Relations in OCL

D.H.Akehurst

University of Kent
D.H.Akehurst@kent.ac.uk

Abstract. OCL is proposed as a query language within the QVT framework. The main QVT submission bases the specification of transformations on the concept of relations. Relations are not first class entities within the OCL. By extending OCL with the concept of Relations it can better serve the needs of the QVT framework. In particular this enables OCL to be used as a semantic interpretation of a QVT transformation language and may even facilitate the use of OCL as a transformation specification language.

1 Introduction

The QVT Merge submission [4] seems to agree that the specification of model transformations be based (at some level) on the concept of Relations. There is also some agreement that OCL [3] be used as an expression language within such specifications.

OCL is based partly on the mathematical foundation of Set Theory; of which Relation Theory is considered to be a part. Indeed if we look at the specification language Z [6], which is also based on Set Theory, binary relations are considered to be a major part of its use, as are Schemas which define n-ary relations. This paper proposes the addition of relations as a type within the OCL. The technical details are based partially on the use of relations within Z but adapted to be inline with the conventions of OCL and take also into consideration the approach to relations found in Databases and Relation Algebra.

OCL already has the concept of n-tuples with named parts; it follows that a relation in OCL can be a set of n-tuples that conform to a common type. Seeing as there can be frequent use of binary relations this paper proposes to also incorporate the notion of a binary relation as a special type of n-ary relation (Figure 1).

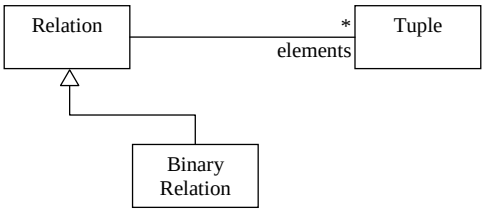

Figure 1 Relations

The work described in this paper is not complete; rather, it is a proposal with some initial ideas of how the incorporation of relations into OCL could be achieved.

Section 2 gives a brief overview of the relational operators found in Set Theory, Z and Relational Algebra. Section 3 provides a proposal for incorporating n-ary

relations into OCL and Section 4 extends the proposal to also incorporate binary relations. Section 5 looks at interpreting the semantics of the relation specification language defined in the QVT-Merge submission using the OCL relations defined in this paper.

2 Relation Theory

A Binary Relation is defined to be a set of pairs, i.e. a subset of a Cartesian product. This can of course be extended to n-ary Relations being a set of n-Tuples, formed from the Cartesian product of n sets. It is at this point that relation theory splits into two strands; Binary relation theory which deals with operations on and properties of binary relations; and Relational Algebra [2] handling functions and properties of n-ary Relations used within computing under the domain of Databases and SQL.

2.1 Binary Relations (Set Theory)

If S and T are sets then the type of a binary relation R is defined as follows:

$R : \mathbb{P}(S \times T)$, or R is a Set $\{ (s,t) \mid s \in S \wedge t \in T \}$

I.e. a binary relation is a set of pairs of elements, the first part of each pair is drawn from the set S and the second from part T. The definition of a relation is either given as an explicit set of pairs, or by using a predicate to define a subset of the full product of sets S and T.

Functions **dom** and **ran** are defined to give the sets of elements that are related. **dom** gives the domain, i.e. those elements from S that are related to something in T and **ran** gives the range, i.e. those elements from T that are related to something in S.

Additional useful functions on relations are:

- domain/range restriction
- domain/range subtraction (anti-restriction)
- relational image
- relational inverse
- relational composition
- reflexive closure
- symmetric closure
- n compositions
- transitive closure
- reflexive transitive closure
- relational overriding

Plus any of the set operations, which when applied to a relation, are interpreted as being applied to the set of elements.

There are also a number of properties of relations that can be defined:

- homogeneous
- heterogeneous
- reflexive
- symmetric
- anti symmetric
- asymmetric
- transitive
- equivalence

If we then introduce the notion of a function as a particular type of relation or the property ‘functional’ on relations, we get another set of properties (which can also be seen as properties of a relation):

- functional / injective
- partial

- total / surjective
- bijective (injective and surjective)

2.2 Z

The language Z uses set theory and binary relations as its foundation; however it also offers a notion of n-ary relations in the form of a Schema. This gives names to each part of an n-tuple, and the Schema defines a type for all such n-tuples.

