Comparison of Categorical Foundations of Object-Oriented Database Model

Tomáš Hruška and Petr Kolenčík*

Department of Computer Science and Engineering
Faculty of Electrical Engineering and Computer Science
Technical University of Brno
Božetěchova 2, 612 66 Brno, Czech Republic
{hruska,kolencik}@dcse.fee.vutbr.cz

Abstract. The present development in the database area is highly influenced by the object-oriented principles of data modeling. On the contrary to the previously successful relational approach, it lacks rigorous theoretical support. This problem is mainly due to the fact that the semantics description of such features as generalization, specialization, encapsulation, and inheritance is not trivial by using just the means of the set theory. Therefore, many researchers focused on finding suitable theoretical means for the modeling of complex objects. This paper is concerned with the comparison of categorical modeling in the database area and proposes a limit data model that enables to represent the objects, relationships, multiple inheritance, class polymorphism and virtual methods in the categorical setting. The aim to formalize these notions arose from the actual need. We would like to use it to support the development of an object-oriented database model closely following the standard ODMG'93. This model proposes several useful features to the database modeling concerning the object hierarchies and modeling roles [13] that require a specific approach and it seems that the category theoretical framework fulfills these objectives.

1 Introduction

Recently, database researchers stressed the need to design an unified object-oriented database model. There have been proposed the standard [2] called ODMG'93 which gives an informal description of data definition and manipulation and defines bindings to C++ and Smalltalk (this paper follows the revised release 1.2). But on the contrary to the relational database model, based on the set theory, the object-oriented approach lacked rigorous mathematical foundations supporting formal semantics of data structure definition and manipulation. As a result, researchers proposed several frameworks defining the model-theoretic semantics of the object-oriented paradigm.

Category theory (CT) seems to be an useful device to express these concepts in an understandable and unified way. Diskin and Goguen in their manifests [4, 9] pointed out that CT is a formalism supporting a high level of abstraction that helps to simplify

* Supported by the grant of the Czech Grant Agency No. 102/96/0986 Object-Oriented Database Model and by the grant of the Czech Ministry of Education No. 0630 Modeling of Inheritance in the Object-Oriented Database Model.

proofs and gives a new understanding of studied problems. Diskin emphasized the fact, that objects are considered homogeneous and all the information about the structure of classes is defined by using morphisms. Therefore, in contrast to the set theory and first order logic, it provides devices more suitable for the description of complex objects, their properties and relationships.

The attempts to give formal semantics to the object-oriented model came from different directions. The formal specification of the semantics of the extended entity-relationship model can be found in [8]. However, the set theory used there seems to be too flat to describe the rich structures and high-level semantics of the object-oriented model in a simple and clear way. Based on the results of the type-theory, Bertino and Martino provided the Chimera object-oriented database model [11] with formal foundations and typing. In [14], a novel logic was presented claiming capability to model all the aspects of the object-oriented paradigm. Another promising approach is presented in [10]. It uses an algebraic specification and emphasizes proofs rather than models. The research done in these directions gives us the possibility to compare the set theoretic, logic-based, algebraic and categorical approach. CT can be actually used in any of these approaches with the advantage of objectiveness and high abstraction that helps to describe the object-oriented paradigm in a clear and unified way.

In this paper, we will build on the results of the categorical approach to data specifications as presented in [5], [7], [17], [18], [19] and [21]. In these works, classes and database objects are basically represented as categorical objects, operations as arrows, generalization and specialization constructions as limits. For our purpose , we will use just the basic notions of CT as presented in [1].

The paper is organized as follows: The next section reviews the current research in the area of the object-oriented modeling based on CT. We discuss advantages and disadvantages of the existing models. Section 3 compares these models with a presentation of limit data model. Section 4 describes, how to define the semantics of class polymorphism. Section 5 deals with relationships in the object-oriented model satisfying required integrity constraints.

2 Existing Categorical Models

In this section, we briefly introduce the problem of data specification by using the theoretical means of CT. Particularly, distinctions, advantages and disadvantages of current approaches are discussed with respect to the problem domain they were intended for. Our final objective is to extend the existing categorical framework to capture some new features and use it to provide the formal semantics for a database model similar to that one proposed in the standard ODMG'93 [2].

2.1 Typegraph Model

In the typegraph model of [21], the database schema is represented by a small category and the class hierarchy and data manipulation are modeled by using *product*, *equalizer*, *pullback* and *pushout* constructions. The category is not defined directly, but generated by the typegraph, and the instances are functors from this category to an extension category. The category **Set** (of sets and maps between them) is used only as one possibility to model the actual values stored in the database (this approach is followed also in [5] and [17]).

Data are represented in the typegraph model in a similar way as in [5]. The type hierarchy is built on so called *simple types* which are the nodes of a directed acyclic graph that represents the inheritance hierarchy. The instances of these types (or classes) in the typegraph are sets. The notion of a subclass is modeled as an arrow and can be interpreted as an injective mapping between the instance sets of the types. The attributes of classes are modeled again by using arrows from classes to other simple types. Tuijn demonstrated this approach to data modeling by using an example shown in Fig. 1. This example will help us to understand the formal definition of the simple typegraph.

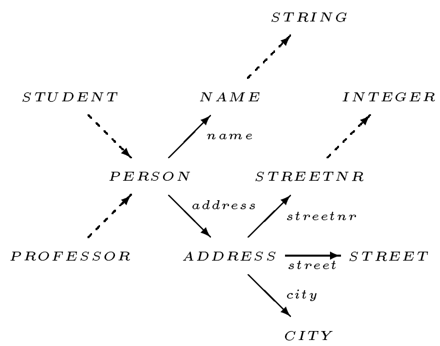


Fig. 1. Example of inheritance hierarchy

Fig. 1 shows the formal definition of a database storing information about persons, students and professors. Notice the distinction of arrows. The solid ones represent the attributes of classes while the dashed ones represent the subclass relationships. For example, $STUDENT$ and $PROFESSOR$ are inherited from the type $PERSON$. Similarly, $NAME$ is derived from $STRING$ and $STREETNR$ is a subtype of $INTEGER$. However, the arrow denoted as $name$ models an attribute of $PERSON$ and this property is inherited to $STUDENT$ and $PROFESSOR$. The inheritance is modeled as a composition of arrows in the typegraph. Formally, the typegraph is defined as follows:

Definition 1 (**Typegraph**) A simple typegraph is a quadruple $\mathcal{G} = (\mathcal{S}, \mathcal{H}, \mathcal{A}, \mathcal{P})$ where

