Electronic Communications of the EASST
Volume 9 (2008)

Proceedings of the Workshop
Ocl4All: Modelling Systems with OCL
at MoDELS 2007

Analyzing Semantic Properties of OCL Operations by Uncovering Interoperational Relationships

Mirco Kuhlmann and Martin Gogolla

17 pages

Analyzing Semantic Properties of OCL Operations by Uncovering Interoperational Relationships

Mirco Kuhlmann[1] **and Martin Gogolla**[2]

[1] mk@informatik.uni-bremen.de [2] gogolla@informatik.uni-bremen.de,
http://db.informatik.uni-bremen.de/
University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany

Abstract: The OCL (Object Constraint Language) as part of the UML (Unified Modeling Language) is a rich language with different collection kinds (sets, multisets, sequences) and a large variety of operations defined thereon. Without negating the strong correlation between both fields we can say that these operations have their origin partly in logic (like the operations forAll and exists) and partly in computer science, in particular database systems (like the operation select). Some of these operations may be expressed in terms of other operations. This paper presents a systematic study of relationships which hold between OCL features like the mentioned operations. Apart from presenting the relationships between operations in a conceptual way, the relationships are described by a formal metamodel allowing systematic and computer supported access to the operation relationships by querying an underlying formal description.

Keywords: UML, OCL, Collections, Equivalences, Semantics, Operations, Benchmark

1 Introduction

The aim of this paper is to compile and discuss the majority of relationships which hold between OCL (Object Constraint Language) [WK03] operations on collections in a single place. Partly, these relationships are mentioned in the OCL standard, but there they are distributed over several sections. However, some more interesting relationships do not show up in the OCL standard, but are presented here. The motivation for our work is to state and clarify the basic semantic relationships between OCL operations on collections. Collections play a central role in OCL, and, for example, the universal and existential quantifiers are understood in OCL as collection operations. Therefore it seems necessary to study whether usual properties from logic hold in OCL as well and how the connection to other OCL operations looks like.

Another reason for our study is that we plan to develop an OCL benchmark as a quality check for an OCL evaluation engine. With the upcoming of MDA (Model Driven Architecture) and MDD (Model Driven Development), OCL becomes more and more popular, partly as a pure expression language, partly as a language for expressing transformations, and more and more OCL engines show up. We consider the relationships discussed in this paper as a starting point for such an OCL benchmark, because they can be understood as pairs of OCL expressions which have to deliver the same evaluation results in every situation. In this context we defined technically oriented equivalences as well. Such a benchmark could be relevant for a variety of OCL tools for which a comparison can be found in [TRF03]. Related work includes the Dresden OCL compiler [HDF00] compiling OCL into Java code, the OCLE system having a similar scope as the USE tool (see Sect. 3) but no automatic snapshot facility [Chi01], the Kent Modeling Framework KMF [AP05] allowing to use OCL for own Java projects, the Octopus [Kla05] OCL 2.0 syntax checker, BoldSoft's tool ModelRun [Bol02], HOL-OCL an interactive proof environment for OCL [Bru07], the KeY system [ABB$^+$00] based on TogetherJ and allowing interactive verification of OCL properties, a recent approach compiling OCL to C# [Arn05], and work translating (a simplified version of) OCL into the theorem prover PVS [KFdB$^+$05]. Few commercial UML tools (e.g., Poseidon, MagicDraw, MaxUML, Together, XMF-Mosaic) provide basic OCL support.

The rest of this paper is structured as follows. Section 2 introduces the relationships in a conceptual way and presents the basic relationships in tabular form. We distinguish between database related, logic and functional programming related collection operations, whereas Subsect. 2.5 focuses on more technical equivalences. Section 3 introduces a metamodel for the relationship implemented in our USE system. This so-called relationship warehouse can be used to formally query the OCL collection operations and the relationships. Section 4 finishes the paper with a short conclusion.

2 Relationships between OCL Collection Operations

In this section we consider the database related operations *select* and *reject*, the logic operations *exists*, *forAll* and *one* as well as the operation *collect* which is related to functional programming. All of them are collection operations defined on all kinds of OCL collections, i.e., sets, bags and sequences, and require an OCL expression for evaluation.

Table 1 presents a categorized list of relationships between the mentioned operations. The left side shows an operation call with an OCL expression e on a collection c. The right side shows an equivalent OCL expression. This implies the interchangeability of both sides within an OCL expression. We neglect the definition of iterators (like `e` in c`->forAll(e|...)`) in almost all operation calls, because they do not play an explicit role except for the alternative expressions of the operation *one*, in which two iterator declarations are required. The indexing of an expression refers to the context of its evaluation (as, e.g., e_x in c`->forAll(x,y|`e_x `and` e_y `implies x = y))`.

Every operation shown on the left side of Tab. 1 features an equivalent *iterate* expression. Table 2 lists all equivalences related to the operation *iterate*. The column 'Operation' presents the considered collection operations and the collections (set, bag and sequence) for which the equivalence holds. The general name *Collection* indicates that the alternative expression can be applied to all three collection types. The variables *et* resp. *ct* represent the type of expression e resp. collection c (*et* for element type, *ct* for collection type).

The order of the relationship kinds in Tab. 1 reflects the structure of the following sections. At first we examine the relationships between the database related and the logic operations separately. Then we analyze the interdisciplinary equivalences proceeding with the operation *collect* and its particular properties.