As with binary relations, there are a number of operations that can be performed on Schemas as follows:

- conjunction
- disjunction
- negation
- quantification
- hiding
- composition

Within the Z schema language there is the notion of renaming which enables a specification to rename one (or more) of the parts in a schema to a new name.

2.3 Relational Algebra

A relational algebra (RA) is a set of operators that take relations as their operands and return a relation as their result. There are eight main operators (defined in [2]), called: *Select*; *Project*; *Intersect*; *Difference*; *Join*; *Divide*; *Union*; and *Product*. However, these eight are not independent and three (Intersect, Join and Divide) can be defined in terms of the other five, which are called primitive operators (see [2]). Consequently, in order for a query language to be considered fully expressive, it must support as a minimum, the primitive operators [1]: Union, Difference; Product; Project; and Select. A definition of a relation and these five operators (taken from [2]) is given below:

Relation:	Is a mathematical term for table, which is a set of tuples; a relation with arity k is a set of k-tuples.
Union:	Returns a relation containing all tuples that appear in either or both of two specified relations.
Difference:	Returns a relation containing all tuples that appear in the first and not the second of two specified relations.
Product:	Returns a relation containing all possible tuples that are a combination of two tuples, one from each of two specified relations.
Project:	Returns a relation containing all (sub) tuples that remain in a specified relation after specified attributes have been removed.
Select:	Returns a relation containing all tuples from a specified relation that satisfy a specified condition.

The interested reader is referred to [2] and [5] for further details on relational algebras.

3 N-ary Relations

When handling n-ary relations, elements of the relation can either be n-tuples with ordered parts, or n-tuples with each part named. In relational algebra in order to distinguish between the multiple parts of an n-tuple, each part is named, and thus each dimension of the relation is also named; we suggest a similar approach be taken for incorporating n-ary relations into OCL.

There are two aspects to consider when introducing the new concept of relations into OCL:

1. How do we specify or construct a relation?
2. What operations can be performed on a relation?

The following sub-sections address these issues.

3.1 Constructing a relation

A literal set in OCL is defined using the keyword 'Set' followed by the elements to be contained in the set:

```
let s1 = Set{ 1,2,3 }
```

Thus in OCL we could construct a literal relation in a similar manner as follows:

```
let r1 = Relation{ Tuple{x=1,y=1,z=1},
                   Tuple{x=2,y=4,z=8},
                   Tuple{x=3,y=9,z=27} }
```

A second way in which we could construct a relation is to convert a Set of tuples into a relation. i.e.

```
let r2 = Set{ Tuple{x=1,y=1,z=1},
              Tuple{x=2,y=4,z=8},
              Tuple{x=3,y=9,z=27} }->asRelation()
```

The result of such an operation on a set would of course be undefined if the elements of the set were not all tuples of a single conforming type. An alternative method of converting a set into a relation would be to construct an identity relation from the Set, i.e.:

```
Set{ 1, 2, 3 }->identityRelation() =
     Relation{ Tuple{first=1,second=1},
               Tuple{first=2,second=2},
               Tuple{first=3,second=3} }
```

However, as we have n-ary relations it would be nice to extend the identity operation to facilitate construction of a relation between n parts, e.g. using an iterator:

```
Set{1,2,3}->identityRelation( x,y,z | true ) =
     Relation{ Tuple{x=1,y=1,z=1},
               Tuple{x=2,y=2,z=2},
               Tuple{x=3,y=3,z=3} }
```

Which enables us to construct relations of n parts, to name each part, and at the same time constrain the elements of the relation using the condition part of the iterator, e.g.:

```
Set{1..5}->identityRelation( x,y,z | x < 4 ) =
     Relation{ Tuple{x=1,y=1,z=1},
               Tuple{x=2,y=2,z=2},
               Tuple{x=3,y=3,z=3} }
```

3.2 Relations Types

As well as constructing relations from explicit literal elements it is often useful to define the relation using a predicate expression rather than literal values. In this case it

is necessary to give the explicit types and names of the parts of the relations along with an expression to define members of the relation, e.g.:

```
Relation{ x:Integer, y:Integer, z:Integer |
              x*x = y and x*x*x = z }
```

If we were to try and define the set of elements for a relation defined like this we would end up with an infinite set as there are an infinite number of Integers.