1. $\mathcal{S}$ is the set of simple types representing the nodes of the typegraph;
2. $\mathcal{H} \subseteq \mathcal{S} \times \mathcal{S}$ is a partial order which defines the hierarchy of the simple types in $\mathcal{S}$;
3. $\mathcal{A}$ is the set of attribute labels;
4. $\mathcal{P} : \mathcal{S} \to \mathbf{2}^{\mathcal{A} \times \mathcal{S}}$ associates to each simple type a number of attributes together with their types. It satisfies $\forall s_1, s_2 \in \mathcal{S} : s_1 \neq s_2 \Rightarrow \pi_{\mathcal{A}}(\mathcal{P}(s_1)) \cap \pi_{\mathcal{A}}(\mathcal{P}(s_2)) = \emptyset$, where $\pi_{\mathcal{A}}(\mathcal{P}(s))$ is defined as $\{a \in \mathcal{A} \mid \exists s' \in \mathcal{S} : (a, s') \in \mathcal{P}(s)\}$. $\square$

The simple typegraph is used to generate a category. Its objects are the nodes of $\mathcal{G}$ and the arrows are defined by $\mathcal{H}$ and $\mathcal{P}$. Tuijn denotes this category as $\mathcal{G}^*$ because it represents a closure of $\mathcal{G}$ with respect to the composition of arrows, which is a concept well known in CT. However, the distinction of the dashed and solid arrows is not preserved in $\mathcal{G}^*$. Arrows are generated by the subclass and attribute edges $\mathcal{H}$ and $\mathcal{P}$, respectively, without distinction. We will give a deeper insight to it in Sect. 3.3 and 3.4. According to the object-oriented modeling, the arrows generated by $\mathcal{H}$ correspond to *inheritance* and these generated by $\mathcal{P}$ represent *aggregation*. Tuijn used terms *simple inheritance* and *specification* for these features. For example, $name$ is inherited from $PERSON$ to $STUDENT$ and $PROFESSOR$ following the subclass arrows. However, $street$ is a property of $ADDRESS$ and $address$ itself is a property of $PERSON$. Therefore, a morphism $street \circ address$ should be added to the specification as a property of $PERSON$. Tuijn pointed out, that in other formalisms and most programming languages a slightly different notation is used: $s.address.street$ instead of $street \circ address(s)$.

In this setting, a query mechanism is proposed as an algebra. The semantics of its key operator (the limit construction) is defined in a declarative way. Tuijn proved, that all the operations of the relational algebra, the powerset, nest and unnest operators of nested data model, and the transitive closure can be expressed in the typegraph language.

In addition to the data specification and manipulation by using the limit constructions, Tuijn was concerned with the fundamental characterization of object identity and inheritance in a general context (independently on some specific model). He pointed out, that the object identity is closely connected to the problem of decomposition of views while the subclassing and inheritance has a lot in common with an updatability of views. This approach brings the possibility of reasoning about the database schema on a high level of abstraction. However, it is built on the view-based data modeling which does not fit the context of the model proposed in the standard ODMG'93. Similar direction leading to an interesting abstraction of inheritance can be found in [3] and [7]. Their approach is more suitable for our purpose.

We can summarize, that this work focuses on the structural character of object-oriented paradigm but does not consider the dynamic aspects. It also does not deal with the other properties such as the data encapsulation, static and dynamic constraint checking and event handling.

2.2 Theory Based and Process Based Approach

The research presented in [7] defines formal semantics for the basic constructs of an object-oriented specification language. They focus on the information system development from the formal specification. Therefore, they use theory presentations in a logic to describe properties and behavior of the objects while the means of CT serve to express the relationships in the inheritance hierarchy. The aggregation, inheritance and particularization are formalized as specification constructs acting on a diagram in the category of theory presentations.

According to their former research, the logic used to formalize the specification building should include a category of signatures, a functor from this category to the category of sets and relations defining consequences for each set of formulas. Then, a theory presentation for such a logic consists of a pair (Θ, Φ) where Θ is a signature and Φ is a collection of formulas for that signature usually referred to as the *axioms* of the specification. The theory presented by (Θ, Φ) consists just of the set of formulas which are consequences of Φ (its theorems). Presentations are not closed under the consequence and, therefore, are generally finite and can be used for specification.

Similarly in [3], the authors use the same categorical framework in which theory presentations are substituted by the process models. The inheritance is modeled by using the limit construction in the same manner in both works. It illustrates the strong unifying property of CT enabling to find interesting generalizations and connections between different aspects of the object-oriented modeling.

Here, we will show just the approach to the multiple inheritance modeling because the core of the model is based on the similar ideas as the typegraph model. In the typegraph model the notion of the multiple inheritance was missing so we naturally move to the features that are more difficult to formalize.. According to [3], the multiple inheritance is viewed as a pushout in the category of objects with partial morphisms (PMOb). For example, an account for saving a foreign currency (***sfc_account***) can be obtained by derivation from a saving account (***s_account***) and a foreign currency account (***fc_account***). These two accounts were both specialization of account (***account***). To express this structure we can use the notion of pushout as shown in Fig. 2.

$$\begin{tikzcd}
 & \textbf{\textit{account}} \arrow[dl] \arrow[dr] & \\
\textbf{\textit{s_account}} \arrow[dr] & & \textbf{\textit{fc_account}} \arrow[dl] \\
 & \textbf{\textit{sfc_account}} &
\end{tikzcd}$$

Fig. 2. Illustration of multiple inheritance as a pushout in PMOb.

The pushout is a kind of limit construction. In the category **Set**, the object ***sfc_account*** would be the union of ***s_account*** and ***fc_account***, if two corresponding arrows in the bottom part of the diamond were inclusions. If two arrows in the upper part of the diamond were also set inclusions then the object ***account*** would be the intersection of ***a_account*** and ***fc_account***. This construction can be easily extended to general pushouts consisting of more base objects. However, the objects in this category are not interpreted as sets but as so called *markovian objects* consisting of a process and markovian observation structure (markovian means that update functions for objects are independent on their states). The arrows are partial morphisms between the processes.

Fig. 2 defines the following semantics: The markovian process and observation structure of ***account*** can be mapped by using the partial morphism to ***s_account*** with the meaning that ***s_account*** defines the same object as ***account*** and extends it with some additional functionality. In such a way, the partial morphisms enable to add new operations to ***s_account*** that work also with the properties of ***account***. Similarly, ***fc_account*** adds some new functionality sharing the properties of ***account*** with the object ***s_account***. The two arrows in the upper part of inheritance diamond are crucial for the definition of ***sfc_account*** since they identify the properties which the objects ***s_account*** and ***fc_account*** share.