2.1 Database Related Operations

Both database related operations (*select* and *reject*) are strongly connected. The first operation selects all elements of a source collection which fulfill a boolean expression, the other rejects them. This fact implies the following relationship.

col `->reject(elem:` $elemtype$ `|` $expr_{elem}$ `)` $\equiv$
col `->select(elem:` $elemtype$ `| not (` $expr_{elem}$ `))`

On the left side all elements fulfilling the expression $expr_{elem}$ are removed from the original source collection. The same result is provided by the selection of all objects not fulfilling the boolean expression. The opposite direction considering *select* as source for translation is defined analogously (see Tab. 1).

The *iterate* expression given below represents an alternative expression for a call of *reject*. The accumulator of type *coltype*, i.e., the type of the collection *col*, is initialized with an empty collection[1]. During the iteration, the accumulator is not changed if an element fulfills the expression $expr_{elem}$. Otherwise the current element is included. Analogously the *select* expression is translated by reversing the including condition.

col `->reject(elem:` $elemtype$ `|` $expr_{elem}$ `)` $\equiv$
col `->iterate(elem:` $elemtype$ `; res:` $coltype$ `= oclEmpty(` $coltype$ `) |`
  `if` $expr_{elem}$ `then res else res->including(elem) endif)`

2.2 Logical Theorems formulated in OCL

The logical relationships between the existential and universal quantification are well-known. In OCL the domain of discourse is represented by the collection on which the operation *forAll* resp. *exists* is invoked on. Based on the same domain of discourse (*col*) we can state the following equivalence.

col `->exists(elem:` $elemtype$ `|` $expr_{elem}$ `)` $\equiv$
`not` col `->forAll(elem:` $elemtype$ `| not (` $expr_{elem}$ `))`

There is at least one element fulfilling the expression $expr_{elem}$ if and only if not all elements falsify the expression. The opposite direction is also valid, because all elements fulfill the boolean expression if and only if there is no element which does not fulfill it (see Tab. 1).

Both operations can be expressed with an *iterate* expression. The translation of *exists* is shown below. We need a boolean accumulator initialized with the value *false*. The update function represents a disjunction of the former accumulator value and the evaluation result of expression $expr_{elem}$. Once this expression is evaluated to *true*, the accumulator remains true during the iteration.

[1] The collection expression *oclEmpty*(T) is defined in USE to create an empty collection of type T.

Kind	**Left**	**Right**
DB	c->reject(...|e)	c->select(...|not e)
related	c->select(...|e)	c->reject(...|not e)
Logic	c->exists(...|e)	not c->forAll(...|not e)
	c->forAll(...|e)	not c->exists(...|not e)
Set	c->one(...|e)	c->exists(...|e) and
		c->forAll(x,y|e_x and e_y implies x = y)
Set	c->one(...|e)	c->exists(x|e_x and c->forAll(y|e_y implies x = y))
Inter-	c->exists(...|e)	c->reject(...|e)->size() < c->size()
disci-	c->exists(...|e)	c->select(...|e)->notEmpty()
plinary	c->forAll(...|e)	c->reject(...|e)->isEmpty()
	c->forAll(...|e)	c->select(...|e) = c
	c->one(...|e)	c->reject(...|e)->size() = c->size() - 1
	c->one(...|e)	c->select(...|e)->size() = 1
Collect	c->exists(...|e)	c->collect(...|e)->includes(true)
	c->exists(...|e)	c->collect(...|e)->asSet()->one(e|e)
	c->forAll(...|e)	c->collect(...|e)->excludes(false)
	c->forAll(...|e)	let s = c->collect(...|e)->asSet() in
		c->notEmpty() implies s->one(true) and s->one(e|e)
	c->one(...|e)	c->collect(...|e)->count(true) = 1

Table 1: Relationships between the considered collection operations

Operation	**Iterate Expression**
`collect`	*c*`->collect(...`|*e*`)`
Set,Bag	*c*`->iterate(...;r:Bag(`*et*`) = oclEmpty(Bag(`*et*`))` | `r->including(`*e*`))`
Sequence	*c*`->iterate(...;r:Sequence(`*et*`) = oclEmpty(Sequence(`*et*`))` | `r->append(`*e*`))` (also possible `r->including(` *e* `))`
Sequence	*c*`->iterate(...;r:Sequence(`*et*`) = oclEmpty(Sequence(`*et*`))` | `Sequence{r,Sequence{`*e*`}}->flatten())`
`exists`	*c*`->exists(...`|*e*`)`
Collection	*c*`->iterate(...;r:Boolean = false`|`r or` *e*`)`
`forAll`	*c*`->forAll(...`|*e*`)`
Collection	*c*`->iterate(...;r:Boolean = true`|`r and` *e*`)`
`one`	*c*`->one(...`|*e*`)`
Collection	*c*`->iterate(...;r:Sequence(Boolean) = Sequence{false,false}`| `if r->first() then Sequence{true,false}` `else Sequence{r->last() and` *e*`,` `(r->last() and` *e*`) xor (r->last() or` *e*`)} endif)->last()`
`reject`	*c*`->reject(...`|*e*`)`
Collection	*c*`->iterate(elem;r:`*ct* `= oclEmpty(`*ct*`)` | `if` *e* `then r else r->including(elem) endif)`
`select`	*c*`->select(...`|*e*`)`
Collection	*c*`->iterate(elem;r:`*ct* `= oclEmpty(`*ct*`)` | `if` *e* `then r->including(elem) else r endif)`

Table 2: Translation of the considered operations to iterate expressions

```
$col$ ->exists(elem: $elemtype$ | $expr_{elem}$ ) ≡
$col$ ->iterate( elem: $elemtype$ ; res:Boolean = false |
   res or $expr_{elem}$ )
```

In the case of operation *forAll* the accumulator is initialized with *true* and the update function changes to a conjunction. Once $expr_{elem}$ is false, the accumulator becomes and stays false.

