The Functorial Data Model - An Extension to Functional Databases

D. A. Nelson & B. N. Rossiter
Computing Science
University of Newcastle upon Tyne, NE1 7RU

M. A. Heather
Department of Law
University of Northumbria at Newcastle, NE1 8ST

May 1994

Abstract

Owing to the recognised deficiencies in relational databases, a new generation of object-oriented database systems are emerging. The major deficiency of many of these new systems is in their formal basis, and in aspects such as views and closure. To aid this cause, a model has been developed based on category theory, known as the functorial, or the categorical product, model. This paper will introduce an example database model using the functorial model, comparing it to the DAPLEX functional model developed in the early eighties.

About the author

David Nelson is a Ph.D. student in the Department of Computing Science with research interests in databases and category theory.

Nick Rossiter is lecturer in the Department of Computing Science with particular interests in databases and systems analysis.

Michael Heather is senior lecturer in law where he has been responsible for computers and law since 1979.

Suggested Keywords

Databases, functional model, object-relational model, category theory, functorial model, student example.

1. Introduction

There are three main concepts which any database model must support in order for it to be a fully defined system: structure, which is the underlying schema behind the database, usually the objects within that database and all inter-object and intra-object associations between them, and the data definition language; rules, which are any restrictions on that schema definition, most notably the normalisation and integrity constraints in relational databases; and manipulation, which is the query language.

Currently, the most widely used database system is the relational model [Date90], but the relational model is based on tables (relations) which is an unnatural concept when applied to real world problems. The newer generation databases such as the object-oriented model [Atkinson90], the nested relational model [Codd79, Roth88] and the functional model [Shipman81] have attempted to alleviate the problems of atomicity by adding more natural association abstractions. The most recent database model, the object-oriented model, provides probably the best data structure for handling real world data, but the object-oriented model introduces problems of its own, namely that it strongly lacks an 'universal' formal basis [Kim90] and other concepts such as a manipulation language as rich as the relational algebra and a view mechanism.

To attempt to add these missing concepts to the object-oriented database model, a theoretical model is being developed. This is known as either the functorial or the categorical product model [Rossiter94], and is based on using textbook categorical constructions to provide a formal basis. Because the functional database model provides what is probably the most natural query language, based on function composition, it is intended that a manipulation language can be developed based on DAPLEX, thereby providing a conceptually natural query language for the object-oriented model.

This paper will therefore first introduce the functional database model, using the DAPLEX model developed by Shipman. An example will be given in both a relational SQL model, and in the functional model. Then, a brief introduction to category theory will be given, to explain the concepts which are used in the functorial model. The functorial model is then introduced, and finally an example using this model is provided.

2. Functional Database Models

The functional model we will look at is that of DAPLEX [Shipman81]. This is the most widely used and well known functional model produced.

2.1 DAPLEX

'The DAPLEX language is an attempt to provide a database system interface which allows the user to more directly model the way he thinks about the problems he is trying to solve' [Shipman81].

DAPLEX is a data definition and manipulation language for database systems based on the functional data model. It claims to be a 'conceptually natural' database interface language, being based on two main concepts, the entity and the function.

Entities provide the type definition of an object within the database, e.g. a Student entity is a Person, where a Person is an ENTITY, an entity being the base type within the system. Within an entity, functions represent its values and its associations to other entities, where a function can return a single valued set, represented by a single headed arrow →, or a multi-valued set, represented by →>. For example:

DECLARE name(Student) → STRING

defines a function on a Student which returns his or her name as a character string, whereas:

DECLARE courses_completed(Student) →> Section

represents a relationship of a Student entity to the 'many' Section entities which a particular student has completed.

Functions can be of two main sorts. There are the data definition functions represented by the declare statement, and the derived data functions, which are used for defining inverses of existing functions, views on the database, aggregation operations, etc., on existing values. For example:

```
DEFINE gradePointAverage(Student) →
        AVERAGE(grade(Student, Section)
                OVER courses_completed(Student))
```

defines a new derived function for a Student which returns that student's average grade for all courses which he or she has completed.

The derived data also provides a constraint and trigger facility, for instance:

```
DEFINE student(Class) → INVERSE OF class(Student)

DEFINE TRIGGER overbooked(Class) → COUNT(student(Class))>45
        SEND MESSAGE(head(dept(Class)), 'Overbooked:', title(Class))
```

This uses the INVERSE OF operator to define a function to return the name of a Student within a particular Class, and then defines a trigger which operates whenever the number of students in that class is greater than 45, sending a message to the head of the department for that class.

Probably the most notable use of derived functions is that they can provide separate user views of a database. Using derived functions, the view of the database can be altered to suit the taste of the user. Derived functions provide the amount of access to the database that the user requires, both for convenience and security. The new user

view is defined in a different name space (intension) from the old view, to only give access to the new view. The PERFORM .. USING construct must be used, which defines how to perform any updates to derived data, and what that update actually means.

The final section of the DAPLEX system is its manipulation, which is carried out by the means of function composition, e.g.

```
FOR EACH Student
        SUCH THAT dname(minor(Student)) = 'Maths'
PRINT name(Student)
```

The manipulation system provides the means of querying, and of update and deletion operations, for example, to change some of the data for the student 'Mary' we could have:

```
FOR THE Student SUCH THAT name(Student) = 'Mary'
BEGIN
        LET minor(Student) = THE Department
                SUCH THAT name(Department) = 'Mathematics'
        EXCLUDE courses_completed(Student) = THE Section
                SUCH THAT sec_no = 1
        INCLUDE courses_completed(Student) =
        {
                THE Section SUCH THAT sec_no = 2,
                THE Section SUCH THAT sec_no = 4
        }
END
```

DAPLEX is a navigational language, all operations are carried out by functions, compared to the subsetting approach of the relational model. The functorial model introduced later is based on both approaches, navigational through the definitions of categories and the use of morphisms (arrows) for representing associations, and subsetting for the current query approach of creating subobjects from objects.

The declaration of the database entities and functions are done at the intensional level of the database, the values of functions comprise the extension of the database. But the intension level is more complex than just described, as a metadata view of the database is provided, whereby the intension of the database can be queried, using the normal manipulation commands. This gives a two level database architecture, represented by the metadata and the data levels.

There has been much research into functional database models in Britain. Kulkarni and Atkinson [Kulkarni87, 88] have developed EFDM, the Extended Functional Data Model, which is an extension of DAPLEX including concepts such as a better view facility and graphical navigation and querying, using the language PS-Algol [Atkinson83], which is a variant of the Algol language with persistent extensions. Gray and Paton [Gray88] have been looking at the development of the DAPLEX system using Prolog, and have lately extended their work to look at the role of this

system for object-oriented databases [Gray92]. Finally, research by Sutton and King [Sutton92] has been looking at the integration of a form of modal logic in functional database systems, for handling incomplete information.

The next two sections will introduce the example, firstly using the entity-relationship model and the SQL data definition and manipulation language to show the database in the relational model, and then a functional version is given using a functional diagram along with DAPLEX definition and manipulation commands.

2.2 The Relational Example

The example that is going to be used is an adaptation from the functional example given in Elmasri's book [Elmasri89]. It has been simplified slightly for the purposes of this paper, and translated both into a relational and a functorial model.