For our purpose, we will use the same patterns of diagrams. However, we will focus on the data structure modeling which better expresses the requirements of the standard ODMG'93. Accordingly, we will work with a category whose objects will represent classes and whose arrows will model attributes, methods, multiple inheritance and relationships. In this setting, we will use the limit constructions to define some additional constrains on the database schema. These constraints will enable to represent the virtual base classes and class polymorphism.

2.3 Graph-Based Semantic Modeling

In [6], Diskin and Cadish proposed a graph-based specification language for the semantic modeling. The main idea was to generalize the relational data model to capture the semantics of object-oriented features. Moreover, the graph-based specification language at the same time possesses a precise semantics. They were inspired by the property of CT which replaces the analytical assertions about elements by the synthetic assertions about diagrams. In their approach, nodes in the diagrams model classes and arrows represent their attributes, subclassing and relationships. The queries are viewed as operations on such diagrams.

Their objective was to provide any kind of graphical schemes with a corresponding logic so that any schema itself can be considered as a theory of the logic. This idea, developed in CT through the concept of a sketch, has a lot in common with the previous models. In addition, they clarified the connection of relational approach and first order logic with a sketch-based framework and extended this kind of reasoning with a rich semantics.

In this context, Diskin and Cadish provided mathematically correct semantics for the notions of object identity, relationships (with distinguishing arbitrary associations from aggregations), specialization and generalization and weak entity type stressing the fact that this approach is universal with respect to simulation of any other formal data specification. On this ground they suggested a new approach to view integration.

The categorical devices used for modeling generalization and specialization are the same as in the typegraph and theory based models. Their aim was to solve the problems of view integration by using the diagrams, sketches and other means of CT. Therefore, they did not considered the behavioral aspects of the object-oriented models neither the multiple inheritance, naming problem, class polymorphism nor method dispatching. However, their work greatly influenced our understanding of the problem of conceptual modeling.

2.4 Summary

The works described in the previous subsections formalize the basic aspects of the object-oriented paradigm as the object identification, inheritance, aggregation and relationships. They view these features from different perspectives, namely emphasizing the data structure and conceptual modeling [5, 6, 18], queries and views [21] and behavior [3, 7].

These approaches are useful for various purposes of integrating the entity-relational and object-oriented framework, addressing the problem of view integration, exploring the modeling power of the categorical models, development of provably correct systems or just formalizing the object-oriented notions in a new way.

The issues which are not covered by these works include the class polymorphism, naming problem, integrity constraints for relationships, internal structure of the database, behavior modeling (in the sense of the object-oriented programming), virtual methods, method dispatching and bulk data types which are all important features of the model defined by the standard ODMG'93.

3 The Limit Data Model

With respect to the results presented in the previous section, we proposed *limit data model* (LDM). The original ideas were developed in [12, 15, 16]. Here, we summarize the main results focusing on the *multiple inheritance*, *conflicts* in attributes and *class polymorphism* which are problems that have not been formally addressed yet using CT from the point of the data structure perspective. We will not deal with the modeling of the methods and bulk data types here. The methods are dealt separately in [16] and the bulk data types are being currently investigated. We will close the paper demonstrating how the *relationships* can be represented in LDM since they are modeled distinctly from the previous works and, we hope, fit nicely the context of ODMG'93.

3.1 Database Intention and Extension

The object-oriented modeling supposes a hierarchy of classes that are built on basic domains by using the *aggregation* and *inheritance* constructions. Such a definition of the database structure can be formally viewed as a *category* consisting of *objects* and *arrows* with additional *commutativity conditions* in accordance with [3, 5, 7] and [21]. We use the term arrow and morphism interchangeably in this paper. The category used to model the database *intention*, i.e. the classes in the inheritance hierarchy, will be denoted as $\mathcal{D}_I$. Objects of $\mathcal{D}_I$ correspond to the *classes* and arrows with additional conditions represent *aggregation*, *inheritance*, *relationships* and *methods*. The category $\mathcal{D}_I$ defines the syntax of the model; the subscript I stands for intention.

Concerning the semantics, we will follow the approach of [5] and [21] giving interpretation of the categorical constructions in the category **Set** . We used LDM to give semantics to the multiple class objects which can acquire and discard types during their lifetime [13]. The means of CT help to reason about the data structures and possible conflicts arising from the multiple class membership.

While $\mathcal{D}_I$ represents the *structure* of the database, specific *objects* (tuples with complex structure), derived from $\mathcal{D}_I$, are formalized as a *model* of the database intention in **Set**. In CT, it is formally expressed as a *functor* $D : \mathcal{D}_I \to \mathcal{D}_E$ representing the *content* of the database or the database *extension*.

D defines appropriate mappings from the classes to their extents, object identifiers (*oids*) to e.g. natural numbers, arrows to methods etc. The commutative conditions in $\mathcal{D}_I$ are required to be preserved also in $\mathcal{D}_E$. Any transaction consisting of a method invocation or operation *update*, *new* or *delete* is expressed as a redefinition of D according to the results given by a function in $\mathcal{D}_E$ used to implement the operation defined in $\mathcal{D}_I$. Thus, we can introduce the notion of persistence into the inherently functional language of CT.

Modeling of $\mathcal{D}_I$ in **Set** is only one possible interpretation of the database schema. In Sect. 2.2, we have seen another interpretation in the category whose objects represented the markovian processes and observation structures and whose arrows were morphisms between these structures. Comparing these approaches, we get universal models which can be applied to the different aspects of the object-oriented paradigm. However, we have not considered this kind of interpretation for our model because we intend to use it for different purpose.

3.2 Building Complex Objects

Considering the interpretation of $\mathcal{D}_I$ in **Set**, aggregated objects can be built by using the *product* construction as have been already pointed out in [21]. Similar approach is described in [5]. To illustrate, let us consider a class $Address$ composed of attributes $street$, $city$ and $postcode$. $Address$ itself is defined in the left part of Fig. 3 as a product $Address = \mathtt{string} \times \mathtt{string} \times \mathtt{string}$. To include an object of the type $Address$ into $Person$, one can simply define projection $\pi^{Person}_{addr} : Person \to Address$ in the diagram in Fig. 3.b. In this paper, the projections are denoted as π. The subscript distinguishes which class is the projection defined for and the superscript informs about the name of the attribute that is referred to.

$$\begin{tikzcd}
\mathtt{string} & \mathtt{string} \\
\boldsymbol{Address} \arrow[u, "\pi^{Address}_{street}"] \arrow[ur, "\pi^{Address}_{city}"'] \arrow[dr, "\pi^{Address}_{postcode}"'] & \\
& \mathtt{string}
\end{tikzcd}$$

(a) Define ***Address***

$$\begin{tikzcd}
& \boldsymbol{Address} \\
\boldsymbol{Person} \arrow[ur, "\pi^{Person}_{addr}"] \arrow[rr, "\pi^{Person}_{name}"'] & & \mathtt{string}
\end{tikzcd}$$

(b) Aggregate ***Address*** in ***Person***

Fig. 3. Building aggregated objects

The semantics of the aggregation itself is defined as follows: an object of the type $Address$ is aggregated in $Person$. $Address$ is stored as a part of $Person$ lacking its own *object identifier* (*oid*); i.e. it can not be referred from the outside of the object $Person$.

