Conceptual Data Modelling from a Categorical Perspective

A. H. M. TER HOFSTEDE*, E. LIPPE, AND P. J. M. FREDERIKS

Department of Information Systems, University of Nijmegen, Toernooiveld 1, NL-6525 ED Nijmegen, The Netherlands

** Present address: Department of Computer Science, The University of Queensland, Brisbane Qld 4072, Australia*

E-mail: arthur@cs.uq.edu.au

For successful information systems development, conceptual data modelling is essential. Nowadays many conceptual data modelling techniques exist. In-depth comparisons of concepts of these techniques are very difficult as the mathematical formalizations of these techniques, if they exist at all, are very different. Consequently, there is a need for a unifying formal framework providing a sufficiently high level of abstraction. In this paper the use of category theory for this purpose is addressed. Well-known conceptual data modelling concepts, such as relationship types, generalization, specialization, collection types and constraint types, such as the total role constraint and the uniqueness constraint, are discussed from a categorical point of view. An important advantage of this framework is its 'configurable semantics'. Features such as null values, uncertainty and temporal behavior can be added by selecting appropriate instance categories. The addition of these features usually requires a complete redesign of the formalization in traditional set-based approaches to semantics.

Submitted June 1995, revised February 1996

1. INTRODUCTION

Conceptual data modelling is imperative for successful information systems development. Currently, many different conceptual data modelling techniques exist (see e.g. [1, 2]). Examples are ER [3] and its many variants, functional modelling techniques, such as FDM [4], and so-called object-role modelling techniques, such as NIAM [5]. Complex application domains, such as meta modelling, hypermedia and CAD/CAM, have led to the introduction of advanced modelling concepts, such as those present in the various forms of Extended ER (see e.g. [6, 7]), IFO [8], and object-role modelling extensions such as FORM [9] and PSM [10, 11].

This plethora of techniques reflects the general situation in the field of information systems development. In [12] this situation is described by the term *Methodology Jungle*. In [13] it is estimated that during the past years, hundreds if not thousands of information system development methods have been introduced. Most organizations and research groups have defined their own methods. Hardly any of them has a formal syntax, let alone a formal semantics. The discussion of numerous examples, mostly with the use of pictures, is a popular style for the 'definition' of new concepts and their behavior. This has led to *fuzzy* and *artificial* concepts in information systems development methods.

To some extent this latter observation is also true for the field of conceptual data modelling. In-depth comparison of concepts of various techniques is complicated by the fact that neither the techniques involved have a formal semantics or completely different formalizations. Consequently a unifying framework for conceptual data modelling techniques seems imperative. Such a framework should be *formal*, in order to avoid ambiguities; offer a sufficiently high level of *abstraction*, in order to concentrate on the meaning of concepts instead of on representational aspects; and be sufficiently *expressive*. The goal of this paper is to define such a unifying framework for conceptual data modelling techniques. This framework should clarify the precise meaning of fundamental data modelling concepts and offer a sufficient level of abstraction to be able to concentrate on this meaning and avoid distractions of particular mathematical representations (in a sense, the well-known *Conceptualization Principle* [14] can also be applied to mathematical formalizations). These requirements suggest category theory (see e.g. [15]) as an excellent candidate. Category theory provides a sound formal basis and abstracts from all representational aspects. Therefore, the framework will be embedded in category theory.

For conceptual data modelling techniques that do have a formal foundation, the framework described may also be of use, as it may suggest natural generalizations and expose similarities between seemingly different concepts. Another interesting application of the use of category theory can be found in the opportunity to consider different interpretations of a modelling technique by considering different categories as semantic target domains. For example, if one wants to study 'null' values in relationship types in a particular data modelling technique, it is natural to consider **PartSet**, i.e. the category of sets and *partial* functions, as a target category. The use of partial functions allows certain

components of a relation to be undefined. In this sense, the approach outlined is more general than approaches as described in [16, 17] where only specific types of categories, topoi, are possible target categories.

The idea of a 'configurable semantics' is an *essential* feature of the unifying framework. The addition of a new dimension (e.g. null values, uncertainty, time) to an existing conceptual data modelling technique now often implies a complete redesign of the existing formalization. In case of a formalization of the involved technique in terms of the presented framework such an addition would only imply a choice of an appropriate target category.

The paper is organized as follows. Section 2 contains a brief introduction to category theory and its historical background. Section 3 describes the essential data modelling concepts, i.e. relationship types, generalization, specialization, and collection types, from a category theoretic point of view. In section 4 two important constraint types, the total role constraint and the uniqueness constraint, are given a categorical semantics. Section 5 presents conclusions and identifies topics for further research.

2. CATEGORY THEORY

This section contains the definition of the categorical constructs and notations needed in the rest of this paper, in order to make it self-contained as much as possible. For an in-depth treatment of category theory the reader is referred to [15].

2.1. Background

A brief history of the origin of category theory can be found in [18]:

> Eilenberg and Mac Lane created categories in the 1940s as a way of relating systems of algebraic structures and systems of topological spaces in algebraic topology. The spread of applications led to a general theory, and what had been a tool for handling structures became more and more a means of defining them. Grothendieck and his students solved classical problems in geometry and number theory using new structures—including topoi—constructed from sets by categorical methods. In the 1960s, Lawvere began to give purely categorical definitions of new and old structures, and developed several styles of categorical foundations for mathematics. This led to new applications, notably in logic and computer science.

Category theory is therefore a relatively young branch of mathematics designed to describe various *structural* concepts from different mathematical fields in a *uniform* way. Category theory offers a number of concepts and theorems about those concepts, that form an abstraction of many concrete concepts in diverse branches of mathematics. As pointed out by Hoare [19]: 'Category theory is quite the most general and abstract branch of pure mathematics'.

In the 1970s and 1980s category theory also found its way into computer science. Applications of category theory can be found in such diverse fields as automata and systems theory, formal specifications and abstract data types, type theory, domain theory and constructive algorithmics. As pointed out by [20], category theory can provide help with at least the following:

- *Formulating definitions and theories.* In computing science, it is often more difficult to formulate concepts and results than to give a proof. As stated by [21], category theory provides a language with a convenient symbolism that allows for the visualization of quite complex facts by means of diagrams.
- *Carrying out proofs.* Once basic concepts have been correctly formulated in a categorical language, it often seems that proofs 'just happen': at each step, there is a 'natural' thing to try and it works.
- *Discovering and exploiting relations with other fields.* Sufficiently abstract formulations can reveal surprising connections.
- *Formulating conjectures and research directions.* Connections with other fields can suggest new questions in one's own field.
- *Unification.* Computing science is very fragmented, with many different subdisciplines having many different schools within them. Hence, the kind of conceptual unification that category theory can provide, is badly needed.
- *Dealing with abstraction and representation independence.* In computing science, more abstract viewpoints are often more useful, because of the need to achieve independence from the overwhelmingly complex details of how things are represented or implemented.

This last item is particularly relevant in the context of this paper. Category theory allows the study of the essence of certain concepts as it focuses on the *properties* of mathematical structures instead of on their *representation*. To illustrate this point, consider for example possible definitions of an *ordered pair*. The well-known Wiener–Kuratowski definition of an ordered pair is:

$$\langle a, b \rangle = \{a, \{a, b\}\}$$

From this definition one can always derive what the first element of the ordered pair involved was, and what its second element was. However, assuming that we deal with sets of natural numbers, the following definition also has this property:

$$\langle a, b \rangle = 2^a 3^b$$

Clearly, both definitions could be used for the definition of an ordered pair as both encompass its essence. However, it is also clear that they are both overspecific. One could speak of two *implementations* of ordered pairs. The definitions prescribe particular representations and do not focus on the underlying essence. They are precisely the kind of definition that category theorists abhor. One might say

that category theory applies the Conceptualization Principle to mathematical formalizations.

Despite the popularity of category theory in some fields of computing science, not many applications in the field of information systems can be found in the literature. Recently, however, it seems that this is changing. Categorical formalizations of (aspects of) object orientation (see e.g. [22–24]), object-oriented data models (see e.g. [25, 16]), ER (see e.g. [26]), and the Relational Model (see e.g. [27, 17]) have been proposed. In [28] a categorical framework for the axiomatization of conceptual modelling concepts is described (based on the notion of π-institution). In [16] it is remarked that the uniformity of category theory provides a basis for interesting generalizations in the context of data modelling and that it not only offers insight into well-known operators but also allows for the definition of new operators, which would be far from trivial in other formalisms.

2.2. Basics

This section presents the definitions of the basic concepts of category theory as far as they are important for the rest of this paper. Most of these definitions are adapted from [15].

A directed multigraph is a directed graph where there may be multiple edges with the same direction between two nodes.

DEFINITION 2.1. A directed multigraph $\mathcal{G}$ consists of a set of nodes $\mathcal{G}_0$ and a set of edges $\mathcal{G}_1$. The source and target of an edge can be found by application of the functions source and target, respectively. The notation $f{:}A \rightarrow B$ implies that f is an edge with $\mathsf{source}(f) = A$ and $\mathsf{target}(f) = B$. □

The following definition defines a category as a special kind of multigraph.

DEFINITION 2.2. A *category* C is a directed multigraph whose nodes are called *objects* and whose edges are called *arrows*. For each pair of arrows $f{:}A \rightarrow B$ and $g{:}B \rightarrow C$ there is an associated arrow $g \circ f{:}A \rightarrow C$, the *composition* of f with g. Furthermore, $(h \circ g) \circ f = h \circ (g \circ f)$ whenever either side is defined. For each object A there is an arrow $\mathsf{Id}_A{:}A \rightarrow A$, the *identity* arrow. If $f{:}A \rightarrow B$, then $f \circ \mathsf{Id}_A = f = \mathsf{Id}_B \circ f$. □

Figure 1 represents a simple example of a category. It is an abstract example: no assumptions about the meaning of the objects and the arrows have been made (and indeed, have to be made!).

In this category the choice of composites is forced: $f \circ \mathsf{Id}_A = f = \mathsf{Id}_B \circ f$. In category theory it is customary to omit the identity arrows in drawings of categories if they do not serve a particular purpose. We will adopt this convention in the rest of this paper. The objects and arrows of a category may also have a concrete interpretation. For example, objects may be mathematical structures such as sets, partially ordered sets, graphs, trees etc. Arrows can denote functions, relations, paths in a graph, etc.

As a concrete example of a category in the context of information systems consider the set of all instantiations of a data base, and all possible updates on these instantiations. The instantiations may serve as objects, and the updates as arrows of the corresponding category. Each object has an identity arrow, if one considers the 'neutral' update, i.e. the update that does not change an instantiation at all, to be a normal update. One can easily verify that this indeed constitutes a category. Arrow composition is associative as update composition is associative. Also, the neutral update serves as a neutral element with respect to arrow composition: an update composed with a neutral update simply yields that update.

In the context of this paper, some set-oriented categories are important. The most elementary and frequently used category is the category **Set**, where the objects are sets and the arrows are total functions. The objects of **Set** are not necessarily finite. The category whose objects are *finite* sets and whose arrows are total functions is called **FinSet**. The category **PartSet** concerns sets with *partial* functions, while the category **Rel** has sets as objects and binary *relations* as arrows.

Some arrows have special properties. We consider three important kinds of arrows: *monomorphisms*, *epimorphisms* and *isomorphisms*.

DEFINITION 2.3. An arrow $f{:}A \rightarrow B$ is a *monomorphism* if for any object X of the category and any arrows $x, y{:}X \rightarrow A$, if $f \circ x = f \circ y$, then $x = y$. □

Figure 2 illustrates the definition of a monomorphism.

A monomorphism in the category **Set** captures the idea of an injective function. In the category **PartSet** a monomorphism describes a total and injective function.

DEFINITION 2.4. An arrow $f{:}B \rightarrow A$ is an *epimorphism* if for any object X of the category and any arrows $x, y{:}A \rightarrow X$, if $x \circ f = y \circ f$, then $x = y$.

Figure 3 illustrates the definition of an epimorphism.

In the category **Set** an epimorphism corresponds to a surjective function.

An epimorphism is a monomorphism in the *dual category*. A dual category of a category C, denoted as C^{op}, has the same objects as C and as arrows all arrows of C inverted, i.e. if $f{:}A \rightarrow B$ is an arrow in C then $f^{\mathrm{op}}{:}B \rightarrow A$ is an arrow of C^{op}. As a result the composition of arrows in the

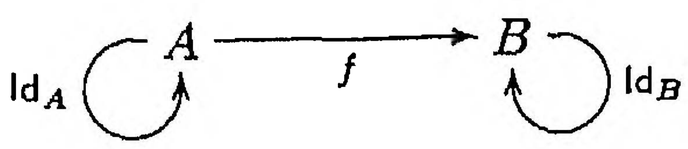


FIGURE 1. A simple example of a category.

