A Categorical Framework for Conceptual Data Modeling: Definition, Application, and Implementation

A.H.M. ter Hofstede, E. Lippe, Th.P. van der Weide

Computing Science Institute
University of Nijmegen
Toernooiveld 1
NL-6525 ED Nijmegen
The Netherlands
{arthur,ernstl,tvdw}@cs.kun.nl

Published as: A.H.M. ter Hofstede, E. Lippe, and Th.P. van der Weide. A Categorical Framework for Conceptual Data Modeling: Definition, Application, and Implementation. Technical Report CSI-R9512, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, October 1995.

Abstract

For successful information systems development, conceptual data modeling is essential. Nowadays a plethora of techniques for conceptual data modeling exist. Many of these techniques lack a formal foundation and a lot of theory, e.g. concerning updates or schema transformations, is highly data model specific. 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 modeling concepts, such as relationship types, generalization, specialization, and collection types 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. Applications of the framework in the context of schema transformations and improved automated modeling support are discussed.

Keywords: Conceptual Data Modeling, Category Theory, Meta Modeling, ER, IFO, NIAM, Schema Transformations, CASE Shell
Classification: 68P99 *(AMS-1991)*, H.1.0. *(CR-1991)*

1 Introduction

Problem area

Conceptual data modeling is imperative for successful information systems development. Currently, many different conceptual data modeling techniques exist (see e.g. [HK87, PM88]). Examples are ER [Che76] and its many variants, functional modeling techniques, such as FDM [Shi81], and so-called object-role modeling techniques, such as NIAM [NH89]. Complex application domains, such as meta modeling, hypermedia, and CAD/CAM, have led to the introduction of advanced modeling concepts, such as present in the various forms of Extended ER (see e.g. [TYF86, EGH+92]), IFO [AH87], and object-role modeling extensions such as FORM [HO92, Hal95] and PSM [HW93, HPW93].

This plethora of techniques reflects the general situation in the field of information systems development. In [AF88] this situation is described by the term *Methodology Jungle*. In [Bub86] it was estimated that during the seventies and early eighties, hundreds if not thousands of information system development methods were introduced. Most organizations and research groups had defined their own methods. Hardly any of them had a formal syntax, let alone a formal semantics. The discussion of numerous examples, mostly with the use of pictures, was 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. This situation has not improved in recent years, it has, perhaps, even deteriorated.

Focus of paper

To some extent this latter observation is also true for the field of conceptual data modeling. Many conceptual data modeling techniques do not have a formal semantics or have completely different formalizations. A lot of theory developed for these techniques, e.g. in the fields of schema transformations, schema evolution, or updates, is highly technique dependent and not transferable. As such a unifying framework for conceptual data modeling 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 modeling techniques and to investigate some of its applications. This framework should clarify the precise meaning of fundamental data modeling 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* [Gri82] can also be applied to mathematical formalizations). 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.

Due to the high abstraction level of the framework it may suggest natural generalizations, expose similarities between seemingly different concepts, and allow for the transfer of results between different techniques. Another interesting application of the use of category theory can be found in the opportunity to consider different interpretations of a modeling 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 modeling 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 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 modeling 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. As will be shown, automated tool support for the framework can be realized and will allow such a choice for desired semantic features.

Choice for category theory

Category theory is 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 [Hoa89]: "Category theory is quite the most general and abstract branch of pure mathematics".

In the seventies and eighties category theory has also found its way into computing 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 [Gog91], 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 [AHS90], 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. As such, 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. [ES91, FSMS91, CSS94]), object oriented data models (see e.g. [Sie90, Tui94]), ER (see e.g. [DJM92]), and the Relational Model (see e.g. [IP94, BSW94]) have been proposed. In [SFMS89] a categorical framework for the axiomatization of conceptual modeling concepts is described (based on the notion of π-institution). In [Tui94] it is remarked that the uniformity of category theory provides a basis for interesting generalizations in the context of data modeling and that it not only offers insight in well-known operators but also allows for the definition of new operators, which would be far from trivial in other formalisms.

Organization of the paper

The paper is organized as follows. Section 2 describes the essential data modeling concepts, i.e. relationship types, generalization, specialization, and collection types, from a category theoretic point of view. In section 3, it is defined which categories provide a meaningful semantics for conceptual data modeling techniques. Further, applications of some of these categories are shown. Section 4 addresses schema transformations in the context of the framework and section 5 focuses on tool support. The tool described in this latter section is centered around a three-level architecture allowing analysts to define their own data modeling technique, to specify schemas in such a technique, and to enter populations of these schemas. Section 6 presents conclusions and identifies topics for further research. Finally, appendix A contains a brief introduction to category theory for those readers unfamiliar with this field of study.

2 Data modeling type constructors

In this section a number of important conceptual data modeling 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 2.1, 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 section 2.2. After the definition of type models, the various data modeling constructs are given a category theoretic definition. These constructs are defined in terms of restrictions on type models.

2.1 Type graphs

Data models[1] 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 (type) constructions are discerned by labeled arrows. Relationship types, for example, are object types, and thus correspond to nodes. The construction of a relationship type is described by arrows with label role. An object type participating via a role in a relationship type is target of such an arrow, 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 2.1

A type graph $\mathcal{G}$ *is a directed multigraph over a label set* {role, spec, gen, elt_role, clt_role} *such that there are no cycles consisting solely of* subtype *edges, i.e. edges with label* spec *or* gen. *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, labeled 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 labeled with spec, then A is a specialization of B, while if e is labeled with gen then B is a generalization of A (and possibly other object types). If edge $e\colon A \to B$ is labeled with clt_role, edge $f\colon A \to C$ is labeled with elt_role, and $\mathsf{clt}(e) = f$, then B is a collection type with as element type C.

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 2.5) 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 (actually, from an OO point of view they are quite natural, see section 3.2). Hence, restrictions, other than on cyclic inheritance structures, will not be imposed.

[1] In this paper, the term data model does not refer to a data modeling *technique*, but to a schema specified according to the conventions of such a technique. Both the term *data model* and the term *schema* will be used throughout this paper.

As an example of how data models can be represented as type graphs, consider figure 2, which shows the type graph of the NIAM data model in figure 1. Entity 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]).

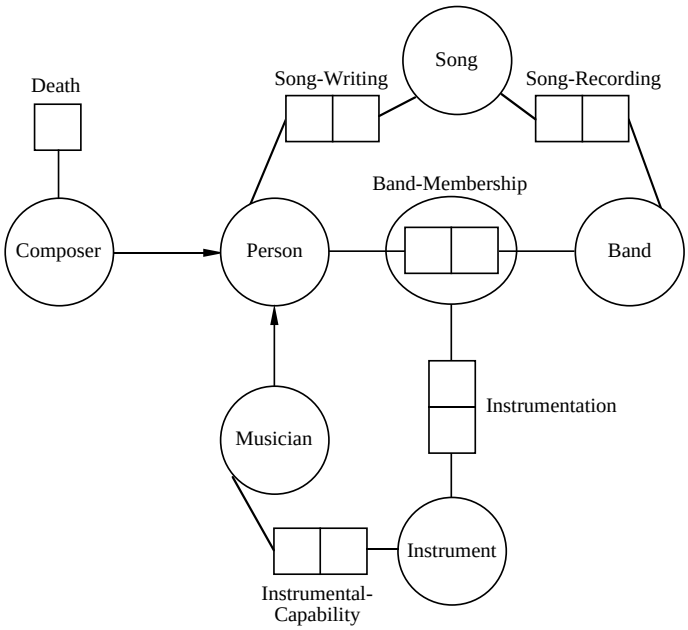


Figure 1: A NIAM data model

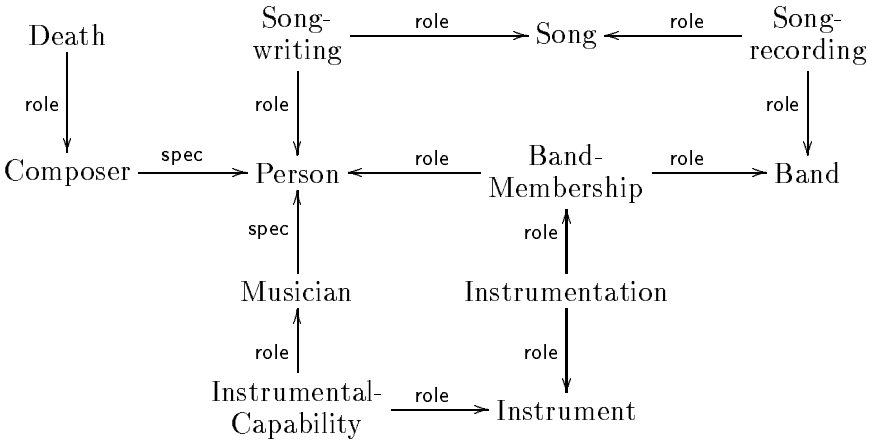


Figure 2: Type graph of the data model of figure 1

2.2 Type models

The semantics of a data model is the set of all its possible instantiations, which are also referred to as *populations*. In our approach, a population is defined as a graph homomorphism from the type graph to an *instance category*, which serves as an underlying calculation domain. First we shortly summarize the definition of a category. A more detailed description can be found in appendix A.

Definition 2.2

A category $\mathcal{C}$ *is a directed multigraph whose nodes are called* objects *and whose edges are called* arrows *such that:*

1. *For each pair of arrows* $f: A \to B$ *and* $g: B \to C$ *there is an associated arrow* $g \circ f: A \to C$*, the* composition *of* f *with* g*. Furthermore,* $(h \circ g) \circ f = h \circ (g \circ f)$ *whenever either side is defined.*
2. *For each object* A *there is an arrow* $\mathsf{Id}_A: A \to A$*, the* identity *arrow. If* $f: A \to B$*, then* $f \circ \mathsf{Id}_A = f = \mathsf{Id}_B \circ f$*.*

□

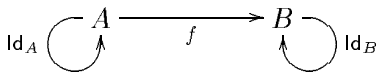


Figure 3: A simple example of a category

In figure 3 a simple example of a category is shown. In this category the choice of composites is forced: $f \circ \mathsf{Id}_A = f = \mathsf{Id}_B \circ f$. A population is referred to as a type model, and is formally defined by:

Definition 2.3

A type model Pop *for type graph* $\mathcal{I}$ *in category* C *is a graph homomorphism* $\mathsf{Pop}: \mathcal{I} \to C$*.* C *is referred to as the* instance category *of model* Pop*.* □

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 homomorphism Pop is sometimes omitted if it is clear from the context.

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 **Fund** of categories. Categories of this class have to fulfill 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**. These requirements will be discussed in section 3.1.

