On Querying UML data models with OCL

D.H.Akehurst and B.Bordbar

University of Kent at Canterbury,
Canterbury, Kent, CT2 7NF
{D.H.Akehurst, B.Bordbar}@ukc.ac.uk

Abstract.
UML is the de-facto standard language for Object-Oriented analysis and design of information systems. Persistent storage and extraction of data in such systems is supported by databases and query languages. UML sustains many aspects of software engineering; however, it does not provide explicit facility for writing queries. It is crucial for any such query language to have, at least, the expressive power of Relational Algebra, which serves as a benchmark for evaluating its expressiveness. The combination of UML and OCL can form queries with the required expressive power. However, certain extensions to OCL are essential if it is to be used effectively as a Query Language. The adoption of the ideas presented in this paper will enable query expressions to be written using OCL, that are elegant and ideally suited for use in conjunction with UML data models. This technique is illustrated by expressing the UML equivalent of an example Relational data model and associated query expressions.

1 Introduction

There is a long-standing approach to data modelling, based on the mathematical concept of relations. This approach is supported by Entity Relationship diagrams [6] [21] as a specification language; by relational databases [9] as a means to provide persistence; and the Standard Query Language (SQL [17]) for querying the data.

More recently the Object-Oriented approach to data modelling has been developed. Similarly, this is supported by the Unified Modelling Language (UML [1]), OO-databases [3] [10] and the Object Query Language (OQL [5]).

UML is the OMG's standard for object oriented modelling and has quickly become the de facto standard for specifying OO systems. A UML diagram (such as a Class Diagram) is typically not sufficient to define all aspects of the specification. Therefore, UML provides a textual Object Constraint Language (OCL [1] [23]), which can be used to express detailed aspects about the modelled system.

OCL was originally designed specifically for expressing constraints about a UML model. However, its ability to navigate the model and form collections of objects has lead to attempts to use it as query language [18] [19] [14] [16].

It is well known that in the case of relational databases, in order for a query language to be useful, it must have the expressive power of a relational algebra [9] [22]. Hence, it follows that the same must be true for OO databases and their

respective query languages. In this respect, the authors of [16] discuss the expressive power of OCL, and infer that OCL in isolation is not as expressive as a relational algebra.

Building upon their approach, this paper makes use of the detailed semantics of UML and OCL to present an indirect method of forming query expressions. We show that this method leads to a technique for forming expressions that are as expressive as those formed using a relational algebra.

The proposed method requires extra UML classes to be added to the model; this can be cumbersome and resource consuming. Since the UML reference model is currently undergoing a major revision [2] [8], the final part of the paper takes the opportunity to propose extensions to OCL, which enable OCL to be used as an ideal Object-Oriented Query Language.

The rest of this paper is organised as follows: Section 2 discusses the relational approach to data modelling and provides a definition of a relational algebra (RA). Section 3 defines the example used throughout the paper. Section 4 discusses the problems of constructing queries using OCL. Section 5 illustrates a method by which UML and OCL in conjunction can provide all the functionality required by a query language. Section 6 proposes some extensions to the OCL core that would enable OCL query expression to be much more easily formed. Finally, the paper concludes in section 7 by summarising the work presented.

2 Relational Data Modelling

A long standing technique for modelling data in information systems is to represent data using a set of *table*s and relationships between tables. This approach is supported using Relational Database Management Systems (RDBMS), which manage computerised implementation of the data, tables and relationships.

One of the major features of an RDMBS is the provision of extensive support for the manipulation of data. The standard language for expressing the required manipulations (or queries) is called SQL [17]. The principle behind such query languages, giving them a sound mathematical foundation, is called Relational Algebra [9] [22].

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 [9]), 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 [9]). Consequently, in order for a query language to be considered fully expressive, it must support as a minimum, the primitive operators [7]: Union, Difference; Product; Project; and Select. A definition of a relation and these five operators (taken from [9]) 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 [9] and [22] for further details on relational algebras.

3 Example

As an example, we use through out the paper a data-model that could form part of a database used by an educational institution. The model records information regarding students, courses and teachers, and additionally it relates each student to the courses they study and each course to the teachers who teach the courses. The model can be specified in UML as shown in **Fig. 1**.

