A Unifying Framework for Conceptual Data Modelling Concepts

P.J.M. Frederiks, A.H.M. ter Hofstede, E. Lippe

Department of Information Systems
University of Nijmegen
Toernooiveld 1
NL-6525 ED Nijmegen
The Netherlands
{paulf,arthur,ernstl}@cs.kun.nl

Published as: P.J.M. Frederiks, A.H.M. ter Hofstede, and E. Lippe. A Unifying Framework for Conceptual Data Modelling Concepts. Technical Report CSI-R9410, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, September 1994.

Abstract

For succesful information systems development, conceptual data modelling is essential. Nowadays many techniques for conceptual data modelling exist, examples are NIAM, FORM, PSM, many (E)ER variants, IFO, and FDM. In-depth comparisons of concepts of these techniques is very difficult as the mathematical formalisations of these techniques, if existing at all, are very different. As such 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 are discussed from a category theoretic point of view. Advantages and disadvantages of the approach chosen will be outlined.

Keywords: Conceptual Data Modelling, Category Theory, Meta Modelling
Classification: 68P99 *(AMS-1991)*, H.1.0. *(CR-1991)*

1 Introduction

It seems an undisputed fact that, opposed to most mature scientific disciplines, the discipline of information systems does not have a sound and widely accepted foundation of basic concepts. Numerous and widely diverting views abound in this field of study. To a large extent, this is caused by the lack of formal foundations. The resulting situation has often been referred to as the *Methodology Jungle* [AF88]. In [Bub86] it is estimated that during the past years, hundreds if not thousands of information system development methods have been introduced. Most organisations 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 behaviour. This has led to *fuzzy* and *artificial* concepts in information systems development methods and, to some extent, also in conceptual data modelling techniques.

Conceptual data modelling is imperative for succesful information systems development. Currently, many different conceptual data modelling techniques exist such as ER [Che76] and its many variants, functional modelling techniques, such as FDM [Shi81], and so-called object-role modelling techniques, such as NIAM [NH89]. Complex application domains, such as meta modelling, hypermedia and CAD/CAM, have led to the introduction of advanced modelling concepts, such as present in the various forms of Extended ER (see e.g. [TYF86, EGH+92]), IFO [AH87], and object-role modelling extensions such as FORM [HO92] and PSM [HW93, HPW93].

Essential for creating order in this chaos is a unifying framework. 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 Conceptualisation Principle [Gri82] can also be applied to mathematical formalisations). These requirements suggest category theory (see e.g. [BW90]) 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 [Tui94, BSW94] where only very specific types of categories, topoi, are possible target categories.

The paper is organised as follows. Section 2 contains a brief introduction to category theory and its historical background. Section 3 describes data modelling concepts from a category theoretic point of view. Section 4 discusses advantages and disadvantages of the approach outlined and identifies topics for further research. Finally, for an in-depth treatment of the formal definition of the framework described in this paper we refer to [LHF94].

2 Category Theory

In this section we briefly discuss the historical background of category theory and its "traditional" applications in computer science (see also [Lan71, Hoa89]). Then we will present some essential category theoretic definitions for the purpose of this paper. For an in-depth treatment of category theory the reader is referred to [BW90].

2.1 Background

Category theory is a relatively young branch of mathematics which originated from algebraic topology, and designed to describe various *structural* concepts from different mathematical fields in a *uniform* way [Fok92]. 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 [Hoa89]: "Category theory is quite the most general and abstract branch of pure mathematics".

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 was, and what the 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. As such, they are precisely the kind of definition that category theorists abhor. One might say that category theory applies the Conceptualisation Principle to mathematical formalisations.

In the seventies and eighties category theory has 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. Not many researchers have applied category theory to information systems. Recently, however, it seems that this is going to change (see e.g. [SFMS89, DJDR94, Sie90, Tui94]). Some of the advantages of the application of category theory to data modelling are mentioned in [Tui94]:

> An important motivation for a categorical basis for data modeling is that category theory allows to define constructs in a very uniform way which provides a basis for interesting generalizations. This way of working often offers new insight in well-known operators as well as it accounts for the introduction of new operators which were far from trivial in other formalisms. An additional feature of using category theory is that it is fundamental and graph oriented which is suitable to describe data modeling concepts.

To these advantages might be added that category theory facilitates recognition of similar constructions and reduces proof obligations (as the dual of each theorem is also true).