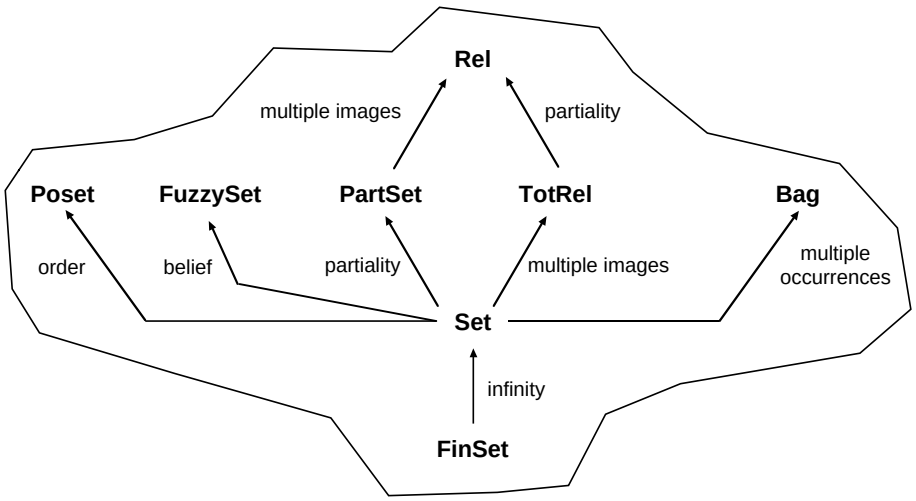


Figure 4: The class of categories **Fund**

In figure 4, some examples of categories in **Fund** are shown. In this figure each arrow is labeled with 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 categories **PartSet** and **Set** the objects are sets, but in **PartSet** the arrows do not have to be total functions as in **Set**. As will be shown in section 2.3, the category **PartSet** should be considered as calculation domain if one is interested in the study of "null"-values in relationship types. Other categories in figure 4 are:

- The instance category **TotRel** where the objects are sets and the arrows total relations. This category may be useful in the context of multi-valued attributes.
- The instance category **Bag** where the objects are bags (multisets) and the arrows total functions, such that the frequency of an original never exceeds the frequency of an image. This category can be chosen when objects cannot be distinguished on the basis of their properties.
- The instance category **Poset** where the objects are partially ordered sets and the arrows monotonous (i.e. order-preserving) functions. This category may be of importance in the context of systems with a weak notion of time, i.e. only the relative age of instances is to be taken into account.
- The instance 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 $f: \langle S, \sigma \rangle \to \langle T, \tau \rangle$ is a function $f: S \to T$ such that $\sigma \leq \tau \circ f$. The category **FuzzySet** plays an important role for systems that have to deal with uncertainty.

In the remainder of this section, the various data modeling constructions will be discussed from a categorical point of view. Some of these constructions will impose extra requirements on type models.

2.3 Relationship types

One of the central concepts in conceptual data modeling is the concept of *relationship type*. A relationship type represents an association between object types and may be n-ary in some data modeling techniques (where $n \geq 1$), 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. A simple example of a binary relationship type represented conform the ER conventions is shown in figure 5. The associated type graph is shown in figure 6.

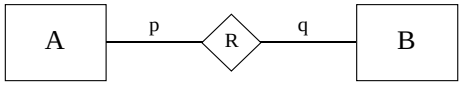


Figure 5: A simple ER schema

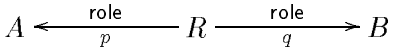


Figure 6: Type graph of figure 5

Usually, relationships are instantiated as mappings from the involved roles to values. Our approach is slightly more general. The actual representation of relationship instances becomes irrelevant, it is sufficient to have the components available via *access functions*. As an example consider the interpretation of the following population of relationship R from figure 5 in the category **FinSet**.

$\mathsf{Pop}(R)$

p	q
a_1	b_1
a_2	b_1

According to definition 2.3, 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!):

$$\begin{aligned} \mathsf{Pop}(p) &= \{r_1 \mapsto a_1, r_2 \mapsto a_2\}, \\ \mathsf{Pop}(q) &= \{r_1 \mapsto b_1, r_2 \mapsto b_1\}. \end{aligned}$$

In this approach, the two relationship instances have an identity of their own (r_1 and r_2), and the functions p and q can be applied to retrieve the respective components. The populations of the object types A and B are omitted.

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

conventional data modeling techniques these cases are equivalent as instances of relationship types are uniquely determined by their components. In OO approaches, however, these cases are not equivalent as objects are not necessarily uniquely determined by their properties.

Next we consider the instantiation of a relationship type in the context of instance category **PartSet**. As remarked before, this category allows certain components of relationship instances to be undefined:

$$\begin{aligned} \mathsf{Pop}(p) &= \{r_2 \mapsto a_2\}, \\ \mathsf{Pop}(q) &= \{r_1 \mapsto b_1, r_2 \mapsto b_1\}. \end{aligned}$$

In this population, relationship instance r_1 represents a case of incomplete knowledge as it does not have a corresponding object playing role p.

As a final example, the instance category **Rel** is chosen. 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:

$$\begin{aligned} \mathsf{Pop}(p) &= \{r_2 \mapsto a_1, r_2 \mapsto a_2\}, \\ \mathsf{Pop}(q) &= \{r_1 \mapsto b_1, r_2 \mapsto b_1, r_2 \mapsto b_2\}. \end{aligned}$$

2.4 Subtype relationships

Many conceptual data modeling 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 modeling 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]. For an in-depth discussion of these notions, refer to [HP95].

2.4.1 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.

As an example of specialization consider the IFO schema of figure 7 (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 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.

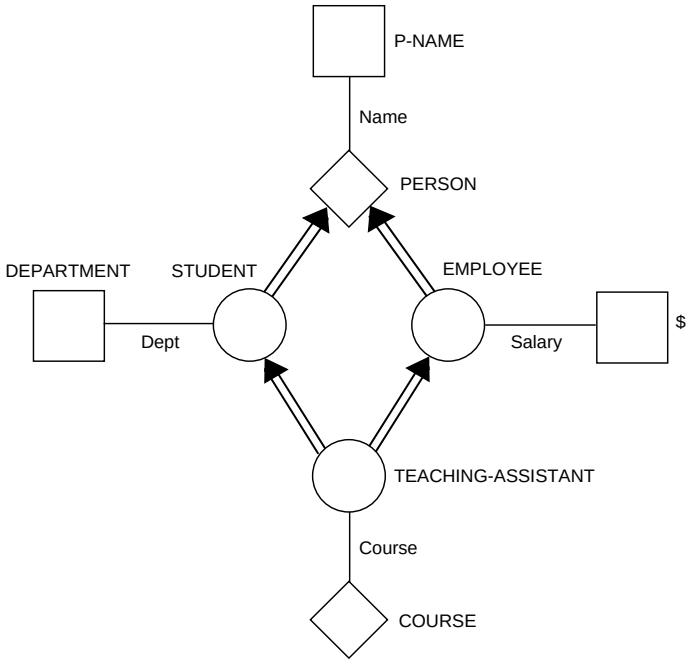


Figure 7: An example of a subtype hierarchy in IFO

A special form of inheritance is identity inheritance. Basically, two approaches exist. In the first approach, objects can be instances of more than one object type. In the second approach, objects can occur in one object type only. Corresponding objects are related via injective functions. This approach is in line with most OO data models and has as main advantage that structure and representations can be separated more effectively. Consequently, a subtype relation is required to be mapped onto a monomorphism (in the category **Set** a monomorphism corresponds to an injective function, see appendix A).

However, problems may occur as a result of multiple inheritance: an object should always correspond with precisely one object in each supertype. As an example of this problem, consider figure 8, where teaching assistant *TA999* corresponds to two different persons: *Richards* and *Jones*. To solve such problems, subtype diagrams are required to commute.

2.4.2 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, HPW93].

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 9. 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

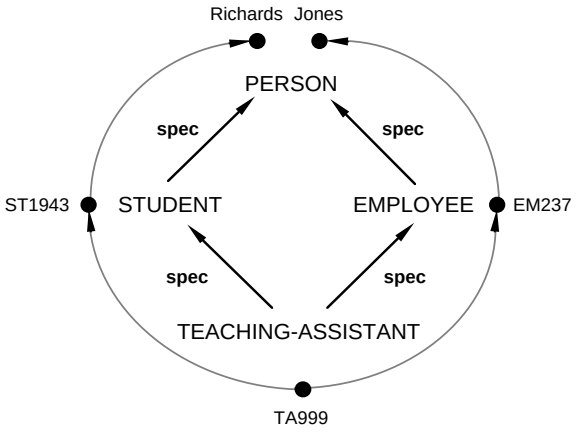


Figure 8: A non-commutative diagram

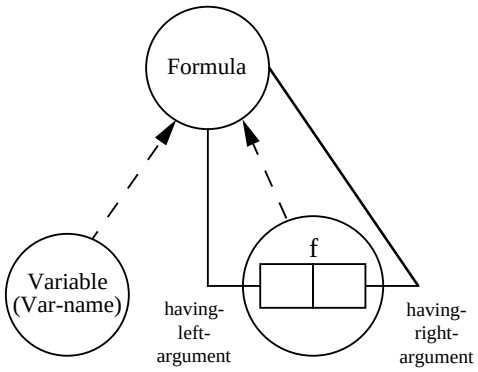


Figure 9: An example of generalization in PSM

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 to factor out common properties of object types. These properties can then be related to the generalization of these types.

A straightforward approach to incorporating generalized types in type models is defining them as coproducts of the corresponding specifiers. The generalized object type then has to be mapped on a coproduct in the instance category and the generalization arrows should correspond to the canonical injections relating instances of specifiers to their generalized version. Of course, as the coproduct represents a *disjoint* union in **Set**, this formalization implies that specifiers have to be disjoint. In some data modeling techniques (including PSM) this is not necessarily true. For example, the schema of figure 10 shows specifiers of a generalised object type that are *not* disjoint. In this schema, the object types *Plant-eater* and *Flesh-eater* have the *Omnivores* in common.

This problem can be solved by using the general notion of colimit. Let G be a generalized type with a set V of specifiers. The solution starts with the observation that the collection of instances of G is completely determined by subtype relationships between instances of specifiers of G. The following definitions give a formal description of the restriction of a population to

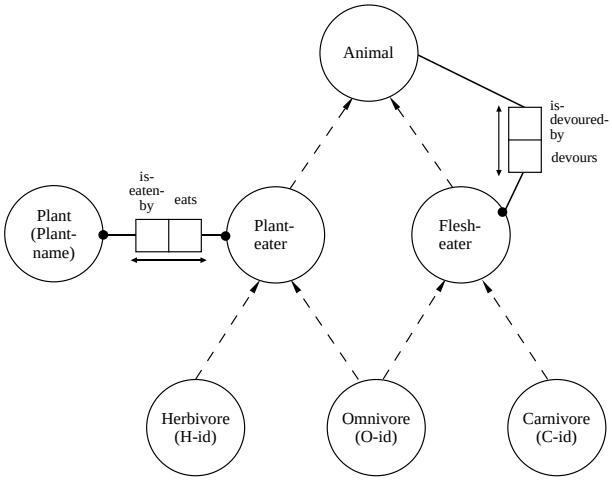


Figure 10: Another example of generalisation

contain only the relevant subtype relations among specifiers in V. First, in the type graph all relevant nodes and subtype edges are determined.

Definition 2.4

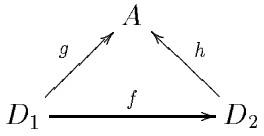
Given a graph $\mathcal{G}$ and a set of nodes $N \subseteq \mathcal{G}_0$, then the domination of $\mathcal{G}$ by N *is equal to a subgraph D of $\mathcal{G}$ that is defined as follows: The edges of D are the edges from $\mathcal{G}_1$ that occur on a directed path that ends in a node $n \in N$. The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

Next, populations are to be restricted to these relevant nodes. For a concrete population of a type graph, this leads to a concrete diagram in the corresponding instance category.

Definition 2.5

Given a graph homomorphism $D{:}\mathcal{G} \to C$ and a set of nodes $V \subseteq \mathcal{G}_0$. Let $\mathcal{G}_V$ be the domination of $\mathcal{G}$ by V. Then the domination of D by V *is equal to the restriction of D to $\mathcal{G}_V$.* □

The problem to be solved now is that instances may have various manifestations (in different object types). These manifestations are to be unified by the introduction of canonical representations (identifiers) for corresponding occurrences. The instance universe U^V_{Pop} contains the canonical representations of all instances of a set V of object types in a type model Pop. Category theoretically, instance universes can be captured by the notion of colimit. A colimit is defined for a diagram, referred to as the *base* of the colimit, and further consists of an object, the *apex*, and a set of arrows to this apex, one for each object in the diagram. For each arrow f from object D_1 to object D_2 in the diagram, the diagram consisting of this arrow and the arrows g and h from D_1 and D_2 to the apex, should commute (see the following diagram).