Initial work made a distinction between Finite and Infinite Relations; however, since submitting the paper for review, further work has developed a more satisfactory approach. (This approach also provides a solution to the problem originally discussed in section 5.1 of this paper, requiring a mechanism for stating that pairs of elements from two containers of objects are related, see section 5.1.)

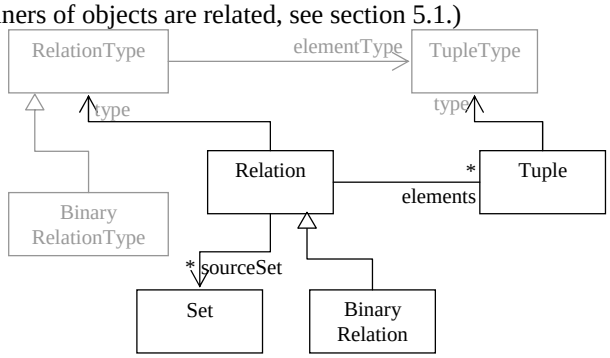

Figure 2 - Relation and RelationType

The approach makes a distinction between a relation and its type, i.e. similarly to the notions of:

```
Tuple{x=1, y=2}
```

and

```
TupleType(x:Integer, y:Integer)
```

We can have the notions of:

```
Relation{ Tuple{x=1,y=1}, Tuple{x=2,y=2}, Tuple{x=3,y=3 } }
```

and

```
RelationType(x:Integer, y:Integer | expression )
```

If we look to the definition of functions in Z literature, they make use of the notion of source and target sets, from which the elements of the function are drawn. Thus for any relation type we can define a relation by setting the source sets for each domain of the relation, e.g.:

```
let
  R3 = RelationType( x:Integer, y:Integer, z:Integer |
                     x*x = y and x*x*x = z ),
  r3 = R3{ x=Set{1..3}, y=Set{1..9}, z=Set{1..27} }
in
  r3 = Relation{ Tuple{x=1,y=1,z=1},
                 Tuple{x=2,y=4,z=8},
                 Tuple{x=3,y=9,z=27} }
```

The elements of relation r3 are formed by restricting the elements of the cross product of the source sets to those elements that meet the expression given in the relation type R3.

As a consequence, all the relation operations can be defined on (finite) relations, although some performance gains may be achievable by making use of the expression given in the type of each relation. Where necessary the default expression for the type of a relation (e.g. when constructed explicitly) can either be set to 'true' or an expression that tests for membership in an explicit set of tuples.

3.3 Operations on Relations

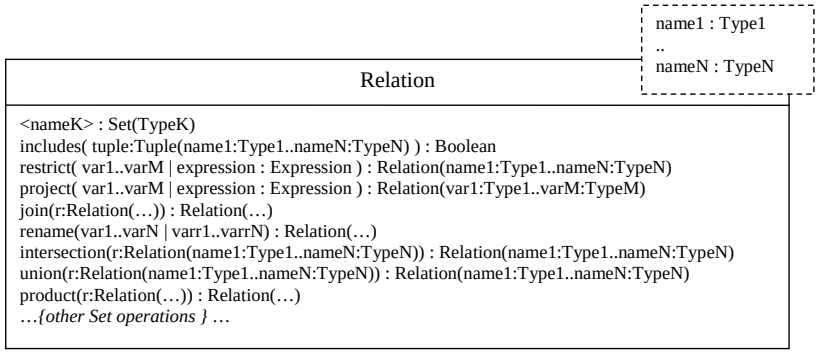

Figure 3 Relation Operations

The previous section has already mentioned a requirement for two operations, one to test if a particular tuple is a member of a relation and one to restrict a relation to eliminate members. Another important operation is the n-dimensional version of the **dom** and **ran** operations known as **project** in relational algebra. Figure 3 shows a Template Relation class with a number of possible operations.

Some operations such as inverse do not have any meaning when applied to n-ary relations; Some operations such as union and intersection can be interpreted as if they are identical to the similar set operations - operating on the set of elements.

