Derived classes as a basis for views in UML/OCL data models

Balsters, H.

Derived Classes as a basis for Views in UML/OCL Data Models

H. Balsters
University of Groningen
Faculty of Management and Organization
P.O. Box 800
9700 AV Groningen

h.balsters@bdk.rug.nl

SOM-theme A Primary processes within firms

Abstract
UML is the de facto standard language for analysis and design in object-oriented frameworks. Information systems, and in particular information systems based on databases and their applications, rely heavily on sound principles of analysis and design. Many present-day database applications employ object-oriented principles in the phases of analysis and design due to the advantages of expressiveness and clarity of such languages as UML. Database specifications often involve specifications of constraints, and the Object Constraint Language (OCL) - as part of UML - can aid in the unambiguous modelling of database constraints. One of the central notions in database modelling and in constraint specifications is the notion of a database view. A database view closely corresponds to the notion of derived class in UML. This paper will show how the notion of a derived class in UML can be given a precise semantics in terms of OCL. We will then demonstrate that the notion of a relational database view can be correctly expressed as a derived class in UML/OCL. A central part of our investigation concerns the generality of our manner of representing relational views in OCL. An important problem that we address in this respect is the representation of product spaces and relational joins. Joins are often essential in view definitions, and we shall demonstrate how we can express Cartesian products and joins within the current framework of UML/OCL language by employing the notions of derived class. As a consequence, OCL will be shown to be equipped with the full expressive power of the relational algebra, offering support for the claim that OCL can be useful as a general query language within the framework of the UML/OCL data model.

Keywords
UML, derived classes, OCL, information systems design, database modelling, database views, query languages

Introduction

Information systems, and in particular information systems based on databases and their applications, rely heavily on sound principles of analysis and design. This paper focuses on particular principles of analysis and design related to database applications. Following [BP98], we can state that object-oriented (OO) modelling can prove to be very beneficiary in (relational) database applications. A database is a permanent, self-descriptive repository of data stored in files. A database is self-descriptive in the sense that it not only contains the data, but also a description of the data structure, or *schema.* In databases, the data usually change rapidly, while the schema stays relatively static. A database management system (DBMS) consists of software managing access to the data. DBMSs provide generic functionality for a broad range of applications; one of the foremost features of a DBMS is the availability of a *query language* offering an interactive means for reading and writing data from the database. A relational database has data represented as tables, and a relational DBMS manages access to tables of data and associated structures in highly effective and efficient manner. (Relational databases use SQL as a data manipulation language, and tables are called *relations* in SQL.) Relational database applications can benefit substantially from OO modelling. The OO paradigm provides a uniform framework for both the design of database code and programming code. Database and their applications can thus be developed in one and the same conceptual framework. In fact, one can say that integrating relational databases into object-oriented applications is state of the art in software development practice. OO data models offer high-level modelling primitives leading to clear and concise specifications of database schemas. A high-level description of a database schema in terms of an OO data model can easily be mapped to a relational database schema employed by a conventional relational DBMS [BP98]. Hence, the analysis and design stage of a (relational) database can be separated in a clear and meaningful fashion.

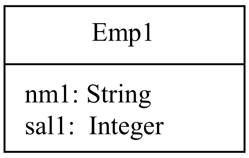
The most important OO modelling language is UML, being the de facto standard for OO analysis and design of information systems [OMG99]. Recently, researchers have investigated possibilities of UML as a modelling language for (relational) databases. [BP98] describes in length how this process can take place, concentrating on schema specification techniques. [DH99, DHL01] investigate further possibilities by employing OCL (the Object Constraint Language [WK99]) for specifying constraints and business rules within the context of relational databases. Some researchers take a very general approach investigating possibilities of UML/OCL; e.g., [AB01] treat OCL as a general query language for UML data models, and [EP00] use OCL as a general language for business modelling. Current research, however, has not yet shown an effective way to deal with an important aspect of (relational) database modelling, namely modelling of so-called *database views*. A (database) view is a derived table (or derived relation, in SQL), meaning that a view does not exist as a physical relation; rather a view is defined by an expression much like a query [GUW02]. Views, in turn, can be queried as if they existed physically, and in some cases, we can even modify view content. That is, a user is offered the impression that a view is some base relation inside the database, but in fact it is a derived (or *virtual*) relation defined in terms of the actual base relations constituting the database. View definitions are an important asset in database applications, because users are usually only interested in a part of the database, and not in the complete underlying corporate database. Hence, it is important that users have access to that part of the database considered relevant for their category of database applications. Another application area for views can be found in the realm of Federated Databases, where legacy databases are to interoperate by employing a so-called mediating system. This mediating system can be considered as an integration of a set of certain database views defined on the component legacy database systems. The research in this paper is actually inspired by work on modelling federated databases as a system of database views [BB00].

Till now, researchers have not been fully complete in capturing the notion of database view within the UML/OCL data model. The idea is that OCL provides expressiveness in terms of relatively abstract set definitions that should prove to be sufficient to capture the general notion of (relational) database view. This idea of employing abstract object-oriented set definitions to captures views and constraints has also been pursued on the full level of object-oriented databases, be it not in the context of UML/OCL language, but rather in the context of an experimental OODB user language in combination with an underlying theoretical semantics [BBZ93, BV92]. In the more specific context of relational databases and OCL, [DH99] offer a framework for representing constraints within the relational data model, but a general notion of database view is lacking. Database views and query languages are strongly related topics, since views are basically no more than named queries. [GR97] was one of the first papers to investigate the possibilities of a query language for UML; further investigations can be found in [AB01] and [MC99]. [AB01] have attempted to demonstrate that OCL can offer the basis for a general query language for UML data models by showing how to represent Cartesian products and projections in OCL, thus paving the way to the claim that OCL has the same expressive power as the so-called relational algebra [D00, GUW02]. By demonstrating such a result, one could also claim to have a basis for representing views within OCL. The approach taken in [AB01], however, has some shortcomings. First of all, their solution is rather complex, leading the authors to propose certain language extensions to OCL as an alternative solution to the problem. Furthermore, in [MC99] it is claimed -on theoretical grounds- that OCL cannot have the expressive power of the so-called relational calculus. Since the expressiveness of the relational calculus and the relational algebra are equivalent [D00], there seemingly is an inconsistency between the two results offered in [AB01] and [MC99]. The inconsistency, however, is based on a different approach on treating Cartesian products in UML/OCL. We will discuss both [MC99, AB01] in detail, pointing out that both approaches contain certain flaws that can be remedied by a new approach adopted in our paper. In particular, we will offer an alternative approach for establishing the result that the expressiveness of OCL includes that of the relational algebra. We will do so by showing how to offer the notion of *derived class* a formal basis within the framework of UML/OCL, and subsequently use this notion of derived class to represent the notions of Cartesian product and (relational) join. This result will establish that OCL includes the expressiveness of the relational algebra, and we will do so in a relatively simple manner and without resorting to language extensions of OCL. Once we have established the result that OCL includes the expressiveness of the relational algebra, then we also have provided a basis for representing the general notion of (relational) database view.