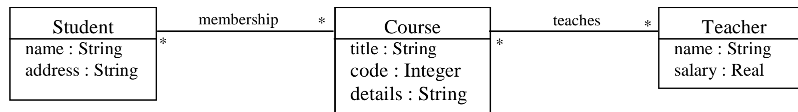
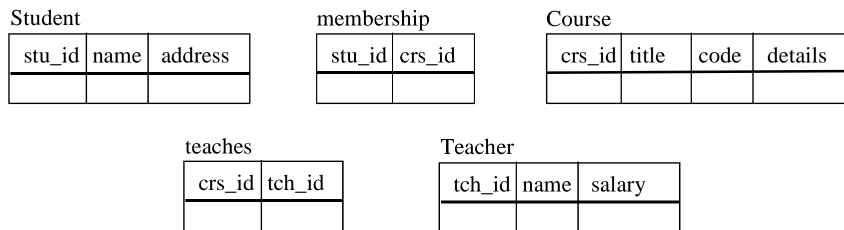


Fig. 1. Example specified in UML

The data-model shown in **Fig. 1** can be mapped to a Relational Database (RDB) using the technique suggested in [3]. The mapping is formed such that each Class and each Association forms a Table definition and the tables are related by including the appropriate foreign keys, as shown in **Fig. 2**. The keys for each of the tables Student, Course, and Teacher are respectively named stu_id, crs_id and tch_id.



Student

stu_id	name	address

membership

stu_id	crs_id

Course

crs_id	title	code	details

teaches

crs_id	tch_id

Teacher

tch_id	name	salary

Fig. 2. Example expressed as an RDB

Using this model, assume that there is a requirement to provide a list that shows which students are taught by which teachers, irrespective of which course.

With respect to the RDB definition, the required list of teachers and students can be generated using an SQL statement as follows:

```
SELECT DISTINCT s.name, t.name                                        (1)
   FROM Student s, Teachers t, membership m, teaches ts
   WHERE t.tch_id = ts.tch_id
   AND   ts.crs_id = m.crs_id
   AND   m.stu_id = s.stu_id;
```

This Query produces a table showing the names of each teacher and student pair. If we subsequently required more information regarding either the teacher or student in any particular pairing, the SQL statement would need to be changed to include the additional information.

Ideally, we would like the list to contain the pairs of teacher and student objects, rather than just a list containing pairs of their names. Thus, any requirement for additional information (about teachers or students) can be navigated to, from the results of the query, rather than having to alter the specification of the query to return the additional information.

4 OCL as a Query Language (QL)

Integrating OO models of data into RDBMS is state of the art in software development. While SQL serves as a query language in RDBMS, OQL [5], the most famous query language for object databases, does not have a precise semantics. Although there are many variations, there is no fully accepted equivalent to SQL for querying an OO data model.

Blaha and Premerlani [3] introduce the Object Navigation Notation (ONN) for navigation of OMT [20] models (a predecessor to UML). UML has instead OCL, which in addition to being useful as a navigation and constraint language poses a natural choice for use in making queries on the model.

Work such as [11] shows how OCL, when used to specify integrity constraints (well-formedness rules) on the model, can be mapped to the equivalent SQL for the specification of integrity constraints on the tables of an RDB. Their work shows how each OCL language construct can be mapped to an equivalent declarative SQL statement. Their continuing work in [12] discusses the provision of tool support for their approach.

In order for OCL to be considered a fully expressive QL, we must be able to translate any SQL statement into an equivalent OCL expression.

The authors of [14] point out certain obstacles regarding the use of OCL as a QL. In particular they recognise that most Object-Oriented modelling environments that include a QL, provide as a basic type a facility for structured aggregation (such as a tuple or “struct”) and give as examples [4], [13] and [15]. As it stands, OCL does not provide such a feature.

Mandel and Cengarle’s work [16] explicitly addresses the expressiveness of OCL in the context of its use as a query language. They conclude that OCL is not expressive enough to define all of the operations required by a relational algebra and hence it does not form an adequate query language.

As it stands, OCL supports three out of the five required RA operations. Union, Difference, and Select are all supported directly by operations defined on the OCL collection types. However, the operations Product (or Cartesian Product) and Project are not supported, and cannot be supported as they directly require a facility for structured aggregation or a notion of tuples (see [14]).

5 UML and OCL as a Query Language

Based on the semantics of UML, in this section we demonstrate the UML and OCL together *can* form expressions with the functionality of the RA operators. The key is to provide the concept of a tuple.