For the relational version, we will firstly use an entity-relationship diagram [Layzell89], to give a diagrammatic representation of the database. The table types are then given, and three searches are provided in SQL [Date90] to show a broad range of queries. The natural language versions of these queries are as follows:

1. Print the names of all students who have minored in the mathematics department;

2. Print the names of all courses which student 'Mary' takes;

3. Print the names of all people who are male.

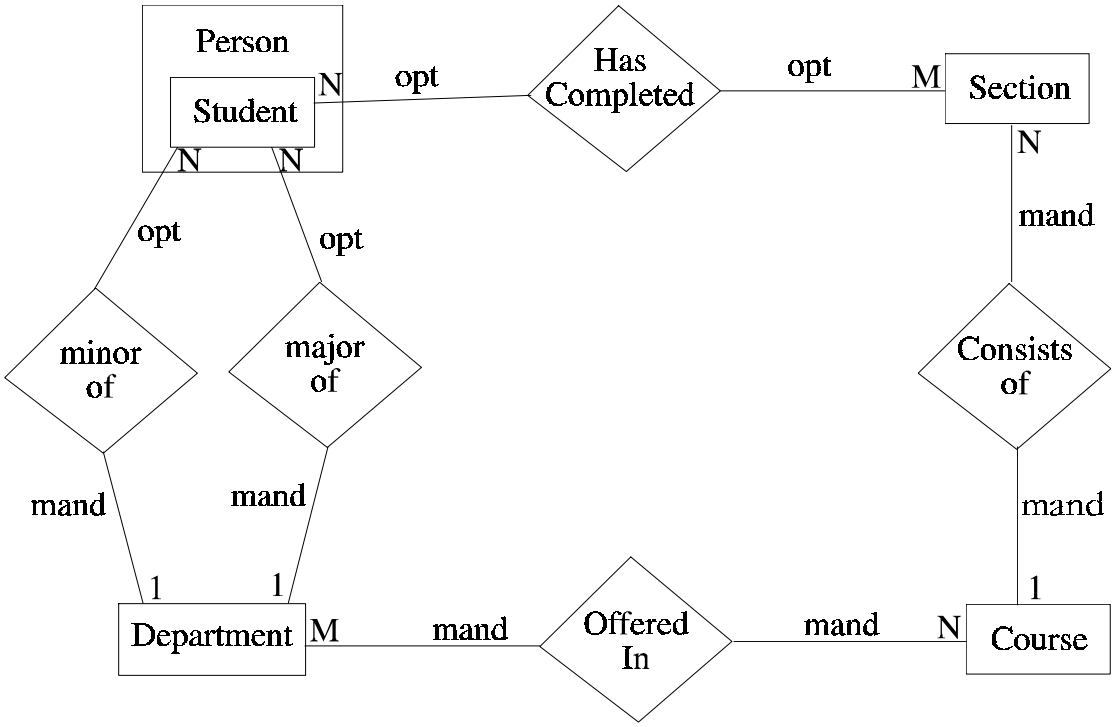

Fig 1 : Entity-Relationship Diagram

Briefly, the entity-relationship (E-R) diagram in figure 1 shows five entities (Person, Student, Section, Course, Department) and the binary relationships between them, which each have a name, a functionality and a membership class. The Student entity is shown as contained within a Person to indicate subtyping, or inheritance.

The next stage is to convert this diagram into relational table types. Firstly, each entity in the E-R model becomes a relation, or table, in the relational model. The two many-to-many relationships each become tables, and the minor and major relationships both become tables in the relational model, as the membership class of the 'many' part is optional. Thus, the table types are as follows:

Person = ssn + name + bdate + sex

Student = ssn* + class

Department = dname + dphone + doffice

Course = c_no + cname + cdesc

Section = sec_no + c_no* + year + qtr

OfferedIn = dname* + c_no*

HasCompleted = ssn* + sec_no* + grade

Minor = ssn* + dname*

Major = ssn* + dname*

The table types show the names of the tables (which begin with a capital letter), and the set of attributes within each table (separated by plus symbols). Underlined attributes identify the primary key of that table, and an asterisk identifies the foreign key, i.e. the attribute or attributes which relate that table to the primary key of another table.

Note the main drawbacks with the relational model. Extra table types have to be defined to deal with the atomicity and normalisation restrictions, and all information about associations between tables has been lost apart from primary and foreign key definitions. The functional model does not suffer from these drawbacks.

There is also a problem with the representation of subtypes in the relational model, namely how they should be represented. The best way seems to be to just store the specialisation attributes of the subtype (Student) in the subtable, and still use the generalised class (Person) for storing the other information. Alternatively, we could store all information for the supertype within the subtypes table, so the Student table would contain all the information in the Person table, and then a Student would not have an entry in the Person table, to save duplication of information. This though causes querying anomalies, as shall be seen below in the following SQL queries.

1. Print the names of all students who have minored in the mathematics department:

```
SELECT name FROM Person, Minor
WHERE Person.ssn = Minor.ssn AND dname = 'Maths'
```

2. Print the names of all courses which student ‘Mary’ takes:

```
SELECT cname FROM Course, HasCompleted, Person, Section
WHERE Course.c_no = Section.c_no AND
        Section.sec_no = HasCompleted.sec_no AND
        Person.ssn = HasCompleted.ssn AND
        Person.name = ‘Mary’
```

3. Print the names of all people who are male:

```
SELECT name FROM Person WHERE sex = ‘M’
```

The first query is a simple join on the Person and minor tables, creating a new table with a single attribute ‘name’ where its tuples are the names of all those students who have minored in ‘Maths’. Note that although the natural language query asks for students, the way the subtypes are represented makes it sufficient to directly query on the Person table, because if a person is not a student then that persons key attribute is not in the Minor table, and so the join excludes this person from the relationship. If a student had all information stored in the Student table, then this query would have been incorrect, we would have had to alter it to:

```
SELECT name FROM Student, Minor
WHERE Student.ssn = Minor.ssn AND dname = ‘Maths’
```

The second query raises some interest, especially in the number of tables that need to be joined in order to perform the query. This is quite a complex query in relational systems, even though it is intuitively simple to visualise. The final query once again shows the benefit of how the example handles subtyping, in that it will create a table containing all names of people, and thus will include all students, whose sex is male. If we had not handled subtyping in this way, then the query would have been more complex as we would have had to search both the Person and the Student tables.

We now proceed onto the functional example.

2.3 The Functional Example

Figure 2 shows the equivalent functional diagram to the previous E-R diagram. It shows the five entities, Person, Student, Department, Course, Section, and all functions within them, where the name of the arrow represents the name of the function, and the entity type at the head of the arrow represents that function’s type. The single headed arrows are single-valued functions, i.e. return one entity, and the double headed arrows are multi-valued functions, i.e. return a set of entities. The relationship between Student and Person is shown as an isA arrow, and note also the function grade(Student, Section), which defines a function which ‘joins’ two entities together, in effect giving a function with a composite domain.

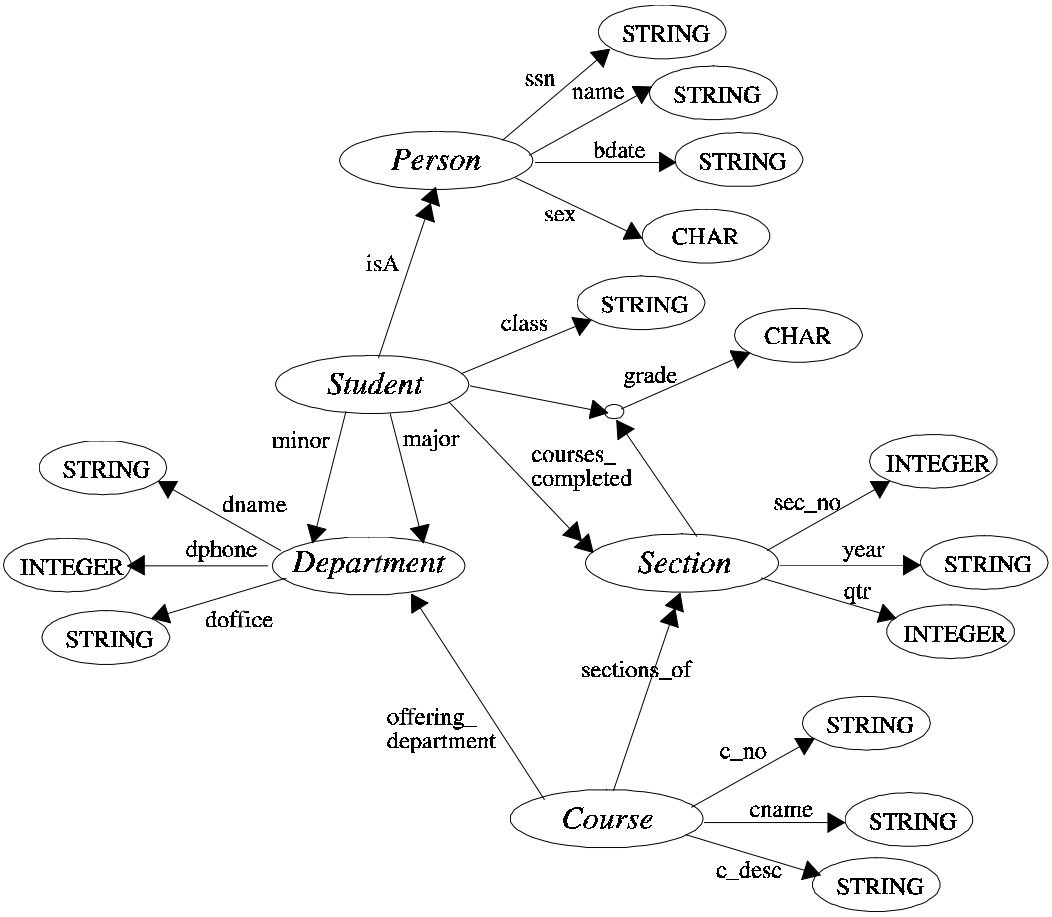

Fig 2 : The Functional Diagram

The corresponding data definition statements are easy to infer from the diagram, but they are listed below for convenience.