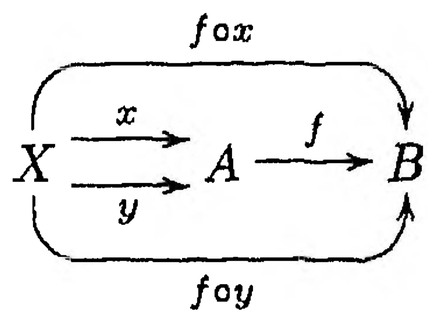


FIGURE 2. Illustration of the definition of a monomorphism.

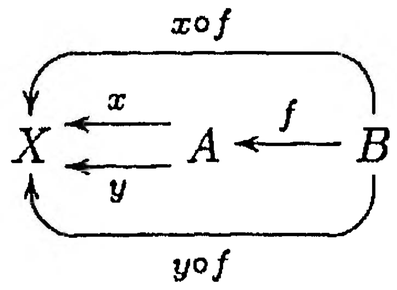


FIGURE 3. Illustration of the definition of an epimorphism.

dual category is defined on the inverted arrows. The concept of duality in category theory is very important as it reduces proof obligations: the dual of a theorem is also a theorem.

The category theoretic equivalent of the set theoretic concept of a bijective function is called an *isomorphism*. In a mathematical context isomorphism means indistinguishable in form. As remarked in [29]:

> Isomorphisms are important in category theory since arrow-theoretic descriptions usually determine an object to within an isomorphism. Thus isomorphisms are the degree of 'sameness' that we wish to consider in categories.

DEFINITION 2.5. An arrow $f:A \to B$ is said to be an *isomorphism* if an arrow $g:B \to A$ exists such that $f \circ g = \mathrm{Id}_B$ and $g \circ f = \mathrm{Id}_A$. Arrow f is called the *inverse* of arrow g and vice versa. If such a pair of arrows exists between two objects A and B, A is *isomorphic* with B, which is denoted as $A \cong B$. The identity arrows are the *trivial isomorphisms*.

There are also some objects with special properties.

DEFINITION 2.6. An object T of a category $\mathcal{C}$ is called a *terminal* object if there is exactly one arrow $A \to T$ for each object A of $\mathcal{C}$. Terminal objects are denoted by 1. The dual notion, an object of a category that has a unique arrow to each object (including itself), is called an *initial* object and denoted as 0.

As terminal (initial) objects are isomorphic, one usually speaks of *the* terminal (initial) object of a certain category.

The initial object in **Set** is the empty set. The terminal objects in **Set** are all singleton sets. In the category **Rel** the empty set is both initial and terminal.

2.3. Diagrams

Many categorical definitions and proofs employ diagrams. As remarked before, quite complex facts can be visualized by the use of these diagrams. The following definition defines what a diagram is.

DEFINITION 2.7. Let $\mathcal{I}$ and $\mathcal{G}$ be graphs. A *diagram* in $\mathcal{G}$ of *shape* $\mathcal{I}$ is a homomorphism $D:\mathcal{I} \to \mathcal{G}$ of graphs. $\mathcal{I}$ is called the *shape graph* of the diagram D.

The following example, taken from [15], illustrates some subtleties involving the concept of diagram.

EXAMPLE 2.1. Let $\mathcal{G}$ be a graph with objects A, B and C and arrows $f:A \to B$, $g:B \to C$, and $h:B \to B$. The following diagram

$$A \xrightarrow{f} B \xrightarrow{g} C$$

can then be formally defined, using the shape graph $\mathcal{I}$,

$$1 \xrightarrow{u} 2 \xrightarrow{v} 3$$

as the homomorphism $D:\mathcal{I} \to \mathcal{G}$ with $D(1) = A$, $D(2) = B$, $D(3) = C$, $D(u) = f$, and $D(v) = g$. The following diagram is just like D (has the same shape) except that v goes to h and 3 goes to B.

$$A \xrightarrow{f} B \xrightarrow{h} B$$

The following diagram has a different shape graph as the two diagrams considered before.

$$A \xrightarrow{f} B \circlearrowright h$$

Formally it corresponds to a diagram $E:\mathcal{J} \to \mathcal{G}$, where the shape graph $\mathcal{J}$ is defined by

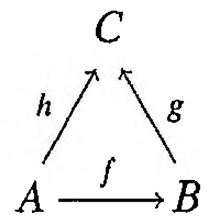
$$1 \xrightarrow{u} 2 \circlearrowright w$$

with $E(1) = A$, $E(2) = B$, $E(u) = f$ and $E(w) = h$. □

The notion of a *commutative* diagram plays a central role in category theory. Categorical proofs and definitions often use diagrams and prove or require them to commute. Commutative diagrams are the categorist's way of expressing equations.

DEFINITION 2.8. A diagram is said to commute if every path between two objects in its image determines through composition the same arrow. □

EXAMPLE 2.2. The following diagram commutes if and only if h is the composite $g \circ f$.



□

2.4. Products and coproducts

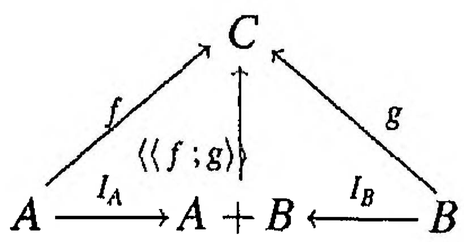
In the disjoint union of a number of sets, elements originating from different sets can always be distinguished. The disjoint union of two sets can be defined in several ways. A possible definition of the disjoint union $A + B$ of two sets A and B is

$$A + B = \{\langle a, 0\rangle \mid a \in A\} \cup \{\langle b, 1\rangle \mid b \in B\},$$

with canonical injections I_A and I_B, i.e. $I_A(a) = \langle a, 0\rangle$ and $I_B^2(b) = \langle b, 1\rangle$. The categorical definition of a *coproduct* (also referred to as *sum*) generalizes this definition. In particular, it does not prescribe a representation.

DEFINITION 2.9. A *coproduct* of two objects A and B in a category consists of an object $A + B$ together with arrows $I_A:A \to A + B$ and $I_B:B \to A + B$ such that for any arrows

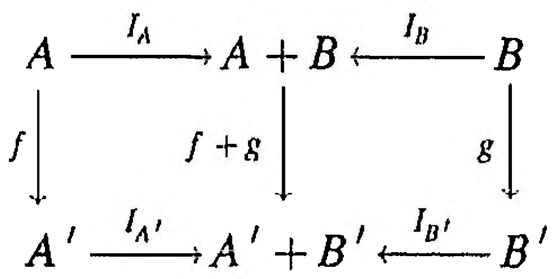
$f:A \to C$ and $g:B \to C$, there is a unique arrow, denoted as $\langle\langle f;g\rangle\rangle:A+B \to C$, for which the following diagram commutes:



$I_A:A \to A+B$ and $I_B:B \to A+B$ are called *injection arrows* of the sum. □

The definition of a coproduct can be generalized, in a straightforward manner, to be applicable to any number of objects in a category. Coproducts can also be defined for arrows. In the category **Set**, the coproduct of two arrows $f:A \to A'$ and $g:B \to B'$ is a function $f+g:A+B \to A'+B'$. If this function is applied to an element x of the disjoint union $A+B$ it either yields $f(x)$ or $g(x)$, depending on whether x originates from A or B, respectively.

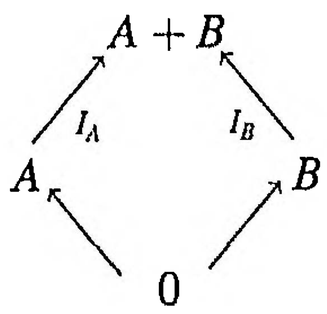
DEFINITION 2.10. A *coproduct* of two arrows $f:A \to A'$ and $g:B \to B'$ is an arrow $f+g:A+B \to A'+B'$ such that the following diagram commutes:



□

Sums in the category of sets have special properties they do not have in most other categories. One such property is that sums in **Set** are *disjoint*. In a disjoint sum the sum injection arrows must be monomorphisms.

DEFINITION 2.11. Let A and B be two objects in a category with an initial object 0 and a coproduct $A+B$. Then the following diagram commutes.



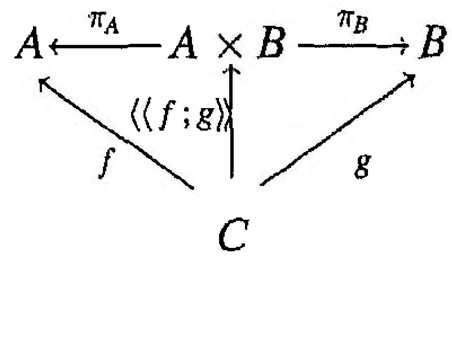
If this diagram is a pullback (i.e. it is a universal commutative cone, see definition 2.15) and the canonical injections I_A and I_B are monomorphisms, then the coproduct $A+B$ is a *disjoint coproduct*. □

In several interesting categories (e.g. **Set**) monomorphisms are *complementable*:

DEFINITION 2.12. An arrow $f:A \to B$ is *complementable* iff a $g:C \to B$ exists such that B is isomorphic with $A+C$ with f and g as the sum injection arrows. In this case g is a *complement* of f. The object C is frequently denoted as $B-A$.

The dual notion of coproduct is *product*. In the category **Set** a product corresponds to the notion of a cartesian product with associated projection functions.

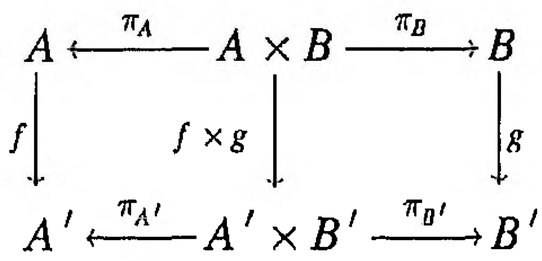
DEFINITION 2.13. A *product* of two objects A and B in a category consists of an object $A \times B$ together with arrows $\pi_A:A \times B \to A$ and $\pi_B:A \times B \to B$ such that for any arrows $f:C \to A$ and $g:C \to B$, there is a unique arrow, denoted as $\langle\langle f,g\rangle\rangle:C \to A \times B$, such that the following diagram commutes:



□

As with coproducts, this definition can be extended to arrows in a straightforward manner.

DEFINITION 2.14. A *product* of two arrows $f:A \to A'$ and $g:B \to B'$ is an arrow $f \times g:A \times B \to A' \times B'$ such that the following diagram commutes:



□

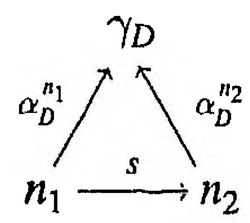
2.5. Limits and colimits

Limits and colimits are dual notions. Both concepts are very general and often used in category theory.

A *limit* is the categorical version of the concept of an equationally defined subset of a product. A product, therefore, is a special kind of limit. A *colimit* is the categorical version of a quotient of a sum by an equivalence relation. A coproduct, therefore, is a special kind of colimit. Only the definition of a colimit is given as the general notion of limit is not important in the context of this paper.

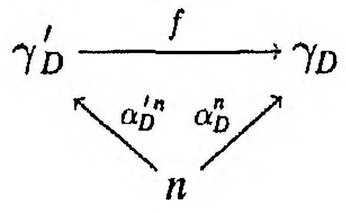
DEFINITION 2.15. Let $\mathcal{G}$ be a graph and $\mathcal{C}$ be a category. Let $D:\mathcal{G} \to \mathcal{C}$ be a diagram in $\mathcal{C}$ with shape $\mathcal{G}$. A *cocone* with *base* D is an object γ_D (*apex*) together with a family $\{\alpha_D^n\}$ of arrows of $\mathcal{C}$ indexed by the nodes of $\mathcal{G}$, such that $\alpha_D^n:n \to \gamma_D$ for each node of $\mathcal{G}$. The arrow α_D^n is the *component* of the cocone at n. The cocone is written as $\{\alpha_D^n\}:D \to \gamma_D$, or simply $\alpha_D:D \to \gamma_D$.

The cocone is *commutative* if for any arrow $s:n_1 \to n_2$ of $\mathcal{G}$, the following diagram commutes.



If $\alpha'_D:D \to \gamma'_D$ and $\alpha_D:D \to \gamma_D$ are cocones, an *arrow*

from the first to the second is an arrow $f: \gamma'_D \rightarrow \gamma_D$ such that for each node n of $\mathcal{G}$, the following diagram commutes.



A commutative cocone with base D is called *universal* if it has a unique arrow to every other commutative cocone with the same base. A universal cocone, if such exists, is called a *colimit* of the diagram D. □

3. DATA MODELLING TYPE CONSTRUCTORS

In this section a number of important conceptual data modelling concepts are given a category theoretic foundation. First, however, it is necessary to define a uniform syntax of conceptual data models that is as general as possible. In section 3.1, conceptual data models are defined by means of *type graphs*. The semantics of a data model is the set of possible populations, i.e. instantiations of its structure. Populations are formalized via the notion of *type models*, defined in subsection 3.2. After the definition of type models, the various data modelling constructs are given a category theoretic definition. These constructs are defined in terms of restrictions on type models.