The following subsections start by explaining a way of supporting the notion of an n-tuple using the UML concept of AssociationClass and n-ary Association. This is followed by an explanation of how to use the concept to provide the functionality of the Product and Project operations. The section concludes by illustrating that the approach enables the expression of our example query.

5.1 LinkObjects as Tuples

As illustrated by an extract of the UML meta-model [1] shown in **Fig. 3**. An instance of an Association is called a *Link* and is defined by the UML standard to be a *tuple of object references*. Dually, the instance of an AssociationClass is called a *LinkObject*. The object references are modelled within the UML meta-model as *LinkEnds* and each Link contains an *ordered* list of LinkEnds.

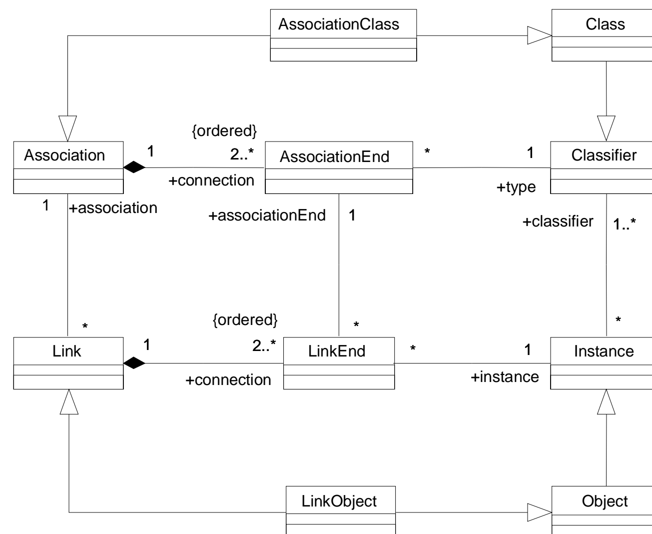


Fig. 3. Extract from the UML meta-model showing AssociationClass and LinkObject

Given this definition of the meta-model, the concept of a LinkObject can be used to represent a tuple of other objects. The relationship between the elements (coordinates) of the tuple is expressed by the Link and the object representing the tuple, is the connected LinkObject. **Fig. 4**a illustrates an object diagram showing three objects – a, b and c – that are involved in the tuple (a,b,c).

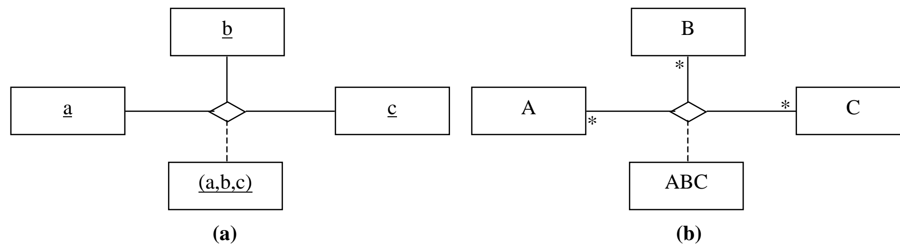


Fig. 4. Class and instance specification of a tuple.

The appropriate class specification defining a type for the tuple object can be given using an AssociationClass as shown in **Fig. 4**b. The multiplicity of the AssociationEnds is defined as '*' to allow each involved object to partake in any number of tuples.

The above describes support for 3-tuples; the approach is extensible, supporting n-tuples with n-ary associations for any n.

An issue to consider at this point, is how to determine the ordering of a tuple; for example, is the first co-ordinate of the tuple of type A, B or C? There is nothing within the notation for n-ary associations that indicates the ordering of its ends, even though the UML meta-model does define that the ends are ordered. Generally, this is not a problem as each end is uniquely named and these names are used within OCL expressions to navigate to the co-ordinate objects rather than an index number.

5.2 Cartesian Product using UML and OCL

The above approach describes how to support tuples using the UML concept of a Link. In this subsection, we present an indirect method for providing the functionality of the Product operator over three sets. In a similar way, one can support the operation over n sets for any value of n.

By definition, the product of three sets S, T and U is the set of all tuples (s,t,u) such that s is in S, t is in T and u is in U.

Since OCL is a side effect free language an OCL expression cannot create new objects. Thus, the result of the Product operation must be formed by selecting appropriate tuples from a set in which they already exist.

