Extending an Object-Oriented Model: Multiple Class Objects

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. Experiences of both researchers and practitioners with the development of object-oriented database systems help us to evaluate the real contributions of object-oriented modeling principles and show the limits and possibilities of improvements of the object-oriented model. One of the research areas where such a need was recognized concerns the class membership. In most object-oriented data model objects must belong to a single most specific class. However, the real world situations often break this presumption. The aim of this paper is to discuss the issues relating to an extension of modeling concepts to capture the objects that can belong to the multiple most specific classes. Our final objective is to present a framework for the formal object-oriented models that will provide the rigorous theoretical tool to solve issues concerning conflicts that can arise in structural components of objects. Most importantly, we show that the structural conflicts can be solved from the context determined by the static typing.

1 Introduction

The real world situations often break the common presumption of the object-oriented design where objects have a structure determined for all lifetime. In the conceptual design, we often use the multiple inheritance for the model situations in which entities can play several roles. However, this approach does not see an object as a single entity acquiring and discarding roles. According to [14], this approach to the object typing leads to the loss of the possibility to build highly abstract data models because it does not correspond to the actual way the real world entities behave. These problems influence the design, implementation and application level of the object-oriented database systems. For an appropriate study of these questions, some theoretical tools were used (see [2]) to avoid any unexpected conflicts.

To illustrate, consider a database of persons at a university. Suppose that a public library represents a part of this university. A person can play a lot of different roles, such as employee, a student, a teacher or a reader. The roles played by the objects can change during their lifetime.

* 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.

Assume that there are stored personal codes of students and distinct codes of readers in the database. Furthermore, suppose that a former reader enters this university. The system should automatically resolve conflicts in the attribute *code*. Of course, many cases of this situation may occur, and the database designer should not be forced to rewrite each time the database scheme to get a new class defined by the multiple inheritance. For example, the new class would be forced for the role of *student–reader*. The preferred solution lies in the existence of objects being of the two (generally more) types.

They are two basic reasons for this solution. First, we avoid the combinatorial explosion of the classes defined by using the multiple inheritance (in the cases when many classes can be combined). Second, we do not want to define some new functionality for the combined class. We just expect that it possesses attributes of both students and readers. However, we must decide how to manage the name conflicts due to the presence of attributes and methods of the same name. For this purpose, we deal with the *multiple inheritance* modeling. The multiple inheritance will be then used to represent the *multiple class objects*.

We present another approach to solve the problem of the multiple class objects by using the *category theory* (CT). In their manifestos [3, 6], Diskin and Goguen pointed out that the category theory is the formalism especially suited for this purpose. According to these manifestos, we try to show that the approach based on CT brings more simplicity and clarity, in addition, makes the reasoning easier. Specifically, we show, that the structural conflicts can be solved from the context determined by the static typing. Advantages of CT will be illustrated in terms of the formal database model called LDM (Limit Data Model, see Sect. 2), designed to support the development of the object-oriented database system at the author's department.

The main reason for using CT as a formal tool for the object-oriented modeling is that it offers a formal, graphical and powerful abstract language for the modeling of objects (this idea is developed especially in [4]). This property gives a new view and helps to see nontrivial properties which would be difficult to discover otherwise (examples can be found in [5, 10, 11, 13]).

Moreover, all categorical constructions specify objects in terms of their relationship to other objects, and this specification results in a clearer declarative approach with an immediate operational equivalent. In this paper, we use only the basic notions of the category theory (see [1]). The original ideas were introduced in [8, 9, 7]. Here, we focus on the problem of an attribute access for the classes defined by the multiple inheritance and the semantics of the operations creating objects and acquiring and discarding their types.

The paper is organized as follows. Section 2 gives the definition of the inheritance in LDM with an emphasis on the multiple inheritance, attribute access, and disambiguity resolution. Section 3 explains, how to create objects with a single most specific class. Section 4 deals with the semantics of the type changing operations *acquire* and *discard*. Finally, Section 5 presents the semantics of the attribute access for the objects with multiple most specific classes together with the proof that conflicts can be resolved using the static typing in LDM.

2 Inheritance in the Limit Data Model

In general, the database intention is modeled as category $\mathcal{D}_I$ in LDM. In $\mathcal{D}_I$, the categorical objects represent *classes* while arrows with additional conditions are used to model an *aggregation*, *inheritance*, *relationships* and *methods*.