Hence, corresponding instances in different object types will be related to the same instance in the instance universe. The requirement that the apex is minimal, category theoretically captured by the requirement that a unique arrow from the apex should exist to every other object that also satisfies the commutation requirements, ensures that the instance universe does not contain any superfluous instances. For a formal definition of a colimit, the reader is referred to appendix A.

Definition 2.6

The instance universe *determined by a set of object types $V \subseteq \mathcal{G}_0$ in a given type model* Pop*, denoted as U^V_{Pop}, is the apex of the colimit with as base the subtype diagram dominated by V.*

□

In a type model Pop, each generalized object G with specifiers in V has to be mapped onto the instance universe U^V_{Pop} and the generalization arrows onto the corresponding colimit injections.

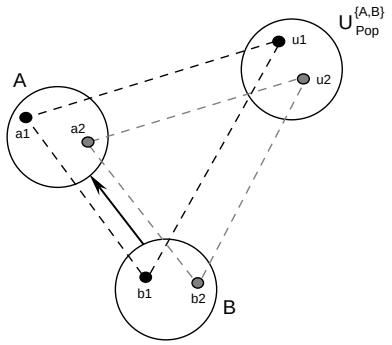


Figure 11: Illustration of an instance universe

Example 2.1 As an illustration of the previous definition consider figure 11. In this figure, there is a subtype relation between objects A and B, relating b_1 to a_1 and b_2 to a_2. Hence, a_1 and b_1 should correspond to one instance in the instance universe $U^{\{A,B\}}_{\mathsf{Pop}}$, e.g. u_1, and a_2 and b_2 also, e.g. u_2. The dotted lines illustrate the commutation requirements. A slightly more complex example of an instance universe can be found in figure 12.

□

In [LH94] it is proven that in a category that has disjoint coproducts the colimit of a diagram consisting of complementable monomorphisms, which is true for the subtype diagram of definition 2.6, always exists. The associated arrows are then also complementable monomorphisms. This result is important as some categories have disjoint coproducts, 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 coproducts 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.

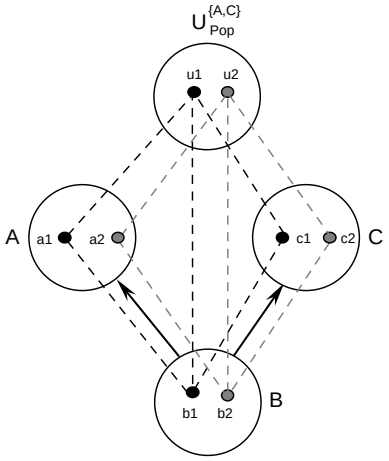


Figure 12: Another illustration of an instance universe

2.5 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.

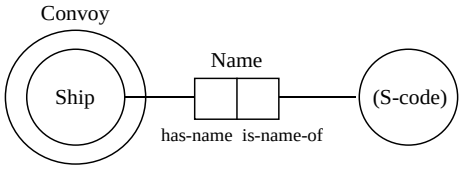


Figure 13: An example of a collection type in PSM

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.

The approach to the categorical formalization of collection types is based on an alternative treatment of collection types 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*.

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

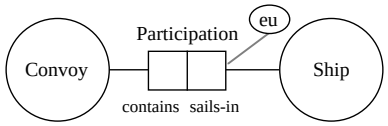


Figure 14: A translation of the Convoy Problem

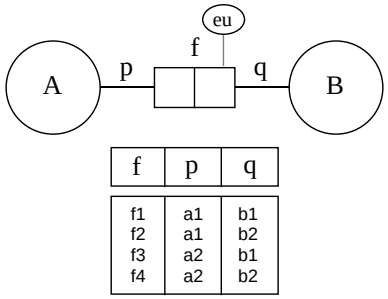


Figure 15: A population violating the existensional uniqueness constraint

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 $\{b_1, b_2\}$.

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.

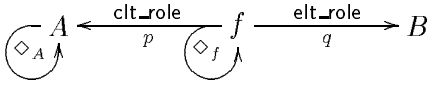


Figure 16: A solution for collection types

Category theoretically, roles p and q of the type graph of figure 15 satisfy the *extensionality property* in a certain type model iff the identity arrows Id_A and Id_f are the only possible substitutions for $\Diamond_A$ and $\Diamond_f$ in the following equations (see also figure 16):

$$\begin{aligned} \Diamond_A \circ p \circ \Diamond_f &= p, \\ q \circ \Diamond_f &= q. \end{aligned}$$

In these equations $\Diamond_A$ and $\Diamond_f$ correspond to permutations on A and f respectively. If no other permutations than the identity permutations satisfy the equations, no instances of A can be interchanged in f without loss of information. In that case the existensional uniqueness constraint is not violated. 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:

$$\begin{aligned} \diamond_A &= \{a_1 \mapsto a_2, a_2 \mapsto a_1\}, \\ \diamond_f &= \{f_1 \mapsto f_3, f_3 \mapsto f_1, f_2 \mapsto f_4, f_4 \mapsto f_2\}. \end{aligned}$$

3 Semantic interpretations

One of the most important advantages of using a categorical approach to the semantics of conceptual data modeling techniques is that different instance categories can be used. Depending on the semantic features that are to be studied, e.g. null values, uncertainty, time, an appropriate instance category can be chosen.

In this section first the requirements that instance categories should satisfy are outlined, then various examples of instance categories and their applications are discussed.

3.1 Valid instance categories

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

- All finite coproducts and products must exist.
- Coproducts must be disjoint (see appendix A).

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 modeling 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. This leads to the following definition of a valid instance category:

Definition 3.1

A category C is a valid instance category *iff all finite products and coproducts exist, coproducts are disjoint, and a functor, i.e. a graph homomorphism preserving identities and composition, F: **FinSet** $\to C$ exists such that for all $M_1, M_2: \mathcal{G} \to$ **FinSet** it holds that $M_1 = M_2 \Leftrightarrow F \circ M_1 = F \circ M_2$.* □

The following instance categories are valid: **FinSet**, **Set**, **PartSet**, **Rel**, **FuzzySet**. A description of various category theory constructs and proofs for these categories can be found in [LH94].

3.2 Object Orientation

In the past years, interest in object orientation has grown spectacularly. A plethora of methods and techniques (more or less) based on object oriented principles have been proposed in the literature. Unfortunately, there is not much consensus about what object orientation precisely is [Kin89, Dit90] and, worse, firm theoretical foundations for many methods and techniques, especially for those developed to support the analysis phase of system development, are lacking [Jac93, EJW95, CF92, Iiv95]. These shortcomings seem to be particularly true for OO data models. Although formal OO data(base) models exist (see e.g. [AHV95]), there is no commonly agreed upon formal OO *conceptual* data model [ADM+89, Kim91]. The framework presented in the previous section may be used to provide OO conceptual data modeling techniques with a formal foundation. This will be argued in the remainder of this subsection.

OO data models allow for the description of so-called complex values. Complex values can be viewed as finite trees whose internal nodes indicate the use of tuple and set constructors [AHV95]. Types in OO data models may be defined by the use of these constructors. As an example, consider the following type definitions taken from [AHV95]:

class *Person*
  type tuple (name: string, citizenship: string, gender: string);
class *Director* inherit *Person*
  type tuple (directs: set(*Movie*));
class *Actor* inherit *Person*
  type tuple (acts_in: set(*Movie*),
              award: set(tuple(prize: string, year: integer)));
class *Movie*
  type tuple (title: string,
              actors: set(*Actor*),
              director: *Director*);
class *Theater*
  type tuple (name: string, address: string, phone: string).

While such definitions may be satisfactory from a design point of view, they violate the Conceptualisation Principle, for the following reasons. Firstly, the schema contains redundancy as the fact that an actor plays in a certain movie will be stored twice (in the attribute *acts_in* of *Actor* and the attribute *actors* of *Movie*). To remove this redundancy a conceptual irrelevant choice for one of these attributes has to be made (this can be compared to the forced choice for an *owner* in CODASYL). Secondly, properties are described via explicit structural descriptions. Consider for example the award information for actors. The fact that this information is stored as a set of tuples is not relevant from a conceptual point of view. OO conceptual data models should allow complex values to be described without forcing particular representations. Hence, in OO conceptual data models all objects should have their own *identity*, independent of their structural composition. This can be achieved by focusing on the *properties* of complex values. Thirdly, in an analysis model concrete data types, such as string, are not relevant. They also seem to be conflicting with the idea of object identity: do concrete values have an identity independent of their value?

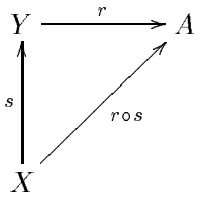
In the framework, complex values have their own identity. In addition to that, they have to satisfy certain properties depending on their type. For example, an instance of a relationship

type, which is the conceptual equivalent of a tuple constructor, has to provide access to its components, while an instance of a collection type, which is the conceptual equivalent of a set constructor, is to be uniquely determined by its elements. By abstracting from representations, the framework is very flexible with respect to type definitions. There are no problems, for example, with an object type which is both a relationship type and a collection type. The framework simply allows such an object type to be viewed from two different perspectives. Structural conflicts can not occur.

In many OO conceptual data models, relationship types are treated as special kinds of object types, often with a restricted use (consider for example OMT [RBP+91]). This is unfortunate as it complicates the view that an OO conceptual data model simply consists of object types. In addition to that, not many OO conceptual data models support the description of objects with set behavior. The importance of collection types for conceptual modeling is stressed in [MS95]. The framework supports both relationship types and collection types in a natural way.

From an OO point of view the arrows labeled with role in a type graph may be viewed as attributes of the object types corresponding to their sources. By choosing as instance category the category **Rel**, attributes may be *multi-valued* (*set-valued*), which is often the case in OO data models [KL89, ZM90]. In such cases the value of an attribute can be a (possibly empty) set of values.

Finally, inheritance of attributes is realized through arrow composition. If an object type X is a subtype of another object type Y, which has attribute A, then X also has access to this attribute via the composition of the subtype arrow and the attribute arrow.



Note that the framework does not restrict the use of inheritance structures apart from excluding cyclic inheritance structures. As a result, a ternary relationship type can be a subtype of a binary relationship type. From an OO point of view, this would simply mean that certain instances of that binary relationship type have an extra attribute.

3.3 Incomplete and uncertain knowledge

Models in **FuzzySet** can be used to model uncertainties. Every object type A is equipped with a function σ_A that describes the probability that an element is a member of that object type. The arrows in **FuzzySet** are total functions, and for each arrow $f: A \to B$ it must hold that $\sigma_A(a) \leq \sigma_B(f(a))$. 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 1, 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.

There are of course other ways of introducing uncertainty in data models. In [KS94, BGP92], for example, probabilistic approaches to uncertainty are considered. For the semantics of these models, an instance category **ProbSet** could be chosen. An object in **ProbSet** is a set of probability functions based on a common domain. A probability function τ assigns values from the interval [0,1] to the values of the common domain, say V, such that $\Sigma_{v \in V} \tau(v) = 1$. The arrows of **ProbSet** are total functions. As a simple example of the use of this category, consider the following table, adapted from [BGP92]:

EMPLOYEE	DEPARTMENT	QUALITY
Jon Smith	Toy	0.4 [Great] 0.5 [Good] 0.1 [Fair]
Fred Jones	Houseware	1.0 [Good]

In this population, DEPARTMENT is a deterministic attribute of EMPLOYEE, while QUALITY is a stochastic attribute. For example, *Jon Smith* has quality *Great* with a probability of *0.4*.