3.1. Type graphs

Data models can be represented by *type graphs* (see also [25] and [16]). The various object types in the data model correspond to nodes in the graph, while the various constructions can be discerned by labelling the arrows. Relationship types, for example, correspond to nodes. An object type participating via a role in a relationship type is target of an arrow labelled with role, which has as source that relationship type. As an object type may participate via several roles in a relationship type a type graph has to be a *multi*graph.

DEFINITION 3.1. A *type graph* $\mathcal{G}$ is a directed multigraph over a label set {role, spec, gen, elt_role, clt_role}. Edges with label spec or gen are called *subtype* edges, The type graph may not contain cycles consisting solely of subtype edges. Further, there is a bijective function clt from edges with label clt_role to edges with label elt_role such that related edges have identical sources. The function type yields the label of an edge.

An edge e, labelled with role, from a node A to a node B indicates that A is a relationship type in which B plays a role. If e is labelled with spec, then A is a specialization of B, while if e is labelled with gen then B is a generalization of A (and possibly other object types). If edge $e: A \rightarrow B$ is labelled with clt_role, edge $f: A \rightarrow C$ is labelled with elt_role and $\text{clt}(e) = f$, then B is a collection type with as element type C (collection types will be explained in depth in subsection 3.5).

The definition of a type graph is very liberal, only cyclic subtype structures are (obviously) excluded. The definition allows a node to be a collection type as well as a relationship type, a binary relationship type to be a subtype of a ternary relationship type, a collection type to have several element types etc. Excluding these 'peculiarities' from data models turns out to be unnecessary from a theoretical point of view as it is possible to give such data models a formal semantics. Hence, restrictions, other than on cyclic subtype structures, will not be imposed.

As an example of how data models can be represented as type graphs, consider the type graph in Figure 5, which represents the NIAM data model in Figure 4. Object types in NIAM are represented as circles, roles as boxes and arrows between circles represent subtype relations (for a complete overview of the graphical conventions of NIAM refer to [5]).

3.2. Type models

The semantics of a data model is the set of all possible instantiations, also referred to as *populations*. In our approach, a population is defined as a *model* from the type graph to a category. A model is a graph homomorphism from a graph to a category (interpreted as a graph).

DEFINITION 3.2. Given a category F, a *type model* for a given type graph $\mathcal{G}$ in F, is a model $M: \mathcal{G} \rightarrow F$. F is referred to as the *instance category* of the model.

A type model maps the object types in the type graph onto objects in the instance category and the edges onto arrows in this category. To avoid notational clutter, the model is sometimes omitted if it is clear from the context. For example, the product of two object types is sometimes written as $A \times B$ instead of $M(A) \times M(B)$.

At this point no requirements on the mapping of edges in relation to their labels is imposed. These requirements will be discussed in the remainder of this section and will lead to the definition of a *valid* type model in subsection 3.6.

The above definition implies that the semantics of a data model depends on the instance category chosen. Not all categories provide a meaningful semantics for data models. Instance categories are required to be members of a class of categories **Fund**. Categories of this class have to fulfill a number of requirements that will be discussed in section 3.7.

In Figure 6, some examples of categories in **Fund** are shown. The label of each arrow denotes a feature that exists in the category that is target of that arrow, but not in the category that is source of that arrow. For example, in the category **PartSet** functions do not have to be total, contrary to the category **Set**. As will be shown in subsection 3.3, this category should be considered if one is interested in the study of 'null'-values in relationship types. Other categories in Figure 6 are:

- The category **TotRel** where the objects are sets and the arrows total relations.
- The category **Bag** where the objects are bags (multisets)

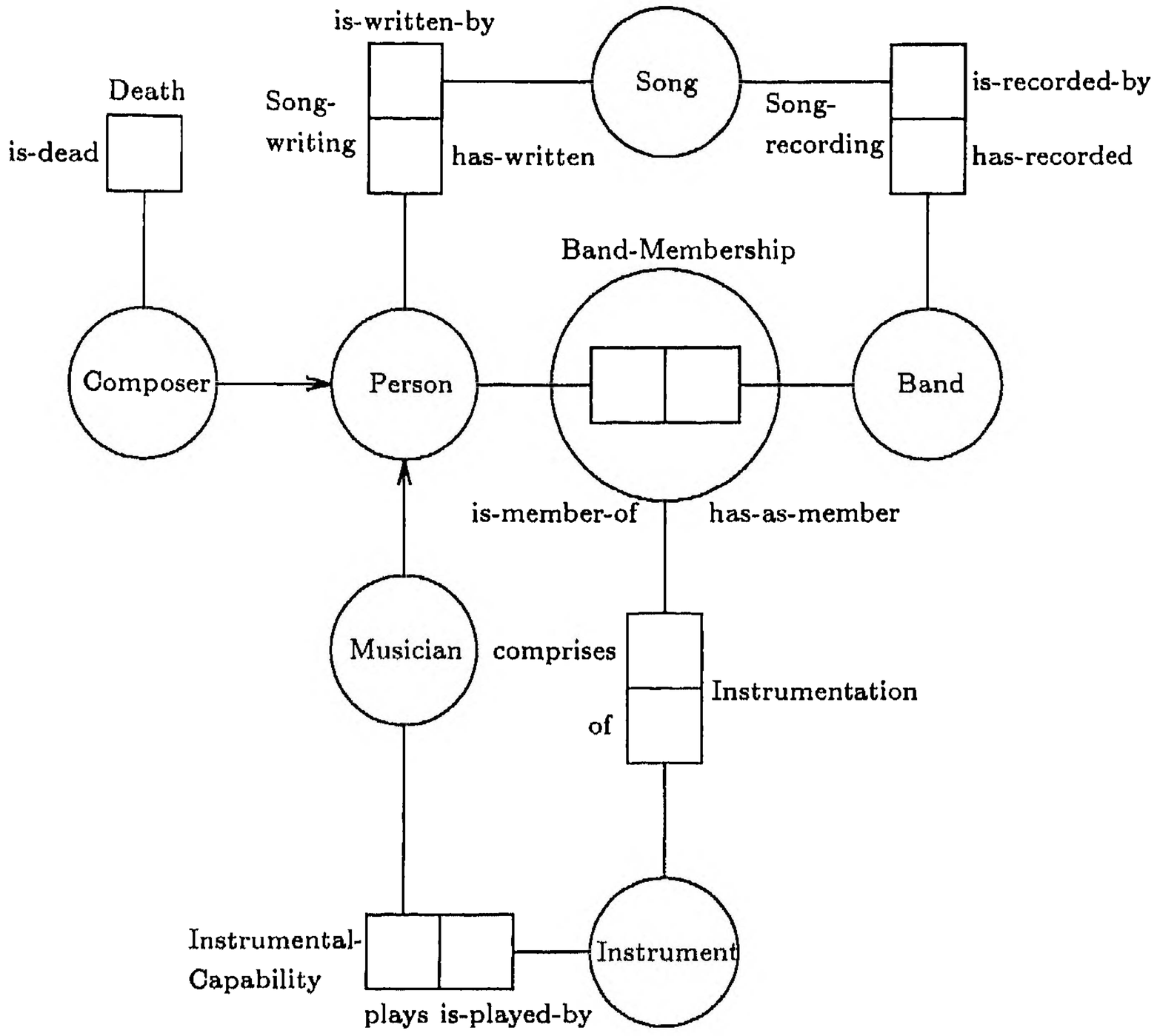


FIGURE 4. A NIAM data model.

and the arrows total functions, such that the frequency of an original never exceeds the frequency of an image.

- The category **Poset** where the objects are partially ordered sets and the arrows monotonous (i.e. order-preserving) functions.
- The category **FuzzySet** where the objects are fuzzy sets and the arrows special total functions on these sets. A fuzzy set is a pair $\langle S, \sigma \rangle$ where S is a set and σ is a total function on S assigning to each element of S the degree of membership. An arrow $f: \langle S, \sigma \rangle \rightarrow \langle T, \tau \rangle$ is a function $f: S \rightarrow T$ such that $\sigma \leq \tau \circ f$.

3.3. Relationship types

One of the central concepts in conceptual data modelling is the concept of *relationship type*. A relationship type represents an association between object types and may be n-ary in some data modelling techniques (where $n \geq 1$), as well as play a role in other relationship types. Yourdon [30] refers to such relationship types as *associative object type indicators*, while in NIAM relationship types participating in other relationship types are called *objectified fact types*. A relationship type consists of a number of roles, capturing the way object types participate in that relationship type.

In the past, relationship types have often been formalized by viewing them as subsets of a cartesian product. This has commonly been referred to as the *tuple-oriented approach*. As an example consider Figure 7 which depicts an ER schema with a relationship type R consisting of roles p and q played by entity types A and B, respectively. A population

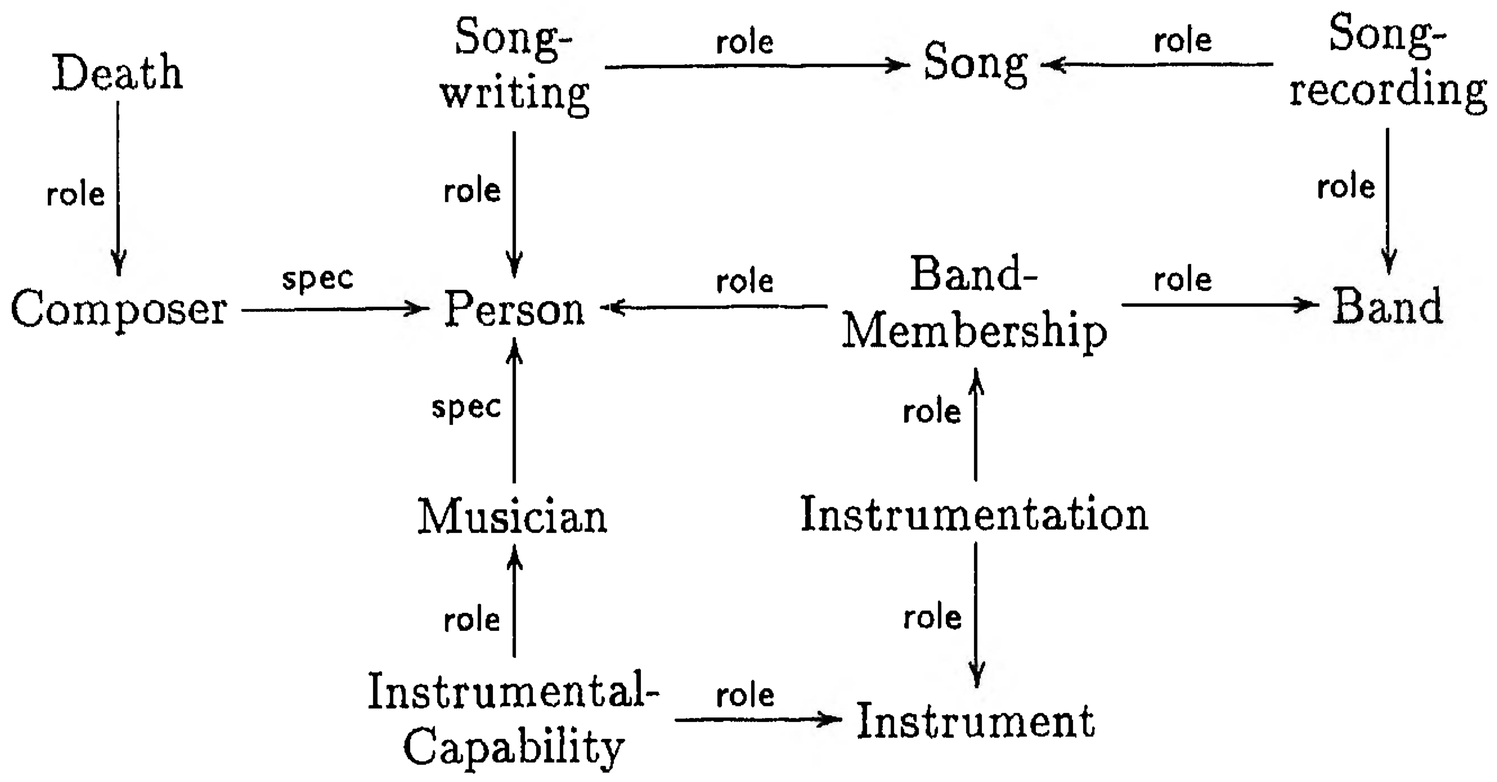


FIGURE 5. Type graph of the schema of Figure 4.