2.2 A Brief Introduction

In order to keep this paper self-contained this section presents the definitions of the category theoretic concepts used. Most of these definitions are adapted from [BW90].

The following definition defines what a category is.

Definition 2.1

A category $\mathcal{C}$ *is a directed multigraph* $\langle \mathcal{O}, \mathcal{A}, \mathsf{source}, \mathsf{target} \rangle$ *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 *in* $\mathcal{O}$ *there is an arrow* $\mathsf{Id}_A : A \rightarrow A$*, the* identity *arrow. If* $f : A \rightarrow B$*, then* $f \circ \mathsf{Id}_A = f = \mathsf{Id}_B \circ f$*.* □

In figure 1 a simple example of a category is shown. 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

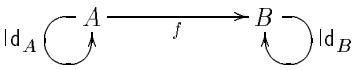


Figure 1: A simple example of a category

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** concerns sets with *relations*. Another interesting category is the category **FuzzySet**, where the arrows are total functions and the objects *fuzzy sets*.

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

Definition 2.2

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 this definition.* □

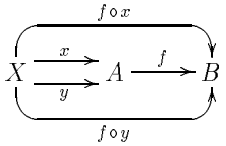


Figure 2: Illustration of the definition of a monomorphism

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 2.3

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 this definition.* □

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

An epimorphism is a monomorphism in the *dual category*. A dual category, denoted as $\mathcal{C}^{\mathrm{op}}$, of a category $\mathcal{C}$ has the same objects as $\mathcal{C}$ and as arrows all arrows of $\mathcal{C}$ inverted, i.e. if $f : A \to B$ an arrow in $\mathcal{C}$ then $f^{\mathrm{op}} : B \to A$ is an arrow of $\mathcal{C}^{\mathrm{op}}$. As a result the composition of arrows in the dual category is defined on the inverted arrows. For an illustration of the concept of duality consider figures 2 and 3. The concept of duality in category theory is very important as it reduces proof obligations: the dual of a theorem is also a theorem.

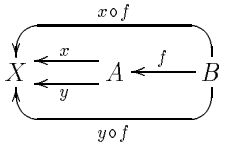


Figure 3: Illustration of the definition of an epimorphism

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 [RB88]:

> 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 2.4

An arrow $f : A \to B$ is said to be an isomorphism *if there exists an arrow $g : B \to A$ such that $f \circ g = \mathsf{Id}_B$ and $g \circ f = \mathsf{Id}_A$. 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. □

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 $\imath_1$ and $\imath_2$, i.e. $\imath_1(a) = \langle a, 0\rangle$ and $\imath_2(b) = \langle b, 1\rangle$. The categorical definition of a *coproduct* or *sum* is the generalization of the notion of disjoint union. In particular, it does not prescribe a representation. First however, it is necessary to define what a *commuting* diagram is. Proofs and definitions in category theory often use diagrams and prove or require them to commute.

Definition 2.5

A diagram is said to commute if every path between two nodes determines through composition the same arrow. □

Definition 2.6

A coproduct *of two objects A en B in a category consists of an object $A + B$ together with arrows $\imath_1 : A \to A + B$ and $\imath_2 : B \to A + B$ such that for any arrows $q_1 : A \to V$ and $q_2 : B \to V$, there is a unique arrow $\langle q_1; q_2\rangle : A + B \to V$ for which the diagram of figure 4 commutes.* □

The definition of a coproduct can straightforwardly be generalized to be applicable to any number of objects in a category.

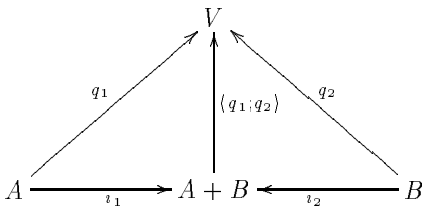


Figure 4: The coproduct of two objects

3 A Unifying Description of Conceptual Data Models

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 formalised via the notion of *type models*, defined in section 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 [Sie90] and [Tui94]). 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* $\{\mathsf{role}, \mathsf{spec}, \mathsf{gen}, \mathsf{elt\text{-}role}, \mathsf{col\text{-}role}\}$ *such that there are no cycles consisting solely of* subtype *edges, i.e. edges with label* spec *or* gen. *Furthermore, there is a bijective function* $\mathsf{Col}_{\mathcal{G}}$ *from edges with label* col-role *to edges with label* elt-role *such that related edges have identical sources. The function* type *yields the label of an edge.* □

The definition of a type graph is very liberal, only cyclic inheritance structures are (obviously) excluded. The definition allows a node to be a collection type (a notion which will be explained in depth in section 3.5) 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 inheritance structures, will not be imposed. Whether all these permitted constructions are really useful from a practical point of view remains an issue for future research.