The only way to ensure that such a set exists is to constrain the model as a whole. For the model shown in **Fig. 4**b, containing classes A, B, C and ABC, the following constraint is sufficient:

```
context ABC inv:
ABC.allInstances->size = A.allInstances->size *
                         B.allInstances->size *
                         C.allInstances->size
```

(**2**)

Strictly speaking, this constraint could be placed anywhere inside the UML model, it constrains the instantiation of the model as a whole, rather than specifically constraining the class ABC; however, we use class ABC as a convenient placeholder. It should be noted that, since we are dealing with models of data, at any fixed point in time there are a finite number of instances of each class, and hence the numerical values in constraint (2) are finite.

The semantics of UML and this constraint result in a model definition, for which the instances of class ABC correspond to the Cartesian Product of the instances of A, B and C; i.e.:

```
ABC.allInstances = A.allInstances x
                   B.allInstances x
                   C.allinstances
```

(**3**)

To see this, notice that the Left Hand Side (LHS) of (3) is a subset of its Right Hand Side (RHS). Each ABC object is a LinkObject connecting A, B and C objects and we have proposed (in the previous subsection) that a Link Object is equivalent to a tuple, so it follows that the LHS $\subset$ RHS.

The UML standard [1, p2-94] states that "There are not two Links of the same Association which connects the same set of Instances in the same way." Due to constraint (2) the LHS and RHS have the same finite number of elements. Since there cannot be two identical instances of ABC the LHS = RHS in equation (3).

A Cartesian Product between two arbitrary Sets of objects can be formed by selecting appropriate instances from the Set of allInstances of an AssociationClass defined in this way.

5.3 Project using UML and OCL

The previous section explained our method for the creation of Cartesian Products. We now build on this to define a method of support for the Project operation. Again, we explain our approach for three classes and projection over 3-tuples, but it is extensible to the general case for n-tuples.

As defined in [16], for a Binary Association (Pair or 2-tuple), the project function must return either the first or second co-ordinates, by simple navigation. The difficult issue with Project occurs when we consider tuples with more than two co-ordinates.

Consider three classes A, B, C and the Cartesian Product of their instances as represented by AssociationClass ABC (**Fig. 4**b). Formally, a projection, $proj_{1,3}$ maps a tuple (a,b,c) into a 2-tuple (a,c). To implement the operation $proj_{1,3}$ there are two issues to address:

1. The AssociationClass that provides a type for the tuple (a,c), and whose instances represent the Cartesian Product of A and C.
2. The specification of the function that maps (a,b,c) onto (a,c).

Following the method defined in the previous section, we must add the AssociaitonClass AC to the model, relating classes A and C, as shown in **Fig. 5**.

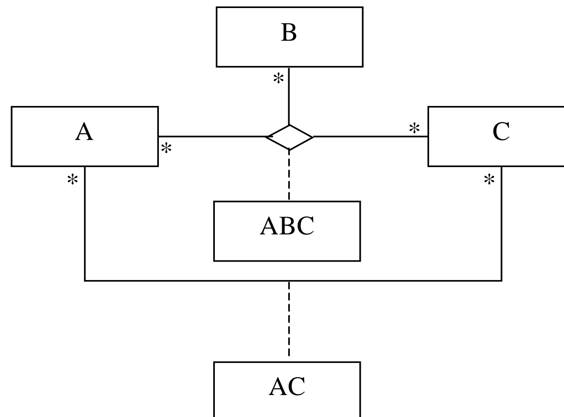


Fig. 5. Extended class specification

The project operation can be subsequently defined as follows:

```
context tuple : ABC

tuple->project_a_c : AC
pre: OclType.allInstances->exists( t | t = AC )
post: result = AC.allInstances->select( tup |
                  tup.a = tuple.a and tup.c = tuple.c ) )
```
(**4**)

The pre condition of the operation ensures that the class AC has been defined. The post condition defines the result to be the tuple of type AC that has its co-ordinates 'a' and 'c' equal to the 'a' and 'c' co-ordinates of this tuple. (Remember that the elements of a LinkObject, the tuple, are navigated to via the rolenames of the Association, rather than by their index value.)