The operation *one* tightens the existence condition. It returns true if the boolean expression evaluates to *true* in context of exactly one element. This fact is reflected in the following relationship, which is only valid for set-valued source collections, because bags and sequences may include equal elements. Beside the existence of at least one element fulfilling $expr_{elem}$ the equality of all fulfilling elements is required.

```
$col$ ->one(elem: $elemtype$ | $expr_{elem}$ ) ≡
$col$ ->exists(elem: $elemtype$ | $expr_{elem}$ ) and
$col$ ->forAll(elem1,elem2: $elemtype$ |
   $expr_{elem_1}$ and $expr_{elem_2}$ implies elem1 = elem2 )
```

The translation of *one* to *iterate* is more complex than the previous definitions, because the accumulator should initially be false, become true if there is one element fulfilling $expr_{elem}$ and become false again if there is another fulfilling element. For this reason we initialize the accumulator with a sequence of two boolean values. The last value represents the desired result value, which can be accessed by appending the sequence operation *last* at the end of the whole OCL expression. It indicates whether exactly one element fulfilling the boolean expression $expr_{elem}$ exists in the source collection *col*. We use the *xor* operation to formulate this requirement. The right side of the expression (`res->last() or` $expr_{elem}$) becomes and remains true as soon as one element fulfills the boolean expression. If another fulfilling element exists in *col*, the left side become true as well (`res->last() and` $expr_{elem}$). This situation implies that the *xor* expression, i.e., the last value of the accumulator sequence, evaluates to *false* and the first value of the accumulator to *true*. The *if* expression assures in this case that the resulting accumulator value *Sequence*{*true*,*false*} does not change anymore.

```
$col$ ->one(elem: $elemtype$ | $expr_{elem}$ ) ≡
$col$ ->iterate(elem: $elemtype$ ;
                res:Sequence(Boolean) = Sequence{false,false} |
   if res->first() then Sequence{true,false}
   else
     Sequence{res->last() and $expr_{elem}$ ,
              (res->last() and $expr_{elem}$ ) xor (res->last() or $expr_{elem}$ )}
   endif )->last()
```

An alternative for the presented solution with two boolean variables would be to use an `iterate` which counts the number of positive elements and finally check whether this counting yields one.

2.3 Interdisciplinary Relationships

In OCL we can join the database related and logic operations. In general, it is possible to translate *forAll*, *exists* and *one* to *select* and *reject*, but not vice versa, because the logic operations do not result in a collection. We can derive a boolean value from a collection, but it is infeasible to construct a collection based on a single boolean value. In the following we will clarify the available equivalences.

col `->exists(elem:` $elemtype$ `|` $expr_{elem}$ `)` $\equiv$
col `->reject(elem:` $elemtype$ `|` $expr_{elem}$ `)->size()` $<$ col `->size()` $\equiv$
col `->select(elem:` $elemtype$ `|` $expr_{elem}$ `)->notEmpty()`

If all elements fulfilling the boolean expression $expr_{elem}$ are rejected and the size of the resulting collection is smaller than the unfiltered collection, the existence of a particular element is guaranteed. We achieve the same result by checking whether the collection which results from selecting all elements fulfilling $expr_{elem}$ is not empty.

col `->forAll(elem:` $elemtype$ `|` $expr_{elem}$ `)` $\equiv$
col `->reject(elem:` $elemtype$ `|` $expr_{elem}$ `)->isEmpty()` $\equiv$
col `->select(elem:` $elemtype$ `|` $expr_{elem}$ `)` $=$ col

All elements of a given collection induce the truth of expression $expr_{elem}$ if and only if all elements are rejected, i.e., the result is empty, or selected, i.e., the result equals the original collection.

For the translation of operation *one* more specific statements are necessary. Exactly one element has to be rejected, i.e., the size of the corresponding result must equal the size of the original collection subtracted by 1. Analogously the operation *select* has to result in a collection including exactly one element.

col `->one(elem:` $elemtype$ `|` $expr_{elem}$ `)` $\equiv$
col `->reject(elem:` $elemtype$ `|` $expr_{elem}$ `)->size()` $=$ col `->size() - 1` $\equiv$
col `->select(elem:` $elemtype$ `|` $expr_{elem}$ `)->size() = 1`

2.4 Features of Collect

Finally we examine the operation *collect* which is related to functional programming. This operation is different from the operations discussed above, because the type of its body expression is not predefined. Any desired ‘function’ may be used to map the elements of the source operation and to collect them in a bag resp. sequence.