The product construction captures this semantics in the following way. Interpreting the diagrams in Fig. 3 in **Set**, the actual values of the attributes $street$, $city$ and $postcode$ are stored within the $Person$ tuple. However, the value of these attributes can not be shared and can be accessed only through the $Person$ by using the composition of projections. For instance, the attribute `Person.addr.street` is accessed by using the projection $\pi^{Address}_{street} \circ \pi^{Person}_{addr}(Person)$.

In CT, the notation is slightly different from that one often used in the object-oriented programming languages as pointed out in [21]. Instead of the point, we use a small circle and write the projections in a reverse order. To help us to see the associated constructions of programming languages, we write the corresponding class in the subscript and the referred attribute in the superscript. To illustrate, the projection $\pi^{Person}_{addr}(Person)$ will denote access to the attribute $addr$ of the class $Person$ which is usually written as `Person.addr` in the programming languages.

3.3 Modeling Classes

In essence, the inheritance is quite similar to the aggregation. It defines objects of the same structure. These are tuples containing all the attributes of some existing class with an additional structure and functionality. However, the aggregation requires to specify the whole path to access the attributes of some enclosed object while inheritance uses a flat *naming schema* to refer the attributes which are brought from the superclasses.

Therefore, the notion of the inheritance can be formalized similarly to the aggregation with the exception that we have to solve the problem of attribute access for the derived classes. The attributes brought from the superclasses can be accessed again by using the composition of projections. The arrow composition is an essential property of CT and according to [1] we use the notation $\pi^B \circ \pi^A(a)$ for a composed projection $\pi^B(\pi^A(a))$. In addition to the aggregation, we will introduce a specific labeling assigning meaningful names to the (composed) attribute projections. The labeling is essential to model the inheritance and will describe the flat naming space in each derived class.

To illustrate, we use the labeling π^{Class}_{name} for the projection accessing the attribute $name$ in $Class$. The projection that defines the relationship to the superclass will be denoted as $\pi_{Class:Super}$. The labeling is designed with a special purpose. The classes defined by the multiple inheritance can potentially cause disambiguities and the notation for the subscripts will be used to resolve these conflicts. For instance, the labeling $\pi^{Class}_{Super::name}$ will distinguish the unique attribute $name$ of $Class$ inherited from $Super$. Before giving a formal definition of the labeling algorithm we will give a deeper insight to the issues caused by the multiple inheritance by using a practical illustration.

Consider a simple university database storing information about persons, students, lecturers and teaching assistants. Let us suppose that we are interested in the person's names and, in addition, it is required to distinguish the students and lecturers by using specific codes. We would also like to know who are the supervisors of the individual teaching assistants. In LDM, this definition will be represented by the diagram in $\mathcal{D}_I$ as shown in Fig. 4.

According to the previous subsection, we recall that $Person$ is defined as a product. It is shown in Fig. 4.a (for simplicity the address attribute is omitted and P^p is used as an abbreviation of $Person$). Having done this, we can start building the derived classes for students, lectures and teaching assistants as presented in the diagram in Fig. 4.b. Yet, this diamond does not add any new property to the classes till the final diagram is composed by using the projections π^S_{code}, π^L_{code} and π^{TA}_{sup}. These projections define the new attributes in the derived classes. Later, these new properties of the classes will be denoted simply as S^+, L^+, and TA^+, as a whole.
Interpreting this categorical definition in **Set**, we have:

$$\begin{tikzcd} & \texttt{string} \\ \boldsymbol{P}^p \arrow[ur, "\pi^P_{name}"'] & \end{tikzcd}$$

(a) The definition of ***P***

$$\begin{tikzcd} & \boldsymbol{P}^p & \\ \boldsymbol{S}^p \arrow[ur, "\pi_{S:P}"] & & \boldsymbol{L}^p \arrow[ul, "\pi_{L:P}"'] \\ & \boldsymbol{TA}^p \arrow[ul, "\pi_{TA:S}"] \arrow[ur, "\pi_{TA:L}"'] & \end{tikzcd}$$

(b) Diamond of multiple inheritance

$$\begin{tikzcd} & & & \texttt{string} & \\ \texttt{int} & & \boldsymbol{P}^p \arrow[ur, "\pi^P_{name}"'] & & \texttt{int} \\ & \boldsymbol{S}^p \arrow[ul, "\pi^S_{code}"] \arrow[ur, "\pi_{S:P}"] & & \boldsymbol{L}^p \arrow[ul, "\pi_{L:P}"'] \arrow[ur, "\pi^L_{code}"'] & \\ & & \boldsymbol{TA}^p \arrow[ul, "\pi_{TA:S}"] \arrow[ur, "\pi_{TA:L}"'] \arrow[rr, "\pi^{TA}_{sup}"'] & & \boldsymbol{L}_{oid} \end{tikzcd}$$

(c) Adding new properties

Fig. 4. Small university database

$$P^p = \texttt{string} : \{name \mid name \in \texttt{string}\},$$
$$S^p = P^p \times \texttt{int} : \{(name, code) \mid name \in \texttt{string}, code \in \texttt{int}\} \text{ and}$$
$$L^p = P^p \times \texttt{int} : \{(name, code) \mid name \in \texttt{string}, code \in \texttt{int}\},$$

both inheriting from the class P. The class teaching assistant of the structure

$$TA^p = L^p \times S^p \times L_{oid} : \{(name, L{::}code, S{::}code, sup) \mid name \in \texttt{string}, \\ L{::}code, S{::}code \in \texttt{int}, sup \in L_{oid}\}$$

inherits from L^p and S^p adding the attribute sup. Notice please that the object of the type L, representing the supervisor, is referred in the class TA by using its *oid*. Of course, this semantics is better described using relationships as shown in [12] which will be discussed later (see Sect. 5).

Concerning the notation, the superscript p in $Class^p$ indicates that this categorical object represents purely the database objects belonging to $Class$ and not to its subclasses. We call the classes denoted as $Class^p$ *pure classes*. However, we should note here that it breaks the requirements of the principle of class polymorphism and we will show how to deal with this problem in Sect. 4.