declare Person() ⇒> ENTITY
declare ssn(Person) ⇒ STRING
declare name(Person) ⇒ STRING
declare bdate(Person) ⇒ STRING
declare sex(Person) ⇒ CHAR

declare Student() ⇒> Person
declare class(Student) ⇒ STRING
declare minor(Student) ⇒ Department
declare major(Student) ⇒ Department
declare courses_completed(Student) ⇒> Section

declare Section() ⇒> ENTITY
declare sec_no(Section) ⇒ INTEGER
declare year(Section) ⇒ STRING
declare qtr(Section) ⇒ INTEGER

```
declare grade(Student,Section) ⇒ CHAR

declare Department() ⇒>> ENTITY
declare dname(Department) ⇒ STRING
declare dphone(Department) ⇒ INTEGER
declare doffice(Department) ⇒ STRING

declare Course() ⇒>> ENTITY
declare c_desc(Course) ⇒ STRING
declare c_no(Course) ⇒ STRING
declare cname(Course) ⇒ STRING
declare offering_department(Course) ⇒ Department
declare sections_of(Course) ⇒>> Section
```

Note the differences to the relational model. It is no longer required to create extra tables, multi-valued functions allow a many-to-many, or one-to-many, relation to be defined, and also there is no longer a need to define primary and foreign keys: all associations between entities are functions, and so the key definitions are no longer required. Later extensions to DAPLEX such as P/FDM [Gray92] have provided key definitions, to aid in the efficiency of persistent data access.

The DAPLEX queries are as follows:

1. Print the names of all students who have minored in the mathematics department:

```
FOR EACH Student SUCH THAT dname(minor(Student)) = 'Maths'
PRINT name(Student)
```

2. Print the names of all courses which student 'Mary' takes:

```
DEFINE sections_in(Section) ⇒ Course AS INVERSE OF
        sections_of(Course)

FOR EACH Course
        SUCH THAT FOR SOME Student
                name(Student) = 'Mary'
PRINT cname(sections_in(courses_completed(Student)))
```

3. Print the names of all people who are male:

```
FOR EACH Person
        SUCH THAT sex(Person) = 'M'
PRINT name(Person)
```

The first query is very simple, but note that instead of using the Person entity, the Student entity can now be used. This is because the DAPLEX system is fully capable of handling the inheritance mechanism correctly. A problem with DAPLEX queries though, is that the result is provided as textual output via a PRINT command. There

are no new entities created, whereas in the relational model a new table would be created. This lack of closure from queries is one of the main drawbacks of DAPLEX.

The second query illustrates the use of derived data, by creating an inverse to an already defined function, and then using that directly within the query. The most notable improvement to this query over the SQL version is that only three entities are composed, whereas the SQL query required a join of four tables.

In the second query, Student is acting both as an entity and a variable. If the query was changed to:

```
FOR EACH Course
        SUCH THAT FOR SOME X IN Student
                name(X) = 'Mary'
PRINT cname(sections_in(courses_completed(X)))
```

where Student is now definitely an entity, and X is a variable bound to each instance in turn of a Student entity, where the value of the function name for that particular student is 'Mary'.

The final query shows that DAPLEX will once again correctly handle inheritance, as a query on all Person entities will also include the Student entities, as the generalised information for a student is stored in the Person entity.

3. An Introduction to Category Theory

Rather than providing a general introduction to category theory, the reader should refer either to Barr and Wells [Barr90] or Simmonds [Simmonds90] for knowledge of the subject. Instead, we will look briefly at the concepts which will be used in the functorial model, which is to be introduced in section four.

3.1 Categories

The two fundamentals in category theory are the *arrow*, or *morphism*, and the *entity*, or *object*. An arrow can be thought of, for the purposes of this paper, as a function in set theoretic terms[1], providing a mapping from a source to a target entity, where the source and target entities appear in some domain and codomain respectively. As in sets, arrows can have an inverse, an identity arrow, and are composable, etc. For a fuller explanation, see [Rossiter94]. The *category*, is a collection of arrows between named entities, or objects. For example, the category **C** has two arrows f and g, where:

$$f : A \rightarrow B \quad g : B \rightarrow C$$

An object in a category where there is only one arrow from it to every other object is termed an *initial object*. Dually, the *terminal object* is one where there is only one arrow to it from every other object in the category.

[1] The arrow can in general represent any mathematical operation.

We use diagrams for displaying arrow composition within categories. *Commuting diagrams* within categories are where there are two paths between two objects, e.g.

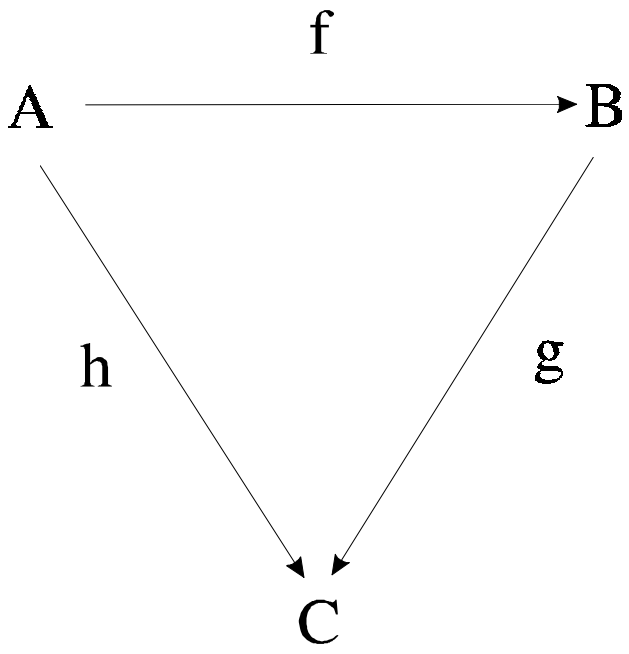

Fig 3 : Commuting Triangle

Thus, the diagram in figure 3 is said to commute when the equations $g \circ f = h$ hold experimentally, i.e. the two paths from object *A* to object *C* can be equated.

3.2 Subcategories

A *subcategory* **E** of a category **D** is one where all of the objects and arrows of **E** can be found in **D**, the sources and targets of arrows in **E** are the same as those in **D**, the identity arrows in **E** are as in **D**, and composition rules for arrows in **E** are as in **D**, i.e.

$$\text{obj}\mathbf{E} \subseteq \text{obj}\mathbf{D} \wedge \text{Hom}\mathbf{E}(p,q) \subseteq \text{Hom}\mathbf{D}(p,q) \quad (p,q \in \text{obj}\mathbf{E})$$

where obj**E** is the set of objects in category **E**, and Hom**E**(*p,q*) is the set of all arrows between objects *p* and *q* in **E**.

There are two special cases of subcategories. When **E** contains all the arrows of **D**, then it is termed a *full* subcategory of **D**, and if **E** has all the arrows of **D**, then it is a *wide* subcategory. Any category is a full wide subcategory of itself.

3.3 Functors

Functors provide a mapping between categories, as does an arrow provide a mapping between objects. More formally, a functor is an arrow between categories that assigns all objects and arrows in the source category to the target category and preserves compositions and identities that are in the source category within the target category.

If the shape of the source and target category is the same, the target category is said to have the *shape* of the functor. A functor which loses information, for instance mapping entities and arrows in the source category to just entities in the target category is termed a *forgetful functor*. Otherwise, a functor which maps all the structure of a source category onto a target category, where the target category may have additional structure itself, is termed a *free functor*.