This explains a method for indirectly supporting a project function that maps 3-tuples of type ABC onto 2-tuples consisting of the first and third co-ordinates. In a similar way, we can support creation of $proj_{2,3}$ and $proj_{1,2}$, which involve associating B to C and A to B respectively.

There are also project functions, $proj_1$, $proj_2$ and $proj_3$, which project a tuple to its individual co-ordinates. These do not require the creation of new links and are supported simply by the navigation semantics of the OCL '.' operator.

Performing a project operation over a set of tuples can be easily supported by making use of the *collect* operation defined for OCL collections.

5.4 Solution to the Example Problem

Our example of section 3 requires an expression that results in a list, relating teachers to their pupils. In this subsection, we use our method to specify the query expression corresponding to the SQL statement (1).

To achieve such an expression, we first modify the original data model (**Fig. 1**) to include a new class, TeacherxStudent, as shown in **Fig. 6**.

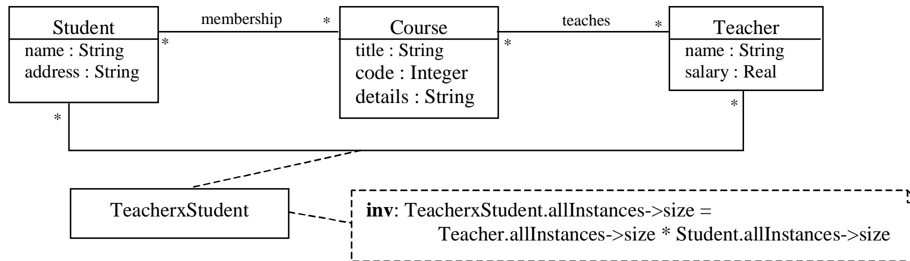


Fig. 6. Extensions to example data model

This AssociationClass specifies a class for tuples of teacher and student objects and defines each member of the Cartesian Product of Student and Teacher objects.

The required list of (teacher, student) pairs can be subsequently specified using an OCL expression as follows:

```
Teacher.allInstances->collect( t |
  t.course.student->collect(s |
    TeacherxStudent.allInstances->select( ts |
      ts.teacher = t and
      ts.student = s ) ) )->asSet
```
(5)

The expression defines two nested *collect* operations, which iterate over each required pair of teacher and student objects. These are used to select the appropriate tuple object from the instances of TeacherxStudent.

This OCL query provides a collection (list) of student and teacher pairs, rather than the pairing of name strings provided by the SQL statement defined earlier. Although it has some complexity, it is functionally similar to the SQL, and we do not need to explicitly state how the data items are joined.

However, the complexity of the model rises as a result of defining additional classes within the UML specification. Ideally, a mechanism should be provided which enables queries to be specified that do not require additional model elements to be defined. The following section proposes essential extensions to the OCL language and pre-defined types that will enable such query expressions to be formed.

6 Extending OCL to be a fully expressive Query Language

In the previous section, we presented our indirect method for using OCL as a query language. However the method requires new components to be added to the original data model; this is cumbersome and undesirable. The approach relies heavily on the use of the *allInstances* operation, which is expensive to implement and controversial with respect to its use on classes such as String and Integer.

To be able to form OCL query expressions that are more seamlessly useable within the context of a UML model, a Tuple type needs to be added to the pre-defined types of the OCL language. This can be easily achieved in a similar manner to the Collection classes.

The following subsections define an appropriate Tuple type and show how instances can be defined as part of an OCL expression. This is in line with the current language semantics, which does allow new instances of collection classes to be specified.

Based on this new OCL Tuple type, the operations Product and Project are defined as operations on (respectively) the Collection and Tuple types. Finally, we show how the provision of this type enables the example query to be easily written as an OCL expression.

6.1 Creating Tuples in OCL

A generic type for tuple objects can be provided using a ParameterisedClass, in a very similar way to the Collection types (Set, Sequence and Bag) already defined. A simple Tuple Class could be defined such that its co-ordinate members are all of the root type OclAny; all tuples would thus belong to the set of *allInstances* of that class.

It would be useful to enable a distinction to be made between tuples of different sizes and between tuples with co-ordinate members of differing types. The concept of a ParameterisedClass can be used to provide a type for tuples that enables this distinction to be made. **Fig. 7** illustrates a UML definition of this Tuple Class.