A type graph representing this (partial) data model could be:

$$\text{DEPARTMENT} \xleftarrow[\text{dep}]{\text{role}} \text{EMPLOYEE} \xrightarrow[\text{qual}]{\text{role}} \text{QUALITY}$$

The population Pop, represented in the table above, can be represented in terms of this type graph and the category **ProbSet**. For convenience, (1) tuples of the form $x \mapsto 0$ are omitted and (2) an assignment a may be given a short name s, for reference purposes, by writing $s : a$.

$$\begin{aligned}
\mathsf{Pop}(\text{EMPLOYEE}) &= \{e_1 : \{\text{Jon Smith} \mapsto 1\}, e_2 : \{\text{Fred Jones} \mapsto 1\}\} \\
\mathsf{Pop}(\text{DEPARTMENT}) &= \{d_1 : \{\text{Toy} \mapsto 1\}, d_2 : \{\text{Houseware} \mapsto 1\}\} \\
\mathsf{Pop}(\text{QUALITY}) &= \{q_1 : \{\text{Great} \mapsto 0.4, \text{Good} \mapsto 0.5, \text{Fair} \mapsto 0.1\}, q_2 : \{\text{Good} \mapsto 1\}\}
\end{aligned}$$

The mapping of the arrows d and q is now straightforward:

$$\begin{aligned}
\mathsf{Pop}(\text{dep}) &= \{e_1 \mapsto d_1, e_2 \mapsto d_2\} \\
\mathsf{Pop}(\text{qual}) &= \{e_1 \mapsto q_1, e_2 \mapsto q_2\}
\end{aligned}$$

In general, there are many issues related to uncertainty in data models. Information may not only be uncertain, but also missing, or vaguely known (i.e. only known to be in a certain set or interval of values). Missing information may have numerous interpretations, e.g. *not applicable*,

unknown, *nonexistent*, or *confidential*. In [Mor90] many of the issues related to imprecise and uncertain information are treated. The use of our approach could lead to the development of appropriate instance categories as a formal foundation for further study of these issues. Such instance categories at least have to incorporate features of the categories **FuzzySet** and **PartSet**.

3.4 Time

In so-called historical or temporal databases the history of the creation and deletion of object instances is maintained (see e.g. [SA86, Sno90]). Historical databases do not forget information and allow previous states of the database to be restored, for example if performed changes turn out to be incorrect or undesirable. Conceptual data modeling techniques exist that support an explicit notion of time and consequently facilitate the implementation of an application in terms of a historical database management system. Examples of such techniques are described in [Ari86] and [TLW91].

The framework can provide a natural semantics for these techniques by the choice of an appropriate instance category. Such an instance category could be the category **TimeSet**, where each object is a set with a function assigning a time interval to each element of that set. The set of time points $\mathcal{T}$ is a totally ordered set and has a maximal element **now**. A time interval is an element $\langle t_1, t_2 \rangle$ from $\mathcal{T} \times \mathcal{T}$ such that $t_1 \leq t_2$. The arrows in this category are functions between the sets of the associated objects such that for each element in the range of that function the time interval is included in the time interval of the origin of that element. This is a necessary requirement and, among others, reflects that composed objects can not outlive their various parts. For example, the fact that a certain person P_1 is enrolled in a certain course C_1 cannot be known before either P_1 or C_1 have come into existence, or after either P_1 or C_1 are no longer "alive".

We conclude with a brief description of an alternative approach to the incorporation of time in the framework. This construction starts with the observation that it is possible to construct a new type model that describes type model changes over time for every valid instance category. Given a category C a new category C^τ can be constructed as follows:

- The objects are total functions $o\colon \mathcal{T} \to C_0$.
- The arrows are total functions $a\colon \mathcal{T} \to C_1$.
- The sources and targets of the arrows are defined by:
  $\mathsf{source}(a) = \lambda t.\mathsf{source}(a(t))$ and $\mathsf{target}(a) = \lambda t.\mathsf{target}(a(t))$.
- Composition is defined by $f \circ g = \lambda t.f(t) \circ g(t)$.
- The identity arrow for an object o is defined by $\mathsf{Id}_o = \lambda t.\mathsf{Id}_{o(t)}$.

It is straightforward to verify that for any valid instance category C, the category C^τ is also a valid instance category.

Given a category C the function @ can be constructed $@: \mathcal{T} \to C^{\mathcal{T}} \to C$ that is defined by $@_t(x) = x(t)$. For every $t \in \mathcal{T}$, $@_t$ is a functor. Application of this functor yields the population valid at time t.

The evolution of a population for a given type graph $\mathcal{G}$ with an instance category C through time can be described as a total function that maps the time domain $\mathcal{T}$ to a population for this typegraph: $M: \mathcal{T} \to \mathcal{G} \to C$. Given such a function M that describes the evolution of a population, a new model $M^{\mathcal{T}}: \mathcal{G} \to C^{\mathcal{T}}$ can be constructed. This model is defined by $M^{\mathcal{T}}(x) = \lambda t.M(t)(x)$, or alternatively by $@_t \circ M^{\mathcal{T}} = M(t)$. The following lemma is then not very difficult to prove (using the fact that @ is continuous, i.e. preserves limits):

Lemma 3.1 If for each $t \in \mathcal{T}$, $M(t)$ is a valid type model then $M^{\mathcal{T}}$ is also a valid type model.

Consequently, it is possible to add time to every instance category. Note that no assumption has been made about the underlying structure of the time domain $\mathcal{T}$. It need not even be (partially) ordered.

4 Schema transformations

The problems of schema equivalence and schema transformations have often been studied in the literature (see e.g. [JNS83, Hul86, Hal91, AS84, Kob86, Hai91, TS93]). For many data modeling techniques, transformations have been described, both on the conceptual level as from the conceptual level to the internal level. Unfortunately, much theory developed in this area seems to be quite data model specific. Furthermore, there is no currently agreed upon definition of schema equivalence, which complicates correctness considerations of schema transformations.

In this section, the application of the categorical framework for schema transformations is considered. The framework offers four main advantages:

1. As category theory aims at discovering and exploiting relations between various fields of study, a lot of theory for transformations is available.
2. Due to the high abstraction level of the framework, the discussion of schema transformations can be kept data model independent.
3. Transformations may be studied with respect to specific instance categories if they are based on the existence of specific semantic features (such as null values). The role these features play in determining the correctness of certain transformations then becomes more explicit.
4. Correctness proofs are relatively straightforward and easy to understand as they do not employ representational properties.

Of course, the issue of schema transformations is a far from trivial one, and a complete treatment is outside the scope of this paper. Rather, the use of three categorical notions for the formal verification of the correctness of schema transformations is illustrated sample wise. The transformations that are described are proven correct with respect to the following categorical notion of schema equivalence:

A schema S_1 is at least as generic as a schema S_2 if each model of S_2 can be translated, via the application of categorical constructs, to a model of S_1. Schemata S_1 and S_2 are equivalent if the reverse is also true.

In section 4.1, the use of coproducts for the object type absorption/extraction transformation is shown. In section 4.2, the use of arrow composition for the nest/unnest transformation is illustrated, and, finally, in section 4.3 it is shown how pullbacks may be used to formally verify the correctness of the join/split transformation.

4.1 Object type absorption - object type extraction

In this section, the object type absorption/extraction transformation (see e.g. [Hal95]) is illustrated. In figure 17 a simple example of this type of transformation is shown. In the left most IFO schema of this figure the binary relationship types f and g between object types A and B may be replaced by a ternary relationship type between A, B, and a concrete object type with as domain the set of instances $\{f, g\}$.

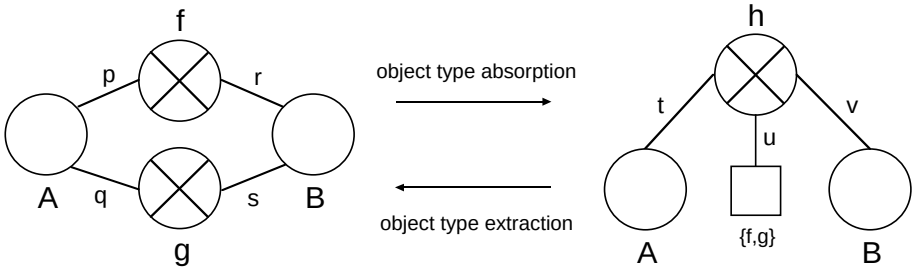
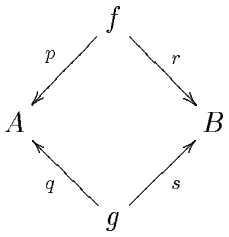


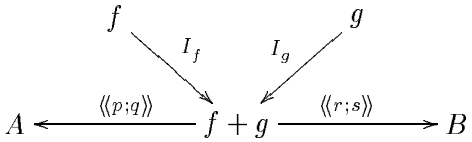
Figure 17: Illustration of the absorption/extraction transformation in IFO

In the following diagram the type graph of the left most IFO schema is shown. This diagram can also be considered as the canonical mapping of the type graph into a category. In the rest of this section, the convention will be adopted that the mapping Pop of the arrows and the object types is omitted. We will show how this population can be transformed into a population of the right most IFO schema. This proves that the left most schema is at least as generic as the right most schema. The transformation the other way around is very similar and will not be shown.

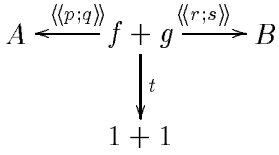


In the first step, the coproduct of the relationship types f and g is constructed. To avoid a loss of information, the canonical injections have to be included in the diagram. The two role arrows p and q can be replaced by the arrow $\langle\langle p; q\rangle\rangle$. Likewise, the role arrows r and s can be replaced by the arrow $\langle\langle r; s\rangle\rangle$. In set-theoretical terms, the arrow $\langle\langle r; s\rangle\rangle$ represents a mapping

from $f + g$ to B composed of the mappings r from f to B and s from g to B, such that r is applied to elements originating from f and s is applied to elements originating from g.



In the second step, the canonical injections I_f and I_g are replaced by a mapping from $f + g$ to $1 + 1$, the coproduct of two terminal objects. Category theoretically, a terminal object is an object such that there is exactly one arrow from each object to that object. Terminal objects in a category can be proven to be isomorphic. In **Set** any one-element set is terminal. In the resulting diagram the arrow t from $f+g$ to $1+1$ is the unique arrow distinguishing the elements from f and g in the coproduct. Hence, the canonical injections have become superfluous and can be omitted. The resulting diagram can be seen as a population of the right most IFO schema:



4.2 Nesting - unnesting

In data modeling techniques where relationship types may play a role in other relationship types, nesting and unnesting are common notions. Unnesting a relationship g via a role r which is played by another relationship type f corresponds to replacing that role by roles of f (see figure 18). Hence, composed objects playing role r are decomposed into their parts. If each instance of relationship type f has to play a role in relationship type g (this is captured by the cardinality constraint *1:* on role s in the left most ER schema of figure 18), relationship type f becomes superfluous after g is unnested.