As an example of how data models can be represented as type graphs, consider figure 6, which shows the type graph of the NIAM data model in figure 5. 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 [NH89]).

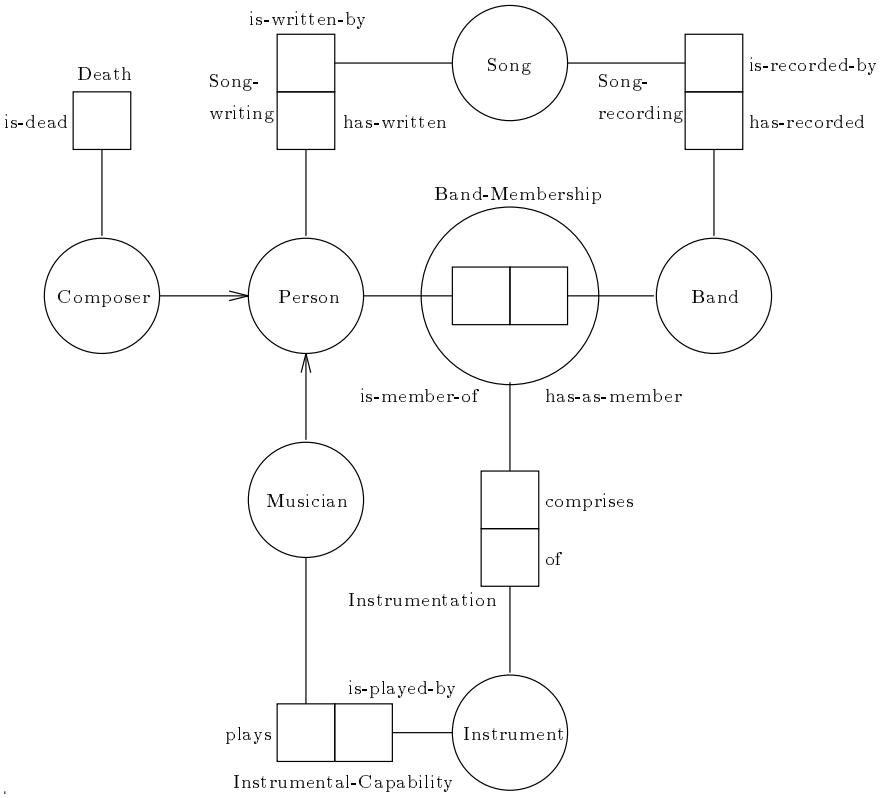


Figure 5: A NIAM data model

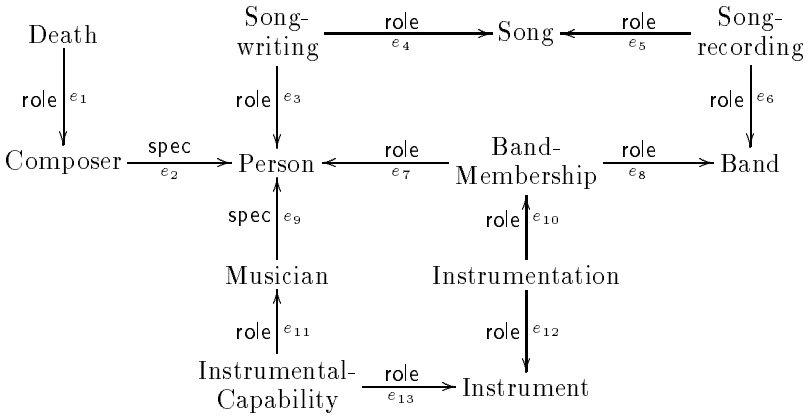


Figure 6: Type graph of the schema of figure 5

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

The above definition implies that the semantics of a data model depends on the target category, referred to as 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 fulfil a number of requirements. Firstly, they should allow for certain categorical constructions (e.g. coproducts should always be defined). Secondly, they should have certain special categorical properties (e.g. coproducts should be *disjoint*), and thirdly, the category **FinSet**, consisting of finite sets and total functions, should act as a "bottom" element of **Fund**, i.e. a member of **Fund** should be able to represent each population representable in **FinSet**. For a complete definition of the first two requirements refer to [LHF94]. The third requirement deserves some further explanation.