Notice, that the structure of S^p, L^p and TA^p can not be directly obtained by using just the means of the product construction as shown in the equations even if this structure naturally expresses the supposed semantics of this example. To get it correctly, we need to impose some additional commutativity conditions on the diagrams in Fig. 4.c. This problem concerns above all the attribute $name$ which should have just one occurrence in the class TA^p although it is inherited twice through the classes S^p and L^p. On the contrary, the attribute *code* must be in two copies in TA^p because it can potentially carry two distinct values. Otherwise, if the attribute *code* had always the same value for S^p and L^p, it should be defined already in P^p as it would be meaningful and unique for each person. In fact, this principle expresses the semantics of *virtual base classes* as defined in [20].

Formally, we describe the virtual base classes in a way category theorists express equations, i.e. through the commutative diagrams. In order to express the required semantics, the diamond of the multiple inheritance in Fig. 4.c should commute for the classes P^p, S^p, L^p and TA^p. In other words, the composite arrows on the two paths leading to the class P should give the same results: $\pi_{S:P} \circ \pi_{TA:S} = \pi_{L:P} \circ \pi_{TA:L}$. Thus we can assign a single label to these two equivalent projections referring to the shared attribute $name$ $\pi^{TA}_{name} = \pi^{S}_{name} \circ \pi_{TA:S} = \pi^{L}_{name} \circ \pi_{TA:L}$. As a result, we ensure that P^p is so called virtual base class and, in essence, the attribute $name$ in S^p and L^p denotes the same value. This construction, which is well known in CT, is called *pullback*. Therefore, the *pullback* enables to model the notion of virtual base classes.

3.4 Attributes as Labeled Projections

Let us summarize, that the strength of the object-oriented model lies in the inheritance construction. On the contrary to the aggregation, the inheritance enables to derive a new class, based on some existing one, bringing the inherited attributes to its name space. However, for the classes defined by the multiple inheritance it is not always possible since we require the attribute names to be unique. In LDM, the inheritance is modeled by using the product construction, arrow composition and projection labeling as suggested in the previous section. The arrow composition will be always unique but the problem of the assigning of unique names to the composite projections should be carefully examined. This issue was not addressed yet in the category theoretical setting.

Considering the previous example, this problem concerns TA^p, where we need to distinguish two distinct attributes of the same identification *code*. The corresponding projections $\pi^{S^p}_{code} \circ \pi_{TA:S}$ and $\pi^{L^p}_{code} \circ \pi_{TA:L}$ are always disambiguous but we can not add just the name *code* to the name space of TA^p because this name denotes two different values. The problem of labeling of the projections will be called the *naming problem*. The conflicting attributes *code* can be distinguished by using the double colon notation *S*::*code* and *L*::*code*.

In general, the naming problem can be resolved as follows. For each class, we define and label composite projections that access all attributes inherited from the superclasses through the *naming algorithm*.

First of all we need to define some auxiliary notions. Let $\pi^{C_1}_1, \pi^{C_1}_2, \ldots, \pi^{C_1}_p$ be all the possible projections accessing the attributes of class C_1. Let us denote the name of the attribute accessed by using $\pi^{C_1}_i$ as $attr(\pi^{C_1}_i)$. Each $\pi^{C_1}_i$ adds the name $attr(\pi^{C_1}_i)$ to the name space of C_1.

Our task is to define aliases for all $\pi^{C_1}_i, i = 1 \ldots p$ in the form $\pi^{C_1}_{prefix::name}$ or $\pi^{C_1}_{name}$. Projections accessing two distinct attributes can not be labeled by the same alias. In this expression, *prefix* is the name of a class and $name = attr(\pi^{C_1}_i)$. We will use $name$ instead of $attr(\pi^{C_1}_i)$ in the next paragraphs. Let $\pi^{C_1}_i$ be a composition

$$\pi^{C_1}_i = \pi^{C_n}_{name} \circ \pi_{C_{n-1}:C_n} \circ \ldots \circ \pi_{C_1:C_2}$$

and let $pref(\pi^{C_1}_i)$ denote the set $\{name, C_n::name, C_{n-1}::name, \ldots, C_1::name\}$ containing all the possible prefixes for the attribute $name$ if it would not cause any conflict. We will need also the following relation: we say that $\pi^{C_1}_i > \pi^{C_1}_j$ if

$$\pi^{C_1}_i = \pi^{C_n}_{name} \circ \pi_{C_{n-1}:C_n} \circ \ldots \circ \pi_{C_{k-1}:C_k} \circ \pi^{C_1}_j.$$

Now we can describe the algorithm itself:

Naming Algorithm

For all $\pi_i^{C_1}$, $i = 1 \ldots p$ and $name = attr(\pi_i^{C_1})$ **do**

if $name \neq attr(\pi_j^{C_1})$ for $\forall j = 1, 2, \ldots, i-1, i+1, \ldots, p$
then $name$ is unique in the name space of C_1 and we assign aliases

$$pref(\pi_i^{C_1}) \text{ to } \pi_i^{C_1}$$

getting equivalent projections $\{\pi_{label}^{C_1} \mid label \in pref(\pi_i^{C_1})\}$ referring $name$;
otherwise we assign aliases

$$pref(\pi_i^{C_1}) - \cup_{\pi_j^{C_1}} pref(\pi_j^{C_1})$$

where the union goes through the set $\{\pi_j^{C_1} \mid attr(\pi_j^{C_1}) = name, \pi_i^{C_1}$ and $\pi_j^{C_1}$ do not commute, $\pi_i^{C_1} > \pi_j^{C_1}\}$.

If $\pi_i^{C_1} > \pi_j^{C_1}$ or $\pi_j^{C_1} > \pi_i^{C_1}$ for any $\pi_j^{C_1}$ such that $attr(\pi_j^{C_1}) = name$ then the attribute $name$ is redefined in the class C_1. (Note: It is a question whether to allow attribute redefinition in the model.)

□

The purpose of this algorithm is to deal with the conflicts in the name space of the classes. Single prefixes are used to make the names of the attributes unique. There are several solutions of this problem. One extreme would be to write the whole path in the inheritance tree to identify an attribute. The aggregation works like this but we would lose the advantage of inheritance that enables to reuse the code and data structures.

Other possibility would be to refer the attributes using the unique name of the class where they were defined for the first time. Such class is often called *source class*. However, it still seems to be too demanding for the programmer. He should not need to write the source class each time referring an attribute. Therefore, a more complicated approach is used in the programming language C++. An attribute is identified by its source class only when it is unavoidable. As an example, consider the equivalent C++ declarations to that ones in Fig.4.c.