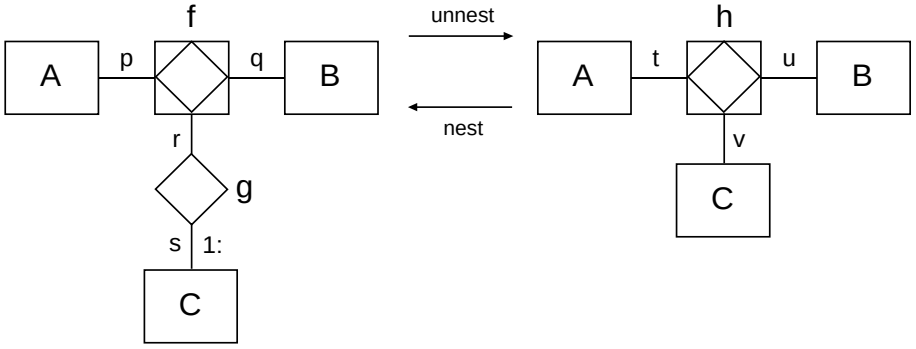
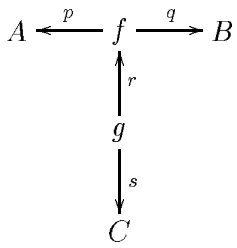
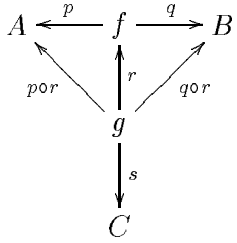


Figure 18: Illustration of the nest/unnest transformation in ER

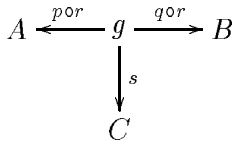
A population of the left most ER schema can be captured by the following diagram:



As arrow composition is always defined in a category, the arrows $p \circ r$ and $q \circ r$ may be added:



Finally, the cardinality constraint, stipulating that each instance of f participates in g, translates categorically to the fact that r should be mapped onto an epimorphism (In **Set**, an epimorphism corresponds to a surjective function). Hence, by omitting the arrows p and q no information loss results:



This diagram may be viewed as a population of the right most ER schema. Again, the reverse transformation is omitted.

4.3 Join - split

In some cases, relationship types with three or more roles may be split without loss of information. This is particularly true in case there is a key on $n-2$ roles, with n the total number of roles of the relationship type. Consider for example the right most NIAM schema of figure 19. In this schema, there is a key on role t of ternary relationship type h. Hence, h may be split into two binary relationship types: one between object types A and B and one between object types A and C. Naturally, there are keys on the roles played by object type A. The reverse transformation is called a join. For equivalence, it is needed that if an instance of A is associated with an instance of B it is also associated with an instance of C and vice versa. This is a result of the fact that the join transformation is based on the inner join (as not many data modeling techniques allow null values in relationship instances). This requirement is captured by the equality constraint on roles p and r in the left most NIAM schema of figure 19. In the rest of this section, focus is on the join transformation.

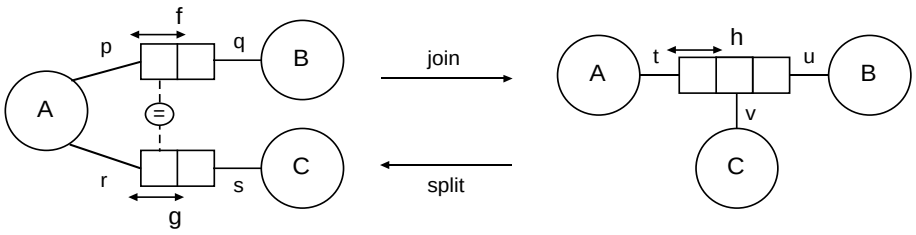
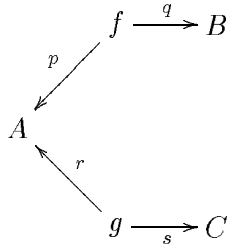
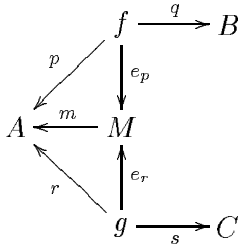


Figure 19: Illustration of the join/split transformation in NIAM

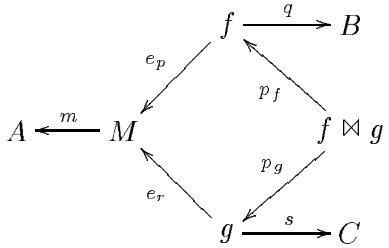
The following diagram shows the general representation of a population of the left most NIAM schema:



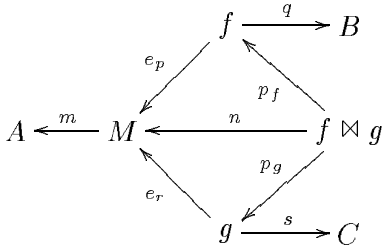
Categorically, the fact that there is an equality constraint on roles p and r translates to the requirement that there is an object M and a monomorphism $m: M \to A$ such that two epimorphisms $e_p: f \to M$ and $e_r: g \to M$ exist with $p = e_p \circ m$ and $r = e_r \circ m$. Object M corresponds to a subobject of A containing the instances of A playing both role p and role r.



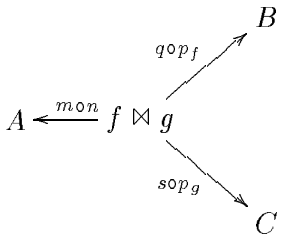
The requirements $p = e_p \circ m$ and $r = e_r \circ m$ imply that the arrows p and q may be omitted from the diagram. Further, the pullback of the arrows e_p and e_r may be added. The pullback corresponds, in set-theoretical terms, to a join. It should be noted, however, that not all valid instance categories have pullbacks (**Rel** is an example of such a category). This is only natural as in some models constraints over several object types should not occur, it would e.g. violate the information hiding principle of Object Orientation.



The pullback of e_p and e_r consists of an object, denoted as $f \bowtie g$, and two arrows p_f and p_g. It is required that the diagram consisting of these four arrows commutes. Hence, there is a unique arrow n such that $n = e_p \circ p_f = e_r \circ p_g$. In the following diagram this arrow is explicitly added:



One can now prove that for every valid instance category, as e_p and e_q are epimorphisms the corresponding pullback arrows p_f and p_g are also epimorphisms (this is not true for every category!). Hence, omission of arrows s and q and the addition of the composed arrows $q \circ p_f$ and $s \circ p_g$ does not lead to a loss of information:



This diagram may be viewed as a population of the right most NIAM schema. Note that $m \circ n$ is a monomorphism (the composition of an arrow with a monomorphism always yields a monomorphism). Hence, the requirement that t, which is to be mapped onto this arrow, should be a key of h is satisfied.

4.4 Epilogue

The transformations described in the previous sections can easily be generalized. For example, the object type absorption/extraction transformation can be applied to a schema with m relationship types between the same n object types. The categorical correctness proofs, however, remain essentially the same.

More importantly, it may be so that some desirable transformations can only be proven correct with respect to specific categories. This was e.g. the case for the join/split transformation, which cannot be applied in the instance category **Rel**. As another example consider the category **PartSet**. In the context of this category, the cardinality constraint specified in the left most ER schema of figure 18 would not be required, as **PartSet** allows for null values in relationship types. Hence, some transformations can be proven correct with respect to every instance category, and some only with respect to certain specific instance categories.

5 Tool support

Many contemporary commercial CASE tools seem to be merely used for documentation and verification purposes (see e.g. [WD90, Ver93]). Partially, this is a result of the inflexibility of these tools: a specific view on systems development is *hard-coded* in them, and therefore cannot be changed or customized to include knowledge that is based upon information engineers' practical experience [VJT92, Wij91]. By consequence, information engineers are left with the problem of finding a way of applying these rigid and inflexible tools in their information engineering practice.

From the mid-1980s onwards, research has focused on this problem. It is claimed that automated tools are preferably built according to a *CASE shell* architecture. Such an architecture allows for the modification and extension of the tool's behavior as the tool includes explicit and adaptable method knowledge. As a consequence, information engineers are able to adapt support tools to their working styles instead of the other way around. Crucial for the development of a CASE shell is the availability of a suitable and formally defined technique for the representation of method knowledge. Such a technique is referred to as a *meta-modeling technique*. Method knowledge represented in a CASE shell according to such a technique is called a *meta-model*.

The concept of a CASE shell is not new. Commercial products such as Toolbuilder of IPSYS Software, Virtual Software Factory of Systematica and MetaDesign of Meta Software Corporation or academic products such as RAMATIC [BBD+89], Metaview [STM88] and MetaPlex [CN89], claim to generate CASE tools tailored to specific methods and organisations. Even a tool exists (MetaEdit [SLTM91]), that supports the modification of meta-models. One of the problems with these shells, however, is that the degree of support of modeling techniques which they offer is limited, due to the low expressive power of the meta-modeling techniques used. Often they are not capable of adequately capturing the *syntax* of models of modeling techniques, and they are never expressive enough to capture the *semantics* of these models. Worse, many of these meta-modeling techniques do not have a formal semantics themselves. Naturally, this results in poor modeling support. In particular, it is nearly impossible to support validation of models.

A satisfactory CASE shell for all types of modeling techniques seems to be unfeasible, due to the inherent complexity and diversity of such techniques. However, a focus on particular types of modeling techniques may allow for the desired degree of flexibility and expressive power. The categorical framework described in this paper may serve as a basis for a data modeling shell: it is formal, expressive, and highly configurable. In this section, the application of the framework for this purpose is investigated. In particular, a prototype, which is currently under development, is discussed and some related computational complexity problems are addressed. The prototype, TRILOGY (which is being developed in C++, WindowsMAKER, and CodeBase), supports the specification of (1) data modeling techniques (see section 5.1), (2) data models (see section 5.2) and (3) populations (see section 5.3).

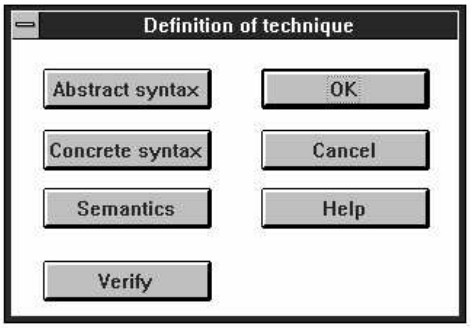


Figure 20: Defining a new data modeling technique

5.1 Describing a data modeling technique

At the top level of TRILOGY, new data modeling techniques may be defined, or existing ones modified. The definition of a new data modeling technique consists of three parts (see figure 20):

1. The definition of the *abstract syntax* of the technique, i.e. its concepts and the rules governing their use.
2. The definition of the *concrete syntax* of the technique, i.e. its representational conventions.
3. The definition of the *semantics* of the technique, which implies a choice for the semantic features required.

The verify option can be used to determine whether there are conflicts between these three parts for a specific data modeling technique. The way these three parts may be specified is discussed in the remainder of this section.

The specification of the abstract syntax of a new data modeling technique boils down to the specification of additional restrictions on type graphs. In some cases these restrictions have already been specified for other data modeling techniques. Suppose for example that a data modeling expert wants to define the syntax of the ER variant of Martin. Many of the rules of the ER variant of Yourdon then apply, hence these rules can be copied (see figure 21). The screen of figure 21 highlights a particular rule for the ER variant of Martin: collection types do not exist. The description field is used for the generation of error messages during schema specification (see section 5.2). In the rule field the formal formulation of the restriction is entered as a LISA-D expression. LISA-D [HPW93] is powerful enough for this purpose as it allows for the specification of arbitrary fixpoint expressions. Such an expression is needed, for example, when cyclic structures in the type graph are to be excluded.

Defining the concrete syntax of a data modeling technique requires the specification of a mapping of its concepts to corresponding (graphical) representations. For example, the mapping may define that entity types are to be represented as squares, relationship types as diamonds, etc. In addition to that, rules have to be specified defining how these concepts may be related.