The database objects are modeled as tuples defined as a product. Within this framework, we can refer to the class attributes only through the projections (this expresses the principle of an encapsulation). To bring attributes from the superclass to the name space of its subclass, as required by the inheritance, we need a device of an arrow composition that is essential in CT. According to [1] we use the notation $g \circ f(x)$ for a morphism $g(f(x))$. Moreover, in a category the objects and the arrows are not named; therefore, we will introduce a labeling function assigning meaningful names to projections that will enable us to model inheritance.

Hereafter, we use the labeling π^{Class}_{name} for the projection accessing the attribute $name$ of the class $Class$. Disambiguities caused by the multiple inheritance will be solved by using $\pi^{Class}_{Super::name}$ denoting the attribute $name$ of the class $Class$ inherited from the superclass $Super$. Finally, the projection that defines a type coercion to the superclass will be written as $\pi_{Class:Super}$. For each subclass, we need to define composite arrows that enable to access the attributes inherited from the superclass and assign appropriate names to the composite projections following the labeling schema.

To give a deeper insight to the problems caused by the multiple inheritance, consider an example of the class *person* $P^p : \{name\}$, *student* $S^p = P^p \times \mathtt{int} : \{(name, code)\}$ and *lecturer* $L^p = P^p \times \mathtt{int} : \{(name, code)\}$, both $student$ and $lecturer$ inheriting from P. The class *teaching assistant* $TA^p = L^p \times S^p : \{(name, L{::}code, S{::}code)\}$ inherits from *lecturer* and *student*.

According to the strong typing of CT, p in $Class^p$ indicates that this categorical object represents purely the database objects belonging to $Class$, not to its subclasses (as required by the principle of the class polymorphism in the object-oriented languages). We call the classes denoted as $Class^p$ *pure classes*.

The attribute $name$ is placed once in the class TA^p even if 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 *students* and *lecturers*, it should be defined in the class *person* only once because it is meaningful and unique for each person. In fact, this principle expresses the semantics of the *virtual base classes* as defined in [12].

Formally, we describe the virtual base classes in a way of category theory which express equations by using of commutative diagrams. Consider Fig. 1. In this figure the diamond of the multiple inheritance for TA^p commutes with the meaning that the composite arrows on two paths leading to P^p give the same results: $\pi_{S:P} \circ \pi_{TA:S} = \pi_{L:P} \circ \pi_{TA:L}$. This means that we can assign a single name to these two equivalent projections referring to the shared attribute $name$ $\pi^{TA^p}_{name} = \pi^{S^p}_{name} \circ \pi_{TA:S} = \pi^{L^p}_{name} \circ \pi_{TA:L}$. As a result, we ensure that P^p is the virtual base class and, in essence, the attribute $name$ in S^p and L^p denotes the same thing. This construction, which is well known in the category theory, is called *pullback*. Therefore, the *pullback* enables to model the notion of virtual base classes.

$$
\begin{tikzcd}
 & & & \mathtt{string} & \\
\mathtt{int} & & P^p \arrow[ur, "\pi^{P^p}_{name}"'] & & \mathtt{int} \\
 & S^p \arrow[ul, "\pi^{S^p}_{code}"'] \arrow[ur, "\pi_{S:P}"] & & L^p \arrow[ul, "\pi_{L:P}"] \arrow[ur, "\pi^{L^p}_{code}"'] & \\
 & & TA^p \arrow[ul, "\pi_{TA:S}"] \arrow[ur, "\pi_{TA:L}"'] & &
\end{tikzcd}
$$

Fig. 1. The diamond of multiple inheritance for teaching assistants.

2.1 Naming Problem

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 inherited attributes to its name space. However, for the classes defined by the multiple inheritance it is not sometimes possible to place all inherited names into one naming space. The inheritance is in LDM modeled by using the product construction, arrow composition and projection labeling. The arrow composition will be always unique but the problem of the assigning of unique names to the composite projections should be carefully examined.

Considering the previous example, this problem concerns TA^p, where we need to distinguish two distinct attributes $code$. 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 because this name denotes two different things. The problem of projections labeling 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 composite projections that access all attributes inherited from the superclasses through the following *naming algorithm*:

Algorithm 1 (**Naming Algorithm**)

For all classes Sup that are superclasses of $Class$ so that have their projections for the attributes already defined and labeled **do**:

1. **if** for any class $X \in \mathcal{D}_I$ $\exists Y \in \mathcal{D}_I$, $X \neq Y$, both distinct from Sup, such that the class Sup contains projections labeled $\pi^{Sup^p}_{X::name}$ and $\pi^{Sup^p}_{Y::name}$

   then $$\pi^{Class^p}_{S::name} = \pi^{Sup^p}_{X::name} \circ \pi_{Class:Sup},$$
   where S is the *source class* of $name$ in which the conflict causing attribute $name$ was defined for the first time (it can be induced from the composition of the projections);

   otherwise, we know that Sup contains the attribute $name$ in the forms $S{::}name$ or $name$ only (where S could be the unique superclass of Sup if the attribute $name$ comes from it). Therefore, we define a labeling of this projection as

$$\pi^{Class^p}_{Sup::name} = \pi^{Sup^p}_{name} \circ \pi_{Class:Sup};$$

2. **then**, we define the projections for the attributes that $Class$ adds itself with $\pi^{Class^p}_{name}$;
3. **finally**, we can identify some common attributes in the form $Sup_1::name$ and $Sup_2::name$ and define a new label $name$ for projections satisfying the commutativity requirement $\pi^{Class^p}_{Sup_1::name} = \pi^{Class^p}_{Sup_2::name}$. We assign the new label $name$ also for projections with unique label $\pi^{Class^p}_{Sup::name}$ (if there arises the name conflict, we report an error message). $\square$

This is precisely what C++ does. To avoid disambiguities, we can always say which inheritance path is meant using the double colon notation with the direct superclass and the attribute name $Sup::name$. However, if a conflict has already occurred in the superclass, we distinguish the attributes by specifying the source class that they come from (without the possibility to write the whole path). In our example, the projections can be computed and labeled as follows:

P : $\pi^{P^p}_{name} : P^p \to \texttt{string}$

S : $\pi^{S^p}_{code} : S^p \to \texttt{int}$

$\pi^{S^p}_{name} \equiv \pi^{S^p}_{P::name} = \pi^{P^p}_{name} \circ \pi_{S:P}$

L : $\pi^{L^p}_{code} : L^p \to \texttt{int}$

$\pi^{L^p}_{name} \equiv \pi^{L^p}_{P::name} = \pi^{P^p}_{name} \circ \pi_{L:P}$

The first projection $\pi^{P^p}_{name}$ for the class $person$ is named according to the second step of the naming algorithm. When we finish the labeling process for the projection of the class *person* we can proceed it for its subclasses. With respect to their structure, classes *student* and *lecturer* are very similar. The attribute $name$ can be accessed through the projection with two equivalent names $\pi^{S^p}_{name}$ and $\pi^{S^p}_{S::name}$ assigned in the first (the branch **otherwise**), and the third step of the naming algorithm.

More interesting naming process runs for the class *teaching assistant* defined by the multiple inheritance:

TA^p : $\pi^{TA^p}_{name} \equiv \pi^{TA^p}_{S::name} \equiv \pi^{TA^p}_{L::name} = \pi^{S^p}_{name} \circ \pi_{TA:S} = \pi^{L^p}_{name} \circ \pi_{TA:L}$

$\text{(according to the commutative condition)}$

$\pi^{TA^p}_{S::code} = \pi^{S^p}_{code} \circ \pi_{TA:S}$

$\pi^{TA^p}_{L::code} = \pi^{L^p}_{code} \circ \pi_{TA:L}$

The attribute $name$ coming from the class *person* can be accessed by using the composition of the projections (following again the branch **otherwise** in the first step of the naming algorithm and then the third step). In this case, we have three possibilities of labeling this composed projection according to the commutativity condition required for the diamond of the multiple inheritance in Fig. 1. However, the attribute *code* has not these properties since it has two different sources S and L in the inheritance hierarchy, so we will leave the two projections, referring to two different things, with distinct labels.

2.2 Class Polymorphism

The inheritance significantly differs from the aggregation also in the following way. An object of a subclass of a class can be used in all places where the class can stand. This is an important property of the object-oriented model that enables to reuse a code and data structures. In our model, this kind of class polymorphism is expressed by using categorical objects called the *generalized classes*, which will be in the text denoted by using the superscript g. They are defined through the cocone construction with the inclusion arrows i.

Simply expressed, for each class we define a categorical object whose type represents the union of the types of its subclasses. For example, in the case of the university database, the category $\mathcal{D}_I$, representing the schema of the database, will have the structure shown in Fig. 2, where the upper square is required to commute. The inclusion arrows leading to the upper square can be easily induced from the inheritance hierarchy in the bottom.

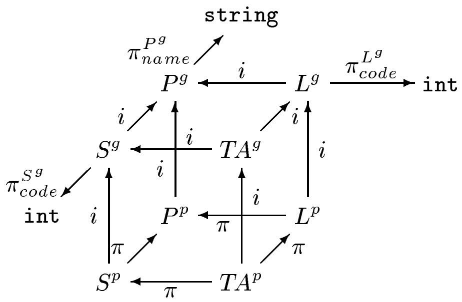

Fig. 2. Inheritance hierarchy of the university database.