3.4 Typing

The selection of elements a in a set A can be represented categorically by $a : \mathbf{1} \rightarrow A$, where **1** is the single category of one discrete object. Typing is added by indicating the category from where the item is taken, so the arrow $a : \mathbf{1}_{\mathbf{C}} \rightarrow A$ makes the element a in set A of type **C**. Typing can also be based on arrows, where, for example, monic arrows in a category or object could be considered of type **M**, where **M** is a category representing the universe of monics.

3.5 Product Cones

3.5.1 Product and Projection

The product and projection operations of relational algebra can be represented categorically by cones [Barr90]. The cone in figure 4 shows the product $A \times C$, with π_l and π_r projection arrows.

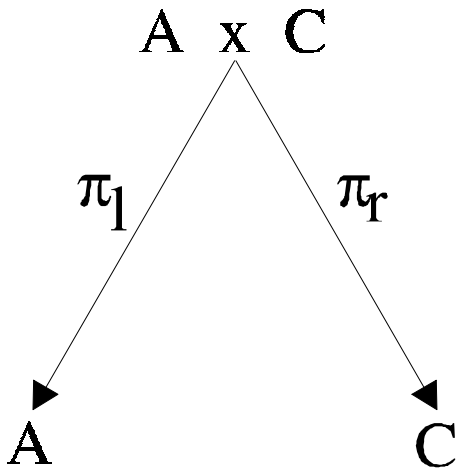

Fig 4 : Product Cone

Figure 5 below shows an alternative representation, where the diagram now commutes. For any object V and arrows $q_1 : V \rightarrow A$ and $q_2 : V \rightarrow C$, there is a product U with projections A and C such that the diagram commutes for the two equations $\pi_l \circ q = q_1$ and $\pi_r \circ q = q_2$.

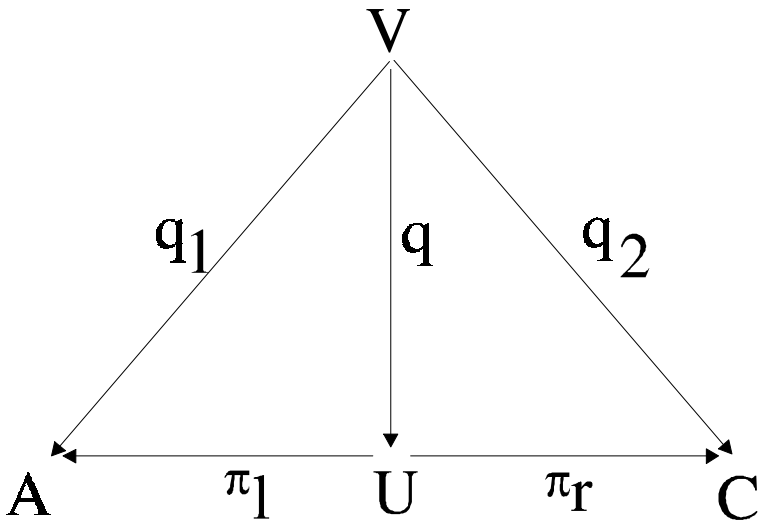

Fig 5 : Commuting Product Cone

3.5.2 Coproduct and Inclusion

The *coproduct*, or *sum*, is the dual of the product, where the coproduct is the disjoint union $A + C$, and i_l and i_r are inclusion arrows.

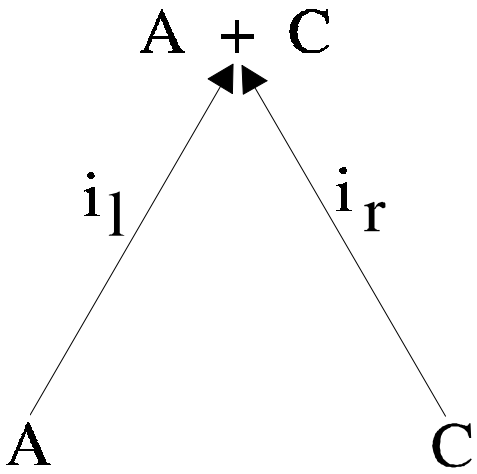

Fig 6 : Coproduct Cone

Note that both product and coproduct need not be binary. The concept can be extended to n-ary products.

3.5.3 Pullbacks

A *pullback* is a product which is restricted over some object. For example, the subproduct $A \times_B C$ over an object B is:

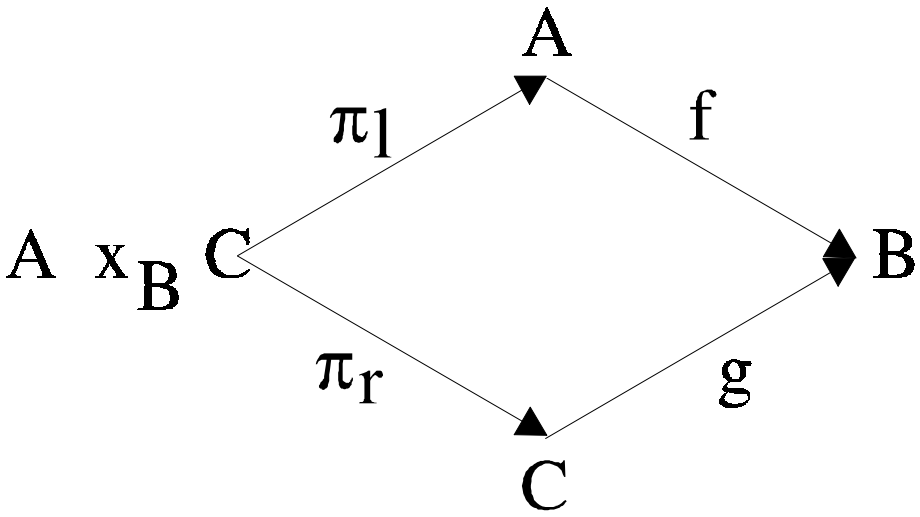

Fig 7 : Pullback

where $f(a) = g(c)$, and $f(a), g(c) \in B$, $a \in A$, $c \in C$. If this diagram commutes then $f \circ \pi_l = g \circ \pi_r$.

3.5.4 Pushouts

The *pushout* is the dual of the pullback, and so is the restricted coproduct. For the diagram in figure 8 below, *S* is the disjoint union of *A* and *C*, restricted over the object *B*.

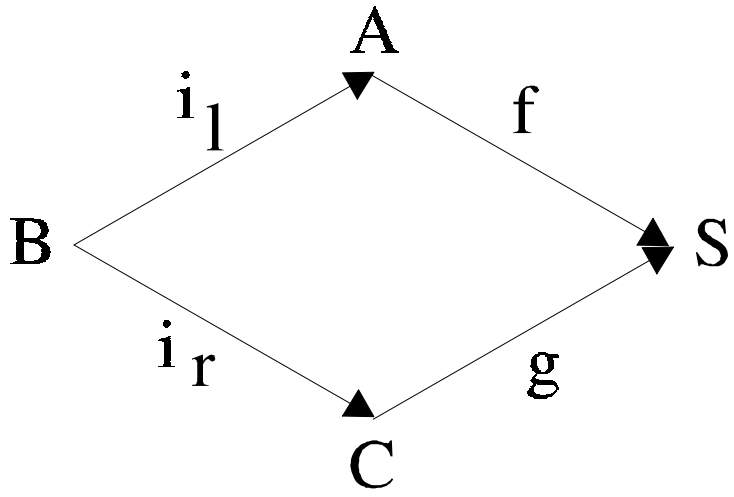

Fig 8 : Pushout

3.6 Limits

A *limit*, if it exists, may be a terminal object of a family of cones [Barr90]. For example, a limit is the infimum for all product cones, where the limit precedes each commuting cone, and only exists if every cone in the family of product cones commutes.

The dual of a limit is the *colimit*, which can be considered the initial object of a family of coproduct cones.

4. The Functorial Model

This new model utilises the concepts of category theory introduced in the previous section, to provide the facilities for object-oriented databases that are described as its minimum objectives in [Atkinson90]. The facilities we will look at in detail are class structures, associations, inheritance, message passing, objects, views and manipulation. For further insight into the functorial model, the reader should refer to the earlier paper which introduces it as the categorical product model [Rossiter94], and provides more of the mathematical definitions than are required in this paper.