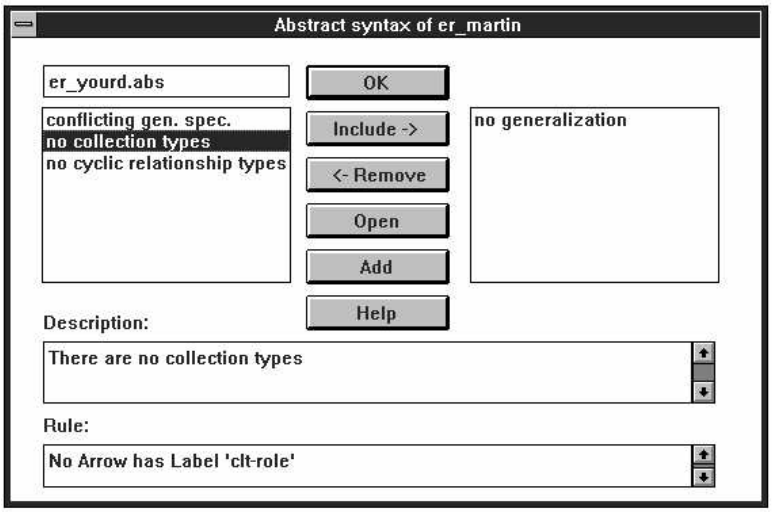


Figure 21: Defining the abstract syntax of a new data modeling technique

For example, the representations of entity types and relationship types should not overlap. In some cases, the specification of such rules requires extra knowledge about the use of these representations in concrete diagrams. If diamonds are used for relationship types, for example, the lines representing the roles may only be connected to the corners of these diamonds. Special locations on graphical symbols, used for defining connections, are often referred to as *handles*.

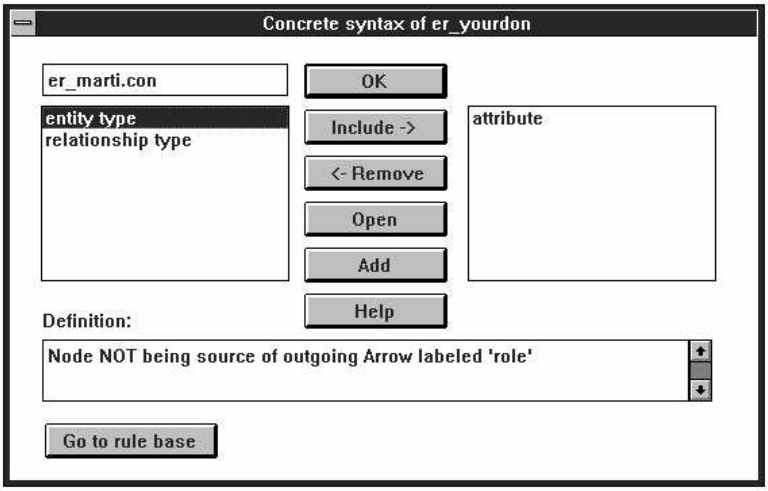


Figure 22: Defining the concrete syntax of a new data modeling technique

In figures 22 and 23, the specification of the representation of data modeling concepts in TRILOGY is illustrated. The screen of the former figure shows an overview of concepts (in this case, some concepts of the ER variant of Martin) and their formal definition (in LISA-D) in terms of the type graph. According to figure 22, an entity type has been defined as a node of a type graph that is not the source of an arrow labeled role. In figure 23, it is shown that entity types are graphically represented as squares. The arrows on the square indicate the relevant

handles (in this case they are used as preferred attachments of lines representing roles). It should be noted that the specification of the rules for graphical representations is only in its initial stage and is not shown. The formal basis of this rule language will be based on work reported in [HVNW92b, HVNW92a].

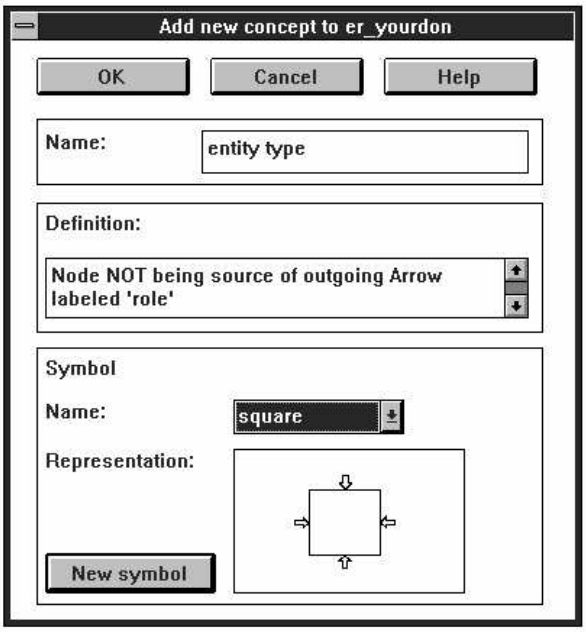


Figure 23: Defining the concrete representation of a concept

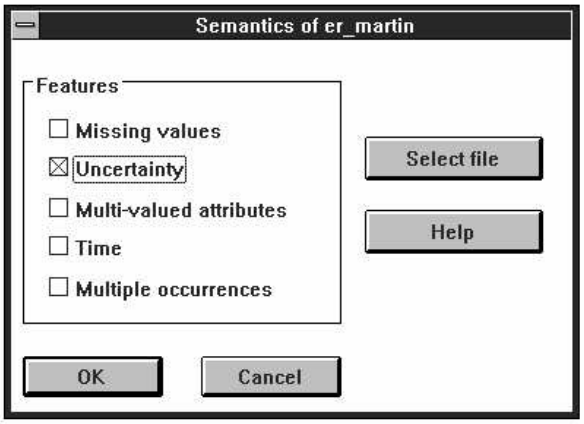


Figure 24: Defining the semantic features of a new data modeling technique

Finally, the semantics of a data modeling technique is defined by the choice of an appropiate instance category. In figure 24, the support for such a choice is shown. Currently, it is not possible to combine different semantic features (such as e.g. time and uncertainty) as it is not yet clear whether constructs in a new instance category can easily be derived from constructs in existing other instance categories. Six instance categories are supported directly: **PartSet**,

FuzzySet, **Rel**, **TimeSet**, **Bag**, and the default category **FinSet**. New instance categories can be defined and incorporated in TRILOGY via the "Select file" option.

5.2 Describing a data model

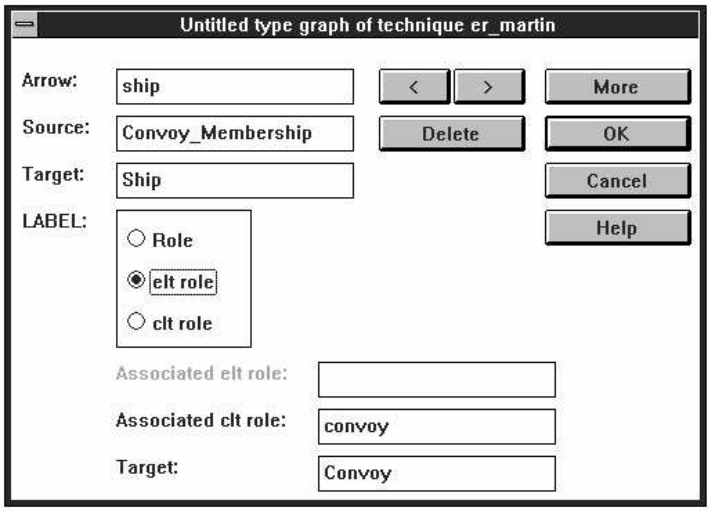


Figure 25: Entering a type graph

In figure 25, it is shown how type graphs are specified in TRILOGY. The definition of a type graph is centered around the arrows (contrary to the formal definition of the type graph, TRILOGY does not allow isolated nodes). For each arrow, the source and the target, as well as its label have to be specified. When the label is of type elt_role, the associated arrow with label clt_role has to be specified (for the bijection clt). As both arrows have identical sources, only the target of the arrow labeled with clt_role has to be specified additionally. When the OK-button is pressed, it is checked whether the type graph satisfies the abstract syntax rules specified for the technique at hand. In this sample case, an error message will be shown as in the ER variant of Martin collection types do not occur (see figure 26).

Clearly, the interface can be improved by allowing type graphs to be entered completely graphically, but currently this has a low priority as emphasis is on the functionality of the tool.

5.3 Populating a data model

Once a data model has been specified according to the conventions of a specific data modeling technique, populations of such a model may be entered. The format of the instances of such a population depends on the underlying instance category. For instances in the category **FinSet** nothing special is required, but for instances in the category **TimeSet**, additional time information is needed (birth and death). Such additional information appears in the "Features" list which is shown when object types are populated (see figure 28, where *Period* is required for objects in **TimeSet**). The "Attributes" list contains the names of the outgoing arrows, which may be thought of as the attributes of the object type. For each object, a number of these

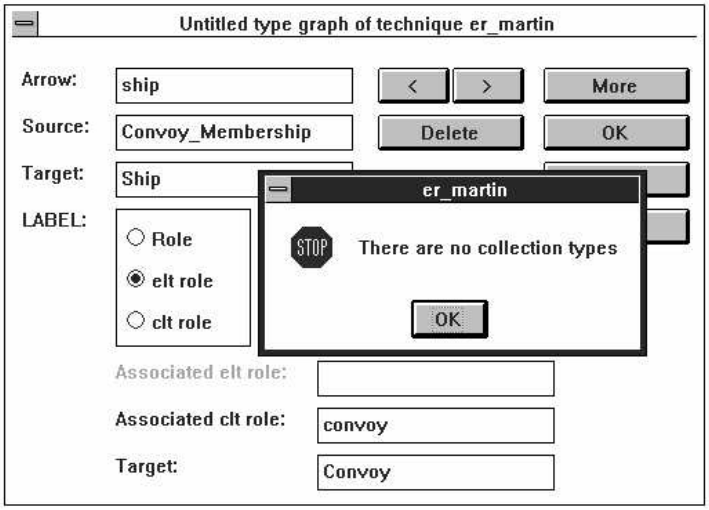


Figure 26: Erroneous type graph

attributes may be added upon creation. In the screen of figure 28, it can be seen that instance *bm_1*, represents the fact that *Mick Jagger* is a member of *The Rolling Stones* from 7/12/62 till now.

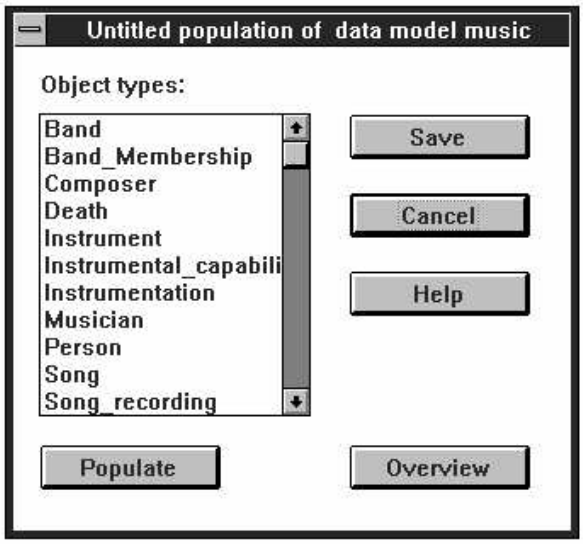


Figure 27: Main screen for entering populations

TRILOGY allows users a complete overview of a population. In figure 29, a population in the instance category **FinSet** of the type graph of figure 2 is shown, while in figure 30 a population of this type graph in **TimeSet** is shown.

It should be noted that from a computational point of view, the validation of a particular population may be quite complex as a result of the categorical formalization of collection types.