The category **FinSet** provides the most intuitive and most standard semantics of data models. Formalisations of data models are usually given in terms of elementary set theory, with finite sets. Such formalisations correspond to a semantics resulting from the choice $\mathcal{F} =$ **FinSet** in definition 3.2. A general framework for conceptual data models should at least be able to encompass such semantics. Therefore, the requirement that **FinSet** should act as a "bottom" element of **Fund** (for the formal translation of this requirement, again refer to [LHF94]). Consequently, the elements of **Fund** each are of a set-like nature.

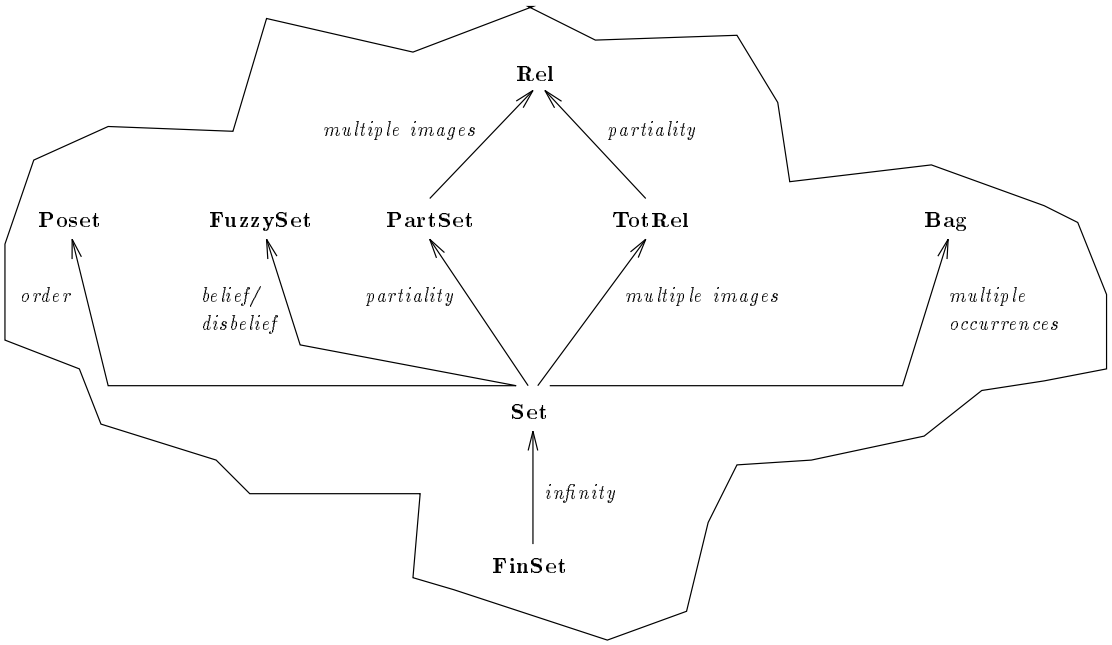


Figure 7: The class of categories **Fund**

In figure 7, 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 section 3.3, this category should be considered if one is interested in the study of "null"-values in relationship types. Other categories in figure 7 are:

- The category **TotRel** where the objects are sets and the arrows total relations.
- The category **Bag** where the objects are bags (multisets) and the arrows total functions.
- 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 total functions on these sets. Popularly speaking, fuzzy sets assign probabilities to the occurrence of elements.

Specific constructions may impose extra requirements on type models. These requirements are discussed in the rest of this section.

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 (with $n \geq 1$), as well as play a role in other relationship types. Yourdon [You89] 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.

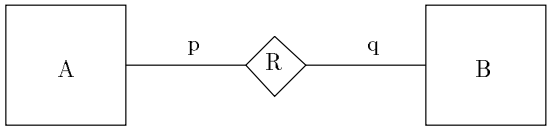


Figure 8: A simple ER schema

In the past, relationship types have often been formalised by viewing them as subsets of a cartesian product. This has commonly been been referred to as the *tuple oriented approach*. As an example consider figure 8. A population 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, algebraic operators do not have important properties such as commutativity and associativity. This observation has led to the *mapping oriented approach* [Mai88] (see also [Fal76]), where relationship instances are treated as functions from the involved roles to values. In this approach, the above sample population would be represented as:

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

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.