4.1 Class

The class construction represents the intension of a database schema. Each class in the database can be represented by a category ($\mathbf{C_i} \mid 1 \leq i \leq n$), where n represents the number of classes in the intension, and $\mathbf{C_i}$ is the name of the i^{th} category in the database.

The arrows within a category $\mathbf{C_i}$ may represent methods (derivations) and dependencies (relationships) within classes, where the name of the arrow gives the name of the method or dependency. A category is a collection of k arrows, $M = \{m_j \mid 1 \leq j \leq k\}$ where m_j has domain $dom(m_j)$ and codomain $cod(m_j)$. $dom(m_j)$ can be structured (i.e. more than one variable) to allow complex methods and dependencies. $\{dom(m_j) \cup cod(m_j)\}$ is the set of variables in the class, which we call V (or obj$\mathbf{C_i}$) with cardinality q. These may be either persistent or memory variables, where the persistent variables $A = \{a_j \mid 1 \leq j \leq n\}$ represent the persistent attributes within the class, and the memory variables $U = \{u_j \mid 1 \leq j \leq p\}$ represent the transient components of the class. Note that A and U are both subobjects of V, and $n + p = q$.

The initial attribute in a category, i.e. an attribute in a category which has arrows to all other attributes in the category can represent the primary key. All persistent attributes in that class are therefore codomains of arrows from that initial object in the category. This can be determined as in [Rossiter94] as the collection of arrows $D = \{f_i \mid 0 \leq i \leq r'\}$ which represent arrows occurring in the universe of functional dependencies and $F = \{m_i \mid 0 \leq i \leq s\}$ representing arrows occurring in the universe of methods. Again, note that D and F are both subobjects of M, and $r' + s = m$.

4.2 Relationships

The association abstraction between classes can be represented in object models by notation based on the entity-relationship (E-R) approach. In category theory, the E-R model can be represented by pullbacks, as in figure 9.

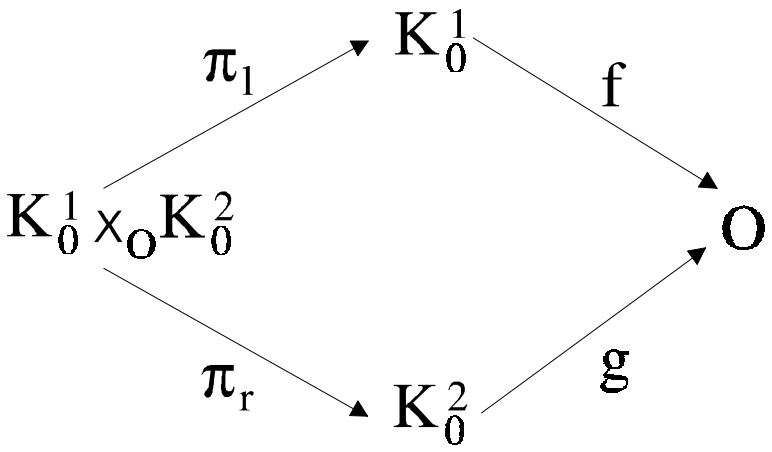

Fig 9 : Relationship as a Pullback

In the diagram of a pullback above, K_0^1 represents the initial object of a category $\mathbf{C_1}$, K_0^2 is the initial object of a category $\mathbf{C_2}$, O is an object representing all possible instances of the relationship between $\mathbf{C_1}$ and $\mathbf{C_2}$, and of the form $\{<k_0^1,k_0^2,o> \mid f(k_0^1) = g(k_0^2), k_0^1 \in K_0^1, k_0^2 \in K_0^2, o \in \wp O\}$, where o is any extra information to be included in the relationship, e.g. for the DAPLEX relationship grade(Student, Section), Student and Section would be categories, and O would be the set of tuples {<ssn, sec#, grade>}.

The product $K_0^1 \times K_0^2$ represents the actual set of tuples $\{<k_0^1, k_0^2>\}$ which would take part in the relationship O, where π_l and π_r project the product into their left and right counterparts respectively. To represent the relationship in the E-R model, the π_l and π_r arrows are typed according to the functionality and membership classes attributed to the relationship. These are defined as follows:

i. If π_l is epimorphic (onto) then the participation of K_0^1 in the relationship is mandatory, otherwise it is optional.

ii. If π_l is monic (one-to-one) then each K_0^1 appears just once in the subproduct with K_0^2.

The rules apply in the same way to π_r, defining K_0^2's participation and functionality in the relationship. These two rules, then, allow us to define both the functionality and the membership class of the relationship, according to table 1, where K_0^1 is the identifier of category or class **A** and K_0^2 is that of **B**.

π_l		π_r		relationship			
epic	monic	epic	monic	participation	mapping	membership class	
				A:C	A:C	A	C
n	n	n	n	N:M	N:M	o	o
y	n	n	n	N:M	N:M	m	o
y	n	y	y	N:1	1:N	m	m
y	y	n	n	1:N	N:1	m	o
y	y	y	y	1:1	1:1	m	m

Table 1 : Arrow Types and their E-R Equivalent

It is easy to see that this can be extended to n-ary relationships, which can be represented by finite products [Rossiter92]. Involuted relationships and multiple relationships between the same two objects can also be handled in an easy manner, usually by giving the relationships different names [Rossiter94].

4.3 Inheritance

Inheritance is where a class has additional information added, in the form of attributes, methods or dependencies. We will look at inheritance by using the categorical concept of the coproduct.

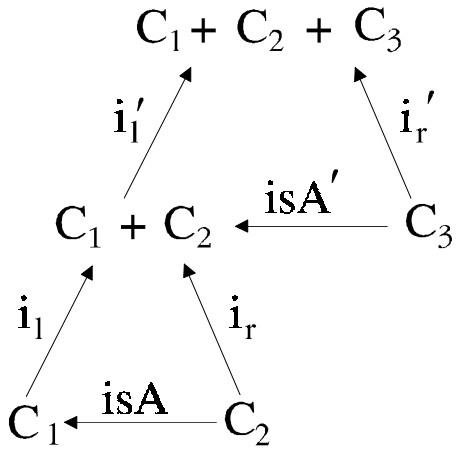

Fig 10 : Inheritance as a Coproduct

The above diagram, figure 10, shows how we can handle multiple levels of inheritance. It shows three classes $\mathbf{C_1}$, $\mathbf{C_2}$ and $\mathbf{C_3}$ where $\mathbf{C_1}$ is a generalisation of $\mathbf{C_2}$ and $\mathbf{C_2}$ is a generalisation of $\mathbf{C_3}$. $\mathbf{C_2}$ is the extra information to be added to $\mathbf{C_1}$ to form the new subclass $\mathbf{C_1 + C_2}$. This disjoint union of $\mathbf{C_1}$ and $\mathbf{C_2}$ represents the final subclass or subcategory, which is an union of the arrows and attributes in $\mathbf{C_1}$ and those in $\mathbf{C_2}$. It is disjoint so that two attributes could not have the same name, otherwise they would be excluded in the coproduct. To have multiple levels of inheritance, we just extend the coproducts, as shown in the diagram.

The i_l and i_r arrows represent the inclusion of the two categories in the disjoint union. The *isA* arrow is just there to indicate the direction of inheritance, although there could

be a dual to this arrow, to have consistency of keys in the new subclass. The subtypes are weak-entity types: they inherit the key from the supertype, as in the following example:

$\mathbf{C_1}$: $a \to b$
$\mathbf{C_2}$: $a \to c$
$\mathbf{C_3}$: $a \to d$

$\mathbf{C_1}$+$\mathbf{C_2}$: $a \to b$
$a \to c$

$\mathbf{C_1}$+$\mathbf{C_2}$+$\mathbf{C_3}$: $a \to b$
$a \to c$
$a \to d$

Note that this model can only support single inheritance, and not multiple inheritance, i.e. **X** *isA* **Y and X** *isA* **Z**. We are currently looking at either extending the coproduct concept, by redefining the disjoint union, or using the free functor to handle this.