This paper is structured as follows. Throughout the paper, we shall use as a running example of a database taken from [GUW02] employing views. Furthermore, all OCL-constructs used in this paper can be found in the basic work of [WK99]. Section 1 offers an introduction to the concept of a view. We will offer some possible attempts to define views in terms of UML/OCL; each attempt will, however, fall short demonstrating that certain at first sight plausible solutions need closer inspection. Section 1 furthermore offers a description of the basic OCL semantics for the notion of derived class in UML. Purpose is to show how the notion of database view can be expressed as a derived class in UML/OCL. A derived class is a device for denoting a virtual class, defined in terms of already existing (base) classes (and possibly other views). Views can be queried independently, with a semantics explained entirely in terms of queries on base classes. Section 2 contains issues concerning more complex view definitions and their representation in UML/OCL. Section 3 discusses the central research question regarding adequacy of our approach to offer a general definition mechanism for database views in UML/OCL. Section 4 treats Joins and Cartesian products in UML/OCL. Section 5 treats expressiveness of OCL as a relational query language, establishing our main result. The paper ends with a short summary of our results.

1. Database views and a simple view representation in UML/OCL

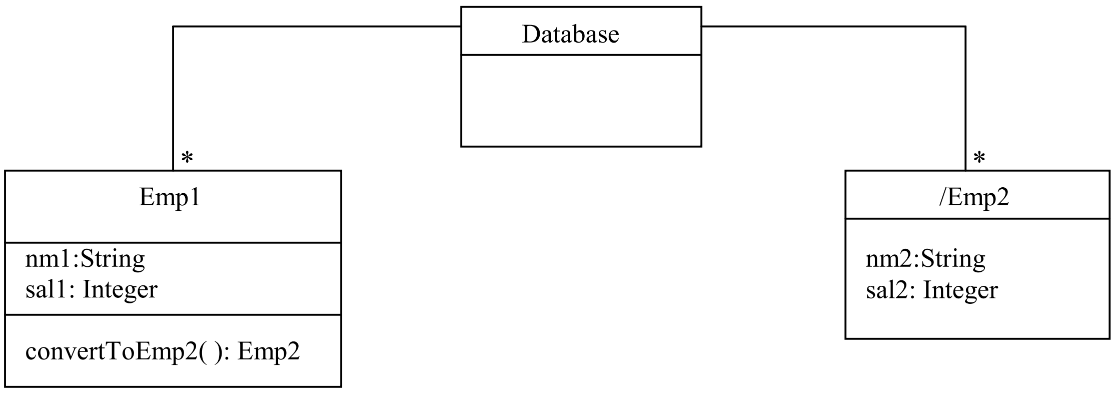
Let's consider the case that we have a class called Emp1 with attributes nm1 and sal1, indicating the name and salary of an employee object belonging to class Emp1



Now consider the case where we want to add a class, say Emp2, which is defined as a class whose objects are completely derivable from objects coming from class Emp1. The calculation is performed in the following manner. Assume that the attributes of Emp2 are nm2 and sal2 respectively (indicating name and salary attributes for Emp2 objects), and assume that for each object e1:Emp1 we can obtain an object e2:Emp2 by stipulating that e2.nm2=e1.nm1 and e2.sal2=(2 * e1.sal1). By definition the total set of instances of Emp2 is the set obtained from the total set of instances from Emp1 by applying the calculation rules as described above. Hence, class Emp2 is a *view* of class Emp1, in accordance with the concept of a view as known from the relational database literature. In UML terminology

[BP98], we can say that Emp2 is a *derived class*, since it is completely derivable from other already existing class elements in the model description containing model type Emp1.

We will now show how to faithfully describe Emp2 as a derived class in UML/OCL in such a way that it satisfies the requirements of a (relational) view. First of all, we must satisfy the requirement that the set of instance of class Emp2 is the result of a calculation applied to the set of instances of class Emp1. The basic idea is that we introduce a class called Database that has associations to classes Emp1 and Emp2. A database object will reflect the actual state of the database, and the system class Database will only consist out of one object in any of its states. Hence the variable *self* in the context of the class Database will always denote the actual state of the database that we are considering. In the context of this database class we can then define the calculation obtaining the set of instances of Emp2 by taking the set of instances of Emp1 as input.



Note that we have used a prefix-qualification by adding a slash to Emp2 indicating that Emp2 is a derived class definition [BP98]. Moreover, we have added an operation, called convertToEmp2, meant to coerce an arbitrary Emp1-object to an Emp2-object. This operation can be defined by the following OCL-specification

```
context   Emp1::convertToEmp2( ): Emp2
post:     self.convertToEmp2.nm2 = self.nm1  and
          self.convertToEmp2.sal2 = (2*self.sal1)
```

We now have all the ingredients necessary to specify the relation coupling the derived class Emp2 to the original class Emp1. This is done by including an invariant specification in the class Database telling us how to calculate the set of instances of Emp2 from the set of instances of Emp1

```
context  Database  inv:
self.Emp2 = self.Emp1→ collect(e:Emp1 | e.convertToEmp2) and
Emp1.allInstances = self.Emp1  and
```

```
Emp2.allInstances = self.Emp2
```

In this way we explicitly specify Emp2 as the result of a calculation performed on Emp1, and we also stipulate that the only Emp1- and Emp2-objects in the database are those obtained from the links starting from the database-object *self*.