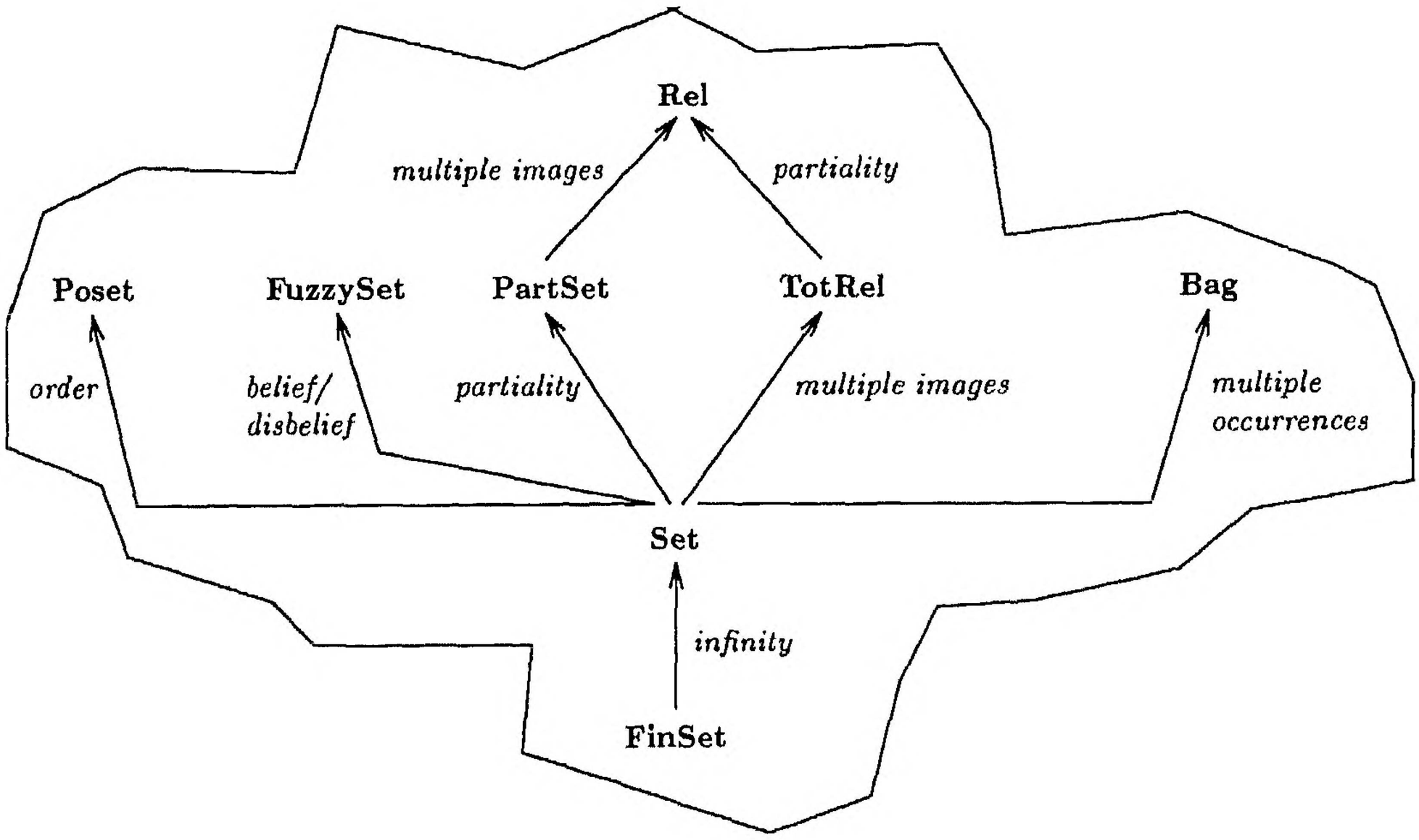


FIGURE 6. The class of categories **Fund**.

of this relationship type, represented in the tuple-oriented approach, could be:

$$\mathsf{Pop}(R) = \{\langle a_1, b_1\rangle, \langle a_2, b_1\rangle\}.$$

The disadvantages of the tuple-oriented approach are obvious: the representation of instances is overly specific. Instances of relationship type R could as well be considered elements of the product $\mathsf{Pop}(B) \times \mathsf{Pop}(A)$ as $\mathsf{Pop}(A) \times \mathsf{Pop}(B)$. A cartesian product imposes an ordering on the various parts of the relation. Consequently, the cartesian product does not have important properties such as commutativity and associativity. This observation has led to the *mapping-oriented approach* [31], where relationship instances are treated as functions from the involved roles to values. In this approach, the above sample population would be represented as:

$$\mathsf{Pop}(R) = \{\{p \mapsto a_1, q \mapsto b_1\}, \{p \mapsto a_2, q \mapsto b_1\}\}.$$

Clearly, this approach does not suffer from the drawbacks of the tuple-oriented approach. No ordering is imposed, while at the same time the various parts of a relation remain distinguishable.

Still, however, one may argue that the mapping-oriented approach imposes unnecessary restrictions. Why do instances have to be represented as *functions*? Is not it sufficient to have access to their various parts? The categorical approach pursues this line of thought. The actual representation of relationship instances becomes irrelevant, their components become available by 'access-functions'. As an example consider the interpretation of the sample population in the category **FinSet**. The type graph of the schema of Figure 7 is shown in Figure 8. Category theoretically, a population corresponds to a mapping from the type graph to an instance category. The sample population therefore, could be represented as (note that there are many alternatives!):

$$p = \{r_1 \mapsto a_1, r_2 \mapsto a_2\},$$
$$q = \{r_1 \mapsto b_1, r_2 \mapsto b_1\}.$$

In this approach, the two relationship instances, r_1 and r_2, have an identity of their own, and the functions p and q can be applied to retrieve the respective components. Note that in this approach it is possible that two different relationship instances consist of exactly the same components.

Apart from **FinSet** it is also possible to choose other instance categories. As remarked before, the category **PartSet** allows certain components of relationship instances to be undefined:

$$p = \{r_2 \mapsto a_2\},$$
$$q = \{r_1 \mapsto b_1, r_2 \mapsto b_1\}.$$

In this population, relationship instance r_1 does not have a corresponding object playing role p.

Another possible choice of instance category is the category **Rel**. In **Rel** the components of relationship instances correspond to sets, as roles are mapped on relations. A relationship instance may be related to one or more objects in one of its components. A sample population

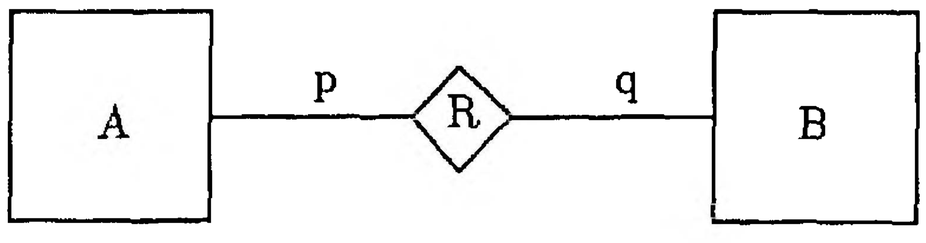


FIGURE 7. A simple ER schema.

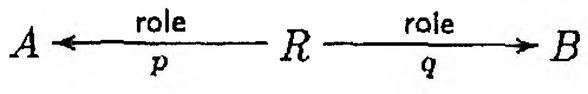


FIGURE 8. Type graph of Figure 7.

could be:

$$p = \{r_2 \mapsto a_1, r_2 \mapsto a_2\},$$
$$q = \{r_1 \mapsto b_1, r_2 \mapsto b_1, r_2 \mapsto b_2\}.$$

3.4. Subtype relationships

Many conceptual data modelling techniques offer concepts for expressing subtype relations. Subtype relations are used to capture inheritance of properties. In the literature many types of inheritance relations exist and the terminology is far from standard. In this section two important types of inheritance relations are considered: specialization and generalization. Many conceptual data modelling techniques contain at least one of these relations, although probably under a different name. The concepts of specialization and generalization in this paper correspond to a large extent to specialization and generalization as defined in IFO [8].

3.4.1. Specialization

Specialization is used when specific facts are to be recorded for only specific instances of an object type. A specialized object type inherits the properties of its supertype(s), but may have additional properties. As such, specialization corresponds to the notion of *subtyping* in NIAM.

As an example of specialization consider the IFO schema of Figure 9 (adapted from [8]). In this schema the boxes represent concrete types, the diamonds represent abstract types and the circles represent subtypes. The double arrows denote specialization relations. Therefore, in this diagram *STUDENT* is a subtype of *PERSON*. The object type *TEACHING-ASSISTANT* is a subtype of both *STUDENT* and *EMPLOYEE*. The subtype hierarchy has been created to express that only for certain types certain facts are to be recorded, e.g. only for employees the salary is relevant. As remarked before, properties are inherited 'downward', e.g. employees have a name as they are also persons.

In set-theoretic terms, the most general formalization of a subtype relation would be to treat it as an injective function. This is more general than requiring that $\mathsf{Pop}(A) \subseteq \mathsf{Pop}(B)$ in the case that A is a subtype of B, as instances may have a different representation in both object types (this is particularly so in object-oriented data models). Therefore, category, theoretically a subtype relation, has to correspond with a monomorphism (recall that in the category **Set** a monomorphism corresponds to an injective function). This is not sufficient, however, for an adequate formalization of specialization relations. Consider for example the following partial population of the schema of Figure 9:

$$\mathsf{Pop}(\text{PERSON}) = \{\text{Jones, Richards}\},$$
$$\mathsf{Pop}(\text{STUDENT}) = \{\text{ST1943}\},$$
$$\mathsf{Pop}(\text{EMPLOYEE}) = \{\text{EM237}\},$$
$$\mathsf{Pop}(\text{TEACHING-ASSISTANT}) = \{\text{TA999}\}.$$

and the following subtype relations (see also Figure 10):

$$I_1 = \{\text{TA999} \mapsto \text{EM237}\},$$
$$I_2 = \{\text{TA999} \mapsto \text{ST1943}\},$$
$$I_3 = \{\text{EM237} \mapsto \text{Jones}\},$$
$$I_4 = \{\text{ST1943} \mapsto \text{Richards}\}.$$

In this sample population, with as instance category **Set**, the instance TA999 of object type *TEACHING-ASSISTANT* corresponds to two instances of *PERSON*: *Richards* as well as *Jones*. Clearly, this is undesirable.

To avoid such problems, subtype diagrams, i.e. diagrams consisting solely of subtype edges, are required to commute. In terms of the presented subtype diagram this would imply that the function composition of I_2 with I_4 should be identical to the function composition of I_1 with I_3 and therefore: $I_4(I_2(\text{TA999})) = I_3(I_1(\text{TA999}))$.

Since the subtype diagram is required to commute, subtypes inherit properties from their supertypes in a unique way. In the example, every teaching assistant inherits the name from its supertype person.

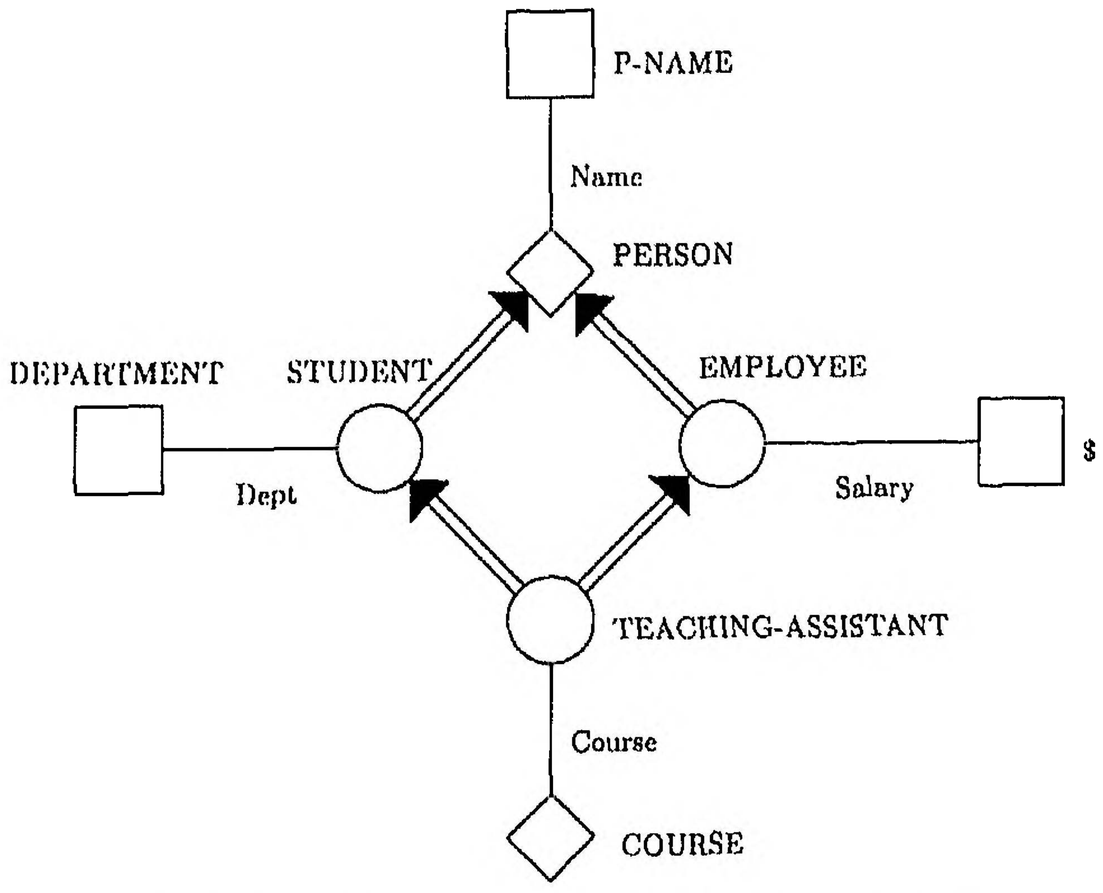


FIGURE 9. An example of a subtype hierarchy in IFO.