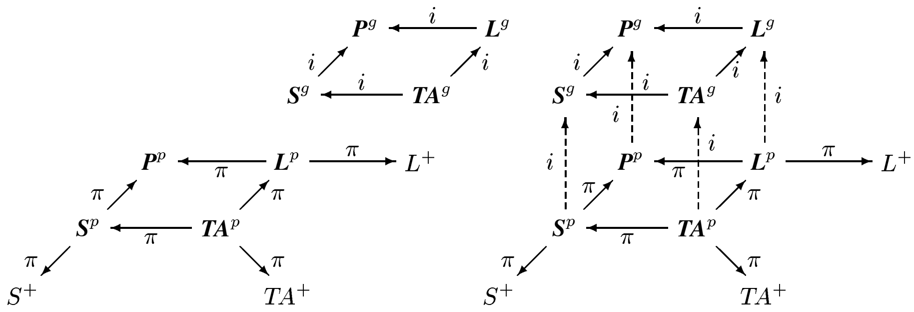
Furthermore, suppose that we have declared the variables `P p; S s; L l;` and `TA ta;`. Now, the attribute access `ta.name` is allowed and represents a shortcut for `ta.S::name` or `ta.L::name`. However, writing an expression such as `ta.code` causes an error message: `'TA::code' is ambiguous. It could be the 'code' in base 'S' of class 'TA', or the 'code' in base 'L' of class 'TA'`. In this case, the compiler requires to distinguish which attribute of `ta.S::code` and `ta.L::code` is meant.

On the other hand, we would like to allow the use of the unique source class in the prefix as, e.g., in the expression `ta.P::name`. Moreover, we should enable to use any meaningful prefix that does not cause the disambiguity. It will give enough freedom to the programmer to specify which attribute is meant and help us to solve the naming problem for polymorphic classes later in Sect. 4.

4 Polymorphism in LDM

The inheritance significantly differs from the aggregation also in another feature. An object of a subclass *can be used in all the places where the class can stand*. It is an important property of the object-oriented paradigm that enables code and data structure reuse. We call this kind of polymorphism *class polymorphism* although similar concept is in the type theory known under the term *subtyping*. There, the term polymorphism is reserved for functions with the types as parameters. However, we prefer the term polymorphism because it is widely used in the area of object-oriented programming in the way we are using it here.

In LDM, the class polymorphism is expressed by using the categorical objects called the *generalized classes*, which will be in the text denoted by using the superscript g. They are defined through the *sum* (or coproduct) construction with inclusion arrows. In the diagram in Fig. 5, the inclusion arrows are denoted as i. For each class we define a categorical object whose type represents the union of the types of its subclasses. In the case of the university database, the intention category $\mathcal{D}_I$ will have basically the structure shown in Fig. 5.c, where the upper square is required to commute.



(a) The bottom *pullback* (b) The upper *pushout* (c) Final inheritance hierarchy

Fig. 5. Semantics of class polymorphism

Let us see in detail what will be the structure of the objects in the upper square shown for clarity as a stand alone diagram in Fig. 5.b. We start from the categorical object P^p in the bottom. In this context, P^p represents database objects of pure type person with only one attribute $name$. Recall, that from this class we derived the classes S^p and L^p extending P^p by some additional property. In this case, it was the attribute $code$ represented in Fig. 5 by the objects S^+ and L^+. Continuing the derivation, we defined the class TA^p by the multiple inheritance adding the attribute sup represented as TA^+.

Furthermore, it is clear that the categorical objects found in the bottom square are, in the category **Set**, represented as sets that have uniform elements. These elements are tuples of the same cardinality. However, the class polymorphism requires the definition of sets containing tuples with possibly different number of members. This property is ensured in the upper square by using the objects with g in the superscript, which are themselves defined as the sum. In **Set**, we get objects with the following structure:

$$
\begin{aligned}
&TA^g = TA^p = \\
&\quad L^p \times S^p \times L_{oid} = \{(name, L{::}code, S{::}code, sup, type) \mid type = \mathtt{TA} \in Type\}
\end{aligned}
$$

$$
\begin{aligned}
S^g = S^p + TA^g \quad &= \{(name, code, type) \mid type = \mathtt{S} \in Type\} \\
&\cup \{(name, L{::}code, S{::}code, sup, type) \mid type = \mathtt{TA} \in Type\}
\end{aligned}
$$

$$
\begin{aligned}
L^g = L^p + TA^g \quad &= \{(name, code, type) \mid type = \mathtt{L} \in Type\} \\
&\cup \{(name, L{::}code, S{::}code, sup, type) \mid type = \mathtt{TA} \in Type\}
\end{aligned}
$$

$$
\begin{aligned}
P^g = P^p + S^g + L^g &= \{(name, type) \mid type = \mathtt{P} \in Type\} \\
&\cup \{(name, code, type) \mid type = \mathtt{S} \in Type\} \\
&\cup \{(name, code, type) \mid type = \mathtt{L} \in Type\} \\
&\cup \{(name, L{::}code, S{::}code, sup, type) \mid type = \mathtt{TA} \in Type\}
\end{aligned}
$$

where $name \in \mathtt{string}$, $code, L::code, S::code \in \mathtt{int}$, $sup \in L_{oid}$ and $Type = \{\mathtt{P}, \mathtt{S}, \mathtt{L}, \mathtt{TA}\}$. The additional member *type* in the structure of classes is a part of the semantics of the sum construction in the category **Set**. For more details see [1]. We use this feature to represent the types of the objects.

Notice, that the inclusion arrows i and the corresponding objects can be easily induced from the inheritance hierarchy and generated automatically in the category $\mathcal{D}_I$. Thus, the definition of the database still retains the objectivity of graph-based specification.

S^g can be viewed as the union of S^p (student tuples) and TA^g (assistant tuples). Similarly, P^g represents the union of S^g, L^g and P^p. The commutativity condition for the upper square ensures that P^g is union (not disjoint union) of S^g and L^g (they have TA^g as their intersection). This construction, which is well known in CT, is called *pushout*. Therefore, the *pushout* enables to model the notion of class polymorphism.

The final remark deals with the naming problem. Within this framework, the classes are represented as generalized classes, which are themselves defined as an union of the pure classes. Therefore, the naming schema should be extended for the generalized classes as well. For each generalized class, we define the projections as the *sums of projections* of the union components giving it the same label in subscript. To illustrate the naming of the generalized class projections, we get

$$\pi^{S^g}_{name} \equiv \pi^{S^g}_{P::name} = \pi^{S}_{P::name} + \pi^{TA}_{P::name} \text{ and}$$
$$\pi^{S^g}_{code} \equiv \pi^{S^g}_{S::code} = \pi^{S}_{S::code} + \pi^{TA}_{S::code}.$$

This categorical construction of the sum of morphisms in **Set** checks first which set does an element $o \in S^g$ come from. It is done with respect to the disjoint union $S^g = S^p \cup TA^p$ in which the member *type* is used to carry the information about the original set. Then, the projection defined for that set is used; i.e. $\pi^{S}_{P::name}$ is used for the objects from S^p and $\pi^{TA}_{P::name}$ is used for the objects from TA^p, respectively.