At first we consider the translation of *collect* to *iterate*. No other translation of this direction is possible, because *select* and *reject* can only result in collections of the same type as the source collection. In contrast to that *collect* may result in bags resp. sequences including elements of any type. Invoking *collect* on a set or bag results in a bag. Otherwise we get a sequence. This implies a distinction within the alternative *iterate* expression. Below we show the equivalence which holds for sets and bags. The accumulator is initialized with an empty bag und updated by including the evaluation result of $expr_{elem}$.

col `->collect(elem:` $elemtype$ `|` $expr_{elem}$ `)` $\equiv$
col `->iterate(elem:` $elemtype$ `;`
`res:Bag(` $exprtype$ `) = oclEmpty(Bag(` $exprtype$ `)) |`
`res->including(` $expr_{elem}$ `) )`

In case of a sequence the accumulator type changes to a sequence. The corresponding expression is accessible in the `collect` part of Tab. 2. There are two alternative expressions for the operation *collect* which are based on sequence-valued source collections. The lower entry refers to another possibility. Instead of appending or including values, we can construct a sequence as well. Thus we replace the whole body of *iterate* by the equivalent expression `Sequence{res,Sequence{` $expr_{elem}$ `}}->flatten().`

A call of *collect* always results in a collection of the same size as the source collection. The database related operations normally result in smaller collections, because they are used as filters. Therefore we cannot state a reasonable translation from *select* or *reject* to *collect*. On the other hand we can state an equivalence by constraining the source collection to defined values. In this particular case it is possible to map all values which should not be selected resp. be rejected to the undefined value. Finally we exclude the undefined value and retrieve the same result as a corresponding *select* resp. *reject* expression.

The definition of equivalences relating *collect* and the logic operations is unproblematic, because we can reuse their body, i.e., the boolean expression, as function mapping every element to a boolean value. The truth values correspond to the evaluation results of the boolean expression in context of the considered elements. Hence the *collect* expression results in a bag resp. sequence of boolean values.

col `->exists(elem:` $elemtype$ `|` $expr_{elem}$ `)` $\equiv$
col `->collect(elem:` $elemtype$ `|` $expr_{elem}$ `)->includes(true)`

At least one element fulfilling $expr_{elem}$ exists if and only if the bag (resp. sequence) resulting from the call of *collect* includes the value *true*. In case of *forAll* the value *false* must not be an element of the collection (`->excludes(false)`). The operation *count* counts the occurrences of a particular element in a collection. With the aid of this operation it is possible to check whether the value *true* occurs exactly one time (`->count(true) = 1`) what corresponds to a call of *one*. The complete definitions are listed in the Collect part of Tab. 1.

The expressiveness of collect opens up the possibility to define additional relationships between the logic operations.

col `->exists(elem:` $elemtype$ `|` $expr_{elem}$ `)` $\equiv$
col `->collect( elem:` $elemtype$ `|`
$expr_{elem}$ `)->asSet()->one(elem | elem)`

Converting a bag (resp. sequence) of truth values to a set, limits the size to two elements (*Set*{*true*}, *Set*{*false*} or *Set*{*true*,*false*}). The operation *one*, invoked on the resulting set, returns *true* if one element represents the value *true*. In this case one element of the original collection *col* fulfills the boolean expression $expr_{elem}$.

```
col ->forAll(elem: elemtype | expr_elem ) ≡
let s = col ->collect(elem: elemtype | expr_elem )->asSet() in
  col->notEmpty() implies s->one(true) and s->one(elem | elem)
```

The alternative expression for the operation *forAll* is more complex. As aforementioned `s->one(elem|elem)` checks whether the value *true* is an element of the resulting set *s*. Beside this requirement we have to assure that no element of *col* does not fulfill $expr_{elem}$, i.e., the value *false* must not be included in *s*. This implies that *s* is a singleton set. The expression `s->one(true)` is true if and only if *s* possesses this property, because every element in *s* fulfills the expression `true`, even if the element represents the value *false*. The call of *forAll* always results in *true* when the source collection is empty. For this reason we have to define an implication. An empty source collection results in a false premise.

2.5 Illustrative and Technical Relationships

This subsection focuses on more special aspects of the Object Constraint Language. First we inspect elementary relationships concerning the operation *iterate* and conversions of singleton collections. Subsequently we state several technical equivalences in terms of an OCL benchmark, which should assure consistent evaluations.

The first presented relationship emphasizes the change of the result variable in an *iterate* expression. By substituting the body *expr* by an equivalent *let* expression including a local variable *res*, we underline the updated value of the accumulator during the iteration.

```
col ->iterate(elem: elemtype ; res: restype = initexpr | expr ) ≡
col ->iterate(elem: elemtype ; res: restype = initexpr | let res= expr in res)
```

In the following we discuss the possibilities to convert a singleton collection into the value of the solely included element. An obvious way is the use of the operation *any*, which is invoked in the first of three alternative expressions.

```
if col ->size()=1
  then col ->any(true)
  else oclUndefined( elemtype ) endif ≡
if col ->size()=1
  then col ->iterate(elem: elemtype ;
        res: elemtype =oclUndefined( elemtype ) | elem )
  else oclUndefined( elemtype ) endif ≡
col ->iterate(elem: elemtype ;
  res:Sequence(OclAny)=Sequence{oclUndefined( elemtype ),false}|
  if res->at(2)=false
    then  Sequence{ elem ,true}
    else  Sequence{oclUndefined( elemtype ),true} endif)->at(1)
```