3.4.2. Generalization

Generalization is a mechanism that allows for the creation of new object types by uniting existing object types. Contrary to what its name suggests, generalization is *not* the inverse of specialization. Specialization and generalization originate from different axioms in set theory [10, 11].

The population of a generalized object type is the union of the populations of the participating object types, referred to as the *specifiers*.

As an example of generalization consider Figure 11. In this schema the graphical conventions of PSM [10] have been used, the dashed lines represent generalization relations. This PSM schema models the construction of simple formulas: a *Formula* may be either a *Variable* or an expression constructed by some function F from simpler formulas. This example demonstrates that generalization can be used for the specification of recursive types. Generalization is also useful when identical properties are relevant for different existing types: these properties can then be related to the generalization of these types.

The application of coproducts yields a possible categorical formalization of generalization. The generalized object

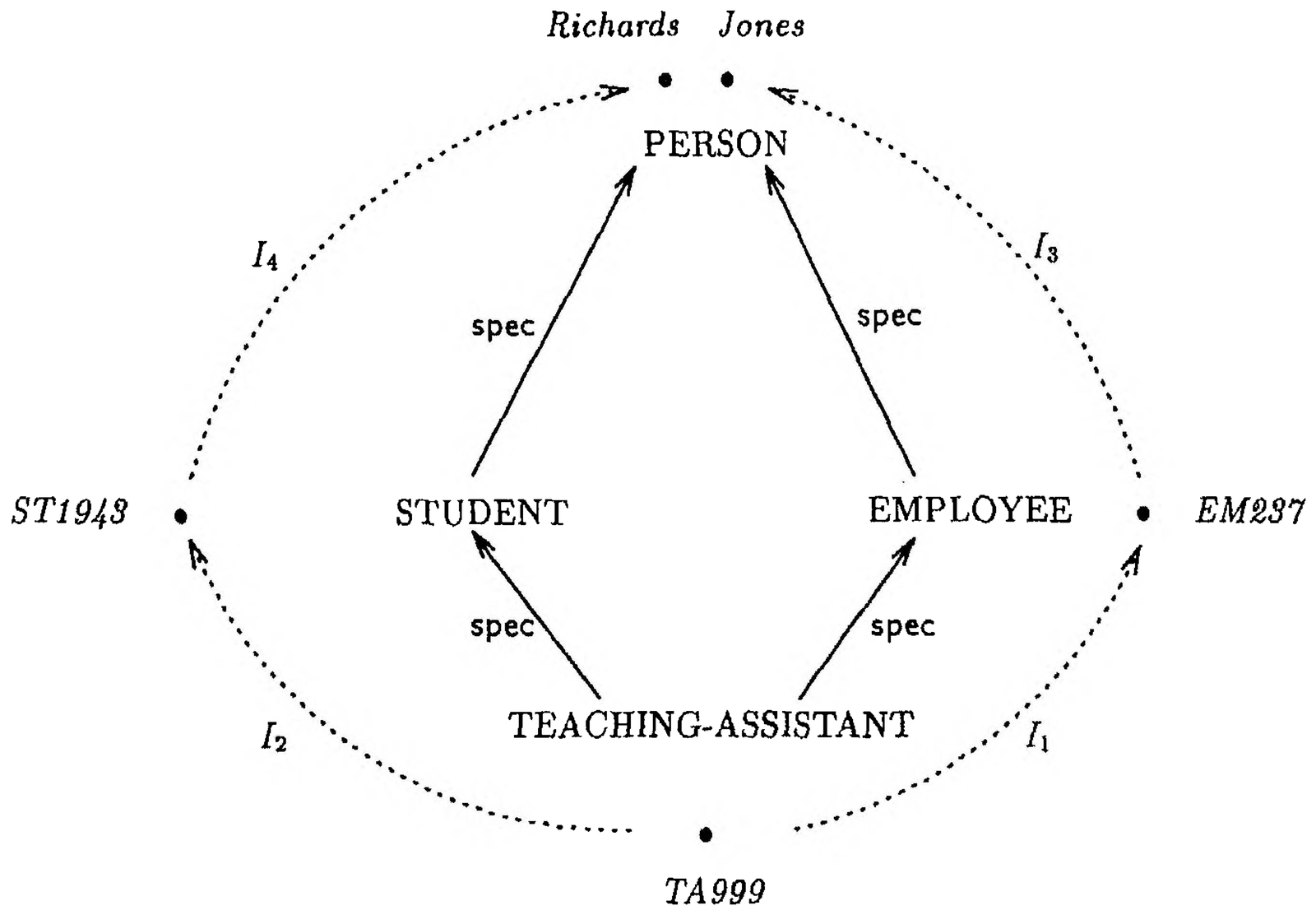


FIGURE 10. A non-commutative diagram.

type has to be mapped on a coproduct in the instance category and the generalization arrows should correspond to the sum injections. Of course, as the coproduct represents a *disjoint* sum in **Set**, this formalization implies that specifiers have to be disjoint. In some data modelling techniques (including PSM) this is not necessarily true. This problem can be solved by using the general notion of colimit.

The solution starts with the observation that the collection of instances of a generalized type with a set of specifiers V is completely determined by the subtype relationships among the subtypes of elements in V. The following definitions give a formal description of a diagram that only contains the relevant subtype relations among subtypes of elements of V.

DEFINITION 3.3. Given a graph $\mathcal{G}$ and a set of nodes $N \subseteq \mathcal{G}_0$, *the subgraph of* $\mathcal{G}$ *dominated by* N is equal to a subgraph D of $\mathcal{G}$ that is defined as follows: The edges of D are the edges from $\mathcal{G}_1$ that occur on a directed path that ends in a node $n \in N$. The nodes of D are the nodes that occur in one of its edges. □

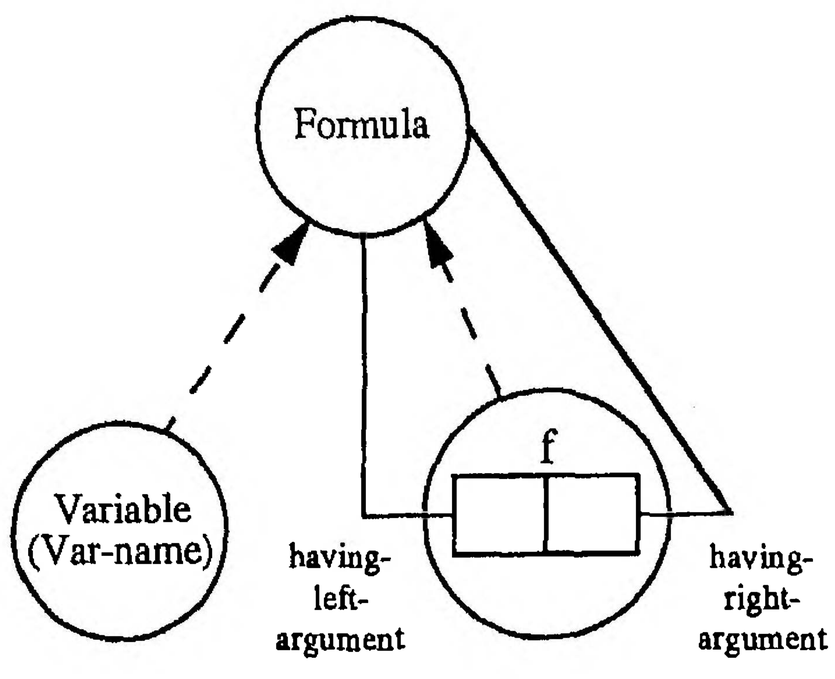


FIGURE 11. An example of generalization in PSM.

DEFINITION 3.4. Given a diagram $D: \mathcal{G} \rightarrow C$ and a set of nodes $V \subseteq \mathcal{G}_0$. Let $\mathcal{G}_V$ be the subgraph of $\mathcal{G}$ dominated by V. Then, D *dominated by* V is equal to D functionally restricted to $\mathcal{G}_V$. □

The instance universe U_M^V represents the collection of all instances of a set V of object types in a model M. The instance universe is used as the generalization of a set V of specifiers.

DEFINITION 3.5. The *instance universe* determined by a set of object types $V \subseteq \mathcal{G}_0$ in a given type model M, denoted as U_M^V, is the apex of the universal cocone with as base the subtype diagram dominated by V. □

In [32] it is proven that in a category that has disjoint sums the colimit of a diagram consisting of complementable monomorphisms, which is true for the subtype diagram of definition 3.5., always exists. The associated arrows are then also complementable monomorphisms. This result is important as some categories have disjoint sums, but do not have all colimits (e.g. **Rel**). Therefore, rather than requiring instance categories to have all colimits, it is required that all finite sums exist and are disjoint, as this is less restrictive.

Finally, it should be pointed out that as a result of the definition of subtype diagrams, the commutativity requirement imposed on these diagrams also applies to generalization.

3.5. Collection types

A *collection type* is an object type of which each instance corresponds to a (nonempty) set of instances of another object type. This latter object type is referred to as the *element type* of the collection type. As sets are identical if,

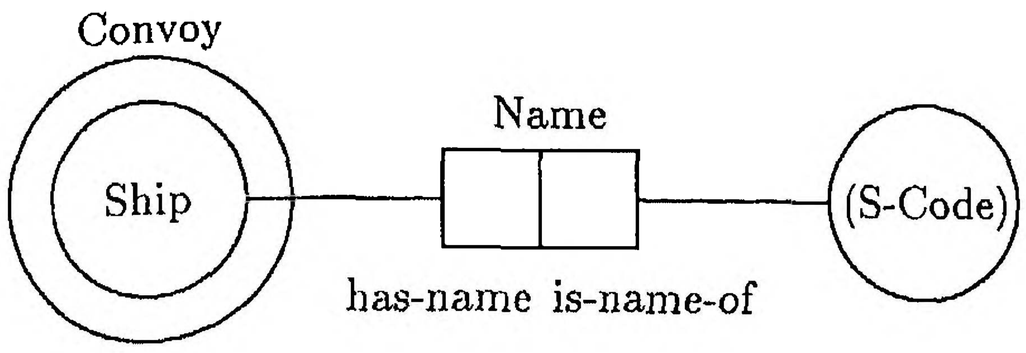


FIGURE 12. An example of a collection type in PSM.

and only if, they contain the same elements, the instances of a collection type are identified by their elements and do not need external identifications. Collection types correspond to *grouping* in IFO, *association* in ECR [7], *grouping classes* in SDM [33], and *power types* in PSM.

As a simple example of the application of collection types consider the schema of Figure 12, which shows a PSM schema of the so-called *Convoy Problem* of [33]. In this schema the object type *Convoy* is a collection type with as element type *Ship*. Ships are identified by a code (*S-code*), while convoys are identified by their constituent ships.

There are several alternatives for a categorical formalization of collection types. One alternative is to require the instance category to be a special kind of category called a topos. This approach has two serious disadvantages, however. First, a topos is a complex type of category, which is not easily understood. Secondly, and more seriously, many interesting categories are not topoi. The use of topoi therefore would imply an extra, very restrictive, requirement on the class of instance categories **Fund**. Another alternative would be the use of *sketches* in order to allow the general specification of algebraic types [15]. Unfortunately, it turns out that such a solution also imposes too many restrictions on **Fund**.

The approach adopted in this paper, does not suffer from the problems outlined in the previous paragraph and is based on an alternative treatment of collection types, as presented in [34]. As pointed out in this paper, collection types become superfluous by the introduction of a new type of constraint, the *existensional uniqueness constraint*, as well as a new identification scheme. As an example consider Figure 13. The existensional uniqueness constraint in this schema expresses that no two convoys may be associated, via role *sails in*, to the same set of ships. As such this constraint captures the extensionality property of sets. Also, the object type *Convoy*, may be identified, via this role, by the object type *Ship*.

