.)

- $trans($`LOOP` x_i `DO` P `END`$, n) =$
  `Sequence{1..vals`$_n$`->at(`i`+1)}->iterate(`
  `i : Integer ; -- iterator, 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 natural 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 $f(n, m) = n + m$, computed by the program

```
X0 <- X1 + X2
```

which can be encoded as the following LOOP-program `P`

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

is translated to the OCL expression obtained by successively calculating *trans* as follows:

1. `Sequence{1..1}->iterate(`
   `i : Integer ;`
   `vals1 : Sequence(Integer) = {0,n,m}`
   `|` $trans($`P`$, 1)$
   `)->first`

2.
```
Sequence{1..1}->iterate(
i : 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(
i : 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(
             i : Integer ;
             vals3 : Sequence(Integer) = vals2
             |   trans(P21,3)
             )
        endif
)   )->first
```

4. ```
Sequence{1..1}->iterate(
i : 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(
             i : 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 $x_1, \ldots, x_r$ are used. For any natural numbers $m_1, \ldots, m_r, m'_1, \ldots, m'_r \in \mathbb{N}$, for any natural number $n \in \mathbb{N}$, if the variable $x_i$ changes its value from $m_i$ to $m'_i$ ($i = 1, \ldots, r$) after the execution of $P$, then* `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 1** *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 natural numbers $m_1, \ldots, m_r, m'_1, \ldots, m'_r \in \mathbb{N}$, for any natural number $n \in \mathbb{N}$, if the variable x_i changes its value from m_i to m'_i $(i = 1, \ldots, r)$ after the execution of P', then `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 `vals`$_k = \{l_0, \ldots, l_r\}$*, then* `vals`$_k$`->at(`$j$`+1)` $= l_j$*,* `vals`$_{k+1}$ *is initialized by trans(P,k) with the value of* `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 $\{l_0, \ldots, l_r\}$ to $\{l'_0, \ldots, l'_r\}$.*

  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-computable 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 [RAT97, 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. <u>`Polygon`</u> `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.[4] But, on the other hand, the feature `allInstances` associated with the type `oclType` of types returns a set; see [RAT97, 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 in 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 [Sch95]), OCL allows only for the definition of primitive recursive functions (or totally undefined functions if `Integer.allInstances` is a valid OCL expression).

5 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 brought a language to describe additional constraints about the objects in the model which are not possible to describe in a graphic way. OCL can also be used as a navigational language. Here it has been 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 it has been shown that in OCL the transitive closure of a relation can be computed by coding the Warshall's algorithm. The resulting code is somehow tricky and neither intuitive nor easy to read. It has been demonstrated that OCL can compute only primitive recursive functions and not any recursive function. In other words, OCL is not equivalent to a Turing machine.

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 [GR97]) or CASL (see [BCKB+98]). 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 specification of model properties in a formal way and for the navigation.

References

[AA93] Paolo Atzeni and Valeria De Antonellis. *Relational database theory.* The Benjamin/Cummings Publishing Company, Inc., 1993.

[4] This is also true for the negation given the closed world assumption.

[AU79] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Sixth ACM Symposium on Principles of Programming Languages (POPL79, proceedings)*, pages 110–117, 1979.

[BCKB+98] M. Bidoit, C. Choppy, B. Krieg-Brückner, P. D. Mosses, and F. Voisin. Concrete syntax for CASL - examples of structured specifications, February 1998. Available at `http://www.brics.dk/Projects/CoFI/Documents/CASL/`.

[Cod72] 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.

[Ger93] Judith L. Gersting. *Mathematical Structures for Computer Science.* Computer Science Press, 3rd edition, 1993.

[Gog98] Martin Gogolla. UML for the Impatient. Research Report 3/98, Universität Bremen, 1998.

[GR97] Martin Gogolla and Mark Richters. On Constraints and Queries in UML. In Martin Schader and Axel Korthaus, editors, *Proc. UML'97 Workshop 'The Unified Modeling Language - Technical Aspects and Applications'*, pages 109–121. Physica-Verlag, Heidelberg, 1997.

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

[Oes97] Bernd Oestereich. *Objektorientierte Softwareentwicklung mit der Unified Modeling Language.* Oldenbourg, 1997.

[Ozk86] Esen Ozkarahan. *Database Machines and Database Management.* Prentice Hall, 1986.

[RAT97] RATIONAL Software Corporation. *Object Constraint Language Specification*, September 1997. Version 1.1. Available at `http://www.rational.com/uml/`.

[RG98] Mark Richters and Martin Gogolla. On Formalizing the UML Object Constraint Language OCL. In Tok-Wang Ling, editor, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98).* Springer, Berlin, LNCS, 1998.

[Sch95] Uwe Schöning. *Theoretische Informatik – kurzgefaßt.* Spektrum Akademischer Verlag, 2nd edition, 1995.

[Ull82] Jeffrey D. Ullman. *Principles of Database Systems.* Computer Software Engineering Series. Computer Science Press, 1982.