The rest of this sub-section addresses the specification of a number of operations on OCL relations. The set of operations specified is not necessarily complete, but includes an important subset of useful operations. Some of the operations are described by giving examples of their use; others are more formally defined in terms of other OCL operations.

project / image / hiding

We can facilitate the project operation by simply extending the notion of project on tuples. I.e. for any tuple, writing '.name' after an expression will return the part of the tuple that is named 'name'. We can do the same for relations, where the result returned by '.name' is the set of objects formed from the set of all tuple parts named 'name' for all tuples in the relation. E.g.

```
r3.x = Set{1,2,3} and r3.z = Set{1,8,27}
```

In addition to a simple project for extracting one of the domains of a relation, we may wish to hide or remove one of the dimensions, i.e. a project to a relation with a subset of the 'columns'. The following OCL shows example uses of the project operation.

```
let
  lectures = Relation{ Tuple{tutor='John', course='Zed'},
                       Tuple{tutor='Dave', course='UML'},
                       Tuple{tutor='Pete', course='Java'} },
  studies = Relation{ Tuple{student='Jane', course='UML'},
                      Tuple{student='Anna', course='Zed'},
                      Tuple{student='Bess', course='Zed'} },
  progCourses = Set{ 'Java' },
  taughtCourses = lectures.course,
  attendedCourses = studies.course,

  -- image where course is from progCourses
  taughtProgCourses = lectures->project( course |
                        progCourses->includes(course)),
  rel = Relation {
    Tuple{tutor='John', course='Zed', student='Anna' },
    Tuple{tutor='John', course='Zed', student='Bess' },
    Tuple{tutor='Dave', course='UML', student='Jane' } }

  -- hide 'course'
  tutors = rel->project(tutor,student| true )
```

includes

Test if a tuple is a member of the relation.

```
context Relation::includes(t:Tuple) : Boolean
 body normalCase: elements->includes(t)
```

restrict / select

In Z there are explicit operators that can be used to restrict the domain and range of a binary relation, e.g. to restrict the range of r3:

```
r3 ◁ {1,2}
```

In relational algebra, such an operation would be formed using a select operation. A similar expression in SQL could be:

```
SELECT x, y, z FROM r3 WHERE x=1 OR x=2
```

Now to do this in an OCL like manner, and at the same time adapt the operation so that the name of the part to be restricted is given, we could introduce an operation in OCL to be used as follows:

```
r3->restrict( x | Set{1,2}->includes(x) )
```

This operation restrict can be defined as a standard select operation on the set of elements, i.e. equivalent to:

```
r3->asSet()->select( t |
   Set{1,2}->includes(t.x) )->asRelation()
```

The name 'select' could be used as the operation name as the operation is similar to the select operation on sets.

compose / natural join

We have a number of options here: the Z approach to joining two relations is to hide the tuple part on which the join is formed; whereas the relational algebra approach is to keep it. *{Which is most useful for OCL? Here we keep it.}*

E.g.

```
let
  rel=lectures->join(studies) =
   Relation {
    Tuple{tutor='John', course='Zed', student='Anna' },
    Tuple{tutor='John', course='Zed', student='Bess' },
    Tuple{tutor='Dave', course='UML', student='Jane' } }
```

We could provide a more general version of the operation that takes an expression parameter giving the conditions of joining. E.g.

```
rel=lectures->join(studies|self.course = studies.course)
```

Although this is using the iterator syntax, it is not using it is the same way. The name 'studies' is being used to refer to an existing variable rather than defining an iterator variable in the context of the following expression. Perhaps a better approach would be to enable explicit specification of expressions and provide a two parameter version of join, e.g.

```
rel=lectures->join( studies,
                    Expression{self,studies |
                       self.course = studies.course}
                  )
```

Although this is not as concise as the previous approach, it does not misuse the standard iterator syntax! Both of these two extended versions of join could run into problems with name clashes.

rename

This operation is slightly problematic to express in OCL without adding additional syntax. We could add additional syntax, and form expressions similar to that of Set theory, Z or RA, however it would be better not to extend the syntax. An example of its use would be to rename the domains of a relation as follows:

```
rel->rename[tutor,course/teacher,subject]
= Relation {
    Tuple{teacher ='John', subject='Zed', student='Anna' },
    Tuple{teacher ='John', subject='Zed', student='Bess' },
    Tuple{teacher ='Dave', subject='UML', student='Jane' } }
```

In fact this operation is essential to the specification of some of the binary relation operations such as 'compose'.