The expression *col* `->any(true)` results in an arbitrary element of the collection *col*. Due to the fact that *col* is a singleton, the result value is deterministic. Collections with either no or at least two elements result in the undefined value. Without changing the meaning we can replace the operation *any* by an *iterate* expression, which results in the value of the designated element after one iteration (see the second alternative expression). The third approach does not need an outer *if* expression. Here we differentiate the cases in the body of an *iterate* expression. An accumulator initialized with a sequence of two *OclAny* values is the starting point for the iteration. The first value of the accumulator represents the result value, which can be accessed by calling the operation *at* afterwards. The second value indicates whether at least one iteration has proceeded. We receive the element value if the source collection *col* includes exactly one element. In all other cases the first element of the accumulator represents the undefined value.[2]

In consideration of the fact that we plan to develop an OCL benchmark for OCL evaluation engines, we propose relationships concerning the conversion of sets and bags to sequences, i.e. we inspect the call of the operation *asSequence* and the allowed result values. Originally the OCL specification makes no statement about the order of the elements in the resulting sequences. To standardize the evaluation of such expressions we state equivalences, which restrict the number of possible evaluation results, but do not dictate any particular order.

```
1.  set ->asSequence() ≡
2.  set ->asBag()->asSequence() ≡
3.  set ->iterate(elem: elemtype ;
      res:Sequence( elemtype )=oclEmpty(Sequence( elemtype ))|
      res->including(elem)) ≡
4.  set ->iterate(u: elemtype ;
      res:Tuple(theSet:Set( elemtype ),theSeq:Sequence( elemtype ))=
        Tuple{theSet: set ,theSeq:oclEmpty(Sequence( elemtype ))}|
      let e=res.theSet->any(true) in
        Tuple{theSet:res.theSet->excluding(e),
              theSeq:res.theSeq->including(e)}).theSeq
```

Expression 1 is the source expression. It converts a set into a sequence. The order of the elements in the resulting sequence should not be affected by a former call of the operation *asBag* on the original set (cp. expression 2). Furthermore the elements of *set* must be considered in the same order as they appear in the result of expression 1 during an iteration. We formulate this requirement by an *iterate* expression in expression 3. Equally the order of the values obtained by a repeated call of the operation *any* is restricted. Expression 4 shows a further *iterate* expression, which is misused to change elements of a tuple. The accumulator is initialized by a tuple comprising two elements. The first element represents the original *set* and the second element an empty sequence. During the iteration the operation *any* chooses arbitrary elements from the set. These elements are sequentially included in the sequence and excluded from the set, respectively. Afterwards we receive the resulting sequence, which should conform to the result of expression 1. The expressions 1, 3 and 4 are equivalent for bag-valued source collections as well.

[2] The last alternative expression only works for flat collections, i.e., when *elemtype* is not a collection. In case of nested collections the sequence has to be substituted by a tuple.

3 Relationship Warehouse

We have implemented a relationship warehouse with the UML-based Specification Environment (USE) [RG01, GBR05]. A USE specification defines a UML class diagram modeling the warehouse, and an object diagram provides the information about OCL standard operations (including the ones considered in Sect. 2) and their relationships. The warehouse was modeled in such a way that precise queries can be defined on it. The purpose of this implementation is to get desired information about relationships and the involved operations quickly and in a comfortable way.

3.1 Overview of the UML Model: The Class Diagram

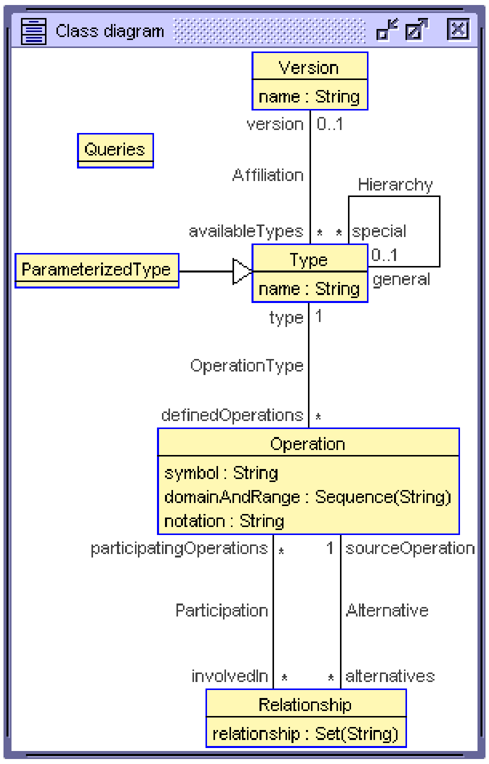

Figure 1: Class Diagram of the Relationship Warehouse

Figure 1 shows the UML class diagram illustrating the concept of the relationship warehouse. The information about the OCL operations and their relationships is synthesized by instantiating that model (see Sect. 3.2).

The class *Version* referring to an OCL specification, e.g., ‘OCL 2.0 (06-05-01)’ [Obj06] or ‘Mark Richters (USE)’ [Ric02] is the starting point for the relationship warehouse. Every OCL specification defines a number of types containing all possible OCL expressions. The association *Hierarchy* is defined to realize subtyping, e.g., *Real* is a subtype of *OclAny*. Hereby, several type hierarchy trees can be created.