To illustrate further the existensional uniqueness constraint, consider the abstract schema of Figure 14. The sample population of this schema violates the existensional uniqueness constraint as both a_1 and a_2 are related, via role

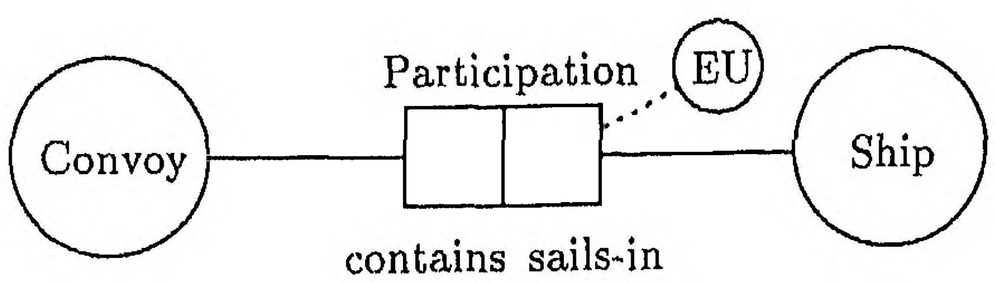
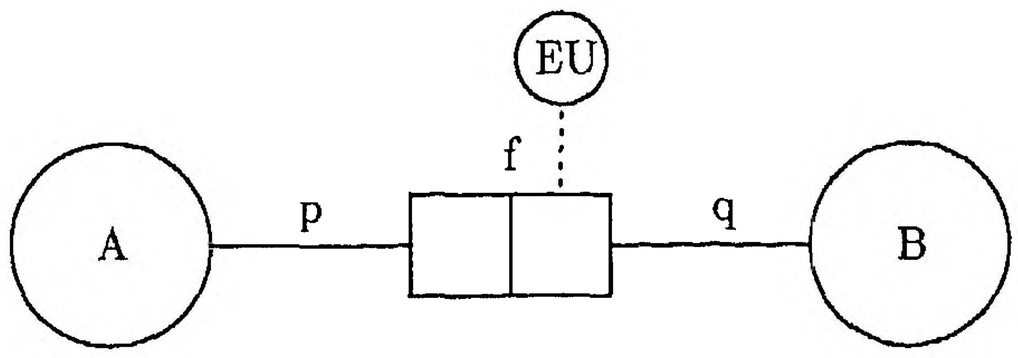


FIGURE 13. A translation of the Convoy Problem.



f	$p(f)$	$q(f)$
f_1	a_1	b_1
f_2	a_1	b_2
f_3	a_2	b_1
f_4	a_2	b_2

FIGURE 14. A population violating the existensional uniqueness constraint.

q, to b_1 and b_2 and therefore both correspond to the set $\{b_1, b_2\}$.

The solution to the categorical formalization of the existensional uniqueness constraint follows from the observation that such a constraint is violated if and only if a non-trivial permutation of the 'set-like' instances exists such that application to the population of the involved relationship type yields *the same* population. In other words, if changing the members of two sets (which have received their own identity!) does not lead to a loss of information, then obviously these two sets have to have identical representations. In the sample population the interchange of a_1 and a_2 in each instance of f, does not lead to a change in the population of relationship type f.

Category theoretically, this requirement states that the existensional uniqueness constraint of the schema of Figure 14 is violated if, and only if, the arrows p and q are mapped onto arrows in the instance category such that non-trivial isomorphisms (i.e. isomorphisms not equal to the identity) $\diamond_A$ and $\diamond_f$ on the objects, corresponding to the collection type A and the involved relationship type f, respectively, can be found for which the following equalities hold (see also the generic type model in Figure 15):

$$\diamond_A \circ p\diamond_f = p,$$
$$q \circ \diamond_f = q.$$

The edges p and q are said to fulfill the *extensionality property*. Obviously, this definition does not impose any requirement on the instance category involved.

As an example of the application of this definition, again consider the sample population of Figure 14. Suppose that the instance category involved is the category **Set**. The following two choices for the permutations $\diamond_A$ and $\diamond_f$ satisfy the imposed requirements, as they are non-trivial

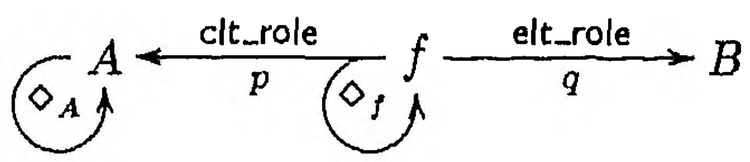


FIGURE 15. A solution for collection types.

isomorphisms and satisfy the two equalities:

$$\Diamond_A = \{a_1 \mapsto a_2, a_2 \mapsto a_1\},$$
$$\Diamond_f = \{f_1 \mapsto f_3, f_3 \mapsto f_1, f_2 \mapsto f_4, f_4 \mapsto f_2\}.$$

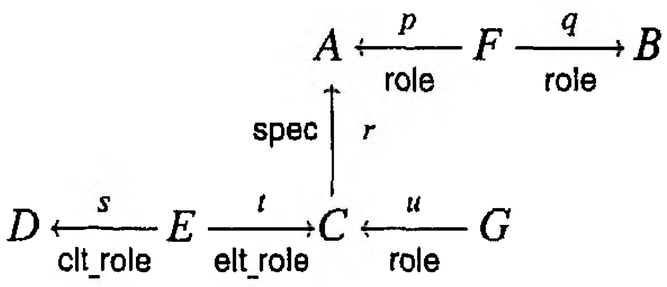
3.6. Valid type models

Now the full definition of a valid type model for a type graph can be presented:

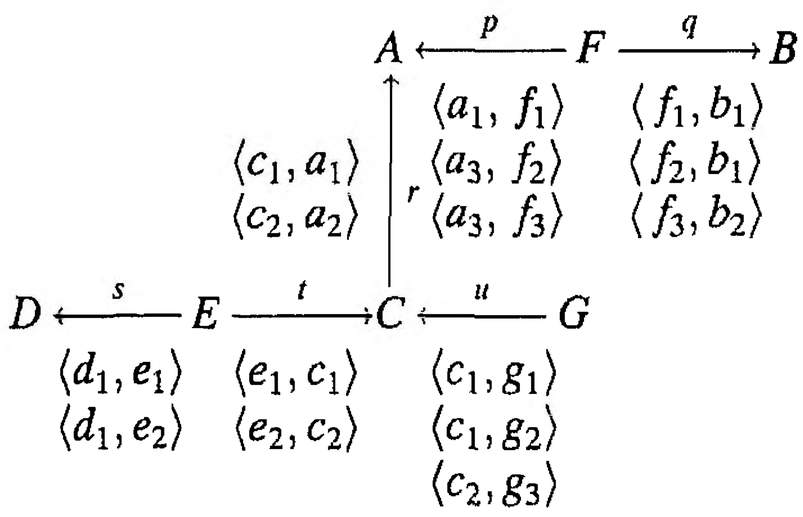
DEFINITION 3.6. A type model $M: \mathcal{G} \rightarrow F$ for a given type graph $\mathcal{G}$ in a category F, is a *valid type model* iff,

1. if x is an edge of $\mathcal{G}$ and $\mathsf{type}(x) = \mathsf{spec}$ then $M(x)$ is a complementable monomorphism.
2. if x is an edge of $\mathcal{G}$ and $\mathsf{type}(x) = \mathsf{gen}$ then $M(x) = \alpha_D^{\mathsf{source}(x)}$, where D is equal to the subtype diagram dominated by the specifiers of $\mathsf{target}(x)$.
3. the subtype diagram of M commutes.
4. if x and y are edges of $\mathcal{G}$, with $\mathsf{clt}(y) = x$ then $M(x)$ and $M(y)$ have to fulfill the extensionality property.

EXAMPLE 3.1. The following type graph describes a simple conceptual data model.



The following is a type model of this type graph in **Set**. The value of the set of elements for each object is equal to the elements that occur in the corresponding arrows and has therefore been omitted from the figure.



This type model is indeed a valid type model. There is one specialization arrow from C to A that is an injective function, and in **Set** all injective functions are complementable monomorphisms. Obviously, the subtype diagram commutes since it only contains one specialization arrow. Collection type D has one instance that represents the set $\{c_1, c_2\}$. It is not difficult to see that s and t fulfill the extensionality property. □

3.7. Valid instance categories

One of the most important advantages of using a categorical approach to the semantics of conceptual data modelling techniques is that different instance categories can be used. The requirements that instance categories should satisfy are listed together with some illustrations.

Instance categories should support the constructions that have been used in the previous sections. This means that every member of **Fund** should have the following properties:

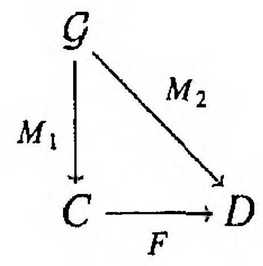
- All finite sums and products must exist.
- Sums must be disjoint.
- An initial object must exist.

Actually, the last requirement is redundant since the initial object is the sum of zero objects. This set of requirements is modest, which implies that there is a large set of possible instance categories.

Some categories, however, are too trivial to be interesting as instance categories, for example the category with only one object and one arrow. Most 'classical' formalizations of conceptual data modelling techniques correspond to a formalization that results from the choice of **FinSet** as instance category. Therefore, it seems reasonable to require that other instance categories have at least the same 'expressive power'. Intuitively, every model in **FinSet** should have a counterpart in other instance categories.

As an introduction to the formalization of this requirement it is useful to define a homomorphism between type models.

DEFINITION 3.7. A *type model homomorphism* between type models $M_1: \mathcal{G} \rightarrow C$ and $M_2: \mathcal{G} \rightarrow D$ is a functor $F: C \rightarrow D$, i.e. a graph homomorphism preserving identities and composition, such that the following diagram commutes:



□

The valid type models and their homomorphisms form a category.

This definition of a type model homomorphism has inspired the following definition of a valid instance category.

DEFINITION 3.8. A category C is a *valid instance category* if all finite products and sums exist, sums are disjoint and there is a functor F: **FinSet** $\rightarrow$ **C** which is a monomorphism in the category of graphs and homomorphisms between graphs.

The following categories are valid instance categories: **FinSet**, **Set**, **PartSet**, **Rel**, **FuzzySet**. A description of various category theory constructs and proofs for these categories can be found in [32].

EXAMPLE 3.2. In several object-oriented databases [35, 36], objects can have multi-valued (or set-valued) attributes. This means that the value of an attribute can be a (possibly empty) set of attribute values. Models in the category **Rel** can be used to model this behavior.

EXAMPLE 3.3. Models in **FuzzySet** can be used to model uncertainties. Every object type A is equipped with a function σ_A that captures the degree of membership of instances. For simplicity's sake, we assume that application of this function yields a probability (for an in-depth treatment of fuzzy sets in a categorical context refer to [15]). The arrows in **FuzzySet** are total functions and for each arrow $f: A \to B$ it must hold that $\sigma_A(a) \leqslant \sigma_B(f(a))$. Therefore, the probability that an individual is an element of a given object type must always be greater or equal to the probability that this individual is an element of one of the subtypes of this object type. Intuitively, this is sensible since if the individual is an element of an object type it must certainly be an element of all supertypes of that type. In addition to that, probabilities of instances of relationship types are less than the probabilities of their parts. If one considers, for example, the relationship type *Band-Membership* in the data model of Figure 4, one finds that the probability that a given person is member of a given band must be less than the probability that that person exists and also less than the probability that that band exists. So models in **FuzzySet** allow the introduction of uncertainty in conceptual data models in a natural way.

4. CONSTRAINTS

Constraints represent restrictions on populations. They exclude populations that do not correspond with a possible situation in the problem domain. Consider for example the NIAM data model of Figure 4. In this data model it may be desirable to express that each person is either a composer or a musician. This implies the specification of a constraint that enforces the populations of these object types to be a cover of the population of the object type person. In general, constraints may be quite complex and special languages for their specification exist (mostly founded in logic).

Two important *types* of constraints that are frequently used in conceptual data modelling techniques are the total role constraint and the uniqueness constraint. These constraint types correpond to a large extent to the cardinality constraints in ER. They are more general, as more than one relationship type may be involved. The semantics of these constraint types is described in the following sections.

4.1. Total role constraint

A total role constraint over a number of roles stipulates that all instances in the object types playing these roles have to participate in at least one of these roles. Total role constraints are important for applications as they determine mandatory/optional properties of objects. For example, in the Relation Model they determine whether a certain column is allowed to contain null-values.

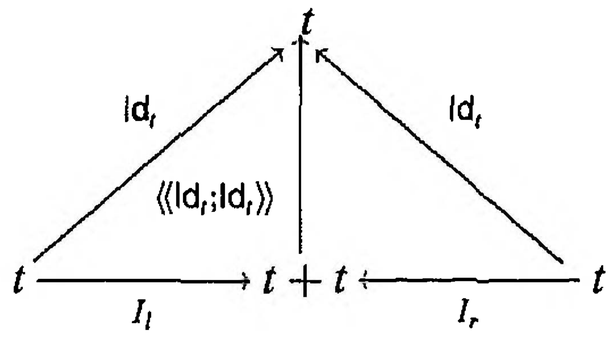
Formally, a total role constraint in a given type graph $\mathcal{G}$ is determined by a set of edges $\tau \subseteq \mathcal{G}_1$. In the simplest example of a total role constraint, τ consists of a single edge e. This total role constraint means that all elements of $\mathsf{target}(e)$ must participate in e. In a model in the category **Set** this implies that $M(e)$ must be a surjective function. More generally we require that $M(e)$ must be an epimorphism.