The only part of the OCL syntax that enables the specification of names, is the first part of the arguments to an iterator operation (as has been used for the other relation operations); these names can be used to identify the original names[1]. However, we need a second list of names to indicate the new names. The second part of an iterator

[1] even if it does misuse the OCL syntax, such names are intended to be variables in the following expression, bound to each element of the set on which the operation is called.

operation is an expression, which cannot directly be used to define a list of names other than as Strings, which could only be evaluated at runtime. To ensure that the static typing of OCL works it is necessary to have the new name values available before runtime.

The following are possible options for renaming syntax, but I don't think any of them are satisfactory. The first requires new syntax, and the others severely misuse syntax!

```
rel[tutor,course/teacher,subject]

rel->rename( tutor, course |
  Relation(teacher:String, subject:String, student:String) )

rel->rename( tutor, course;
             teacher:String, subject:String |  ? )
```

product

This works the same as the product of two sets, how ever the result is a set of tuples formed from the product of two sets of tuples producing a set of tuples of tuples, which is then flattened, hence the original two sets of tuples must have disjoint names for their parts.

Other Set Operations

Other operations such as intersection, union, difference, etc that are found as operations on sets should also be defined for relations.

4 Binary Relations

If we consider binary relations as a special type of n-ary relations it is convenient to define them as subtypes of a 2-ary relation with parts named 'first' and 'second'.

```
BinaryRelationType(X,Y) == RelationType(first:X, second:Y)
```

To make things convenient we also define the literal type Pair to be equivalent to a 2-tuple with named parts 'first' and 'second', e.g.

```
Pair{1,2} = Tuple{first=1, second=2}
```

In the case of the Pair, the OCL literal expression can be simply a shorthand method of writing out the longer tuple version. However, for the BinaryRelation, we can define a number of alternative operations, thus it is necessary to have a separate type, with some conversion operations, e.g.

```
BinaryRelation{ Pair{1,2} }->asRelation() =
           Relation{ Tuple{first=1, second=2} }

Relation{ Tuple{first=1, second=2} }->asBinaryRelation() =
           BinaryRelation{ Pair{1,2} }
```

Although this can return undefined if the element type of the relation elements is not TupleType(first:X, second:Y). If we assume that the type BinaryRelation extends the type Relation, the operations on binary relations can be defined in terms of the operations on the super type. The 'asRelation' and 'asBinaryRelation' can be used to promote or demote a relation to one or other of the types.

4.1 Constructing Binary Relations

As discussed above, we can construct a binary relation by promoting a standard relation using the 'asBinaryRelation' operation. We can also construct one using literal values:

```
BinaryRelation{ Pair{'a',1}, Pair{'b',2}, Pair{'c',3} }
```

or using by using an explicit type and expression:

```
let BR = BinaryRelationType( first:Integer, second:Integer |
                             first*first = second )
in brel = BR{ first=Set{1.3}, second=Set{1..9} }
```

To construct an identity relation from a set and get a binary relation we could provide a separate identity operation on sets, however this is not really necessary as we can construct a standard identity relation and then promote it:

```
Set{1,2,3}->identityRelation(first, second | true)
->asBinaryRelation()
```

Or we could define a simple unparameterised form to have a return type of BinaryRelation:

```
Set{ 1, 2, 3 }->identityRelation() =
     BinaryRelation{ Tuple{first=1,second=1},
                     Tuple{first=2,second=2},
                     Tuple{first=3,second=3}
     }
```

4.2 Binary Relation Operations

We give a definition here of some of the operations on binary relations, others such as the many 'closure' operations could also be defined.

