Analyzing Semantic Properties of OCL Operations by Uncovering Interoperational Relationships

Mirco Kuhlmann and Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
{mk|gogolla}@informatik.uni-bremen.de

Abstract. The OCL (Object Constraint Language) as part of the UML (Unified Modeling Language) is a rich language with different collection kinds (sets, multi-sets, 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.

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. The relationships discussed in this paper can be understood as a starting point for such an OCL benchmark, because the relationships can be understood as pairs of OCL expressions which have to deliver the same evaluation results.

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], 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. 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
Table 1. Relationships between the considered collection operations

<table>
<thead>
<tr>
<th>Kind</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>c-&gt;reject(...</td>
<td>e)</td>
</tr>
<tr>
<td>related</td>
<td>c-&gt;select(...</td>
<td>e)</td>
</tr>
<tr>
<td>Logic</td>
<td>c-&gt;exists(...</td>
<td>e)</td>
</tr>
<tr>
<td></td>
<td>c-&gt;forall(...</td>
<td>e)</td>
</tr>
<tr>
<td>Set</td>
<td>c-&gt;one(...</td>
<td>e)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c-&gt;forall(x,y</td>
</tr>
<tr>
<td>Set</td>
<td>c-&gt;one(...</td>
<td>e)</td>
</tr>
<tr>
<td>Inter-disciplinary</td>
<td>c-&gt;exists(...</td>
<td>e)</td>
</tr>
<tr>
<td></td>
<td>c-&gt;exists(...</td>
<td>e)</td>
</tr>
<tr>
<td></td>
<td>c-&gt;forall(...</td>
<td>e)</td>
</tr>
<tr>
<td></td>
<td>c-&gt;forall(...</td>
<td>e)</td>
</tr>
<tr>
<td></td>
<td>c-&gt;one(...</td>
<td>e)</td>
</tr>
<tr>
<td></td>
<td>c-&gt;one(...</td>
<td>e)</td>
</tr>
<tr>
<td>Collect</td>
<td>c-&gt;exists(...</td>
<td>e)</td>
</tr>
<tr>
<td></td>
<td>c-&gt;exists(...</td>
<td>e)</td>
</tr>
<tr>
<td></td>
<td>c-&gt;forall(...</td>
<td>e)</td>
</tr>
<tr>
<td></td>
<td>c-&gt;forall(...</td>
<td>e)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c-&gt;notEmpty() implies s-&gt;one(true) and s-&gt;one(e</td>
</tr>
<tr>
<td></td>
<td>c-&gt;one(...</td>
<td>e)</td>
</tr>
</tbody>
</table>

Table 2. Translation of the considered operations to iterate expressions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Iterate Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect</td>
<td>c-&gt;collect(...</td>
</tr>
<tr>
<td></td>
<td>c-&gt;iterate(...</td>
</tr>
<tr>
<td>Set, Bag</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c-&gt;iterate(...</td>
</tr>
<tr>
<td>Sequence</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c-&gt;iterate(...</td>
</tr>
<tr>
<td>exists</td>
<td>c-&gt;exists(...</td>
</tr>
<tr>
<td>Collection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c-&gt;iterate(...</td>
</tr>
<tr>
<td>forAll</td>
<td>c-&gt;forall(...</td>
</tr>
<tr>
<td>Collection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c-&gt;iterate(...</td>
</tr>
<tr>
<td>one</td>
<td>c-&gt;one(...</td>
</tr>
<tr>
<td>Collection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c-&gt;iterate(...</td>
</tr>
<tr>
<td></td>
<td>if res-&gt;first() then Sequence(true,false) else Sequence(res-&gt;last() and e, res-&gt;last() or e) endif)-&gt;last()</td>
</tr>
<tr>
<td>reject</td>
<td>c-&gt;reject(...</td>
</tr>
<tr>
<td>Collection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c-&gt;iterate(elem, r: ct = oclEmpty(ct)</td>
</tr>
<tr>
<td></td>
<td>if e then res else res-&gt;including(elem) endif)</td>
</tr>
<tr>
<td>select</td>
<td>c-&gt;select(...</td>
</tr>
<tr>
<td>Collection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c-&gt;iterate(elem, r: ct = oclEmpty(ct)</td>
</tr>
<tr>
<td></td>
<td>if e then res-&gt;including(elem) else res endif)</td>
</tr>
</tbody>
</table>
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.