Discussion: How *not* to represent views

A reader might have the idea that there is an alternative (and rather simple) way to define database views in UML/OCL employing constraints, and without having to introduce the notion of derived class. We wish to discuss this topic here, because it deals with somewhat widespread misconception of what a database view actually is. Consider our example of Emp2 as a database view derived from the base class Emp1. One might be inclined to think that Emp2 could also be defined indirectly by employing suitable constraints. For example, one could introduce Emp2 as an extra model type (hence not as a derived class), and then stipulate the following two constraints

```
context  Emp2  inv:
Emp1.allInstances →
     exists(e1 | e1.nm1 = self.nm2 and 2*e1.sal1 = self.sal2)

context  Emp1  inv:
Emp2.allInstances →
     exists(e2 | e2.nm2 = self.nm1 and e2.sal2 = 2*self.sal1)
```

This way the content of class Emp2 -seemingly- is defined as the desired content of class Emp1, with appropriately changed values for the name and salary components. The thing that is wrong with this approach is that this does not constitute a view definition. This approach rather defines two autonomous base classes that are constrained by one another, and it does not reflect the desired result that Emp2 is a *virtual* class with content that is *derived* from class Emp1 *by calculation*. That is, the desired situation is the one where Emp1 can freely change its contents (due to updates performed by users of the database), *irrespective* of the content of Emp2; the content of the virtual class Emp2 should then be deducible on demand and at any given moment by performing a suitable calculation on the content of Emp1. This reflects the situation that a view is basically no more than a named query result.

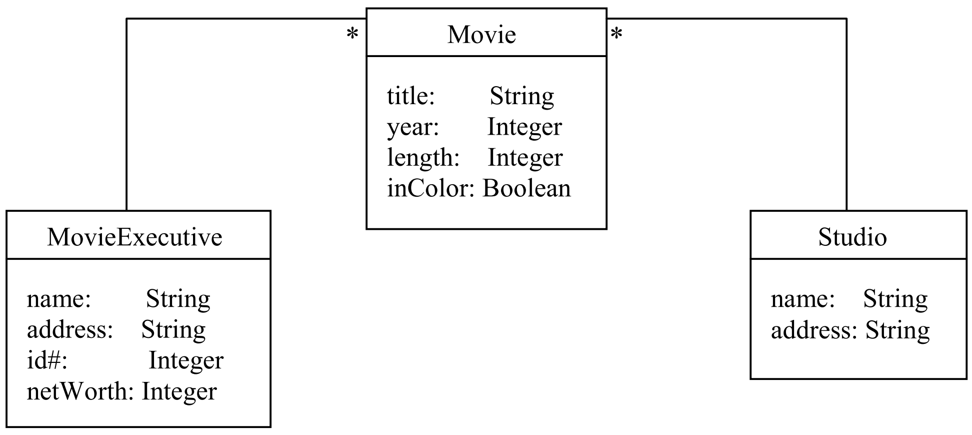
Defining views through constraint definitions is an often made mistake. This mistake, though understandable, leads to a faulty conception of what a view should constitute. A view should constitute a virtual class, completely derivable in terms of existing base classes in the model, at any given moment and on demand. For this reason, we employ the concept of derived class to represent view definitions in UML/OCL.

In the following section we will elaborate on our approach by offering more complex examples where views are constructed from more than one base class.

2. Complex database views in UML/OCL

In this section we will take some examples from the standard database text by [GUW02], and show how we can express complex view representations in UML/OCL. The basic principles developed in the previous section will be applied to a broad class of view definitions, thus demonstrating the general applicability of our approach to express database views in UML/OCL.

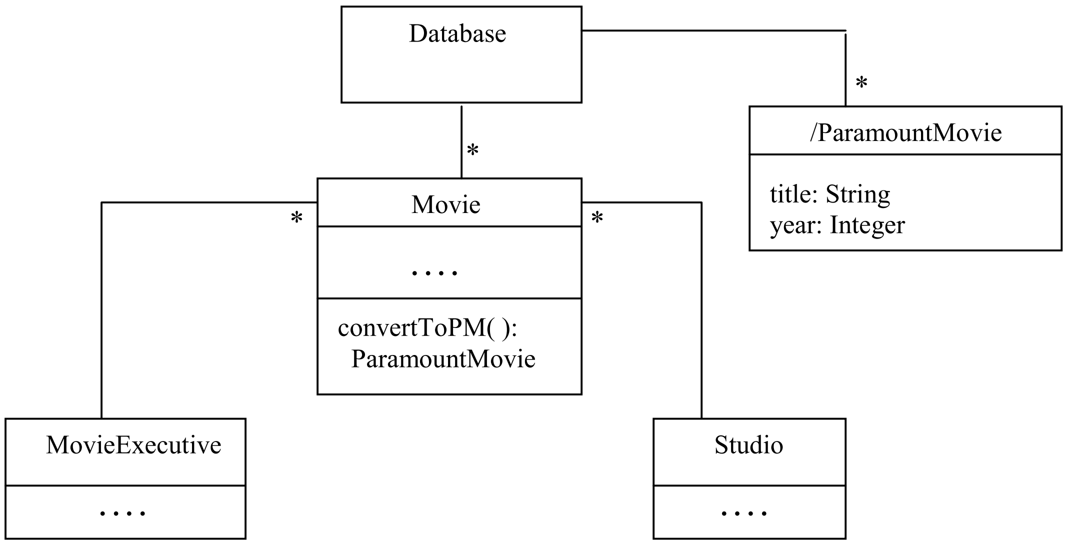
Consider the following base classes (taken from [GUW02])



Assume that we are confronted with the task of defining certain views with these three classes as base classes to be used in the view definitions. Consider, for example, the view ParamountMovie, defined as the class of those movies produced by the Paramount studio. Furthermore, we stipulate that we will only consider attributes title and year in this particular view. This would result, in UML/OCL-terms, in a derived class, where we have to take into account

(1) that we only register attributes title and year
(2) a constraint that the studio name is equal to ‘Paramount’

As in the previous section, we start by introducing a system class called Database with associations to the classes Movie and Studio. We also add the definition of a derived class ParamountMovie representing the desired view definition.