4.4 Message Passing

Message passing is the object-oriented method of calling operations in one object from another object. In categorical terms, it is a function from an object or arrow in one category to another category. For example:

$$\varphi : m_k \to \mathbf{C_1}\ (m_k \in \mathrm{Hom}\mathbf{C_i}(v,v'),\ v,v' \in \mathrm{obj}\mathbf{C_i})$$

takes a value from the arrow m_k in class $\mathbf{C_i}$ and passes it to the class $\mathbf{C_1}$. $\mathrm{Hom}\mathbf{C_i}(v,v')$ is the set of all arrows between v and v' in $\mathbf{C_i}$, and $\mathrm{obj}\mathbf{C_i}$ is the same as the set V mentioned earlier. The second example:

$$\eta : v_i \to \mathbf{C_m}\ (v_i \in \mathrm{obj}\mathbf{C_i})$$

takes a value from the object v_i in category $\mathbf{C_i}$, passing it to category $\mathbf{C_m}$.

4.5 Objects

Objects represent the extension of the database schema. They hold values which must be consistent with the intension. There is a functor D_i from each class, or category, $\mathbf{C_i}$ to the instances for each object type $\mathbf{E_i}$, which again are categories. The functor D_i maps:

i. Each arrow a in $\mathbf{C_i}$ to a set of arrow instances $D_i(a)$ in $\mathbf{E_i}$
ii. Each domain $dom(a)$ in $\mathbf{C_i}$ to a set of instances $D_i(dom(a))$
iii. Each codomain $cod(a)$ in $\mathbf{C_i}$ to a set of instances $D_i(cod(a))$
iv. The key K_0 to a set of instances $D_i(K_o)$
v. Each non-key attribute $(K_i \mid 1 \le i \le r)$ to a set of instances $D_i(K_i)$
vi. Each functional dependence $(f_i \mid 1 \le i \le r)$ to a set of arrow instances $D_i(f_i)$.

There is also a functor $\mathbf{P_i} \rightarrow \mathbf{L_i}$ which maps pullback intension categories to pullback extension categories.

For each class $\mathbf{C_i}$, the functor D_i should preserve limits with respect to the functional dependencies, i.e. a limit that holds in the source category is preserved in the target category. If this is true, then the intension to extension mapping is consistent. Therefore, the diagram in figure 11 below should commute for every cone:

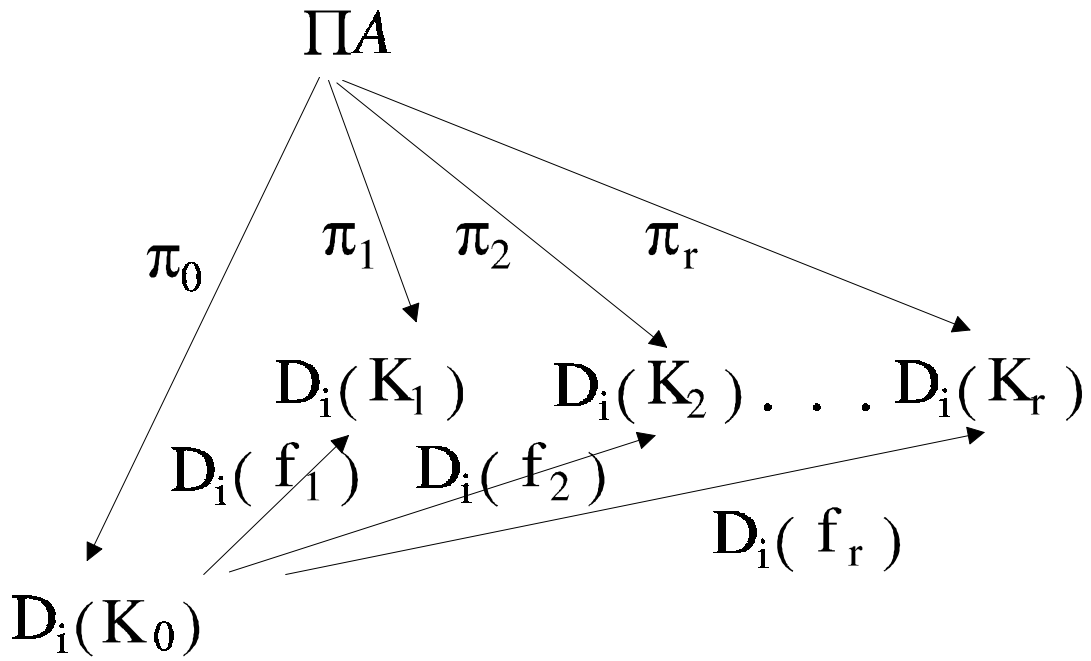

Fig 11 : Limit of Commuting Cones

where ΠA is the limit, π_i is a projection coordinate from ΠA to $D_i(f_i)$ and $D_i(f_i) : D_i(K_0) \rightarrow D_i(K_i)$ are the postulated functional dependencies. The test is that for all non-key attributes $D_i(K_i)$, $D_i(f_i) \circ \pi_0 = \pi_i$.

4.6 Views

In a database system, the view mechanism is used for controlling user access to a database, by providing partial access to data records or objects. In categorical terms, this can be easily represented by using a subcategory, where, for a subcategory $\mathbf{V_j}$ of a category $\mathbf{C_i}$:

$$\mathrm{obj}\mathbf{V_j} \subseteq \mathrm{obj}\mathbf{C_i} \wedge \mathrm{Hom}\mathbf{V_j}(p,q) \subseteq \mathrm{Hom}\mathbf{C_i}(p,q) \quad (p,q \in \mathrm{obj}\mathbf{V_j})$$

That is, the objects in the subcategory $\mathbf{V_j}$ are a subset of category $\mathbf{C_i}$, and the set of all arrows between any two objects in $\mathbf{V_j}$ are a subset of those in $\mathbf{C_i}$. It could of course be a full or wide subcategory, which would allow full access to all of the data or the arrows, or both.

This method only provides a view on a single category $\mathbf{C_i}$ in the database, but with a view mechanism, we would like to be able to provide a view on a collection of objects. This can readily be done by use of the association and composition mechanisms, whereby a new category $\mathbf{J_k}$ would be formed as a product between objects or categories, the latter being termed allegories [Freyd90].

4.7 Manipulation

Operations at both the intra-object and inter-object level can be handled by the arrow in category theory. This is very much equivalent to the approach used in DAPLEX, where functions can be regarded as arrows. The query language we have currently developed is based on the notion of 'sub-setting' categories[2] . This is done by functors, which map between an input and an output category. As the sub-category can be queried further still, we have obtained closure. Composing these functors would allow complex queries to be formed, where the result would be a new category. If we used a forgetful functor, then the result of the query would be a category of values, which can be equated to just providing output as the result of a query, rather than something which can be queried further.

At the moment, the query language is still in its early stages, although we do have a complete query language. Eventually, we hope to have a query system that looks more like a functional query language, such as DAPLEX [Shipman81].

5 The Functorial Example

The diagram in figure 12 shows the product diagram for the functorial data model example, based on the E-R diagram presented earlier. The diagram is very heavily based on the E-R version, most notably that it shows the same entities, now as categories, and the same relationships between them.

The first rectangle shows the Person object represented by a category. It shows the initial object, or key, in italics, with the other persistent attributes being the targets of the arrows from the initial object. The same applies for all other categories in the diagram. Note the extra category, **Student-spec**. This represents the new category which must be joined with the Person category, to form the Student category. Note that in coproduct terms, **Student = Person + Student-spec**. The i_l and i_r arrows are the inclusion arrows, and the *isA* arrow is also shown in the coproduct to define the direction of inheritance.

[2] Note, we use the term sub-set in a very loose sense, we are in fact creating sub-categories.