\[
col \rightarrow \text{reject}(\text{elem} : \text{elemtype} | \text{expr}_{\text{elem}}) \equiv \\
\text{col} \rightarrow \text{select}(\text{elem} : \text{elemtype} | \text{not}(\text{expr}_{\text{elem}}))
\]

On the left side all elements fulfilling the expression \(\text{expr}_{\text{elem}}\) 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 \(\text{coltype}\), i.e., the type of the collection \(\text{col}\), is initialized with an empty collection\(^1\). During the iteration, the accumulator is not changed if an element fulfills the expression \(\text{expr}_{\text{elem}}\). Otherwise the current element is included. Analogously the select expression is translated by reversing the including condition.

\[
col \rightarrow \text{reject}(\text{elem} : \text{elemtype} | \text{expr}_{\text{elem}}) \equiv \\
\text{col} \rightarrow \text{iterate}(\text{elem} : \text{elemtype} ; \text{res} : \text{coltype} = \text{oclEmpty}(\text{coltype}) | \\
\quad \text{if } \text{expr}_{\text{elem}} \text{ then } \text{res} \text{ else } \text{res} \rightarrow \text{including}(\text{elem}) \text{ endif})
\]

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 (\(\text{col}\)) we can state the following equivalence.

\[
col \rightarrow \text{exists}(\text{elem} : \text{elemtype} | \text{expr}_{\text{elem}}) \equiv \\
\text{not} \col \rightarrow \text{forAll}(\text{elem} : \text{elemtype} | \text{not}(\text{expr}_{\text{elem}}))
\]

\(^1\) The collection expression \(\text{oclEmpty}(T)\) is defined in USE to create an empty collection of type \(T\).
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.

$$ col \rightarrow exists(elem: elemtype \mid expr_{elem}) \equiv $$
$$ col \rightarrow iterate(elem: elemtype \mid res:Boolean = false \mid \neg expr_{elem}) $$

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.

$$ col \rightarrow one(elem: elemtype \mid expr_{elem}) \equiv $$
$$ col \rightarrow exists(elem: elemtype \mid expr_{elem}) \land $$
$$ col \rightarrow forAll(elem1,elem2: elemtype \mid expr_{elem1} \land expr_{elem2} \implies elem1 = elem2) $$

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. Its last value indicates the finding of an element fulfilling the boolean expression ($res->last()$ or $expr_{elem}$). The first value turns true if another element fulfills it, i.e., the last value of the accumulator is already true ($res->last()$ and $expr_{elem}$). Finally the last value of the accumulator is true if and only if exactly one element fulfills the expression. We access this value by appending the sequence operation last at the end.

$$ col \rightarrow one(elem: elemtype \mid expr_{elem}) \equiv $$
$$ col \rightarrow iterate(elem: elemtype ; $$
$$ \quad res:Sequence(Boolean) = Sequence{false,false} \mid $$
$$ \quad \text{if res->first()} \text{ then Sequence{true,false} } \quad \text{else} \text{ } $$
$$ \quad Sequence{res->last() \land expr_{elem} \land \neg res->last() \lor expr_{elem}} \quad \text{endif} \rightarrow last() $$
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.

\[
\begin{align*}
\text{col} \rightarrow \text{exists}(\text{elem}: \text{elemtype} \mid \text{expr}_\text{elem}) & \equiv \\
\text{col} \rightarrow \text{reject}(\text{elem}: \text{elemtype} \mid \text{expr}_\text{elem}) \rightarrow \text{size()} < \text{col} \rightarrow \text{size()} & \equiv \\
\text{col} \rightarrow \text{select}(\text{elem}: \text{elemtype} \mid \text{expr}_\text{elem}) \rightarrow \text{notEmpty()} & \equiv \\
\end{align*}
\]