The class Movie is augmented with an operation called convertToPM, which takes a movie-object as input and results in a corresponding object from the class ParamountMovie. This operation is defined in the following OCL-specification

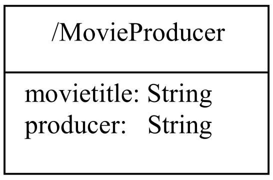
```
context   Movie::convertToPM( ): ParamountMovie
post:     self.convertToPM.title = self.title  and
          self.convertToPM.year  = self.year
```

Furthermore, the system class Database is equipped with the following invariant expressing

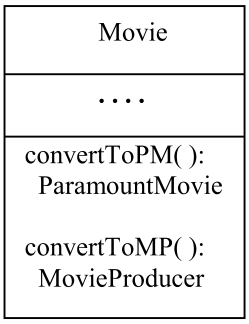
(1) that the derived class ParamountMovie is the result of a calculation (invoking convertToPM) performed on the base class Movie
(2) that the only movie-objects in the database are those obtained from the links starting from the database-object *self.*

```
context  Database  inv:
self.ParamountMovie = self.Movie→
collect(m:Movie | m.studio.name='Paramount'| m.convertToPM)
and
Movie.allInstances  = self.Movie
```

We now treat a slightly more complex view definition involving input from more than one base class. Consider the view MovieProducer defined as the class of objects consisting of movie titles combined with the name of their producer. This view is seemingly more complex than the previous cases, since it involves input from more than one base class. Basically, however, we can achieve our goal in much the same manner as in the previous cases. The derived class MovieProducer is defined by the following model type



and again we assume a database association from the system class Database to this derived class. Input classes for this derived class are the two base classes MovieExecutive and Movie. In this case, however, there already exists a link from the Movie class to the MovieExecutive class. Hence, a single movie-object will suffice to provide information concerning both the movie- and the corresponding movieExecutive-object. We therefore augment the Movie class with an extra operation convertToMP, which converts a movie-object to the corresponding movie producer object



The operation convertToMP is defined in the following OCL-specification

```
context   Movie::convertToMP( ): MovieProducer
post:     self.convertToMP.movietitle = self.title  and
          self.convertToMP.producer = self.MovieExecutive.name
```

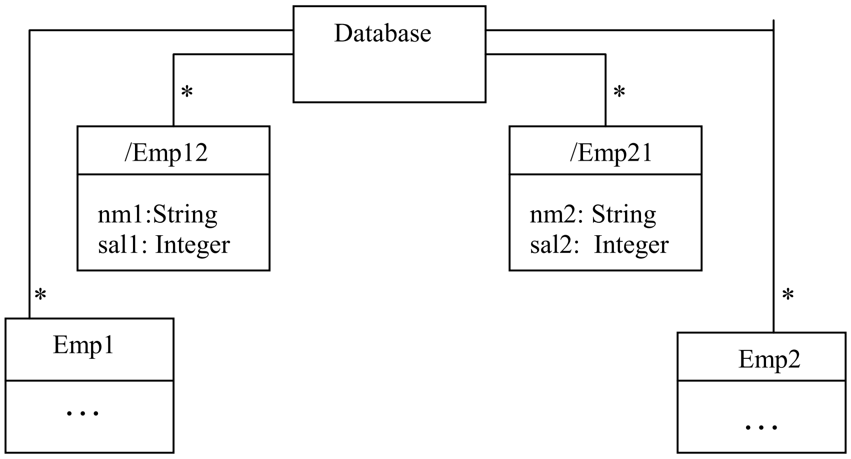
Furthermore, the class Database is augmented with the following invariant expressing that the derived class MovieProducer is the result of a calculation (invoking convertToMP) performed on the base class Movie

```
context  Database  inv:
self.MovieProducer = self.Movie→
                        collect(m:Movie | m.convertToMP)
```

Yet another example concerning inputs coming from more than one base class will be discussed below. This example deals with two input classes that are more or less non-related, in the sense that one base class cannot necessarily be reached from another base class through a sequence of object navigations (as was the case in our previous example concerning multiple class inputs). Consider the following example pertaining to employees, where the classes Emp1 and Emp2 are two base classes, the first with attributes nm1 and sal1, whereas the second class has attributes nm2 and sal2. Assume that we wish to define a view (a derived class), say Emp, consisting of those employee objects with a name value belonging to both the classes Emp1 and Emp2, and with a salary equal to the sum of the two corresponding salary values in class Emp1 and class Emp2, respectively. We assume that name values are unique in both Emp1 and Emp2. Furthermore, we will equip the derived class Emp with attributes nm and sal. Our starting situation deals with two classes

Emp1
nm1: String
sal1:Integer
Emp2
nm2: String
sal2: Integer

In order to achieve our derived class Emp as defined above, we define two auxiliary classes (both of them derived classes): class Emp12 will consist of the Emp1-objects that can be associated to an Emp2-object, and class Emp21 will consist of those Emp2-objects that can be associated to an Emp1-object. Again, this will be done by also introducing a system class called Database with association links to the base classes Emp1 and Emp2