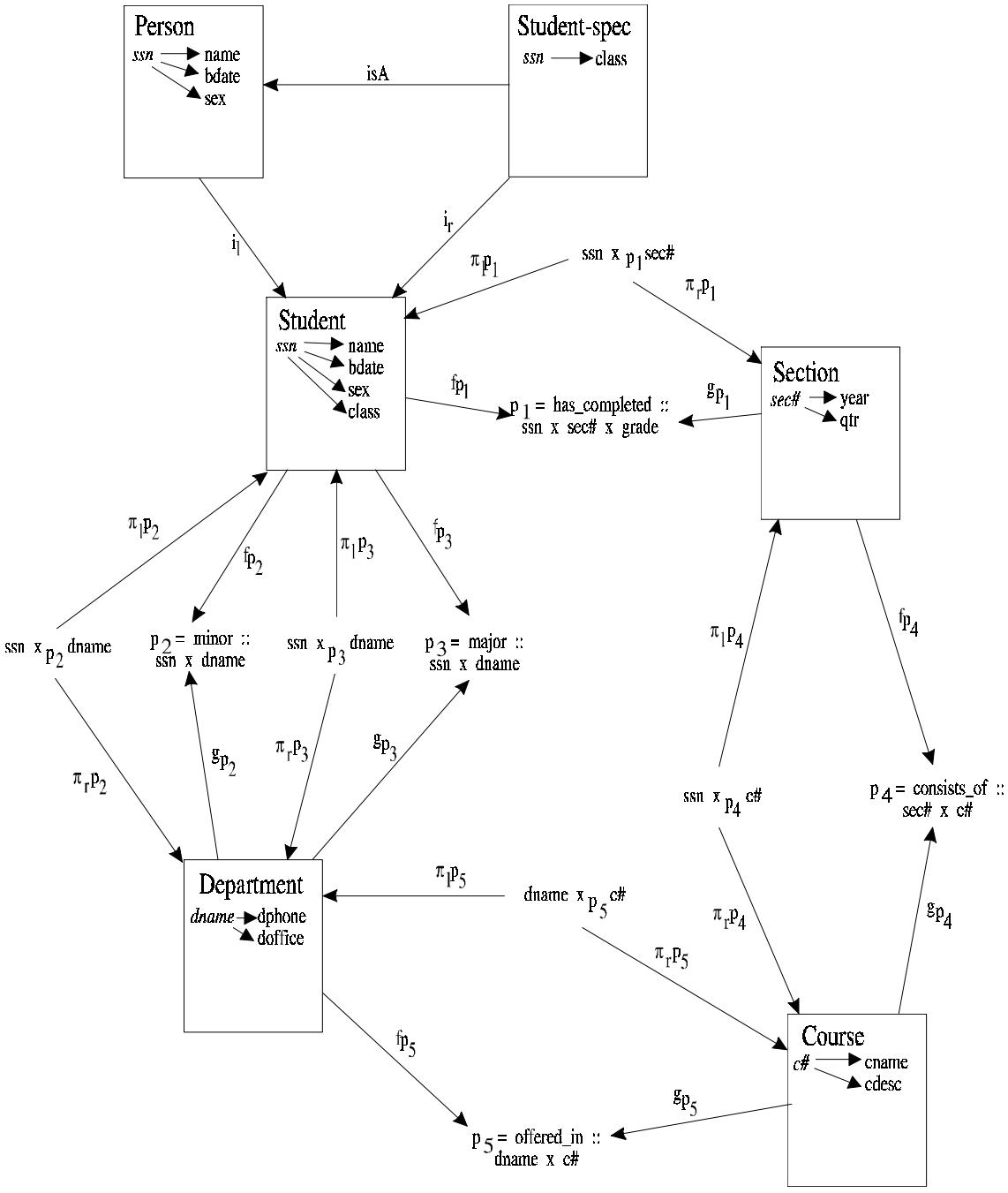

Fig 12 : The Product Diagram

The five relationships between the other four categories are each pullback categories, with four arrows to each pullback. For instance, the relationship p_2 shows a pullback between Student and Department, where *minor* :: *ssn* × *dname* is the name and type (or pool of values) of the product (the object *O* in the previous section), *ssn* $\times_{p2}$ *dname* is the restricted product, $\pi_l p_2$ and $\pi_r p_2$ are the left and right projections of the product into the initial objects of the Student and Department categories respectively, and $f p_2$ and $g p_2$ are functions injecting the initial objects into the pool of values of the product. This is the same for all pullbacks ($p_i \mid 1 \le i \le 5$), which represent relationships between two entities in the E-R sense. The coproduct between Person and Student discussed previously represents a subtype in the E-R model.

The pullback p_1 between Student and Section raises some interest, as it shows the definition of an extra attribute, grade, which is extra information to be stored in the relation. This is in essence a join in the relational model, and is the *o* attribute in the tuple $<k_0^1, k_0^2, o>$ which represents *O*. It is equivalent to the declare grade(Student, Section) function in the functional model.

The product diagram is a commuting diagram, and when all cones commute, then we have consistency in the database intension. The diagrams commute when $f \circ \pi_l = g \circ \pi_r$, for each pullback.

Now that the diagram has been given and explained, all the arrows in the pullbacks must be typed, in order to define the functionality and membership classes of the relationships. These are given below:

$\mathbf{p_1 = ssn \times sec\#}$	**(Student × Section)**
π_l = ¬monic, ¬epic	
π_r = ¬monic, ¬epic	N (opt) : M (opt)
$\mathbf{p_2 = ssn \times dname}$	**(Student × Department)**
π_l = monic	
π_r = epic	N (opt) : 1(mand)
$\mathbf{p_3 = ssn \times dname}$	**(Student × Department)**
π_l = monic	
π_r = epic	N (opt) : 1(mand)
$\mathbf{p_4 = sec\# \times c\#}$	**(Section × Course)**
π_l = isomorphic	
π_r = epic	N (mand) : 1 (mand)
$\mathbf{p_5 = dname \times c\#}$	**(Department × Course)**
π_l = epic	
π_r = epic	N (mand) : M (mand)

For each p_i, the initial objects are given, and then the π_l and π_r projection arrows are typed according to whether they are monic and epic. If an arrow is both monic and epic, then it is isomorphic (in set-based categories). The relationship between the two classes (or categories) is then given in E-R terms, and should therefore be the same as in the E-R diagram.

This typing of arrows is a purely intensional concept, where they represent the type of relationships which actually occur in the extension. In the extension, actual data values are stored rather than name and type information. If the product diagram commutes, then as the functor which maps between the intension and extension should be

structure preserving, the extension will commute, and therefore *integrity* is preserved, otherwise the data is inconsistent with the intension.

This means that we could reproduce the product diagram for the extension, although in this paper we will only show part of it, as in figure 13.

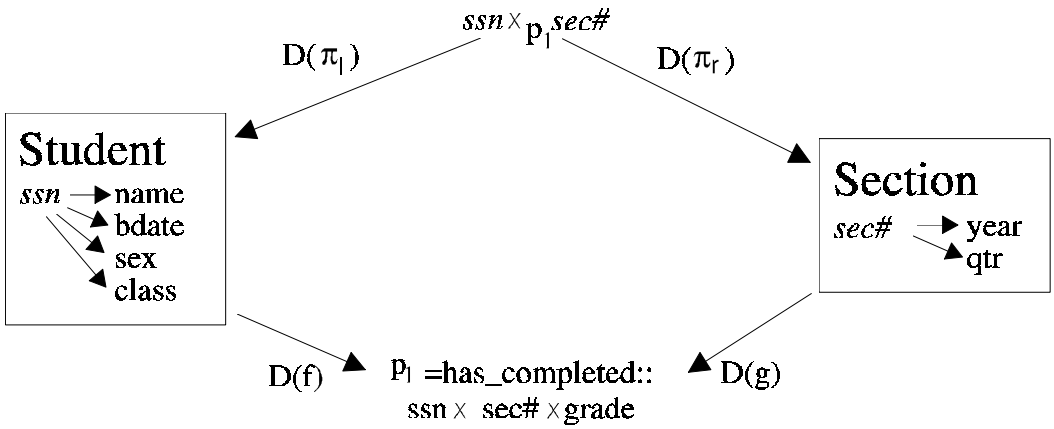

Fig 13 : Pullback in the Extension Schema

This time the arrow names are targets of the functor *D*, but now the names of attributes would be represented by the actual data values in the database. Once again, the diagram would commute, i.e.

$$D(f) \circ D(\pi_l) = D(g) \circ D(\pi_r)$$

and if this was the case for every object in the database, then we could prove that the database is consistent.

Our final task is to present the queries, to match those given in SQL and DAPLEX.