The roots in the type hierarchy are connected to exactly one version. OCL collections are considered as parameterized types.

A type comprises a set of OCL standard operations characterized by their symbol, their domain (the parameter types) and range (the return type), and their notation. The attribute *domainAndRange* collects the formal parameters and the return type in a sequence. The last element in the sequence always refers to the return type.

When there is an alternative expression resulting in the same value as a specific operation call, this alternative can be added as a *Relationship* object linked to the operation. A *Relationship* object stores an equivalence like the ones presented in Tab. 1 and Tab. 2 and is connected to all Operations involved in it. We can navigate from a source operation to all of its alternative expressions towards the participating operations. This makes powerful queries on the warehouse possible.

The attribute *relationship* is defined as set of strings, because different equivalences with the same set of participating operations may exist for a source operation (e.g., `x + y == x - (-y)` and `x + y == y - (-x)` for the addition of integer values additionally involving the subtraction and the operation for changing the sign). The object diagram in the next section will show another, more involved example.

The class *Queries* defines queries in order to retrieve information about operations and relationships with specific properties.

3.2 Storing the Relationships: The Object Diagram

The relationship warehouse is implemented by a large object diagram. Due to the size of the object diagram (we have about 150 operations), it does not make much sense to look at the entire object diagram. Nevertheless we can inspect a part of the object diagram to clarify the possibilities of this realization. Figure 2 presents a small part of the warehouse consisting of a single relationship.

The focus of Fig. 2 is the operation *one* indicated by the symbol `one`. It is defined for the type `Collection` in context of the OCL version `Mark Richters (USE)` [Ric02]. The first element of the sequence available in attribute *domainAndRange* reflects the fact that *one* has to be invoked on a collection (Set, Bag or Seq(uence)). Beside the collection, a boolean expression is required for calling the operation (second element). The attribute *notation* specifies the concrete syntax for the operation *one*. A successful call results in a boolean value, which is declared by the last element of the mentioned sequence.

A single relationship, i.e., an alternative expression, is stored for the source operation *one*. It represents an equivalence listed in the interdisciplinary part of Tab. 1. Three operations are involved in the alternative expression, i.e., the right side of the equivalence: the two collection operations *size* and *select* and the equality defined on integer values. Mark Richters (as well as Standard OCL) specifies a subtype relation between *Integer* an *Real*.

The original object diagram includes all relationships presented in Tab. 1 resp. Tab. 2. Altogether the current warehouse comprises 148 OCL standard operations and 108 equivalences. It can be enriched as needed.