domain / range

```
context BinaryRelation(X,Y)::domain() : Set(X)
  body: self.first

context BinaryRelation(X,Y)::range() : Set(Y)
  body: self.second
```

domainRestrict / rangeRestrict

```
context BinaryRelation(X,Y)::domainRestrict(s:Set(X))
                                : BinaryRelation(X,Y)
  body: self->restrict(first|s->includes(first))

context BinaryRelation(X,Y)::rangeRestrict(s:Set(Y))
                                : BinaryRelation(X,Y)
  body: self->restrict(second|s->includes(second))
```

domainSubtract / range Subtract

```
context BinaryRelation(X,Y)::domainSubtract(s:Set(X))
                                : BinaryRelation(X,Y)
  body: self->restrict(first|s->excludes(first))
```

```
context BinaryRelation(X,Y)::rangeSubtract(s:Set(Y))
                                  : BinaryRelation(X,Y)
  body: self->restrict(second|s->excludes(second))
```

image

```
context BinaryRelation(X,Y)::image(s:Set(X))
                                  : Set(Y)
  body: self.domainRestrict(s).second
```

inverse

```
context BinaryRelation(X,Y)::inverse()
                                  : BinaryRelation(Y,X)
  body: self->rename( first, second | second, first )
```

compose

```
context BinaryRelation(X,Y)::compose(br:BinaryRelation(Y,Z))
                                  : BinaryRelation(X,Z)
  body: self->rename[second/temp]
        ->join( br->rename[first/temp] )
        ->project( first, second | true )
```

5 Correspondence to QVT relations

In the QVT Merge Group submission, there is a running example describing the mapping between a simple UML metamodel and a simple model of XML. We show in this section how the specification of relations using the QVT-Merge notation could be represented using the OCL relations. This does not include a representation of mappings; mappings cause changes to the model and in this paper we are only suggesting extensions to OCL that facilitate relation specification and not suggesting extensions that facilitate actions – i.e. building objects – although such extensions are feasible.

The following is the QVT-Merge specification of a relation between a UML Attribute and its representation in XML as an XML Element.

```
relation Attribute_And_XML {
  domain { (UML.Attribute,a)[name=n, type=t] }
  domain {
    (XML.Element,e)[
      name = "Attribute",
      attrs = {
        (XML.Attribute,xa)[name="name", value = n],
        (XML.Attribute,xa)[name="type", value = t]
      }
    ]
  }
}
```

This can be mapped to a relation type in OCL and subsequently used to check if two objects are related in this way. The corresponding relation type defined is OCL is as follows:

```
Attribute_And_XML =
   RelationType( a:UML::Attribute, e:XML::Element |
    let
      n = a.name,
      t = a.type
    in
     e.name="Attribute" and
     e.attrs->any(xa|xa.name="name").value = n and
     e.attrs->any(xa|xa.name="type").value = t
  )
```

The main difference between the two is the use of a pattern language in the QVT-Merge specification. More differences can be seen in the following example which is the specification of a relation between a UML Class and its XML Element representation.

```
relation Class_And_XML {
 domain{ (UML.Class,c)[name=n, attributes=A, methods=M] }
 domain{
  (XML.Element,e)[
    name="Class",
    attributes={(XML.Attribute)[name="name", value=n]},
    contents = XAM
  ]
 }
 when {
  A->forAll(a | Attribute_And_XML(a, XAM.toChoice()) ) and
  M->forAll(m | Method_And_XML(m, XAM.toChoice()) )
 }
}
```

Which can be expressed as an OCL relation type as follows:

```
Class_And_XML =
   RelationType( c:UML::Class, e:XML::Element |
    let
      n = c.name,
      A = c.attributes,
      M = c.methods,
      XAM = e.contents
    in
     e.name="Class" and
     e.attributes->any(xa|xa.name="name").value = n and
     A->forAll(a |
      XAM->select(e|e.name='Attribute')
      ->exists(e|Attribute_And_XML->includes(Tuple{a=a,e=e}))
     ) and
     M->forAll(m |
      XAM->select(e|e.name='Method')
      ->exists(e|Method_And_XML->includes(Tuple{m=m,e=e}))
     )
  )
```

5.1 Relating elements of containers

There are two things to be said about these relation specifications. Firstly, the 'toChoice' operation is not absolutely necessary – although its use may have performance implications – we can construct a similar expression using a nested 'exists' quantification as is shown in the OCL version.