EXAMPLE 4.1. A total role constraint on the role with name *is-member-of* in the schema of Figure 4, implies that every person has to be a member of a band.

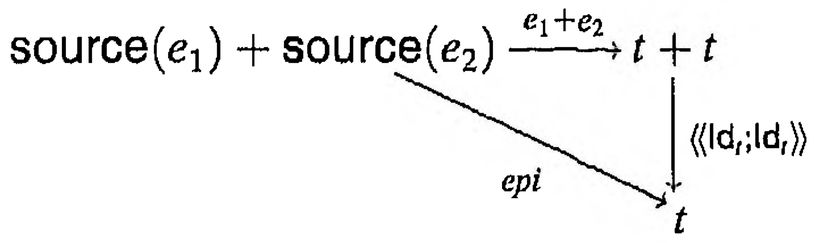
A slightly more complicated example is $\tau = \{e_1, e_2\}$. Two cases can be distinguished, depending on whether both edges have the same target. In the first case both arrows have the same target $t = \mathsf{target}(e_1) = \mathsf{target}(e_2)$. The intuitive meaning of this constraint is that each element of t must participate in at least one of these two edges.

EXAMPLE 4.2. In the context of the schema of Figure 4, a total role constraint on the roles with names *is-member-of* and *has-written* implies that every person either is a member of a band or has written a song or both.

For the semantics of this type of constraint, first construct the sum arrow $e_1 + e_2: \mathsf{source}(e_1) + \mathsf{source}(e_2) \to t + t$. Intuitively speaking every element of t must be present in $\mathsf{target}(e_1 + e_2)$, however, as $t + t$ is a disjoint sum every element is represented twice. Therefore an arrow is needed that maps each element of $t + t$ onto the corresponding element of t. This can be achieved as follows. From the definition of the coproduct it follows that there are two injection arrows $I_l: t \to t + t$ and $I_r: t \to t + t$. Further, there is a unique arrow $\langle\langle \mathsf{Id}_t; \mathsf{Id}_t \rangle\rangle: t + t \to t$, such that the following diagram commutes.



The meaning of the total role constraint is that $\langle\langle \mathsf{Id}_t; \mathsf{Id}_t \rangle\rangle \circ (e_1 + e_2)$ must be an epimorphism.



If $\mathsf{target}(e_1) \neq \mathsf{target}(e_2)$, it is possible that one of these is a subtype of the other or for example that both types have a common supertype. In this case we first inject the elements of the subtype into the supertype and then follow the same procedure as in the previous case. Note that the supertype is always equal to $U_M^{\{\mathsf{target}(e_1),\mathsf{target}(e_2)\}}$.

EXAMPLE 4.3. As an example of this type of total role constraint consider the schema of Figure 16. A total role constraint on the roles with names *receives* and *earns-salary* would imply that every person, which is either a student or an employee or both, either owns a scholarship or earns a salary. This is clearly different from the situation in which every student owns a scholarship and every employee earns

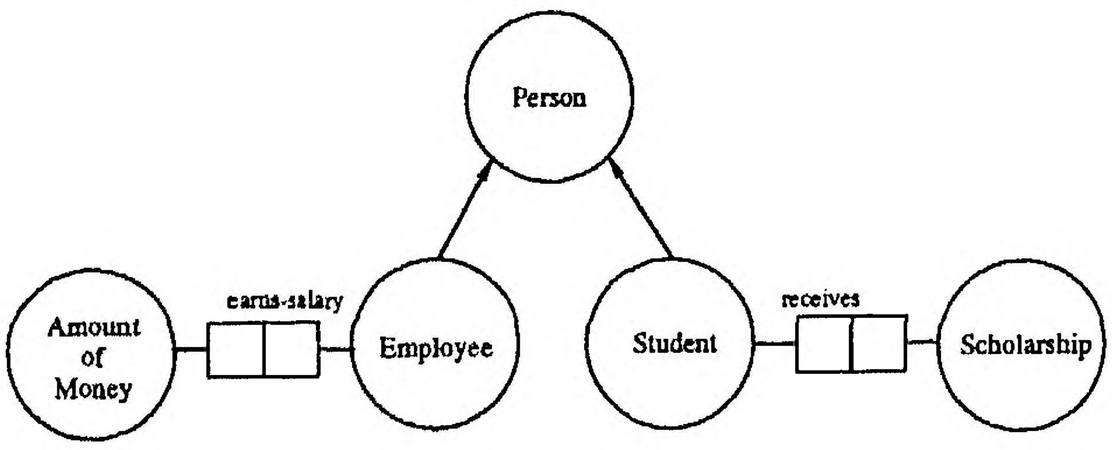
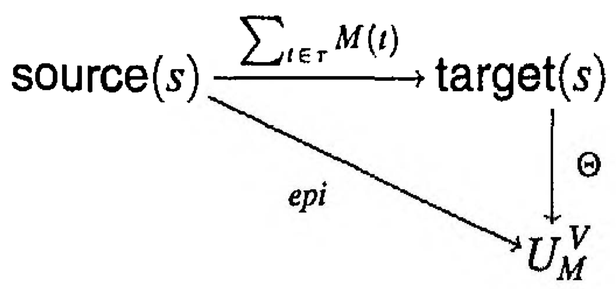


FIGURE 16. Sample schema.

a salary. As students may have different representations as employees, it is necessary to use the colimit construction to identify identical persons. □

The full definition of the semantics of the total role constraint is given below.

DEFINITION 4.1. Given a valid type model M and a total role constraint in the involved type graph $\mathcal{G}$ over $\tau \subseteq \mathcal{G}_1$. Let $s = \sum_{t \in \tau} M(t)$ and $V = \{\mathsf{target}(M(t)) \mid t \in \tau\}$. The definition of the instance universe U_M^V implies that for each $t \in \tau$ an arrow i_t: $\mathsf{target}(M(t)) \rightarrow U_M^V$ exists. Since $\mathsf{target}(s)$ is a coproduct, these i_t determine a unique arrow Θ: $\mathsf{target}(s) \rightarrow U_M^V$. M satisfies the total role constraint τ iff $\Theta \circ s$ is an epimorphism.



□

EXAMPLE 4.4. In example 3.1. take the total role constraint over $\tau = \{p, u\}$. Then $V = \{A, C\}$ and $U_M^V = A$. The sum $p + u$: $F + G \rightarrow A + C$ is the function $\{f_1 \mapsto a_1, f_2 \mapsto a_3, f_3 \mapsto a_3, g_1 \mapsto c_1, g_2 \mapsto c_1, g_3 \mapsto c_2\}$. Then Θ: $A + C \rightarrow A = \{a_1 \mapsto a_1, a_2 \mapsto a_2, a_3 \mapsto a_3, c_1 \mapsto a_1, c_2 \mapsto a_2\}$. The composition $\Theta \circ (p + u) = \{f_1 \mapsto a_1, f_2 \mapsto a_3, f_3 \mapsto a_3, g_1 \mapsto a_1, g_2 \mapsto a_1, g_3 \mapsto a_2\}$ is an epimorphism in **Set** because it is a surjective function. Therefore, the total role constraint over $\tau = \{p, u\}$ is satisfied in this model.

The total role constraint over $\{p\}$ is not satisfied in this model (as a_2 is not in the range of function p), but the total role constraint over $\{q\}$ is. □

The total role constraint can be seen as a generalization of several types of constraints found in conceptual data modelling techniques, such as the collection cover constraint and the subtype cover constraint. The collection cover constraint for a collection type specifies that all instances of its element type should participate in at least one of its instances. The subtype cover constraint specifies that all instances of a given object type should be instances of at least one of a given set of subtypes of that object type.

4.2. Uniqueness constraint

The uniqueness constraint is closely related to the concept of a key over a relation. A uniqueness constraint in a given type graph $\mathcal{G}$ is determined by a set of edges $\tau \subseteq \mathcal{G}_1$.

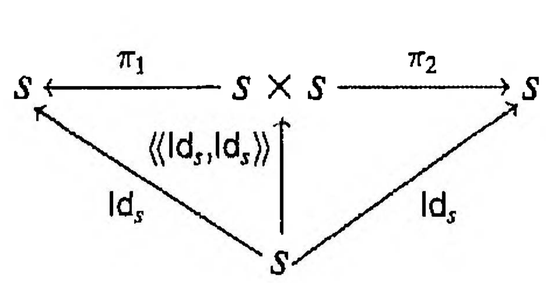
In the most trivial case τ consists of a single edge e. The intuitive semantics is that each element of $\mathsf{target}(e)$ determines at most one element in $\mathsf{source}(e)$. For a model M in the category **Set** this implies that $M(e)$ must be an injective function. More generally, $M(e)$ must be a monomorphism.

EXAMPLE 4.5. A uniqueness constraint on the role with name *is-written-by* in the schema of Figure 4 implies that every song is written by at most one person.

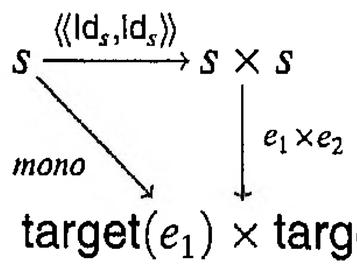
In the next and more interesting case $\tau = \{e_1, e_2\}$ with $\mathsf{source}(e_1) = \mathsf{source}(e_2) = s$. In this case the intuitive semantics is that the combination of an element from $\mathsf{target}(e_1)$ with an element from $\mathsf{target}(e_2)$ determines at most one element in $\mathsf{source}(e_1)$.

EXAMPLE 4.6. Consider a ternary relationship between *Person*, *Duration*, and *Project*, capturing how many hours a certain person has worked for a certain project. A uniqueness constraint on the roles attached to the object types *Person* and *Project* expresses that a person–project combination has at most one associated duration. □

Formally, start by constructing the product arrow $e_1 \times e_2$: $s \times s \rightarrow \mathsf{target}(e_1) \times \mathsf{target}(e_2)$. From the definition of the product it follows that there are two projection arrows π_1: $s \times s \rightarrow s$ and π_2: $s \times s \rightarrow s$. Further, there is a unique arrow $\langle\langle \mathsf{Id}_s, \mathsf{Id}_s \rangle\rangle$: $s \rightarrow s \times s$, such that the following diagram commutes.



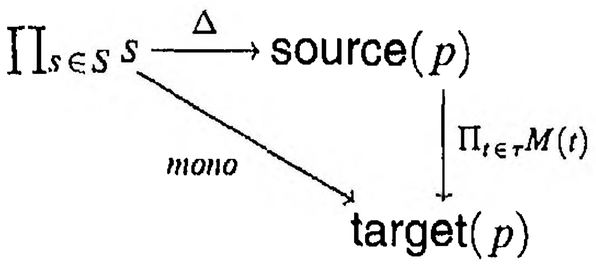
The meaning of the uniqueness constraint is that $(e_1 \times e_2) \circ \langle\langle \mathsf{Id}_s, \mathsf{Id}_s \rangle\rangle$ must be a monomorphism.



The case that $\tau = \{e_1, e_2\}$ with $\mathsf{source}(e_1) \neq \mathsf{source}(e_2)$ is simple, because it is equivalent to the combination of two uniqueness constraints over $\{e_1\}$ and $\{e_2\}$.

The full definition of the semantics of the uniqueness constraint is given below.

DEFINITION 4.2. Given a valid type model M and a uniqueness constraint in the involved type graph $\mathcal{G}$ over $\tau \subseteq \mathcal{G}_1$. Let $p = \prod_{t \in \tau} M(t)$, $S = \{\mathsf{source}(M(t)) \mid t \in \tau\}$. For each $t \in \tau$ there is an arrow π_t: $\prod_{s \in S} s \rightarrow \mathsf{source}(M(t))$. From the definition of the product it follows that these π_t determine a unique arrow Δ: $\prod_{s \in S} s \rightarrow \mathsf{source}(p)$. Then, M satisfies the uniqueness constraint τ iff $p \circ \Delta$ is a monomorphism.



□

As remarked in section 3.3, relationship types behave by default as multisets: the same tuple can be represented multiple times. If this is undesirable, it can be avoided by adding a uniqueness constraint over the roles of the relationship type.