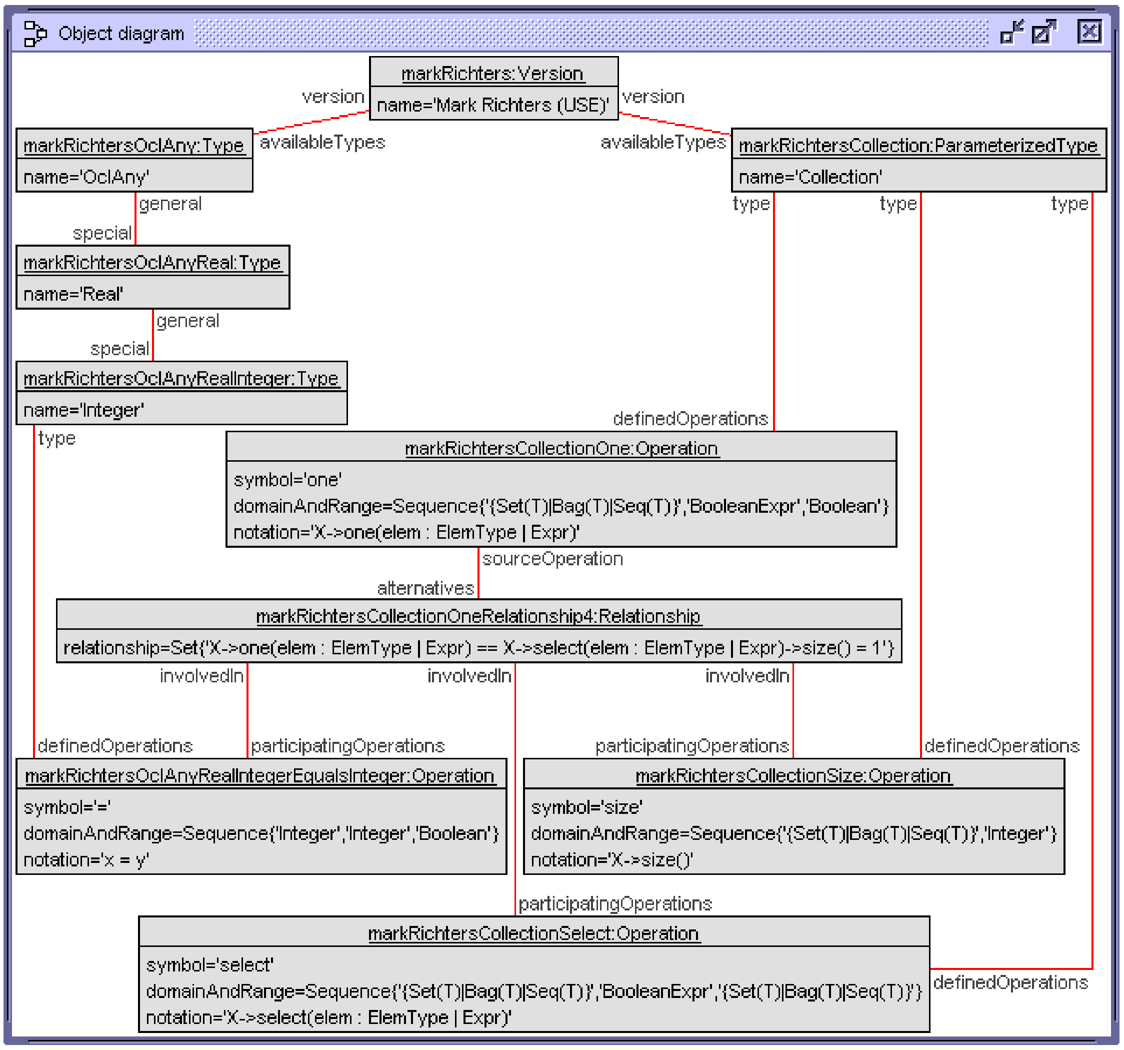
Object diagram
markRichters:Version
name='Mark Richters (USE)'
version
version
availableTypes
availableTypes
markRichtersOclAny:Type
name='OclAny'
markRichtersCollection:ParameterizedType
name='Collection'
general
special
type
type
type
markRichtersOclAnyReal:Type
name='Real'
general
special
markRichtersOclAnyRealInteger:Type
name='Integer'
type
definedOperations
markRichtersCollectionOne:Operation
symbol='one'
domainAndRange=Sequence{'{Set(T)|Bag(T)|Seq(T)}','BooleanExpr','Boolean'}
notation='X->one(elem : ElemType | Expr)'
sourceOperation
alternatives
markRichtersCollectionOneRelationship4:Relationship
relationship=Set{'X->one(elem : ElemType | Expr) == X->select(elem : ElemType | Expr)->size() = 1'}
involvedIn
involvedIn
involvedIn
definedOperations
participatingOperations
participatingOperations
definedOperations
markRichtersOclAnyRealIntegerEqualsInteger:Operation
symbol='='
domainAndRange=Sequence{'Integer','Integer','Boolean'}
notation='x = y'
markRichtersCollectionSize:Operation
symbol='size'
domainAndRange=Sequence{'{Set(T)|Bag(T)|Seq(T)}','Integer'}
notation='X->size()'
participatingOperations
markRichtersCollectionSelect:Operation
symbol='select'
domainAndRange=Sequence{'{Set(T)|Bag(T)|Seq(T)}','BooleanExpr','{Set(T)|Bag(T)|Seq(T)}'}
notation='X->select(elem : ElemType | Expr)'
definedOperations

Figure 2: Part of the Relationship Warehouse

3.3 Querying the Warehouse

The entire object diagram is too large to be visually inspected. Thus,we have to query the relationship warehouse using OCL operations, especially navigation expressions. In the following we exemplify the access of desired information.

Every available Operation is identified by its symbol, the name of its version and its type. We defined the OCL query operation *getOperationBySymbol* to obtain the corresponding *Operation* object. Strictly speaking, there are operations with the same symbol, version and type, but with different formal parameters, e.g., there are two operations with symbol − defined on type *Integer* namely $-(i : Integer) : Integer$ and $-(r : Real) : Real$. In this case we can select the relevant operation by filtering their domain resp. range.

To obtain the *Operation* object, e.g., *one* which was discussed in Sect. 3.2, we need the mentioned arguments:

```
queries.getOperationBySymbol(
  'one', 'Mark Richters (USE)', 'Collection')
```

USE evaluates the OCL expression and displays the resulting value and its type:

```
Set{@markRichtersCollectionOne} : Set(Operation)
```

Starting from this operation we can navigate to the alternative expressions and collect all equivalences:

```
queries.getOperationBySymbol('one', 'Mark Richters (USE)',
  'Collection').alternatives.relationship
```

The resulting bag includes all available equivalences (cp. Tab. 1):

```
Bag{
  'X->one(elem : ElemType | Expr) ==
   X->collect(elem : ElemType | Expr)->count(true) = 1',

  'X->one(elem : ElemType | Expr) ==
   X->exists(elem : ElemType | Expr) and
     X->forAll(elem1, elem2 : ElemType |
       Expr_elem1 and Expr_elem2 implies elem1 = elem2)',

  'X->one(elem : ElemType | Expr) ==
   X->exists(elem1 : ElemType | Expr_elem1 and
     X->forAll(elem2 : ElemType |
       Expr_elem2 implies elem1 = elem2))',

  'X->one(elem : ElemType | Expr) ==
   X->iterate(elem : ElemType;
              res : Sequence(Boolean) = Sequence{false,false} |
     if res->first() then Sequence{true,false}
     else Sequence{res->last() and Expr,
                   res->last() or Expr} endif)->last()',

  'X->one(elem : ElemType | Expr) ==
   X->reject(elem : ElemType | Expr)->size() = X->size() - 1',

  'X->one(elem : ElemType | Expr) ==
   X->select(elem : ElemType | Expr)->size() = 1'} : Bag(String)
```