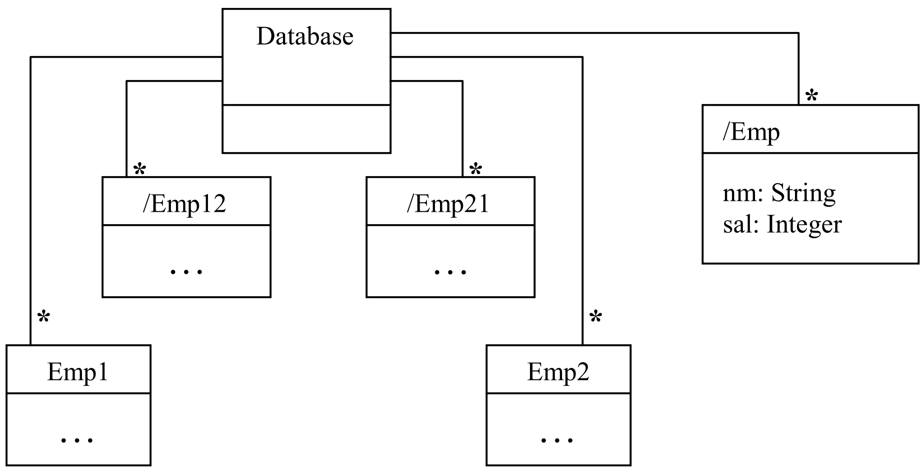
Derived classes Emp12 and Emp21 are defined as follows

```
context  Database  inv:
self.Emp12 = self.Emp1 →
             select(e1:Emp1| self.Emp2 →
                             exists(e2:Emp2| e1.nm1=e2.nm2))
and
self.Emp21 = self.Emp2 →
             select(e2:Emp2| self.Emp1 →
                                exists(e1:Emp1| e1.nm1=e2.nm2))
```

We can now link an Emp1-object to an Emp2-object by introducing an operation, called getE2, within the class Emp12 by the using following constraint

```
context   Emp12::getE2:Emp21
post:     self.getE2.nm2=self.nm1
```

The actual derived class, called Emp, that we are aiming for can now be defined as follows



Within the context of the class Emp12 we introduce an additional operation, called convertToEmp, which links an Emp12-object to an Emp-object

```
context   Emp12::convertToEmp:Emp
post:     self.convertToEmp.nm = self.nm1  and
          self.convertToEmp.sal = self.sal1 + self.getE2.sal2
```

Finally, using the following constraint, we can define the derived class Emp by

```
context  Database  inv:
self.Emp =  (self.Emp12 →
             collect(e1:Emp12| e1.convertToEmp))
             → asSet
```

This example demonstrates that we can also represent a view constructed from two more or less independent base classes. Again, our approach consists of first defining a database system class, and then defining suitable conversion functions to obtain suitable links from base classes to derived classes. An appropriate calculation on the database level using constraints finally yields the desired view definition.

3. Intermediate discussion: generality of our approach

From the three examples offered in the previous sections, we can see how view definitions in general can be expressed in UML/OCL-notation. We have adopted a very broad class of view definitions involving

(1) attribute renaming
(2) attribute value computations
(3) constraint definitions
(4) inputs coming from more than one base class, but related through object navigation
(5) inputs coming from more than one base class, and not related through object navigation

Our approach has adopted the following steps

(1) define a database class as the root class of the system
(2) employ suitable conversion functions in the base classes to coerce from base objects to view objects
(3) within the framework of the database class, define the adequate invariant to express the *computation* of the desired database views

We are now faced with the problem to formulate the generality of our approach. Basically speaking, a view is nothing else than a (named) query. Each view state can be regarded as the answer to an associated query. Any query, in turn, can be associated to a corresponding view definition. Hence, to support the claim that we have offer a general framework for representing relational views in UML/OCL, we will we have to demonstrate, in some sense, that we can express an arbitrary relational query in UML/OCL. If, for example, we can show that OCL has at least the expressive power of the relational algebra, then we can indeed claim that OCL has adequate expressiveness for representing general view constructions. In our case, we will restrict ourselves to views without aggregates and grouping constructions. Our restriction is made, because we want to concentrate on basic results. It will not be a problem, however, to extend our results to views including aggregates and grouping; this can be done in a natural and straightforward manner.

In the section 5, we will demonstrate that OCL actually has the expressive power of the relational algebra (without aggregates and grouping). We shall offer our result by showing how to represent database joins, as well as general Cartesian products in UML/OCL. This problem concerning expressiveness of UML/OCL as a relational query language has been addressed by various researchers [MC99, AB01]; in section 5 we will also discuss the adequacy and correctness of their results and relate those results to our own.

4. Joins and Cartesian products in OCL

This section is concerned with offering a solution to deal with database joins and product spaces in OCL based on the view concept. First we will consider the concept of database join.

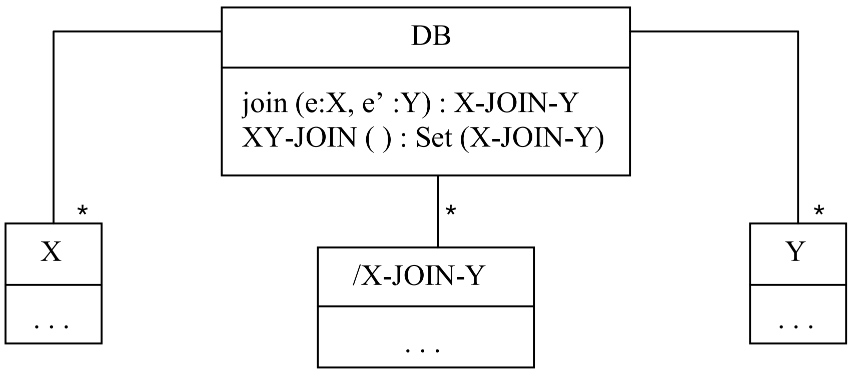
We will then proceed by offering a treatment of Cartesian product spaces represented in UML/OCL and how we can alternatively base joins on this concept of product space.

Joins

A join (also natural join) pairs tuples in two relations that match in some way. This match of two relations X and Y, denoted by X-JOIN-Y, consists of pairing those tuples from X and Y, such that they agree in whatever attributes are common to the schemas of X and Y. More precisely, let a1 , a2 , ..., an be all the attributes that are common in the attribute sections of both X and Y. Then a tuple x and a tuple y are successfully paired if and only if x and y agree with each of the attributes a1 , a2 , ..., an . The result of the pairing consists of a so-called joined tuple which has as attributes all of the attributes obtained from taking the union of the attributes from X and Y. The joined tuple by definition agrees with x for all of the attributes of x, and it agrees with y for all of the attributes of y. When x and y are successfully paired, the joined tuple will agree with both tuples on the attributes they have in common. Joining is an essential operation in databases, and we will show how the natural join can be represented in UML/OCL.