Notice, that to express the sum of projections we need to determine which name will be used to distinguish an attribute uniquely in the subclass. According to Sect. 3.4, we can use the source class prefix to avoid problems of prefix change for the attributes causing conflicts.

5 Relationships

Finally, we show how to represent relationships together with integrity constrains in our model. We will follow the semantics of relationships as defined in the ODMG'93 standard and particularly show how to define the binary relationships, namely one-to-one, one-to-many, and many-to-many. In this model, relationships themselves have no names. On the contrary, named *traversal paths* are defined for each direction of traversal, e.g., a *Lecturer supervises* a *Student*; contrariwise, a *Student is_supervised_by* a *Lecturer*. The traversal paths will be modeled by two distinct arrows in $\mathcal{D}_I$. The fact that two traversal paths both apply to the same relationship is indicated by the keyword *inverse*. It enables to maintain reference integrity by the object database management system.

A relationship can be easily implemented in C++ as a special attribute referring the corresponding object (one-to-one or many-to-one functionality) or set of such objects (one-to-many or many-to-many functionality). Notice, that relationships are just symmetric relations between two sets of objects. The next example, written in accordance with the ODMG'93 syntax, is a modification of the university database. It contains the classes *Student* and *Lecturer* and relationship *to_supervise* of one-to-many functionality with two traversal paths *is_supervised_by* and *supervises*.

Program 1 *Students and Lecturers*

```
interface Student {
  attribute String Name;
  attribute String Branch;
  relationship Lecturer
    is_supervised_by
    inverse Lecturer::supervises
};
```

```
interface Lecturer {
  attribute String Name;
  attribute String Title;
  relationship Set<Student>
    supervises inverse
    Student::is_supervised_by
};
```
□

The relationship *to_supervise* can be represented by the relation $R_{to\_supervise} \subseteq (Student \times Lecturer) \cup (Lecturer \times Student)$ and the two distinct paths $supervises$ and $is\_supervised\_by$ can be represented as the subrelations $R_{supervises} \subseteq (Lecturer \times Student)$ and $R_{is\_supervised\_by} \subseteq (Student \times Lecturer)$, respectively, where the subrelations satisfy $R_{to\_supervise} = R_{supervises} \cup R_{is\_supervised\_by}$. The reference integrity means that the relation $R_{to\_supervise}$ is symmetric. The functionality one-to-many and one-to-one corresponds to the restriction of the subrelation to be a map on one or both traversal paths, respectively. The actual objects stored in the database will have the following structure:

(***oid***, ***Name***, ***Branch***, ***is_supervised_by***)	(***oid***, ***Name***, ***Title***, ***supervises***)
(000,"Williams", "Math", 006)	(006,"Baker", "Dr.", {000})
(001,"Dobson", "Comp.", 007)	(007,"Bond", "MSc.", {001, 002})
(002,"Brown", "Comp.", 007)	

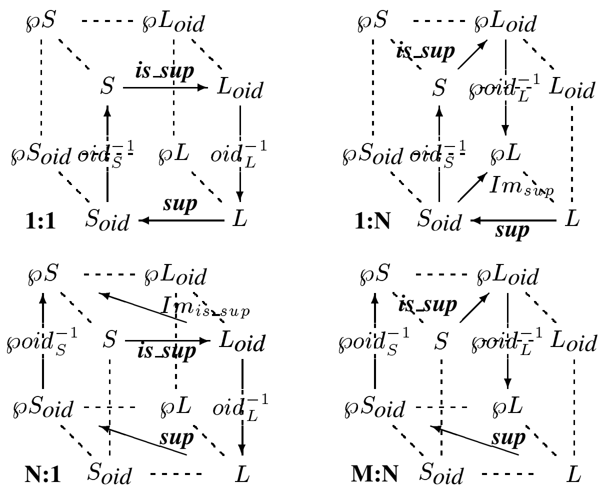
In LDM, the relations used to represent the relationships can be modeled by using the same pattern. We define relations as references to an object or objects depending on the functionality of the corresponding relationship. We use *oids* for this purpose.

The object identifiers can be viewed as an injective mapping assigning unique names to objects. For an object belonging to $Class$ we have: $\pi_{oid}^{Class} : Class \rightarrow Class_{oid}$ where $Class_{oid} \in OID$ is a set of all *oids* of objects belonging to that class. This collection of objects is often called *extent*. *Oids* are essential in the object-oriented paradigm for representation of references to objects, collections of objects and extents of classes. They play also an important role when introducing the notion of persistence into object-oriented programming languages.

To define many-to-one and many-to-many relationships we need to model the attributes which contain a set of *oids*. An additional category theoretical construction called *power object* will be used to represent the set-valued attributes. In **Set**, power objects are sets of all subsets of a given set. We will denote the power object of A as $\wp A$.

As a result, relationships are modeled as arrows returning *oid* of the associated objects. Each traversal path of the relationship corresponds to a distinct projection arrow. The class defining the relationship stands as the source of the arrow and its target is the categorical object representing *oid* or the set of *oids*. From another point of view, an additional attribute will be added to the class definition containing *oid*(s) of the object(s) involved in this relationship.

For example, the relationships of different arity, namely one-to-one, one-to-many, many-to-one and many-to-many, $is\_supervised\_by$ and $supervises$ between *Student* and *Lecturer* can be described by using the following diagrams in which S and L is abbreviation for the classes *Student* and *Lecturer*, respectively. Similarly, ***sup*** and ***is_sup*** stand for the traversal path *supervises* and *is_supervised_by*, respectively.