1. Print the names of all students who have minored in the mathematics department:

A $\rightarrow$ $\mathbf{P_1}$[3]

Hom set = π_{lp2}

Objects = $ssn \times_{p2} dname$, $ssn \mid dname$ = 'Maths'

B $\rightarrow$ **A**

Hom set = { }

Objects = *ssn*

C $\rightarrow$ **Student**

Hom set = $ssn \rightarrow name$

Objects = *ssn*, *name* $\mid$ $ssn \in$ **B**

[3] Category P_i ($1 \leq i \leq 5$) is the pullback category of the relationship p_i, including all arrows in the pullback, the product p_i and the restricted product, and the initial objects of the categories that the pullback is between.

2. Print the names of all courses which student 'Mary' takes:

A $\rightarrow$ **Student**
Hom set = *ssn* $\rightarrow$ *name*
Objects = *ssn*, *name* $\mid$ *name* = 'Mary'

B $\rightarrow$ **A**
Hom set = { }
Objects = *ssn*

C $\rightarrow$ $\mathbf{P_1}$
Hom set = $\pi_{r_{p1}}$
Objects = *ssn* $\times_{p1}$ *sec#*, *sec#* $\mid$ *ssn* $\in$ **B**

D $\rightarrow$ **C**
Hom set = { }
Objects = *sec#*

E $\rightarrow$ $\mathbf{P_4}$
Hom set = $\pi_{r_{p4}}$
Objects = *sec#* $\times_{p4}$ *c#*, *c#* $\mid$ *sec#* $\in$ **D**

F $\rightarrow$ **Course**
Hom set = *c#* $\rightarrow$ *cname*
Objects = *c#*, *cname* $\mid$ *c#* $\in$ **E**

G $\rightarrow$ **F**
Hom set = { }
Objects = *cname*

3. Print the names of all people who are male:

A $\rightarrow$ **Person**
Hom set = *ssn* $\rightarrow$ *name*, *ssn* $\rightarrow$ *sex*
Objects = *ssn*, *name*, *sex* $\mid$ *sex* = 'M'

B $\rightarrow$ **A**
Hom set = { }
Objects = *name*, *sex*

[Note, hom-sets are defined as a set of arrows, each arrow being specified by its name or its source and target.]

The first is simple. It subsets the category $\mathbf{P_2}$, which is the category of the pullback representing the relationship p_2 (i.e. minor) to give us all the π_l arrows, and the *ssn* $\times$ *dname* and *ssn* objects, where the *dname* is 'Maths'. It then removes the arrows and

the product to produce subcategory **B**, and then subsets the **Student** category to give a category containing *ssn* and *name* objects, where a *ssn* is a member of **B**.

The second query is more complex, we first of all produce a category **B** of all **Student** *ssn's* where the name of the student is 'Mary', subset the $\mathbf{R_1}$ pullback to include only those in **B**. We then subset the $\mathbf{P_4}$ pullback to get the sections which Mary takes, testing this for inclusion in the **Course** category, to produce a subset of **Course** which stores the names of courses which 'Mary' takes.

The final query just needs to subset the **Person** category, where the sex object has a value 'Male'. This query depends on the inheritance mechanism functioning correctly, so that all students are taken into account as well.

6. Conclusions

In this paper we have introduced the main concepts of the functorial model, and provided some example queries in this model, as well as in the functional model and the relational model. Table 2 below summarises the major concepts of database systems, identifying how each model copes with them.

The model is still in a relatively early stage of development, a lot still has to be done, namely in the inheritance and manipulation concepts of the model. But, as it stands, it does model most object-oriented aspects in a relatively 'natural' manner. It also does this in a manner which does not lose the best bits of current relational systems, where the pullback still looks like a key-key match between two relations, thus providing more of an object-relational type database.

The next stage, as well as developing the functorial model further, is to actually implement a prototype of the system, using either an object-oriented language such as C++ [Stroustrup91] or a functional language such as Haskell [Hudak92], depending on which contains the best features for categorical programming.

	Relational Model	**Functional Model**	**Functorial Model**
Inter-object associations	Join on primary / foreign keys	Functions	Pullbacks
Intra-object associations	Functional dependencies	N/A	Arrows within category
Integrity constraints	Referential /Entity Integrity	Single/Multi valued functions	Arrow types, object types
Keys	Primary / foreign key definitions	N/A	Initial objects
Normal forms	Normalisation, atomicity and dependency rules	Single/multi-valued sets can break 1NF rule	Arrow types and BCNF tests / 1NF not essential
Relational Join	Relational Algebra	Function composing entities	Pullback / product

Relational Product	Relational Algebra	Function composing entities	Product
Relational Projection	Relational Algebra	Derived functions	Subcategory / product projection
Relational Union	Relational Algebra	Functions between entities	Pushout / coproduct
Relational Difference	Relational Algebra	Derived functions	Subset specifications
Typing	Relational table and attribute types	Entities and function types	Object, arrow and category types
Views	View	Derived functions	Subcategories
Aggregation	Table join	Multi-valued functions	Pullback / Product
Inheritance	Table join	Sub entities	Coproduct or comma category

Table 2 : Comparison of Database Models

7. References

[Atkinson83]
Atkinson M et al. An Approach to Persistent Programming. The Computer Journal, Vol. 26, No. 4, 1983.

[Atkinson90]
Atkinson M et al. The Object-Oriented Database System Manifesto. In Bancilhon92.

[Bancilhon92]
Bancilhon F. The Story of O_2: Implementing an Object-Oriented Database System. Morgan Kaufmann 1992.

[Barr90]
Barr M, Wells C. Category Theory for Computer Science. Prentice Hall International Series in Computer Science, 1990.

[Codd79]
Codd E F. Extending the Database Relational Model to Capture More Meaning. ACM Transactions on Database Systems, Vol. 4, No. 4, December 1979.

[Date90]
Date C J. An Introduction to Database Systems, Volume I. Addison Wesley, 1990.

[Elmasri89]
Elmasri R, Navathe S B. Fundamentals of Database Systems. Addison Wesley World Student Series, 1989.

[Freyd90]
Freyd P J, Scedrov A. Categories, Allegories. North Holland Mathematical Library, Vol. 39, 1990.

[Gray88]
Gray P M D, Moffat D S, Paton N W. A Prolog Interface to a Functional Data Model Database. Lecture Notes in Computer Science, 1988.

[Gray92]
Gray P M D, Kulkarni K G, Paton N W. Object-Oriented Databases: A Semantic Data Model Approach. Prentice Hall International Series in Computer Science, 1992.

[Hudak92]
Hudak P, Fasel H J. A Gentle Introduction to Haskell. SIGPLAN Notices, Vol. 27, No. 5, 1992.

[Kim90]
Kim W. Introduction to Object-Oriented Database Systems. MIT Press, 1990.

[Kulkarni86]
Kulkarni K G, Atkinson M P. EFDM: Extended Functional Data Model. The Computer Journal, Vol. 29, No. 1, 1986.

[Kulkarni87]
Kulkarni K G, Atkinson M P. Implementing an Extended Functional Data Model using PS-Algol. Software Practices and Experiences, Vol. 17, March 1987.

[Layzell89]
Layzell P, Loucopoulos P. Systems Analysis and Development, Third Edition. Chartwell-Bratt Studentlitteratur, 1988.

[Meyer88]
Meyer B. Object-Oriented Software Construction. Prentice Hall International Series in Computer Science, 1988.

[Rossiter92]
Rossiter B N, Heather M A. Applying Category Theory to Databases. University of Newcastle upon Tyne, Technical Report Series, November 1992.

[Rossiter94]
Rossiter B N, Nelson D A, Heather M A. The Product Database Model as a Formal Object Model. University of Newcastle upon Tyne, Technical Report Series, in litt., 1994.

[Roth88]
Roth M A, Korth H F, Silberschatz A. Extended Algebra and Calculus for Nested Relational Databases. ACM Transactions on Database Systems, Vol. 13, No. 4, December 1988.

[Shipman81]
Shipman D W. The Functional Data Model and the Data Language DAPLEX. ACM Transactions on Database Systems, Vol. 6, No. 1, March 1981.

[Simmonds90]
Simmonds H. Notes on Categorical Logic. Logic for Information Technology, SERC, 1990.

[Stroustrup91]
Stroustrup B. The C++ Programming Language, 2nd Edition. Addison Wesley, 1991.

[Sutton92]
Sutton D R, King P J H. Integration of Modal Logic and the Functional Database Model. Lecture Notes in Computer Science, Vol. 618, 1992.