Consider two classes X and Y, represented as model types in a UML diagram (where we have omitted the attributes declarations)

We shall represent the Join by a derived class called X-JOIN-Y



The join condition involves inspection of overlapping attributes occurring in classes X and Y. (We shall assume that in the case of coinciding attributes, the corresponding domain types in classes X and Y are the same.)

```
context   DB  def:
```

```
let    D  =  X.attributes → intersect(Y.attributes)
let    D1 =  X.attributes – D
let    D2 =  Y.attributes – D
```

The attributes of X-JOIN-Y are, by definition, equal to

```
X.attributes → union(Y.attributes)
```

The join of two objects is now specified by

```
context   DB::join(e :X, e′:Y)   : X-JOIN-Y
pre :     D   → forall (d:String | e.d=e′.d)
post:     D1  → forall (d:String | join(e,e′).d = e.d )
          D   → forall (d:String | join(e,e′).d = e.d )
          D2  → forall (d:String | join(e,e′).d = e′.d)
```

This operation constitutes the join calculation of two objects taken from class X and Y respectively. We now offer the definition of the XY-JOIN operator which calculates the set of joinable objects taken from classes X and Y

```
context   DB::XY-JOIN( )  : Set(X-JOIN-Y)
post:   result = (self.X →
                   collect(e:X | self.Y → collect(e′:Y | join(e,e′))))
                   → asSet

context   DB  inv:
self.X = X.allInstances  and
self.Y = Y.allInstances  and
self.X-JOIN-Y = X-JOIN-Y.allInstances  and
self.X-JOIN-Y = self.XY-JOIN
```

This last invariant states that X-JOIN-Y constitutes the complete set of instances of the join of X and Y, due to the following two conditions

(1) the derived class X-JOIN-Y is the result of a calculation (invoking the database operation XY-JOIN) performed on the base classes X and Y
(2) the only X-JOIN-Y objects in the database are those obtained from the links starting from the database-object *self*.

There is also a more general approach to defining joins. This approach is based on first defining the notion of the product space of two classes, and subsequently defining the join of these two classes as a subset on the product. The definition below thus offers a somewhat more basic solution in defining the join-concept in OCL.

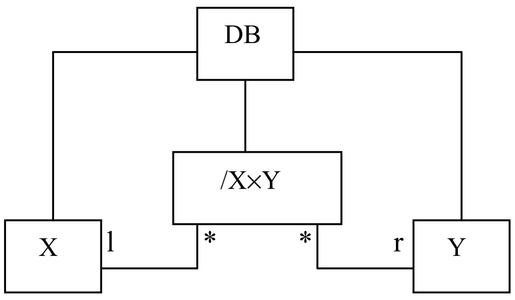
Product spaces in OCL

We now offer a solution to deal with product spaces in OCL. Our solution much resembles the approach we have already adopted to describe joins in OCL. It offers a clean representation of product spaces based on the view concept.

The Cartesian product of two sets X and Y, consists of those pairs that can be formed by taking the first element of the pair to be any element of X, and the second element to be any element of Y. By convention, the first element of the pair precedes the second element. The product of two sets is denoted by X×Y. Should X and Y be relations, then the members of X and Y are tuples, usually consisting of more than one component; the members of X×Y are tuples , with one component for each of the components of the constituent tuples from X and Y. We now show how to represent Cartesian products in UML/OCL.

Again, consider two classes X and Y, represented as model types in a UML diagram (where we have omitted the attributes declarations)

We shall represent the Cartesian product by using a derived class called /X×Y. Again, we will declare a database object DB with associations to the classes X, Y and /X×Y



The derived class X×Y has associations, called l and r, to the classes X and Y respectively.

We now add to the class DB two operations called prod and PROD; prod will indicate the concatenation of two tuples to one tuple, and PROD will deliver the Cartesian product of the two classes X and Y.

```
context  DB   prod(e:X, e′:Y): X×Y
post :        prod(e,e′).l = e  and
              prod(e,e′).r = e′

context  DB   PROD( ): Set(X×Y)
post :   result =  (self.X →
                     collect(e:X | self.Y →
                                collect(e′:Y| prod(e,e′)))
                   → asSet
```

In order to insure that all instances of the classes X and Y should be taken into account, we stipulate the following invariant for the system class DB

```
context   DB  inv:
self.X = X.allInstances  and
self.Y = Y.allInstances  and
self.X×Y = X×Y.allInstances  and
self.X×Y = self.PROD
```

In this manner, X×Y constitutes the complete set of instances of the Cartesian product of X and Y, since

(3) the derived class X×Y is the result of a calculation (invoking PROD) performed on the base classes X and Y
(4) the stipulation that the only X×Y-objects in the database are those obtained from the links starting from the database-object *self.*

Joins revisited

Based on the product space of two classes, we can now offer an alternative definition of the join of two classes. This works as follows.

```
X-JOIN′-Y =
X×Y  →
select(t : X×Y| (t.l.attributes intersect(t.r.attributes)) →
                  forall(d :String| t.l.d=t.r.d))
```

This definition yields the natural join of classes X and Y. Note that this definition of the join-concept is equivalent to the previous join definition given in section 5. By equivalent we mean that each element in the set pertaining to first definition of the join corresponds to exactly one element of the set pertaining to the second definition of the join, and vice versa. Hence, these two sets are not exactly the same (for example the sets are built from different syntactical constructs), but are *isomorphic*. Our second definition has the advantage, however, that it is based on the more simple concept of Cartesian product.

5. OCL as a query language

This section deals with the expressiveness of OCL as a query within the UML/OCL data model. We will show that OCL has at least the same expressiveness as the relational algebra. We will also compare the results of our work with that of other authors concerning this topic. Surprisingly, some authors have achieved results contradicting our own findings. We will discuss adequacy and correctness of their results, and demonstrate some shortcomings in their approach.

The idea of using UML/OCL as a query language has been investigated in various papers [GR97, MC99, AB01]. We will discuss the approaches taken in [MC99, AB01], demonstrating that both papers contain certain shortcomings. We will also show that our solution does not fall prey to these shortcomings, offering an actual basis for OCL as a general query specification language within the context of relational databases.