Alternatively we can use the query operation *getRelationshipBySymbol*, which returns all relationships for an operation in context of a type and version:

```
queries.getRelationshipBySymbol(
  'one', 'Mark Richters (USE)', 'Collection')
```

For getting a more specific result it is possible to formulate more restrictive queries. We can select the equivalences depending on the participating operations. The following expression picks out the alternative expression defined for the source operation *one* in which the operation *select* is involved.

```
queries.getOperationBySymbol(
'one', 'Mark Richters (USE)', 'Collection').alternatives->
  select(participatingOperations.symbol->
    includes('select')).relationship
```

The result of the following expression represents the relationship considered in Fig. 2:

```
Bag{
  'X->one(elem : ElemType | Expr) ==
   X->select(elem : ElemType | Expr)->size() = 1'} : Bag(String)
```

The corresponding query operation *getRelationshipBySymbolAndParticipating* simplifies the selecting expression. Its last argument must be a set of operations, which should be involved in the equivalences.

```
queries.getRelationshipBySymbolAndParticipating(
  'one', 'Mark Richters (USE)', 'Collection', Set{'select'})
```

All expressions explained above need a source operation as starting point. However there are also meaningful queries which do not need any symbols of source operations. For example the available query operation *getRelationshipByParticipating* works backwards. Given a set of operation symbols the query operation returns all relationships in which all stated operations are involved:

```
queries.getRelationshipByParticipating(
  'Mark Richters (USE)', Set{'one'})
```

The operation call results in two relationships. The left side of the equivalences show that *one* is involved in alternative expressions for *exists* and *forAll*:

```
Set{
  'X->exists(elem : ElemType | Expr) ==
   X->collect(elem : ElemType | Expr)->
     asSet()->one(elem | elem)',

  'X->forAll(elem : ElemType | Expr) ==
   let s = X->collect(elem : ElemType | Expr)->asSet() in
   X->notEmpty() implies
     s->one(true) and s->one(elem | elem)'} : Set(String)
```

There are many other possibilities to filter the results, e.g., by considering the formal parameters and return types or the number of operations participating in an alternative expression.

4 Conclusion

We have discussed basic semantic properties of OCL operations on collections. These OCL collection operations play a central role and their relationships should be clearly expressed, which includes the handling of the undefined value. Thereby it is possible to minimize the scope of interpretation caused by informal definitions in the OCL standard. We plan to extend this work and to develop an OCL benchmark which could be used to check the quality of an OCL evaluation engine. With the upcoming of more and more OCL evaluators in the context of MDA and MDD, such a quality assurance mechanism seems necessary to us.

The relationship warehouse presented in the second part of this paper can be extended in different ways. When other OCL versions are added, a slightly modified specification allows for comparing operations in context of different versions. On the other hand we can multiply the possibilities of inspecting the relationships by substituting the strings representing the equivalences for more sophisticated constructs, i.e., instances of the OCL metamodel.

Bibliography

[ABB$^+$00] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I.P. de Guzmán, G. Brewka, and L. M. Pereira, editors, *Proc. 8th European Workshop Logics in AI (JELIA'2000)*, LNCS 1919, pages 21–36. Springer, 2000.

[AP05] Dave Akehurst and Octavian Patrascoiu. The Kent Modeling Framework (KMF). `http://www.cs.kent.ac.uk/projects/ocl`, University of Kent, 2005.

[Arn05] Dave Arnold. OCL/C# Compiler. `www.ewebsimplex.net/csocl/`, ewebsimplex, 2005.

[Bol02] Boldsoft. Boldsoft OCL Tool Model Run. `www.boldsoft.com`, Boldsoft, Stockholm, 2002.

[Bru07] Achim D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. Ph.d. thesis, ETH Zurich, March 2007. ETH Dissertation No. 17097.

[Chi01] D. Chiorean. Using OCL Beyond Specifications. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Proc. UML'2001 Workshop Rigorous Development*, pages 57–68. LNI, GI, Bonn, 2001.

[GBR05] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005.

[HDF00] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, pages 278–293. Springer, LNCS 1939, 2000.

[KFdB$^+$05] M. Kyas, H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons, and H. Kugler. Formalizing UML models and OCL constraints in PVS. *Electr. Notes Theor. Comput. Sci.*, 115:39–47, 2005.

[Kla05] KlasseObjecten. The Klasse Objecten OCL Checker Octopus. `www.klasse.nl/english/research/octopus-intro.html`, Klasse Objecten, 2005.

[Obj06] Object Management Group, Inc. *Object Constraint Language - OMG Available Specification, Version 2.0*, Mai 2006. http://www.omg.org/cgi-bin/doc?ptc/06-05-01.

[RG01] Mark Richters and Martin Gogolla. OCL - Syntax, Semantics and Tools. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, pages 43–69. Springer, Berlin, LNCS 2263, 2001.

[Ric02] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*, volume 14 of *BISS Monographs*. Logos, Berlin, 2002.

[TRF03] A. Toval, V. Requena, and J.L. Fernandez. Emerging OCL Tools. *Software and Systems Modeling*, 2(4):248–261, 2003.

[WK03] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 2003. 2nd Edition.