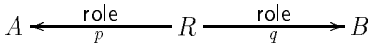


Figure 9: Type graph of figure 8

Still, however, one may argue that the mapping oriented approach imposes unnecessary restrictions. Why do instances have to be represented as *functions*? Isn't 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 8 is shown in figure 9. 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!):

$$\begin{aligned} p &= \{f_1 \mapsto a_1, f_2 \mapsto a_2\} \\ q &= \{f_1 \mapsto b_1, f_2 \mapsto b_1\} \end{aligned}$$

In this approach, the two relationship instances 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. Therefore, there is a difference between the situation where a key is specified on the whole relationship type, and the situation where no key is specified (in NIAM these cases are equivalent as instances of fact types are uniquely determined by their 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:

$$\begin{aligned} p &= \{f_2 \mapsto a_2\} \\ q &= \{f_1 \mapsto b_1, f_2 \mapsto b_1\} \end{aligned}$$

In this population, relationship instance f_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 could be:

$$\begin{aligned} p &= \{f_2 \mapsto a_1, f_2 \mapsto a_2\} \\ q &= \{f_1 \mapsto b_1, f_2 \mapsto b_1, f_2 \mapsto b_2\} \end{aligned}$$

3.4 Inheritance Relations

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 [AH87].

3.4.1 Specialization

Specialization is used when specific facts are to be recorded for specific instances of an object type only. 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.

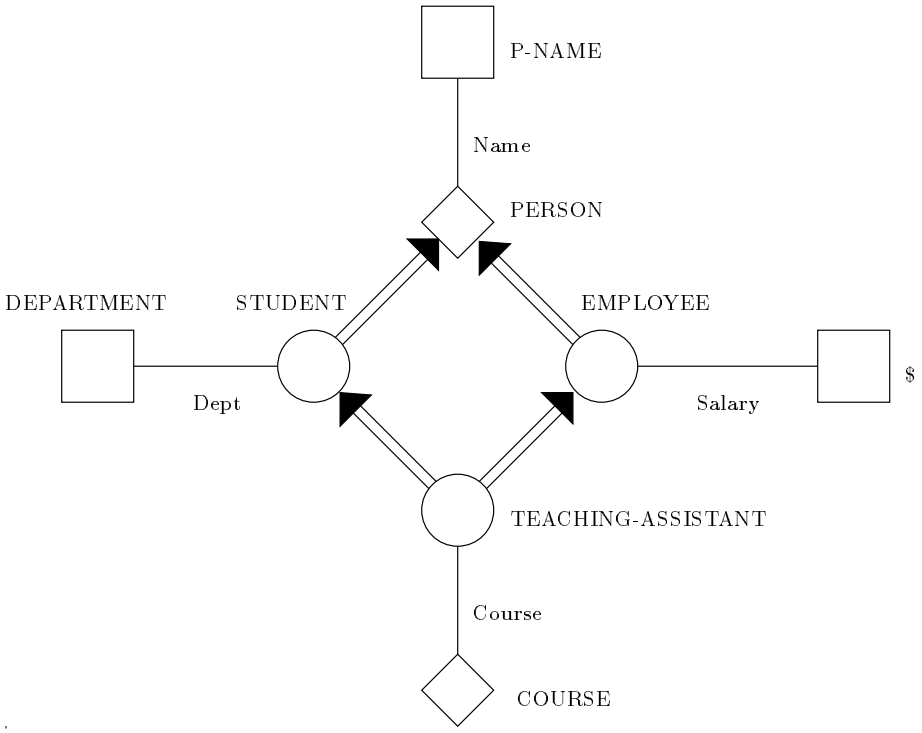


Figure 10: An example of a subtype hierarchy in IFO

As an example of specialization consider the IFO schema of figure 10 (adapted from [AH87]). 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 is 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 formalisation 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 formalisation of specialization relations. Consider for example the following partial population of the schema of

figure 10:

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

and the following subtype relations (see also figure 11):

$$\begin{aligned} I_1(\text{TA999}) &= \text{EM237} \\ I_2(\text{TA999}) &= \text{ST1943} \\ I_3(\text{EM237}) &= \text{Jones} \\ I_4(\text{ST1943}) &= \text{Richards} \end{aligned}$$

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.