Feasibility of our approach

Basically, a query language has the same expressive power as the relational algebra, when that language supports the following operations [D00, GUW02]

1. The usual set operations -*union*, *intersection*, and *difference*- applied to relations
2. Operations that remove parts of a relation: *selection* eliminates some rows (tuples), and *projection* eliminates some columns
3. Operations that combine the tuples of two relations, including *Cartesian product*, which pairs the tuples of two relations in all possible ways, and the *join operation*, which selectively pairs tuples from two operations with the same attribute values for those attributes that belong to both relations
4. An operation called *renaming* that does not affect the tuples of a relation, but changes the name of the relation schema; i.e., the names of the attributes of the relation, or the name of the relation itself.

We claim that we can represent all operations mentioned above, hence demonstrating that UML/OCL offers sufficient support for general query specifications as offered in the relational data model. In the context of our approach, the problem of representing relational algebra operations will be reduced to the problem of defining a suitable derived class that can handle the corresponding relational operation.

We shall now discuss the four categories of operations as mentioned above.

1. The set operations -*union*, *intersection*, and *difference*:
These operations are standard in OCL, be it that these operations are only defined when the underlying types of the sets involved are the same [WK99]. In the case of relations, that is exactly the kind of restriction what we wish to employ: in the relational algebra (and also in SQL) we can only take the union, intersection, and difference of relations that have the same component structure (i.e. the same underlying type).

2. The operations *selection* and *projection*:
Selection is a standard construction in OCL with the same effect as the selection operation in the relational algebra. Projection in the relational algebra involves selecting a subset of the set of attributes of the relation in question, and then taking only those components into account that pertain to that particular subset of attributes. This operation is not directly supported in OCL (only projection on a single attribute is supported), but can be simulated in a series of steps. In our approach, the context of the problem would be to define a derived class C' containing a subset of the attributes of some base class C. This can easily be done by adding an operation to the class C, called convertToC' (say), that has as its sole task to convert an object from C to an object in C'. The definition of convertToC' is offered by stipulating that

```
self.convertToC'.a = self.a
```

for each attribute `a` occurring in the subset of attributes in question.

3. The operations *Cartesian product* and *join*:
The previous section was entirely devoted to the representation of Cartesian products and joins in UML/OCL. We have shown that for any given two classes X and Y, we can represent the Cartesian product and the join of these two classes as derived classes X×Y and X-JOIN-Y, respectively.

4. The *renaming* operation:
Representing the renaming operation in the context of our approach, would reduce to the problem to define a derived class C' containing a renamed set of the attributes of some base class C. This can easily be done by adding an operation to the class C, called convertToC' (say), that has as its sole task to convert an object from C to an object in C'. The definition of convertToC' is offered by stipulating that

```
self.convertToC'.a' = self.a
```

for each attribute combination `a', a` where `a'` is the renamed version of the attribute `a`.

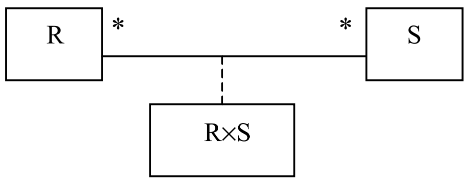
We have, hence, shown that it is possible to represent the expressive power of the relational algebra within the framework UML/OCL data model employing the concept of derived class.

We now proceed by discussing the approaches taken in other papers to represent the relational algebra in the UML/OCL data model.

The other approaches: adequacy and correctness aspects

Basic results regarding expressiveness of OCL as a query language are mainly found in the papers [MC99, AB01]. In [AB01] an alternative solution for representing Cartesian products is offered, aimed at disclaiming the result offered in [MC99] that OCL does not have the expressive power of the relational calculus. We will first treat the approach offered in [AB01], and then discuss the treatment offered in [MC99].

The approach taken in [AB01] has as its major shortcoming (as the authors themselves also indicate) that the solution offered is rather complex, leading the authors to propose certain language extensions to OCL as an alternative solution to the problem. Basically, [AB01] show that by using a suitable *association class* it is possible to represent Cartesian products of class types. For example, consider two arbitrary class types R and S (i.e., non-related by any given association in the model), then it is possible to introduce an association class R×S as follows



A constraint is then added pertaining to this particular association class R×S

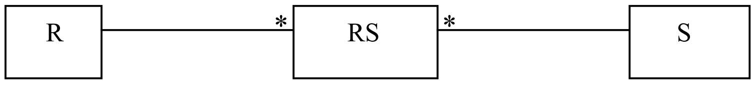
```
context  R×S  inv:
 R×S.allInstances → size = (R.allInstances→size) *
                            (S.allInstances→size)
```

This constraint establishes the desired properties of the association by indirectly saying that each of the elements of classes R and S are indeed represented in some tuple of the class

R×S. In this way [AB01] claim they have R×S as a correct representation for the Cartesian product of R and S. We now proceed by offering our comment on the adequacy and correctness of this solution. First of all, it should be noted that this solution demands the introduction of a separate new model type (like R×S), each time a Cartesian product is needed. In the second place, the constraint offered above is not sufficient to represent all relevant properties of Cartesian products. For example, using this combination of an association class R×S and the accompanying constraint offered above, there is no difference to be made between a Cartesian product R×S and a Cartesian product S×R: hence, the anti-symmetry property of Cartesian products is lost. One could argue that the ordering of the two components is implicitly given by navigation on the class names R and S, but this still leaves another problem to be solved. Another problem posed by the solution is that it is not clear how to represent the Cartesian product of a class with itself (reflexivity); i.e., how do we represent a Cartesian product like R×R, and how do we distinguish between the ordering of the two components (since navigation through class names will not help here)? The main problem both with anti-symmetry and reflexivity is that no distinction has been made by projecting on a left- and a right component when traversing from the product to its component classes; a situation that our solution (cf. section 4) does take into account. Furthermore, in [AB01] it is argued that the solution they have offered (especially due to the rather cumbersome way of dealing with projections on classes, as well as joins) suggests that it might be better to opt for a language extension of OCL to deal with Cartesian products. We conclude by saying that the solution offered in [AB01] is not adequate (and also not completely correct) to deal with Cartesian products, due to the introduction of new model types, its indirect way of dealing with product definitions, and a not completely correct treatment of relevant properties (such as reflexivity and anti-symmetry) of Cartesian products.