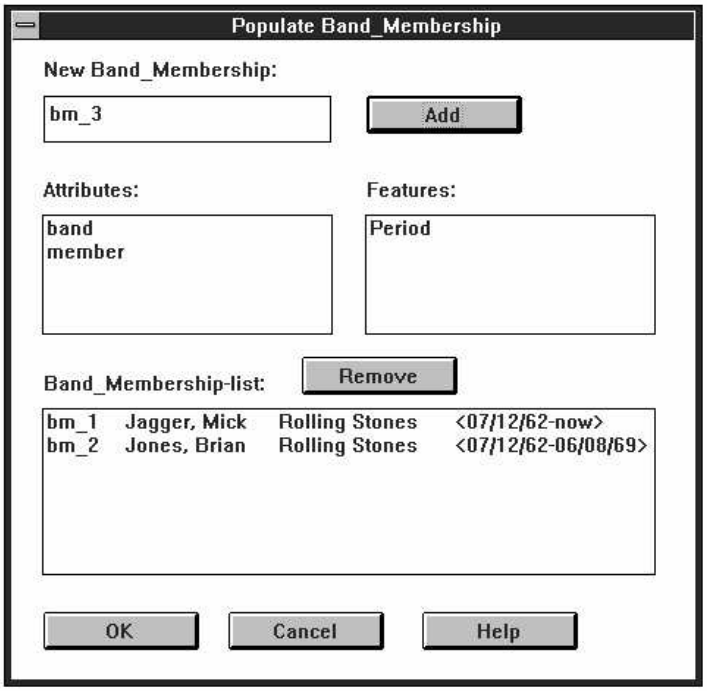


Figure 28: Populating an object type

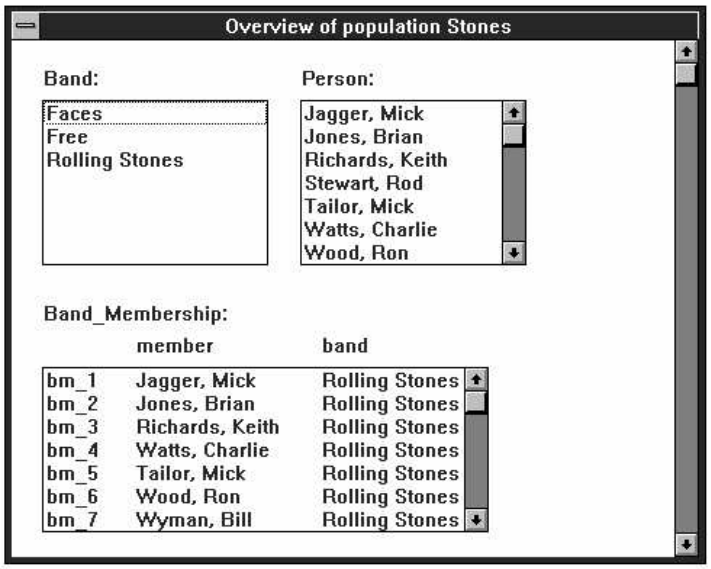


Figure 29: Overview of a population specified in **FinSet**

While this formalization is very general, it also boils down, in set-theoretic terms, to finding all bijections on the set of instances of an element type. Clearly, this makes it an exponential problem. Of course, for particular, often used, instance categories an efficient solution can be implemented.

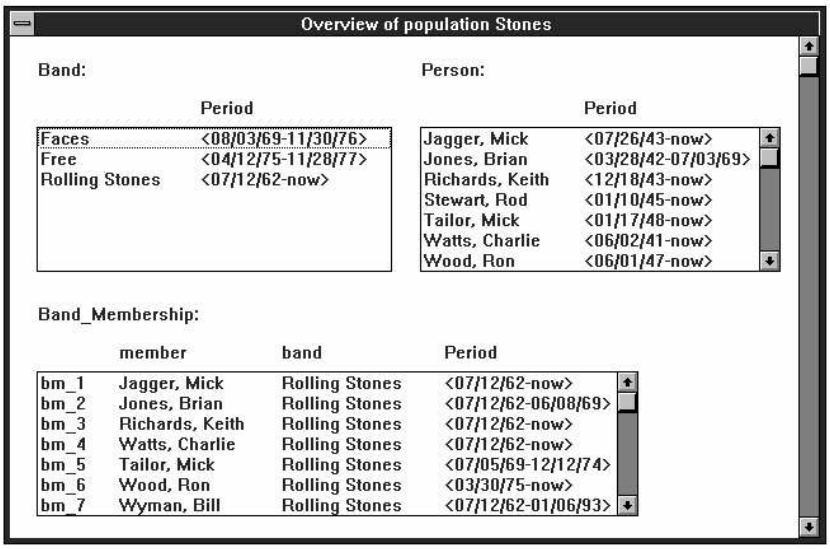


Figure 30: Overview of a population specified in **TimeSet**

Another source of computational complexity is the derivation of the colimit of a subtype diagram, needed in case of generalization. In [RB88], an algorithm, written in the functional programming language ML, is provided for calculating arbitrary (finite) colimits in a category. This reference, however, does not provide explicit complexity considerations and it seems that in general hardly any attention in the categorical literature has been paid to efficient computations of categorical constructs. From a practical point of view, it is obvious that the algorithm in [RB88] can be made more efficient by the use of formal transformation techniques. For example, *memoization* (see e.g. [Par90]), can be applied to remember results of complex computations which are used in several places. In [RB88], function calls such as `f(y)` occur repeatedly, even when `f` is a complex procedure. By adding an assignment `let fy = f(y)` and substituting `f(y)` by `fy` in the algorithm for computing co-equalizers (described on page 89), a significant reduction in computing time can be achieved [2].

5.4 Future developments

Future developments aim at a more graphically oriented user interface and support for transformations to implementation models, such as concrete OO database models and the Relational Model. In addition to that, focus will be on complexity reduction of the computation of categorical constructs, in particular the colimit. It is not yet clear to what extent the high complexity of the validation of the extensionality property and the computation of colimits imposes serious practical limitations. Concrete case studies should provide insight in this matter.

Finally, not many contemporary CASE tools pay attention to the modeling *process*: how should models be constructed? For data modeling techniques, modeling procedures exist that support analysts during information analysis (see e.g. [NH89, Hal95]). The commercially available CASE tool InfoModeler [Asy94], supports such a procedure for the data modeling technique

[2] The authors would like to thank Jacques Sauloy for bringing this to their attention.

FORM. Incorporation of support for the process of data modeling in TRILOGY will also be subject of future research.

6 Conclusions and further research

This paper presents a unifying framework for conceptual data modeling 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 modeling techniques it can be seen as a generalization of them. 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.

As has been shown, the model that is described here subsumes object-oriented data models. The subtypes in our approach are analogous to subclasses. Attributes can be modeled 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 [Tui94]. As complex type constructors are incorporated in the framework, this approach also shows how they may be incorporated in a natural way in OO conceptual data models.

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 modeling techniques, since the mapping of these techniques into the framework automatically defines a semantics for these features.

Concrete applications of the framework can not only be found in its configurable semantics, but also in the field of schema transformations and improved automated modeling support. The framework allows a focus on the essence of schema transformations as it is data model independent. Further, its application may also reveal that some transformations are instance category dependent as they assume the existence of certain semantic features. The tool TRILOGY allows analysts to define their own data modeling techniques. TRILOGY supports validations of data models, as populations may be entered and checked. More research is needed into complexity reductions of computations of categorical constructs needed for population checks. These computations become even more critical when constraints are involved (see [LH94]).

Finally, compared with other approaches that use category theory [Tui94, BSW94] 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 aproaches that are usually limited to topoi or cartesian closed categories.

Acknowledgement

The authors would like to thank Rob Bosman for his assistance in the implementation of the tool described in section 5. The comments and suggestions of the anonymous referees have led to a substantial improvement of the paper. Among others, more emphasis has been put on applications of the framework.

A Category Theory

This section contains the definition of the categorical constructs and notations used in 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 [BW90].

A.1 Basics

This section presents the definitions of the basic concepts of category theory as far as they are important. Most of these definitions are adapted from [BW90].

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

Definition A.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 $f\colon A \to B$ 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 A.2

A category *$\mathcal{C}$ is a directed multigraph whose nodes are called* objects *and whose edges are called* arrows. *For each pair of arrows $f\colon A \to B$ and $g\colon B \to C$ there is an associated arrow $g \circ f\colon A \to C$, 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 $\mathsf{Id}_A\colon A \to A$, the* identity *arrow. If $f\colon A \to B$, then $f \circ \mathsf{Id}_A = f = \mathsf{Id}_B \circ f$.* □

In category theory it is customary to omit the identity arrows in drawings of categories if they do not serve a particular purpose. This convention is adopted throughout this paper. The objects and arrows of a category may have abstract as well as concrete interpretations. 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.

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

Definition A.3

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

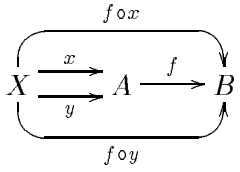


Figure 31: Illustration of the definition of a monomorphism

Figure 31 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 A.4

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

Figure 32 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 $\mathcal{C}$, denoted as $\mathcal{C}^{\mathrm{op}}$, has the same objects as $\mathcal{C}$ and as arrows all arrows of $\mathcal{C}$ inverted, i.e. if $f: A \to B$ is 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. The concept of duality in category theory is very important as it reduces proof obligations: the dual of a theorem is also a theorem.

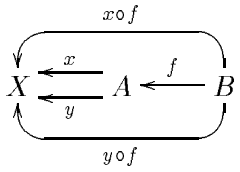


Figure 32: Illustration of the definition of an epimorphism

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 A.5

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

There are also some objects with special properties.

Definition A.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.

A.2 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 A.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 [BW90], illustrates some subtleties involving the concept of diagram.

Example A.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

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

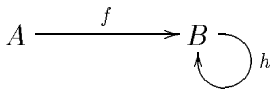
can then be formally defined, using the shape graph $\mathcal{I}$,

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

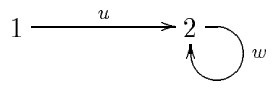
as the homomorphism $D: \mathcal{I} \to \mathcal{G}$ with $D(1) = A$, $D(2) = B$, $D(3) = C$, $D(u) = f$, and $D(v) = g$. The following diagram is just like D (has the same shape) except that v goes to h and 3 goes to B.

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

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



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



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

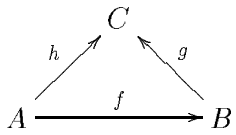
The notion of a *commutative* diagram plays a central role in category theory. Categorical proofs and definitions often use diagrams and prove or require them to commute. Commutative diagrams are the categorist's way of expressing equations.

Definition A.8

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

□

Example A.2 The following diagram commutes if and only if h is the composite $g \circ f$.



□

A.3 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

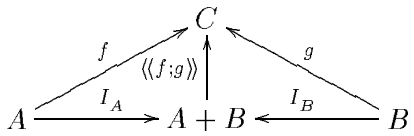
$$A + B = \{\langle a, 0\rangle \mid a \in A\} \cup \{\langle b, 1\rangle \mid b \in B\},$$

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

Definition A.9

A coproduct *of two objects A en 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:

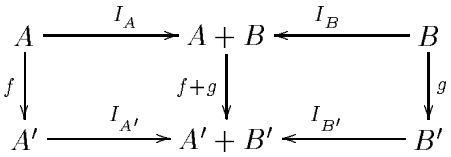


□

The definition of a coproduct can straightforwardly be generalized 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 whether x originates from A or B respectively.