Secondly, the relation is not necessarily complete; it allows elements to exist in the set XAM that are not mapped to either a UML.Attribute or a UML.Method. It may have been the intention of the original specification not to include such a constraint; however, it could be included, in which case there would also need to be a 'forAll' quantification over the elements stating that there should exist either an attribute or method that is related to each one, i.e.:

```
XAM->forAll(e |
     A->exists(a | Attribute_And_XML(a, e)
  or M->exists(m | Method_And_XML(m, e)      )
```

This pattern of constraints that check that the contents of two sets (often containers) are related to each other commonly occurs in relation specifications. In fact it occurs so frequently that it would be very useful to have a more concise way of stating the constraint. The 'toChoice' operation suggested by the QVT-Merge submission goes part way to providing this, but it is not a complete solution.

In set theory (or Z), if the relation[2] 'Attribute_And_XML' is referred to as AX, and there is a set attributes A to be related to a set elements E we can write expressions such as:

```
AX ( A ) = E
```

This says that the set of elements to which some attribute in A is related (i.e. the relational image of A under AX) is equal to the set E; which basically says that all attributes in A are related to some element in E. We may wish to vary this in some cases by stating that there is a subset relationship rather than equality, e.g.

```
AX ( A ) ⊆ E
```

Saying that all attributes are related to an element, but there may be elements in E that are not related to an attribute.

This operation 'image' looks as though it meets our requirement; however it is an operation on binary relations and is thus directional. It does not treat the sets A and E as equal citizens in the context of the expression; ideally we want an expression of the form:

```
AX.ff( A, E )
```

Due to the split notions of RelationType and Relation we can easily construct an expression that checks if the objects in sets A and E are related in the appropriate manner, e.g. the expression:

```
Attribute_And_XML{ A, E }->isBijective()
```

will construct a relation from the objects in A and E, restricted by the expression defined in the relation type 'Attribute_And_XML' and then test (using the isBijective operation) if every object in A and E is related and that every object is related to only one object from the other set. Variations such as 'isFunctional' or 'isTotal' could also be used.

[2] Z does not make the relation and relation type distinction that is discussed in this paper.

This enables a more succinct specification of Class_And_XML as follows:

```
Class_And_XML =
   RelationType( c:UML::Class, e:XML::Element |
    let
      n = c.name,
      A = c.attributes,
      M = c.methods,
      XAM = e.contents
    in
     e.name="Class" and
     e.attributes->any(xa|xa.name="name").value = n and
     Attribute_And_XML{ A, E }->isBijective()
   )
```

6 Conclusion

This paper has proposed that relations be included in OCL as basic types corresponding to the Collection and Tuple types along with the necessary notion of a RelationType corresponding to CollectionType and TupleType. The paper has illustrated the use or specification of a number of useful relation operations within OCL. Finally the paper has shown that these OCL relations can be used to give part of the semantics to a QVT transformation specification language.

Specific points to note are: that some relation operations may require additional OCL syntax (e.g. rename and join) in addition to the obvious additional syntax required to specify a relation. The RelationType concept enables relations to be specified using a predicate expression and source sets from which the elements are drawn.

By looking at the use of relations in the context of QVT specifications, the paper shows the use of relation types and operations to specify that two sets of objects are related by a specific relation; a specification construct commonly required in transformations specifications.

References

[1] Codd E. F., "Relational Completeness of Database Sublanguages," *Data Base Systems*, vol. 6 of Courant Computer Symposia Series, pp. 65-98, 1972.

[2] Date C. J., *An Introduction to Database Systems (Introduction to Database Systems 7th Ed)*: Addison Wesley Publishing Company, ISBN 0201385902, 1999.

[3] OMG, "Response to the UML 2.0 OCL Rfp (ad/2000-09-03), Revised Submission, Version 1.6," Object Management Group, ad/2003-01-07, January 2003.

[4] OMG, "Revised submission for MOF 2.0 Query / Views / Transformations RFP (ad/2002-04-10), QVT-Merge Group, Version 1.0," Object Management Group, April 2004.

[5] Ullman J. D., *Principles of Database Systems*: Pitman Publishing Ltd, ISBN 0273084763, 1980.

[6] Woodcock J. and Davies J., *Using Z: Specification, Refinement, and Proof*: Prentice Hall, ISBN 0-13-948472-8, 1996.