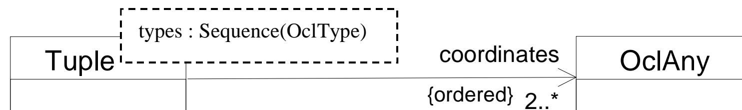


Fig. 7. Definition of a Generic Tuple Class

To support the Project and Product operators, it is necessary to be able to create tuple objects within an OCL expression. Unfortunately, this directly contravenes the OCL policy of no side effects. However, an OCL expression can include the explicit creation of a Collection object. This is not only described as possible in the UML version 1.3 standard, it is also used as part of the definition of the OCL collection classes, for example, as taken from the definition of subSequence :

```
Sequence{lower..upper}->forAll( index | ...
```
(**6**)

Thus, we propose that the semantics should support the creation of Tuple objects in a similar manner, as follows:

```
Tuple { a, b, c }
```
(**7**)

Semantically, the types for each of the co-ordinate members of a Tuple created in this way, can be deduced from the types of the objects used to instantiate it. Thus, the tuple instantiated in equation (7) can be deduced to be an instance of a specific tuple type Tuple(Sequence(A,B,C)), assuming that objects a, b and c are instances of classes A, B and C.

6.2 Definition of a Product Operation

Given the provision of this Tuple type, providing a Product operation is an easy addition to the Collection classes, defined as follows:

```
context c1:Collection(T1)

c1->product(c2:Collection(T2))
      : Collection( Tuple(Sequence{T1,T2}) )
post: result = c1->collect( t1 |
                  c2-> collect( t2 |
                    Tuple {t1,t2} ) )
```

(8)

The Cartesian Product between more than two sets can be formed from multiple nested products of two sets. Semantically, an extension to the *product* operation could be defined, which flattens out nested 2-Tuples into appropriate n-Tuples. For example, given three sets S, T and U, the product:

```
S.product(T).product(U)
```

(9)

gives 3-tuples of the form ((s,t), u). The extension proposes that this be flattened to a tuple of the form (s, t, u).

6.3 Definition of a Project operation

The Project operator can be supported by providing appropriate operations on the Tuple class itself. The Project operation extracts specific elements from its target tuple. Extraction of single elements should result in that element and extraction of multiple elements should result in another tuple, containing the required elements.

This functionality is provided as two separate project operations, defined as follows:

```
context tuple : Tuple( types : Sequence(OclType) )

tuple->project( index : Integer) : types->at(index)
post: result = tuple.coordinates.at(index)

tuple->project( indices: Sequence(Integer) )
         : Tuple( indices->collect(i|types.at(i)) )
pre: indices->size > 1
post: result =
    Tuple { indices->collect( i| tuple.project(i) ) }
```

(10)

The first of these operations takes a single index value as a parameter and returns the object contained at the co-ordinate position of the requested index. The second takes a sequence of indices as a parameter and returns a new Tuple that is formed from the combination of objects at the co-ordinate position of each index in the parameter.

6.4 Ideal solution for Example Problem

If we provide OCL with the above extension, adding the concept of tuple and the related operation definitions of *product* and *project*, then we can easily write powerful and concise OCL query expressions.

For example, considering the example of section 3, the required collection of pairs indicating which students are taught by which teachers, can be formed using the following OCL expression:

```
Teacher.allInstances->collect( t |
  (Set{t}).product( t.course.student ) )
```
(**11**)

The resulting collection contains the set of tuples formed by collecting the union[1], for each teacher, of all products of a teacher and the student objects navigable to from it.

This gives the ideal solution of pairs of student and teacher objects. An expression equivalent to the SQL, giving a collection of pairs of teacher and student names is specified as follows:

```
Teacher.allInstances->collect( t |
  (Set{t.name}).product( t.course.student.name ) )
```
(**12**)

7 Conclusion

In this paper, we have shown that for OCL to be effectively and elegantly used as a Query Language, certain extensions are required – primarily the addition of a Tuple type.

Using as a benchmark the five primitive operators of Relational Algebra, this paper has shown that the combination of UML and OCL are expressive enough to form expressions functionally equivalent to those formed using Relational Algebra.

However, since OCL was not originally designed to be a query language, it lacks certain technical concepts required for writing query expressions, which can be provided by extending the underlying UML model.

Adding extra components to the underlying model is cumbersome and undesirable in the context of a query language. To avoid such measures, this paper has proposed and demonstrated the use of some extensions to OCL that are essential, if it is to be effectively used as a query language.