Definition A.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:*



□

Coproducts in the category of sets have special properties they do not have in most other categories. One such property is that coproducts in **Set** are *disjoint* (for a categorical formalization of this notion refer to [BW90]). In a disjoint coproduct the coproduct arrows must be monomorphisms. In a category with disjoint coproducts all monomorphisms are *complementable*:

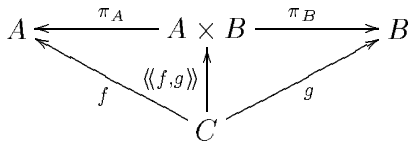
Definition A.11

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 coproduct 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 A.12

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:*

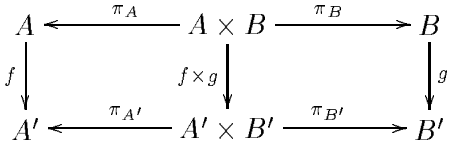


□

As with coproducts, this definition can be straightforwardly extended to arrows.

Definition A.13

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:*



□

A.4 Limits and colimits

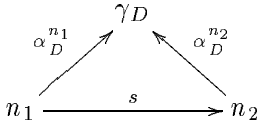
Limits and colimits are dual notions. Both concepts are very general and often used in category theory.

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.

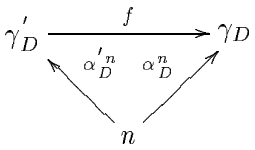
Definition A.14

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 $f: \gamma_D' \to \gamma_D$ such that for each node n of $\mathcal{G}$, the following diagram commutes.*



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

References

[ADM+89] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S.B. Zdonik. The object-oriented database system manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD-89)*, pages 40–57, Kyoto, Japan, 1989. Elsevier Science Publishers.

[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.

[AHS90] J. Adámek, H. Herlich, and G.E. Strecker. *Abstract and Concrete Categories.* Pure and applied mathematics. John Wiley & Sons, New York, New York, 1990.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading, Massachusetts, 1995.

[Ari86] G. Ariav. A Temporally Oriented Data Model. *ACM Transactions on Database Systems*, 11(4):499–527, December 1986.

[AS84] A.D. Atri and D. Sacca. Equivalence and mapping of database schemes. In *Proceedings of the Tenth International Conference on Very Large Data Bases*, pages 187–195, 1984.

[Asy94] Asymetrix. *InfoModeler User Manual.* Asymetrix Corporation, 110-110th Avenue NE, Suite 700, Bellevue, WA 98004, Washington, 1994.

[BBD+89] P. Bergsten, J.A. Bubenko, R. Dahl, M. Gustafsson, and L-Å. Johansson. RAMATIC - a CASE shell for implementation of specific CASE tools. Technical report, SISU, Stockholm, Sweden, 1989. First draft of a contribution to section 4.4 of the TEMPORA T6.1 report.

[BGP92] D. Barbará, H. Garcia-Molina, and D. Porter. The Management of Probabilistic Data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):487–502, October 1992.

[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.

[CF92] D. de Champeaux and P. Faure. A comparative study of object-oriented analysis methods. *Journal of Object-Oriented Programming*, 5(1):21–33, March 1992.

[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.

[CN89] M. Chen and J.F. Nunamaker Jr. MetaPlex: An integrated environment for organization and information systems development. In J.I. DeGross, J.C. Henderson, and B.R. Konsynski, editors, *Proceedings of the Tenth International Conference on Information Systems*, pages 141–151, Boston, Massachusetts, December 1989.

[CSS94] J.F. Costa, A. Sernadas, and C. Sernadas. Object inheritance beyond subtyping. *Acta Informatica*, 31(1):5–26, January 1994.

[Dit90] K.R. Dittrich. Object-oriented database systems: The next miles of the marathon. *Information Systems*, 15(1):161–167, 1990.

[DJM92] C.N.G. Dampney, M.S.J. Johnson, and G.P. Monro. An Illustrated Mathematical Foundation for ERA. In C.M.I. Rattray and R.G. Clark, editors, *The Unified Computation Laboratory*, pages 77–83, Oxford University Press, 1992. The Institute of Mathematics and Its Applications.

[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.

[EJW95] D.W. Embley, R.B. Jackson, and S.N. Woodfield. OO Systems Analysis: Is It or Isn't It? *IEEE Software*, 12(3):19–33, July 1995.

[ES91] H.-D. Ehrich and A. Sernadas. Object concepts and constructions. In G. Saake and A. Sernadas, editors, *Proceedings of the IS-CORE Workshop'91 (Informatik-Berichte 91-03)*, pages 1–24, Braunschweig, Germany, 1991. Technische Universität Braunschweig.

[FSMS91] J. Fiadeiro, C. Sernadas, T. Maibaum, and G. Saake. Proof-theoretic semantics of object-oriented specification constructs. In R. Meersman, W. Kent, and S. Khosla, editors, *Object-oriented databases: analysis, design and construction*, pages 243–284, Amsterdam, The Netherlands, 1991. North-Holland.

[Gog91] J.A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.

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

[Hai91] J.-L. Hainaut. Entity-generating Schema Transformation for Entity-Relationship Models. In *Proceedings of the 10th International Conference on the Entity-Relationship Approach*, Lecture Notes in Computer Science, San Mateo, California, 1991. Springer-Verlag.

[Hal91] T.A. Halpin. A Fact-Oriented Approach to Schema Transformation. In B. Thalheim, J. Demetrovics, and H.-D. Gerhardt, editors, *MFDBS 91*, volume 495 of *Lecture Notes in Computer Science*, pages 342–356, Rostock, Germany, 1991. Springer-Verlag.

[Hal95] T.A. Halpin. *Conceptual Schema and Relational Database Design*. Prentice-Hall, Sydney, Australia, 2nd edition, 1995.

[HK87] R. Hull and R. King. Semantic Database Modelling: Survey, Applications and Research Issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[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.

[HP95] T.A. Halpin and H.A. Proper. Subtyping and Polymorphism in Object-Role Modelling. *Data & Knowledge Engineering*, 15:251–281, 1995.

[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, October 1993.

[Hul86] R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal of Computing*, 15(3):856–886, 1986.

[HVNW92a] A.H.M. ter Hofstede, T.F. Verhoef, E.R. Nieuwland, and G.M. Wijers. Integrated Specification of Method and Graphic Knowledge. In *Proceedings of the Fourth International Conference on Software Engineering and Knowledge Engineering*, pages 307–316, Capri, Italy, June 1992. IEEE Computer Society Press.

[HVNW92b] A.H.M. ter Hofstede, T.F. Verhoef, E.R. Nieuwland, and G.M. Wijers. Specification of Graphic Conventions in Methods. In B. Theodoulidis and A. Sutcliffe, editors, *Proceedings of the Third Workshop on the Next Generation of CASE Tools*, pages 185–215, Manchester, United Kingdom, May 1992.

[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.

[Iiv95] J. Iivari. Object-orientation as structural, functional and behavioural modelling: a comparison of six methods for object-oriented analysis. *Information and Software Technology*, 37(3):155–163, 1995.

[IP94] A. Islam and W. Phoa. Category Models of Relational Databases I: Fibrational Formulation, Schema Integration. In M. Hagiya and J.C. Mitchell, editors, *Theoretical Aspects of Computer Software, International Symposium TACS'94*, volume 789 of *Lecture Notes in Computer Science*, pages 618–641, Sendai, Japan, April 1994. Springer-Verlag.

[Jac93] I. Jacobson. Is Object Technology Software's Industrial Platform? *IEEE Software*, pages 24–30, January 1993.

[JNS83] S. Jajodia, P.E. Ng, and F.N. Springsteel. The problem of equivalence for entity-relationship diagrams. *IEEE Transactions on Software Engineering*, 9(5):617–630, 1983.

[Kim91] W. Kim. Object-oriented database systems: strengths and weaknesses. *Journal of Object-Oriented Programming*, pages 21–29, July/August 1991.

[Kin89] R. King. My Cat is Object-Oriented. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Frontier Series, pages 23–30. Addison-Wesley, Reading, Massachusetts, 1989.

[KL89] W. Kim and F.H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications.* ACM Press, Frontier Series. Addison-Wesley, Reading, Massachusetts, 1989.

[Kob86] I. Kobayashi. Losslessness and semantic correctness of database schema transformation: another look of schema equivalence. *Information Systems*, 11(1):41–59, 1986.

[KS94] Y. Kornatzky and S.E. Shimony. A probabilistic object-oriented data model. *Data & Knowledge Engineering*, 12(2):143–166, 1994.

[LH94] E. Lippe and A.H.M. ter Hofstede. A Category Theory Approach to Conceptual Data Modeling. Technical Report CSI-R9415, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, December 1994. Accepted for publication in *RAIRO Theoretical Informatics and Applications*.

[Mor90] J.M. Morrissey. Imprecise Information and Uncertainty in Information Systems. *ACM Transactions on Information Systems*, 8(2):159–180, April 1990.

[MS95] R. Motschnig-Pitrik and V.C. Storey. Modelling of set membership: The notion and the issues. *Data & Knowledge Engineering*, 16(2):147–185, August 1995.

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

[Par90] H. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development.* Springer-Verlag, Berlin, Germany, 1990.

[PM88] J. Peckham and F. Maryanski. Semantic Data Models. *ACM Computing Surveys*, 20(3):153–189, September 1988.

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

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[SA86] R. Snodgrass and I. Ahn. Temporal Databases. *IEEE Computer*, 19(9):35–42, 1986.

[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, University of Twente, Enschede, The Netherlands, 1990.

[SLTM91] K. Smolander, K. Lyytinen, V-P. Tahvanainen, and P. Marttiin. MetaEdit: A Flexible Graphical Environment for Methodology Modelling. In R. Andersen, J.A. Bubenko, and A. Sølvberg, editors, *Proceedings of the Third International Conference CAiSE'91 on Advanced Information Systems Engineering*, volume 498 of *Lecture Notes in Computer Science*, pages 168–193, Trondheim, Norway, May 1991. Springer-Verlag.

[Sno90] R. Snodgrass. Temporal Databases Status and Research Directions. *SIGMOD Record*, 19(4):83–89, December 1990.

[STM88] P.G. Sorenson, J.-P. Tremblay, and A.J. McAllister. The Metaview System for Many Specification Environments. *IEEE Software*, 5(2):30–38, March 1988.

[TLW91] C. Theodoulidis, P. Loucopoulos, and B. Wangler. A conceptual modelling formalism for temporal database applications. *Information Systems*, 16(4):401–416, 1991.

[TS93] M. Tresch and M.H. Scholl. Schema Transformation without Database Reorganization. *SIGMOD Record*, 22(1), March 1993.

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

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

[Ver93] T.F. Verhoef. *Effective Information Modelling Support.* PhD thesis, Delft University of Technology, Delft, The Netherlands, 1993.

[VJT92] I. Vessey, S.L. Jarvenpaa, and N. Tractinsky. Evaluation of Vendor Products: CASE Tools as Methodology Companions. *Communications of the ACM*, 35(4):90–105, April 1992.

[WD90] G.M. Wijers and H.E. van Dort. Experiences with the use of CASE-tools in the Netherlands. In B. Steinholz, A. Sølvberg, and L. Bergman, editors, *Proceedings of the Second Nordic Conference CAiSE'90 on Advanced Information Systems Engineering*, volume 436 of *Lecture Notes in Computer Science*, pages 5–20, Stockholm, Sweden, May 1990. Springer-Verlag.

[Wij91] G.M. Wijers. *Modelling Support in Information Systems Development.* PhD thesis, Delft University of Technology, Delft, The Netherlands, 1991.

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

[ZM90] S.B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, San Mateo, California, 1990.

Contents