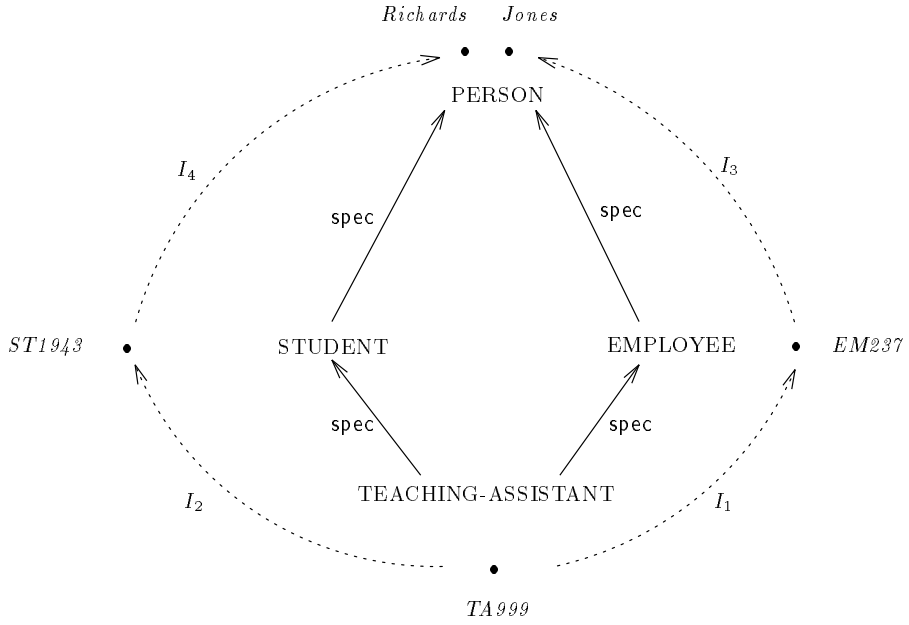


Figure 11: A non-commuting diagram

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: $I_4(I_2(\text{TA999})) = I_3(I_1(\text{TA999}))$.

Finally, inheritance is category theoretically solved by the fact that arrow composition is always defined.

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 [HW93].

The population of a generalized object type is the union of the populations of the participating object types, referred to as the *specifiers*. Typically, properties are propagated "upward" in a generalization hierarchy instead of "downward" (see also [AH87]). This also implies that the identification of a generalized object type depends on the identification of its specifiers.

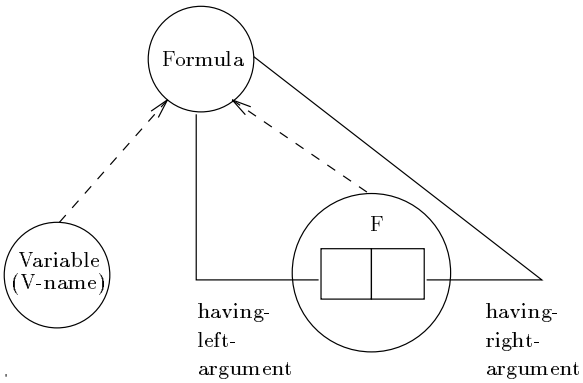


Figure 12: An example of generalization in PSM

As an example of generalization consider figure 12. In this schema the graphical conventions of PSM [HW93] 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 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 formalisation of generalization. The generalized object type has to be mapped on a coproduct in the instance category and the generalization arrows should correspond to the canonical injections. Of course, as the coproduct represents a *disjoint* sum in **Set**, this formalisation implies that specifiers have to be disjoint. In some data modelling techniques this is not necessarily true. A more general formalisation of generalized object types, which is also capable of dealing with non-disjoint specifiers, is presented in [LHF94]. This formalisation requires familiarity with some complex notions of category theory, e.g. *colimits*.

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

3.5 Collection Types

A *collection type* is an object type which instances correspond to (nonempty) sets of another object type, referred to as its *element type*. As sets are identical if 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 [EGH+92], *grouping classes* in SDM [HM81], and *power types* in PSM.