The other approach that we will discuss, is found in [MC99], where it is claimed -on theoretical grounds- that OCL cannot have the expressive power of the so-called relational calculus. Since the expressiveness of the relational calculus and the relational algebra are equivalent [D00], there seemingly is an inconsistency between the two results offered in [AB01] and [MC99]. We shall show that the results in [MC99] contain certain flaws, thus leaving the desired result that OCL has the expressive power of the relational calculus open for validation.

Again, assume that we have two arbitrary class types R and S (i.e., non-related by any given association in the model). In [MC99] a modeltype RS is then introduced



augmented with the following (database-, or root-) constraint

```
context  Database  inv:
R.allInstances →
   Union(S.allInstances) →
      forall(r,s:oclAny |
        if r.oclType.name = s.oclType.name
        then true
        else RS.allInstances → exists(t:RS | t.R=r  and  t.S=s))
```

where `oclAny` denotes the root of the context of the UML model [WK99].

This constraint indirectly says that each of the elements of classes R and S are indeed represented in some tuple of the class RS. In this way [MC99] claim they have RS as a correct representation for the Cartesian product of R and S. We now proceed by offering our comment on the adequacy and correctness of this solution. First of all, it should be noted that this solution demands (as also was the case in [AB01]) the introduction of a separate new model type (like RS), each time a Cartesian product is needed. In the second place, the constraint offered above is not sufficient to represent all relevant properties of Cartesian products; the combination of an association class RS and the accompanying constraint offered above yields no difference between a Cartesian product RS and a Cartesian product SR. Hence, also in this case the anti-symmetry property of Cartesian products is lost. Reflexivity also poses a problem: how to represent the Cartesian product of a class with itself? (The definition of RR is not at all clear from the definition of Cartesian product in the setting of [MC99].) The main problem (as was the case in [AB01]) both with anti-symmetry and reflexivity is that no distinction has been made by projecting on a left- and a right component when traversing from the product to its component classes; a situation -again- that our solution (cf. section 4) does take into account. We conclude by saying that the solution offered in [MC99] is not adequate (and also not completely correct) to deal with Cartesian products, due to the introduction of new model types and a not completely correct treatment of relevant properties (reflexivity and anti-symmetry) for Cartesian products. The conclusion drawn in [MC99] that OCL does not have the expressive power of the relational calculus, is partly based on their treatment of Cartesian product (introducing complexities when trying to define arbitrary projections on attributes). This paper proposes a contrary result, by demonstrating (cf. previous section) that by offering a different treatment of products (and joins) of classes -based on derived classes rather than introduction of certain new model types- that OCL has the expressive power of the relational algebra (and hence the relational calculus).

6. Summary

Information systems, and in particular information systems based on databases and their applications, rely heavily on sound principles of analysis and design. This paper focuses on particular principles of analysis and design related to database applications. Our main aim is to show how the notion of a derived class in UML can be given a precise semantics in terms of OCL. We then demonstrate that the notion of a relational database view can be correctly expressed as a derived class in UML/OCL. A central part of our investigation concerns the generality of our manner of representing database views in OCL. An important problem that we address in this respect is the representation of Cartesian product spaces and relational joins. Joins are often essential in view definitions, and we shall demonstrate how we can express products and joins within the current framework of UML/OCL language by employing the notions of derived class. As a consequence, OCL will be shown to be equipped with the full expressive power of the relational algebra, offering support for the claim that OCL can be useful as a general query language within the framework of the UML/OCL data model.

Acknowledgements

Special thanks go to my colleague Bert de Brock for proof reading my solutions, and helping me out with various problems regarding the connection between OCL and the relational data model.

References

[AB01] Akehurst, D.H., Bordbar, B.; On Querying UML data models with OCL; «UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, 2001, Proceedings. Lecture Notes in Computer Science 2185, Springer, 2001

[BB00] Balsters, B., de Brock, E.O.; A general framework for the design of federated database systems; SOM Research Series 01A26, University of Groningen, 2000

[BBZ93] Balsters, B., de By, R.A., Zicari, R.; Sets and constraints in an object-oriented data model; Proceedings Seventh European Conference on Object-Oriented Programming (ECOOP), Kaiserslautern, Germany, July, 1993.

[BP98] Blaha, M., Premerlani, W.; Object-oriented modeling and design for database applications; Prentice Hall, 1998

[BV92] Balsters, B., de Vreeze, C.C.; A semantics of object-oriented sets; Third International Workshop on Database Programming Languages (DBPL; eds. Abiteboul, Kannelakis), Morgan Kaufmann Publishers, California USA, 1992.

[D00] Date, C.J.; An introduction to database systems; Addison Wesley, 2000

[DH99] Demuth, B., Hussmann, H.; Using UML/OCL constraints for relational database design; «UML»'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, 1999, Proceedings. Lecture Notes in Computer Science 1723, Springer, 1999

[DHL01] Demuth, B., Hussmann, H., Loecher, S.; OCL as a spevcification language for business rules in database applications; «UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, 2001, Proceedings. Lecture Notes in Computer Science 2185, Springer, 2001

[EP00] Eriksson, H., Penker, M.; Business modeling with UML; OMG Press, Wiley, 2000

[GR97] Gogolla, M., Richters, M.; On constraints and queries in UML; Proceedings UML'97 Workshop "The Unified Modeling Language – Technical Aspects and Apllications", 1997

[GUW02] Garcia-Molina, H., Ullman, J.D., Widom, J.; Database systems; Prentice Hall, 2002

[MC99] Mandel, L., Cengarle, M.V.; On the expressive power of OCL; FM'99 – Formal Methods, World Congress on Formal Methods in the Development of Computing Science; Lecture Notes in Computer Science 1708, Springer, 1999

[OMG99] Object Management Group; Unified Modelling Language Specification, version 1.3; June 1999; http://omg.org

[WK99] Warmer, J.B., Kleppe, A.G.; The object constraint language; Addison Wesley, 1999