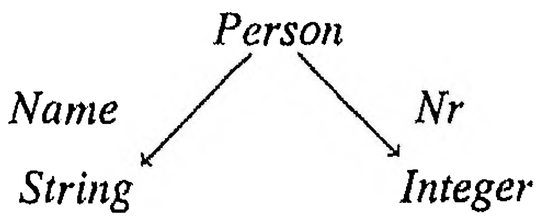
EXAMPLE 4.7. Take for instance, in example 3.1., the uniqueness constraint over $\tau = \{p, q\}$. Intuitively speaking, this constraint should be satisfied since every combination from A and B determines at most one element of F. The arrow $\Delta: F \to F \times F = \{f_1 \mapsto \langle f_1, f_1\rangle, f_2 \mapsto \langle f_2, f_2\rangle, f_3 \mapsto \langle f_3, f_3\rangle\}$. The product $p \times q: F \times F \to A \times B =$

$$\begin{aligned}\{\ & \langle f_1, f_1\rangle \mapsto \langle a_1, b_1\rangle, \quad \langle f_1, f_2\rangle \mapsto \langle a_1, b_1\rangle,\\ & \langle f_2, f_1\rangle \mapsto \langle a_3, b_1\rangle, \quad \langle f_2, f_2\rangle \mapsto \langle a_3, b_1\rangle,\\ & \langle f_3, f_1\rangle \mapsto \langle a_3, b_1\rangle, \quad \langle f_3, f_2\rangle \mapsto \langle a_3, b_1\rangle,\\ & \langle f_1, f_3\rangle \mapsto \langle a_1, b_2\rangle,\\ & \langle f_2, f_3\rangle \mapsto \langle a_3, b_2\rangle,\\ & \langle f_3, f_3\rangle \mapsto \langle a_3, b_2\rangle \quad \}\end{aligned}$$

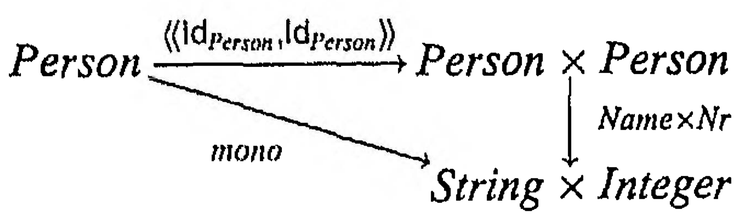
The composition $(p \times q) \circ \Delta = \{f_1 \mapsto \langle a_1, b_1\rangle, f_2 \mapsto \langle a_3, b_1\rangle, f_3 \mapsto \langle a_3, b_2\rangle\}$ is a monomorphism, because it is an injective function. Therefore, the uniqueness constraint over $\tau = \{p, q\}$ is satisfied.

The separate uniqueness constraints over $\{p\}$ and $\{q\}$ are not satisfied because p and q are not monomorphisms. □

EXAMPLE 4.8. Models in **PartSet** give a way to handle missing values. Suppose that persons are identified by their names. Two different persons with identical names receive an additional number to distinguish them.



The arrow Nr is a partial function, because persons with a unique name do not have a number. Suppose that we want to express that every person must be uniquely identified by a combination of name and number. This can be achieved by putting a uniqueness constraint over $\{Name, Nr\}$.



The arrow $\langle\langle \mathrm{Id}_{Person}, \rangle\rangle \mathrm{Id}_{Person} = \{p \mapsto \langle p, p\rangle \mid p \in Person\}$. The arrow $Name \times Nr$ is interesting, since it maps the tuple $\langle p, p\rangle$ for a person p whose Nr is undefined to the tuple $\langle Name(p), \perp\rangle$. The uniqueness constraint holds if $(Name \times Nr) \circ \langle\langle \mathrm{Id}_{Person}, \mathrm{Id}_{Person}\rangle\rangle$ is a monomorphism, i.e. a total injective function. This implies that two persons with the same name must have different numbers, which was indeed the requirement we tried to express. □

Some conceptual data modelling techniques, among others NIAM, allow uniqueness constraints over more than one relationship type. Such a uniqueness constraint expresses a key over a derived relationship type which is a join of the relationship types involved. Therefore, the semantics of this type of uniqueness constraint is completely determined by the way the join condition has to be computed. As joins can be specified categorically by the use of pullbacks, we do not consider such uniqueness constraints explicitly. It should be remarked, however, that some categories do not have pullbacks (e.g. **Rel**). In other words, the introduction of this type of uniqueness constraint leads to a further restriction on **Fund**.

5. CONCLUSIONS AND FURTHER RESEARCH

This paper presents a unifying framework for conceptual data modelling techniques. The framework is based on category theory due to its formality and its high level of abstraction. As has been pointed out, mathematical formalizations should not impose representational choices but instead focus on the *essence* of concepts.

Since the framework contains most important concepts of existing data modelling techniques it can be seen as a generalization of these techniques. Therefore, the framework can be used to compare different conceptual modelling techniques. As very few limitations are imposed upon type graphs several restrictions that exist in other techniques can be lifted. For example, a ternary relationship type may be a subtype of a binary relationship type as the categorical semantics only requires subtype instances to have corresponding supertype instances.

An important property of the framework is its 'configurable semantics'. Features, such as null values, uncertainty and temporal behavior can be added to the models by selecting an appropriate instance category. The addition of such features to traditional (e.g. set or logic-based) semantics usually requires a complete redesign of the formalization. This property is also useful for experimenting with these features in traditional data modelling techniques, since the mapping of these techniques into the framework automatically defines a semantics for these features.

Compared with other approaches that use category theory [16, 17] the current framework is simpler as it only uses basic categorical notions. This makes the framework easier to understand. Furthermore, the range of possible instance categories is wider than in those approaches that are usually limited to topoi or cartesian closed categories.

The model that is described here is very similar to object-oriented data models. The subtypes in our approach are analogous to subclasses. Attributes can be modelled using roles. Attribute inheritance could be incorporated explicitly

in the type model by adding an attribute that is defined in a given type to all its subtypes. The value of this subtype attribute arrow in the type model is the composition of the original attribute arrow with the subtype arrow from the subtype to the supertype. The resulting model is similar to that of [16].

Several extensions to the current framework are the topic of our current research. It seems to be possible to define most relational database operators within the current framework. Further it appears straightforward to incorporate other types of constraints such as the exclusion, equality and subset constraint.

ACKNOWLEDGEMENTS

The authors would like to thank Denis Verhoef for his comments on an earlier version of this paper. The comments and suggestions of the referees, which have led to a number of improvements, are also gratefully acknowledged.

REFERENCES

[1] Hull, R. and King, R. (1987) Semantic Database Modelling: Survey, Applications and Research Issues. *ACM Computing Surveys*, **19**, 201–260.

[2] Peckham, J. and Maryanski, F. Semantic Data Models. (1988) *ACM Computing Surveys*, **20**, 153–189.

[3] Chen, P. P. (1976) The Entity-Relationship Model: Toward a Unified View of Data. *ACM Trans. Database Sys.*, **1**, 9–36.

[4] Shipman, D. W. (1981) The Functional Data Model and the Data Language DAPLEX. *ACM Trans. Database Syst.*, **6**, 140–173.

[5] Nijssen, G. M. and Halpin, T. A. (1989) *Conceptual Schema and Relational Database Design: a fact oriented approach*. Prentice-Hall, Sydney, Australia.

[6] Teorey, T. J., Yang, D. and Fry, J. P. (1986) A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, **18**, 197–222.

[7] Engels, G., Gogolla, M., Hohenstein, U., Hülsmann, K., Löhr-Richter, P., Saake, G. and Ehrich, H.-D. (1992) Conceptual modelling of database applications using an extended ER model. *Data Knowledge Engineering*, **9**, 157–204.

[8] Abiteboul, S. and Hull, R. (1987) IFO: A Formal Semantic Database Model. *ACM Trans. Database Systems*, **12**, 525–565.

[9] Halpin, T. A. and Orlowska, M. E. (1992) Fact-oriented modelling for data analysis. *Journal of Information Systems*, **2**, 97–119.

[10] ter Hofstede, A. H. M. and van der Weide, Th. P. (1993) Expressivenes in conceptual data modelling. *Data & Knowledge Engineering*, **10**, 65–100.

[11] ter Hofstede, A. H. M., Proper, H. A. and van der Weide, Th. P. (1993) Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, **18**, 489–523.

[12] Avison, D. E. and Fitzgerald, G. (1988) *Information Systems Development: Methodologies, Techniques and Tools*. Blackwell Scientific Publications, Oxford.

[13] Bubenko, J. A. (1986) Information System Methodologies—A Research View. In Olle, T. W., Sol, H. G. and Verrijn-Stuart, A. A. (eds), *Information Systems Design Methodologies: Improving the Practice*, pp. 289–318. North-Holland, Amsterdam.

[14] van Griethuysen, J. J. (ed), (1982) *Concepts and Terminology for the Conceptual Schema and the Information Base*. Publ. nr. ISO/TC97/SC5–N695, ISO, ISO Central Secretariat, Geneva.

[15] Barr, M. and Wells, C. (1990) *Category Theory for Computing Science*. Prentice-Hall, Englewood Cliffs, NJ.

[16] Tuijn, C. (1994) *Data Modeling from a Categorical Perspective*. Ph.D. thesis, University of Antwerp, Antwerp, Belgium.

[17] Baclawski, K., Simovici, D. and White, W. (1994) A categorical approach to database semantics. *Mathematical Structures in Computer Science*, **4**, 147–183.

[18] McLarty, C. (1992) *Elementary Categories, Elementary Toposes*. Volume 21 of *Oxford Logic Guides*. Clarendon Press, Oxford.

[19] Hoare, C. A. R. (1989) Notes on an Approach to Category Theory for Computer Scientists. In. Broy, M. (ed), *Constructive Methods in Computing Science*. Volume 55 of *NATO Advanced Science Institute Series*, pp. 245–305. Springer-Verlag, Berlin.

[20] Goguen, J. A. (1991) A categorical manifesto. *Mathematical Structures in Computer Science*, **1**, 49–67.

[21] Adámek, J., Herrlich, H. and Strecker, G. E. (1990) *Abstract and Concrete Categories*. Pure and applied mathematics. John Wiley and Sons, New York.

[22] Ehrich, H.-D. and Sernadas, A. (1991) Object concepts and constructions. In Saake, G. and Sernadas, A. (eds), *Proceedings of the IS-CORE Workshop'91 (Informatik-Berichte 91-03)*, pp. 1–24. Technische Universität Braunschweig, Braunschweig.

[23] Fiadeiro, J., Sernadas, C., Maibaum, T. and Saake, G. (1991) Proof-theoretic semantics of object-oriented specification constructs. In Meersman, R., Kent, W. and Khosla, S. (eds), *Object-oriented databases: analysis, design and construction*, pp. 243–284. North-Holland, Amsterdam.

[24] Costa, J. F., Sernadas, A. and Sernadas, C. (1994) Object inheritance beyond subtyping. *Acta Informatica*, **31**, 5–26.

[25] Siebes, A. (1990) *On Complex Objects*. Ph.D. thesis, University of Twente, Enschede.

[26] Dampney, C. N. G., Johnson, M. S. J. and Monro, G. P. (1992) An Illustrated Mathematical Foundation for ERA. In Rattray, C. M. I. and Clark, R. G. (eds), *The Unified Computation Laboratory*, pp. 77–83. Oxford University Press, Oxford.

[27] Islam, A. and Phoa, W. (1994) Category Models of Relational Databases I: Fibrational Formulation, Schema Integration. In Hagiya, M. and Mitchell, J. C. (eds), *Theoretical Aspects of Computer Software, International Symposium TACS'94*, volume 789 of *Lecture Notes in Computer Science*, pp. 618–641. Springer-Verlag, Sendai.

[28] Sernadas, C., Fiadeiro, J., Meersman, R. and Sernadas, A. (1989) Proof-theoretic conceptual Modelling: the NIAM Case Study. In Falkenberg, E. D. and Lindgreen, P. (eds), *Information System Concepts: An In-depth Analysis*, pp. 1–30. North-Holland, Amsterdam.

[29] Rydeheard, D. E. and Burstall, R. M. (1988) *Computational Category Theory*. Prentice-Hall, Englewood Cliffs.

[30] Yourdon, E. (1989) *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs, NJ.

[31] Maier, D. (1988) *The Theory of Relational Databases*. Computer Science Press, Rockville, MD.

[32] Lippe, E. and ter Hofstede, A. H. M. (1994) *A Category Theory Approach to Conceptual Data Modeling*. Technical Report CSI-R9415, Computing Science Institute, University of Nijmegen, Nijmegen.

[33] Hammer, M. and McLeod, D. (1981) Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, **6**, 351–386.

[34] ter Hofstede, A. H. M. and van der Weide, Th. P. (1994) Fact Orientation in Complex Object Role Modelling Techniques. In Halpin, T A. and Meersman, R. (eds), *Proceedings of the First International Conference on Object-Role Modelling (ORM-1)* pp. 45–59, Townsville, Queensland, Australia.

[35] Kim, W. and Lochovsky, F. H. (eds) (1989) *Object-Oriented Concepts, Databases, and Applications*. ACM Press, Frontier Series. Addison-Wesley, Reading, Queensland, Australia.

[36] Zdonik, S. B. and Maier, D. (eds) (1990) *Readings in Object-Oreinted Database Systems*. Morgan Kaufmann, San Mateo.