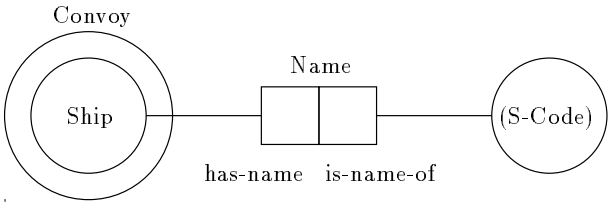


Figure 13: An example of a collection type in PSM

As a simple example of the application of collection types consider the schema of figure 13, which shows a PSM schema of the so-called *Convoy Problem* of [HM81]. 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 formalisation of collection types. One alternative is to use *exponents*. Set theoretically such a solution would correspond to the representation of subsets by means of a *characteristic function*. This approach has two serious disadvantages however. Firstly, an exponent is a complex category theoretic notion, which is not easily understood. Secondly, and more seriously, exponents do not exist in many interesting categories. The use of exponents 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 [BW90]. 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 [HW94]. 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 14. 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*.

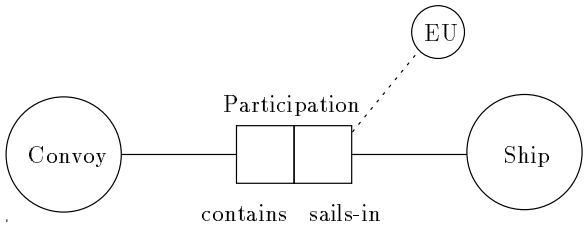
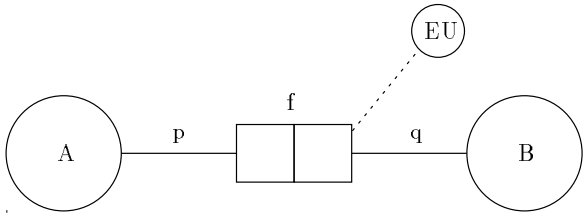


Figure 14: A translation of the Convoy Problem

To further illustrate the existensional uniqueness constraint, consider the abstract schema of figure 15. The sample population of this schema violates the existensional uniqueness constraint as both a_1 and a_2 are related, via role q, to b_1 and b_2 and therefore both correspond to the set $\{b_1, b_2\}$.



X	$p(X)$	$q(X)$
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 15: A population violating the existensional uniqueness constraint

The solution to the categorical formalisation 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.

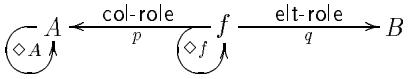


Figure 16: A solution for collection types

Category theoretically, this requirement states that the existensional uniqueness constraint of the schema of figure 15 is violated if and only if the arrows of the type graph of figure 16 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 and the involved relationship type respectively, can be found for which the following equalities hold:

$$\begin{aligned} \Diamond A \circ p \circ \Diamond f &= p \\ q \circ \Diamond f &= q \end{aligned}$$

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 15. 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 isomorphisms and satisfy the two equalities:

$$\begin{aligned} \Diamond A &= \{a_1 \mapsto a_2, a_2 \mapsto a_1\} \\ \Diamond f &= \{f_1 \mapsto f_3, f_3 \mapsto f_1, f_2 \mapsto f_4, f_4 \mapsto f_2\} \end{aligned}$$

4 Conclusions and Further Research

This paper presented 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 formalisations should not impose representational choices but instead focus on the *essence* of concepts. The framework described has been very general in the sense that 1) advanced conceptual data modelling concepts are incorporated, 2) very few syntactic restrictions on data models are imposed, and 3) the semantic target domain is not fixed.

The abstraction from representational issues allows the framework to be liberal with respect to syntax. 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.

The notion of instance category allows the framework to be liberal with respect to semantics. The framework offers opportunities for studying specific features in data models by offering a choice of corresponding categories as instance category. For example, uncertainty can be studied by considering **FuzzySet** as instance category.

Finally, the framework does not use many complex categorical notions compared with similar approaches. Still however, a certain mathematical maturity is required for a complete understanding. This is notably so for the full framework as presented in [LHF94], where also some well-known constraint types are incorporated, which uses some complex categorical notions such as colimit.

Further research is needed for a general treatment of constraints and to answer the question whether application of the framework yields interesting (in)equivalence results (category theory offers many concepts and results for equivalence in a mathematical context). In addition to that, the framework might provide a more general formal foundation for the meta model hierarchy described in [FO94]. Also an in-depth study of the various members of the class of instance categories **Fund** seems to be worthwile.