Concerning the reference integrity required by the keyword `inverse`, we can define its semantics easily in the first three cases. If the corresponding diagram commutes the reference integrity is preserved. It can be equivalently stated as the following expressions over the arrows: $oid_S^{-1} \circ sup \circ oid_L^{-1} \circ is\_sup = id_S$ for one-to-one relationship, where id_S is the identity arrow of S. $Im_{sup} = \wp oid_L^{-1} \circ is\_sup \circ oid_S^{-1}$ for one-to-many and $Im_{is\_sup} = \wp oid_S^{-1} \circ sup \circ oid_L^{-1}$ for many-to-one. The arrow

$Im_f(y) : codom(f) \rightarrow \wp dom(f)$ denotes the *inverse image* function assigning a set of elements $\{x \mid f(x) = y\}$ to every image y. The reference integrity of many-to-many relationship is a little more complicated. Therefore, we will show just the results here. It can be summarized as shown in the following table:

Relationships	Arrow Model		Reference Integrity
1 : 1	$is\_supervised\_by :$ $supervises :$	$S \rightarrow L_{oid}$ $L \rightarrow S_{oid}$	$oid_S^{-1} \circ sup \circ oid_L^{-1} \circ is\_sup = id_S$
1 : N	$is\_supervised\_by :$ $supervises :$	$S \rightarrow \wp L_{oid}$ $L \rightarrow S_{oid}$	$Im_{sup} = \wp oid_L^{-1} \circ is\_sup \circ oid_S^{-1}$ $Im_{sup} : S_{oid} \rightarrow \wp L$
N : 1	$is\_supervised\_by :$ $supervises :$	$S \rightarrow L_{oid}$ $L \rightarrow \wp S_{oid}$	$Im_{is\_sup} = \wp oid_S^{-1} \circ sup \circ oid_L^{-1}$ $Im_{is\_sup} : L_{oid} \rightarrow \wp S$
M : N	$is\_supervised\_by :$ $supervises :$	$S \rightarrow \wp L_{oid}$ $L \rightarrow \wp S_{oid}$	$c : [S \rightarrow \wp L_{oid}] \rightarrow [L \rightarrow \wp S_{oid}]$ $c : (x, y) \mapsto (y, x)$ bijection

6 Conclusions

In [17], Nelson and Rossiter propose so called product model in which classes are modeled as a stand alone categories. The notion of inheritance in the product model is based on the categorical sum construction with considerable disadvantages. Diskin and Cadish preferred the sketch-based approach. Similar flavor has the work of Tuijn based on limits. However, none of these works dealt in depth with inheritance, i.e. multiple inheritance, naming problem and class polymorphism.

We have extended the previous models to capture the semantics of the multiple inheritance, virtual base classes, class polymorphism and methods (not shown here because of limited space). The semantics of the aggregation, object identification, relationships and virtual methods was also formalized in the setting of LDM yet retaining objectiveness of graph-based specification.

In this paper, we have shown how to formalize the notion of the class by using the product construction and the notion of the virtual base class by using the means of the pullback. In this context, the class polymorphism was modeled as a categorical sum which was extended to a pushout in the case of the multiple inheritance. Finally, we described relationships as arrows and semantics of the keyword `inverse` as commutative conditions in the category $\mathcal{D}_I$.

This approach is concerned with the mathematical foundations of the conceptual modeling and we used it to reason about the data structures and possible conflicts arising from the multiple inheritance or caused by the objects that can change roles during their lifetime [13]. In [21], Tuijn shown the computational power of his typegraph model (see Sect. 2.1). These results can be equally applied to LDM. The final objective is to define the semantics of an object-oriented database model similar to that one defined in ODMG'93. The problem of queries and definition of bulk data types will be the subjects of our future research.

References

1. BARR, M., WELLS, C.: Category Theory for Computing Science. Prentice Hall, 1995.
2. CATTELL, R.: The Object Database Standard ODMG-93, Release 1.2. Morgan Kaufmann Publishers, 1996.
3. COSTA, J., SERNADAS, A., SERNADAS, C.: Object Inheritance Beyond Subtyping. Acta Informatica, 1994.
4. DISKIN, Z., CADISH, B.: Algebraic Graph-Oriented = Category Theory Based. Manifesto of categorizing database theory. Tech. Rep. 9406, Frame Inform Systems, Riga, Latvia, 1994.
5. DISKIN, Z., CADISH, B.: Databases as graphical algebras: Algebraic graph-based approach to data modeling and database design. Tech. rep., Frame Inform Systems, Riga, Latvia, 1996.
6. DISKIN, Z., CADISH, B.: Variable Sets and Functions Framework for Conceptual Modeling: Integrating ER and OO via Sketches with Dynamic Markers. In Proceedings of 14th International Conference OOER'95 on Object-Oriented and Entity-Relationship Modelling, M. P. Papazoglou, Ed., vol. 1021 of LNCS, Springer, 1995, pp. 226–237.
7. FIADEIRO, J., SERNADAS, C., MAIBAUM, T., SAAKE, G.: Proof-theoretic semantics of object-oriented specification constructs. In Object-oriented Databases: Analysis, Design and Construction. North-Holland, 1990.
8. GOGOLLA, M.: An Extended Entity-Relationship Model, Fundamentals and Pragmatics. Springer-Verlag, LNCS 767, 1994.
9. GOGUEN, J.: A Categorical Manifesto. Mathematical Structures in Computer Science, 1(1):49–67, 1991.
10. GOGUEN, J. A., MALCOLM, G.: Extended Abstract of a Hidden Agenda. In Proceedings, Conference on Intelligent Systems: a Semiotic Perspective, J. Meystel, A. Meystel, and R. Quintero, Eds., 1996.
11. GUERRINI, G., BERTINO, E., BAL, R.: A Formal Definition of The Chimera Object-Oriented Database Model. Tech. Rep. IDEA.DE.2p.011.01, ESPRIP project 633, 1994.
12. HRUŠKA, T., KOLENČÍK, P.: Semantics of Object Identification in Object-oriented Database Model. In Proceedings of Scientific Conference Electronic Computers and Informatics, Faculty of Electrical Engineering and Informatics of Technical University Košice, 1996, pp. 243–249.
13. HRUŠKA, T., KOLENČÍK, P.: Extending an Object-Oriented Model: Multiple Class Objects. In Proceedings of 16th International Conference ER'97 on Conceptual Modelling. Accepted to publication.
14. KIFER, M., LAUSEN, G., WU, J.: Logical Foundations of Object-Oriented and Frame-Based Languages. Journal of the ACM, 42(4):741–843, 1995.
15. KOLENČÍK, P.: Classes in Object-Oriented Model as Categorical Objects. In Proceeding of the works of PhD students, FEECS TU of Brno, 1996, pp. 75–77.
16. KOLENČÍK, P.: Methods in The Object-Oriented Model as Sums of Morphisms. In Preceeding of International Conference MOSIS'97, 1997, pp. 83–88.
17. NELSON, D., ROSSITER, B.: Prototyping a categorical database in P/FDM. In Second International Workshop on Advances in Databases and Information Systems ADBIS'95, 1995.
18. PIESSENS, F., STEEGMANS, E.: Categorical data-specifications. Theory and Applications of Categories, 1:156–173, 1995. Available at `http://www.tac.mta.ca/tac/`.
19. SIEBES, A.: On Complex Objects. PhD thesis, University of Twente, 1990.
20. STROUSTRUP, E.: The annotated C++ reference manual. Addison Wesley, 1992.
21. TUIJN, C.: Data Modeling from a Categorical Perspective. PhD thesis, Antwerpen University, 1994.