These extensions, which are in line with the existing semantics of OCL, enable us to write concise queries without modifying the underlying UML model.

Acknowledgement

We would like to thank Stuart Kent and Nigel Dalgliesh for their valuable comments. The second author wishes to acknowledge the generous support of the EPSRC project (GR/M69500).

---

[1] Due to the flattening semantics of collect.

References

1. Object Management Group; *Unified Modelling Language Specification, version 1.3*; OMG ad/99-06-08 (June 1999); http://www.omg.org.
2. Object Management Group; *UML 2.0 RFI*; OMG Document ad/99-08-08 (August 1999); http://www.omg.org.
3. M.Blaha, W.Premerlani; *Object-Oriented Modeling and Design for Database Applications*; Prentice Hall Inc., ISBN: 0-13-123829-9 (1998).
4. P. Butterworth, A. Otis, J. Stein; *The GemStone Object Database Management System*; Communications of the ACM, 34(10) (1991) pp 64 - 77.
5. R. G. G. Cattell; *The Object Database Standard: ODMG 2.0*; Morgan Kaufmann Publishers, Inc. (1997).
6. P. P. Chen; *The Entity-Relationship Model - Toward a Unified View of Data*; ACM Transactions on Database Systems, 1(1) (1976).
7. E. F. Code; *Relational completeness of database sub-languages*; In Data Base Systems, R.Rustin, Ed., Prentice Hall, Englewood Cliffs (1972) pp 65 - 98.
8. S. Cook, A. Kleppe, R.Mitchell, B.Rumpe, J.Warmer, A.Wills; *The Amsterdam Manifesto on OCL*; (December 1999); http://www.trireme.com/amsterdam/.
9. C. J. Date; *An Introduction to Database Systems (Introduction to Database Systems, 7th Ed)*; Addison Wesley Publishing Company, ISBN: 0201385902 (August 1999).
10. C. J. Date, H. Darwin; *Foundation for Object/Relational Databases: The Third Manifesto*; Addison Wesley Publishing Company, ISBN: 0201309785 (June 1998).
11. B. Demuth, H. Hussmann; *Using UML/OCL Constraints for Relational Database Design*; Proceedings of «UML» '99 - The Unified Modelling Language: Beyond the Standard (October 1999) pp 598 - 613.
12. H. Hussmann, B. Demuth, F. Finger; *Modular Architecture for a Toolset Supporting OCL*; Proceedings of «UML» 2000 – The Unified Modeling Language: Advancing the Standard (October 2000) pp 278 - 293.
13. O. Deux; *The O2 System*; Communications of the ACM, 34(10) (1991) pp 34-48.
14. M.Gogolla, M.Richters; *On Constraints and Queries in UML*; Proc. UML'97 Workshop `The Unified Modeling Language - Technical Aspects and Applications' (1997).
15. C. Lamb, G. Landis, J. Orenstein, D. Weinreib; *The ObjectStore Database System*. Communications of the ACM, 34(10) (1991) pp 50 - 63.
16. L. Mandel, M. V. Cengarle; *On the Expressive Power of OCL*; FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, Springer LNCS 1708 (September 1999) pp 854 - 874.
17. J. Melton, A. R. Simon; *Understanding the New SQL: A Complete Guide*; Morgan Kaufmann Publishers, Inc., ISBN: 1558602453 (1994).
18. Jan Nordén; *Bold Executable Model Architecture*; (June 2000); http://www.boldsoft.com/products/whitepapers/index.htm.
19. ModelRun; *Boldsoft modelling tool*; http://www.boldsoft.com/products/modelrun/.
20. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen; *Object-oriented Modeling and Design*; Prentice Hall International Paperback Editions, ISBN: 0136300545 (March 1991).
21. B. Thalheim; *Fundamentals of Entity-relationship Modeling*; Springer-Verlag Berlin and Heidelberg GmbH & Co. KG, ISBN: 3540654704 (December 1999).
22. J. D. Ullman; *Principles of Database Systems*; Pitman Publishing Ltd, ISBN: 0273084763 (1980).
23. J. B. Warmer, A. G. Kleppe; *The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley Object Technology Series)*; Addison Wesley Publishing Company; ISBN: 0201379406 (October 1998).