References

[AF88] D.E. Avison and G. Fitzgerald. *Information Systems Development: Methodologies, Techniques and Tools*. Blackwell Scientific Publications, Oxford, United Kingdom, 1988.

[AH87] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[BSW94] K. Baclawski, D. Simovici, and W. White. A categorical approach to database semantics. *Mathematical Structures in Computer Science*, 4:147–183, 1994.

[Bub86] J.A. Bubenko. Information System Methodologies - A Research View. In T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, editors, *Information Systems Design Methodologies: Improving the Practice*, pages 289–318. North-Holland, Amsterdam, The Netherlands, 1986.

[BW90] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Che76] P.P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[DJDR94] C.N.G. Dampney, M. Johnson, P. Dazely, and V. Reich. A higher order "commuting loop" structure that supports very large information systems data and proces architecture. In *Proceedings of the IFIP TC8 Working Conference*, Bonel University, Gold Coast QLD, Australia, May 1994.

[EGH+92] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H-D. Ehrich. Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering*, 9(4):157–204, 1992.

[Fal76] E.D. Falkenberg. Concepts for Modelling Information. In G.M. Nijssen, editor, *Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*, pages 95–109, Freudenstadt, Germany, January 1976. North-Holland.

[FO94] E.D. Falkenberg and J.L.H. Oei. Meta Model Hierarchies from an Object-Role Modelling Perspective. In T.A. Halpin and R. Meersman, editors, *Proceedings of the First International Conference on Object-Role Modelling (ORM-1)*, pages 218–227, Magnetic Island, Australia, July 1994.

[Fok92] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Technical University of Twente, Enschede, The Netherlands, 1992.

[Gri82] J.J. van Griethuysen, editor. *Concepts and Terminology for the Conceptual Schema and the Information Base*. Publ. nr. ISO/TC97/SC5-N695, 1982.

[HM81] M. Hammer and D. McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[HO92] T.A. Halpin and M.E. Orlowska. Fact-oriented modelling for data analysis. *Journal of Information Systems*, 2(2):97–119, April 1992.

[Hoa89] C.A.R. Hoare. Notes on an Approach to Category Theory for Computer Scientists. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 55 of *NATO Advanced Science Institute Series*, pages 245–305. Springer-Verlag, 1989.

[HPW93] A.H.M. ter Hofstede, H.A. Proper, and Th.P. van der Weide. Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, 18(7):489–523, 1993.

[HW93] A.H.M. ter Hofstede and Th.P. van der Weide. Expressiveness in conceptual data modelling. *Data & Knowledge Engineering*, 10(1):65–100, February 1993.

[HW94] A.H.M. ter Hofstede and Th.P. van der Weide. Fact Orientation in Complex Object Role Modelling Techniques. In T.A. Halpin and R. Meersman, editors, *Proceedings of the First International Conference on Object-Role Modelling (ORM-1)*, pages 45–59, Townsville, Australia, July 1994.

[Lan71] S. Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag, New York, New York, 1971.

[LHF94] E. Lippe, A.H.M. ter Hofstede, and P.J.M. Frederiks. Category-Theoretic Foundations of Conceptual Data Modelling. Technical Report, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, 1994.

[Mai88] D. Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland, 1988.

[NH89] G.M. Nijssen and T.A. Halpin. *Conceptual Schema and Relational Database Design: a fact oriented approach*. Prentice-Hall, Sydney, Australia, 1989.

[RB88] D.E. Rydeheard and R.M. Burstall. *Computational Category Theory*. Prentice-Hall, New York, New York, 1988.

[SFMS89] C. Sernadas, J. Fiadeiro, R. Meersman, and A. Sernadas. Proof-theoretic Conceptual Modelling: the NIAM Case Study. In E.D. Falkenberg and P. Lindgreen, editors, *Information System Concepts: An In-depth Analysis*, pages 1–30, Amsterdam, The Netherlands, 1989. North-Holland/IFIP.

[Shi81] D.W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[Sie90] A. Siebes. *On Complex Objects*. PhD thesis, Technical University of Twente, Enschede, The Netherlands, 1990.

[Tui94] C. Tuijn. *Data Modeling from a Categorical Perspective*. PhD thesis, University of Antwerpen, Antwerpen, Belgium, 1994.

[TYF86] T.J. Teory, D. Yang, and J.P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *Computing Survey*, 18(2):197–222, 1986.

[You89] E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.