If all elements fulfilling the boolean expression \( \text{expr}_\text{elem} \) 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 \( \text{expr}_\text{elem} \) is not empty.

\[
\begin{align*}
\text{col} \rightarrow \text{forAll}(\text{elem}: \text{elemtype} \mid \text{expr}_\text{elem}) & \equiv \\
\text{col} \rightarrow \text{reject}(\text{elem}: \text{elemtype} \mid \text{expr}_\text{elem}) \rightarrow \text{isEmpty()} & \equiv \\
\text{col} \rightarrow \text{select}(\text{elem}: \text{elemtype} \mid \text{expr}_\text{elem}) & = \text{col} \\
\end{align*}
\]

All elements of a given collection induce the truth of expression \( \text{expr}_\text{elem} \) 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.

\[
\begin{align*}
\text{col} \rightarrow \text{one}(\text{elem}: \text{elemtype} \mid \text{expr}_\text{elem}) & \equiv \\
\text{col} \rightarrow \text{reject}(\text{elem}: \text{elemtype} \mid \text{expr}_\text{elem}) \rightarrow \text{size()} = \text{col} \rightarrow \text{size()} - 1 & \equiv \\
\text{col} \rightarrow \text{select}(\text{elem}: \text{elemtype} \mid \text{expr}_\text{elem}) \rightarrow \text{size()} & = 1 \\
\end{align*}
\]

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 and updated by including the evaluation result of expr\_elem.

\[
\text{col} \rightarrow \text{collect}\left(\text{elem: elemtype | expr\_elem}\right) \equiv \text{col} \rightarrow \text{iterate}\left(\text{elem: elemtype} ; \text{res:Bag( exprtype ) = oclEmpty(\text{exprtype} ) |} \text{res}\rightarrow \text{including( expr\_elem )}\right)
\]

In case of a sequence the accumulator type changes to a sequence and the operation including to append. 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 values, we can construct a sequence as well. Thus we replace the whole body of iterate by the equivalent expression \text{Sequence}\{\text{res,Sequence( expr\_elem }\}\} \rightarrow \text{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.

\[
\text{col} \rightarrow \text{exists}\left(\text{elem: elemtype | expr\_elem}\right) \equiv \text{col} \rightarrow \text{collect}\left(\text{elem: elemtype | expr\_elem}\right) \rightarrow \text{includes(true)}
\]

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

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

### 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 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 \( \text{Version} \) 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 \( \text{Hierarchy} \) is defined to realize subtyping, e.g., \( \text{Real} \) is a subtype of \( \text{OclAny} \). 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.

**Fig. 2.** Part of the Relationship Warehouse

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(quence)). Beside the collection, a boolean expression is required for
calling the operation (second element). The attribute notation specifies the con-
crete 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 opera-
tions and 108 equivalences. It can be enriched as needed.

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\rightarrow one(elem:ElemType|Expr) ==
X\rightarrow collect(elem:ElemType|Expr)\rightarrow count(true) = 1’,

‘X\rightarrow one(elem:ElemType|Expr) ==
X\rightarrow exists(elem:ElemType|Expr) and
X\rightarrow forall(elem1,elem2:ElemType|
Expr\_elem1 and Expr\_elem2 implies elem1 = elem2)’,

‘X\rightarrow one(elem:ElemType|Expr) ==
X\rightarrow exists(elem1:ElemType|Expr\_elem1 and
X\rightarrow forall(elem2:ElemType|
Expr\_elem2 implies elem1 = elem2))’,

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

‘X\rightarrow one(elem:ElemType|Expr) ==
X\rightarrow reject(elem:ElemType|Expr)\rightarrow size() = X\rightarrow size() - 1’,

‘X\rightarrow one(elem:ElemType|Expr) ==
X\rightarrow select(elem:ElemType|Expr)\rightarrow 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\rightarrow one(elem:ElemType|Expr) ==
X\rightarrow select(elem:ElemType|Expr)\rightarrow 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.

```java
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:

```java
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`:

```java
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 mechanisms 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.

References