The categorical object S^g can be viewed as an union of the sets S^p ($student$ tuples) and TA^g ($assistant$ tuples). Similarly, the object P^g represents union of the sets S^g, L^g and P^p. The commutativity condition for the upper square ensures that P^g is the union (not the disjoint union) of S^g and L^g (they have TA^g as intersection). This construction is in the category theory known as the *pushout*. This means that disjoint extents of S^g and L^g can be expressed using the pushout.

Within this framework, the classes are represented as the generalized classes, which are themselves defined as an union of the pure classes (i.e. the categorical objects with the superscript p that represents the tuples of one type that are stored in the database). Therefore, the naming scheme should be extended for the generalized classes as well. For each generalized class, we define the projections as *sums of projections* of the union components giving it the same label in the subscript. For example, we get $\pi^{S^g}_{name} \equiv \pi^{S^g}_{P::name} = \pi^{S^p}_{name} + \pi^{TA^p}_{name}$ and $\pi^{S^g}_{code} = \pi^{S^p}_{code} + \pi^{TA^p}_{S::code}$. First, this categorical construction of the sum of morphisms in **Set** (the category of sets and mappings between them) determines the set to which an element $o \in S^g$ belongs (with respect to the disjoint union $S^g = S^p \cup TA^p$); then it uses the projection defined for the determined set ($\pi^{S^p}_{name}$ for the objects from S^p and $\pi^{TA^p}_{name}$ for the objects from TA^p).

Now, having defined the generalized classes, the semantics of the object identification can be described in a category as an injective mapping from objects to the set of OIDs that are currently in use (in a length explanation can be found in [7]). OIDs are essential in the object-oriented model for representation of references to the database objects, collections of objects and extents of classes. They also play an important role when introducing persistence into the object-oriented database model.

3 Creating the Database

So far, we have dealt only with the *metadata* of the database and expressed the database schema as a category $\mathcal{D}_I$. Next, we explain how to describe the actual data stored in the database and formalize the semantics of the operations that enable us to create the objects and change their types.

Specific objects (tuples with complex structure) derived from the class definition $\mathcal{D}_I$ are formalized as a *model* of the database intention in the **Set**. **Set** is the category of sets and mappings between them [1]. In the category theory, it is formally represented by using a *functor* D, to another category $\mathcal{D}_E$ representing the content of the database or the database extension.

The functor is defined as a pair of functions. One of them assigns the objects of the first category to the objects of the second category, and the second function does the same job for arrows. Informally, in our case the functor D defines the appropriate mappings from classes of $\mathcal{D}_I$ to their extents in $\mathcal{D}_E$, OIDs to the natural numbers (e.g.), the morphisms to the methods. The commutative conditions of $\mathcal{D}_I$ are required to be preserved also in $\mathcal{D}_E$.

Any transaction consisting of a method invocation or an operation *update*, *new* or *delete* is expressed as a change of D according to the results given by the function in $\mathcal{D}_E$ used to implement the operation defined in $\mathcal{D}_I$. In this way, we can introduce the notion of persistence into the inherently functional language of the category theory.

The semantics of *new* can be specified in the following way: we define a new functor, $D' : \mathcal{D}_I \to \mathcal{D}_E$, that satisfies $D'(c) = D(c)$, for all classes $c \in \mathcal{D}_I - \{Class\}$, and redefine its value for $Class$ by adding a new tuple, *obj*, to the set $D'(Class) = D(Class) \cup \{obj\}$. However, obj needs to be added also to the extents of all the superclasses of the class $Class$: $\forall Sup \in \mathcal{D}_I$, such that $\exists \pi_{Class:Sup} D'(Sup) = D(Sup) \cup obj$.

For example, we show how to create a new object obj of the type S^p initialized to *obj* $= (James, 007)$ (the first attribute denotes the name of the student and other denotes his code in the database). The state of the database before the execution of *new* is shown in Fig. 3. For simplicity, we have omitted object identifiers in this figure.

Recall that the diamond in Fig. 3 is the upper square in Fig. 2, which represents the database intention $\mathcal{D}_I$. D maps the classes to their actual extents in the database represented by $\mathcal{D}_E$. Let us show how these extents are composed by using the sum construction; for instance, the tuple $(Paul, 002, 002)$, representing a teaching assistant, is included to the extents of all classes in the hierarchy. Similarly, *Mark* and *Tom* are included also to the extent of *persons*. Finally, P^g contains all the tuples in our small database.

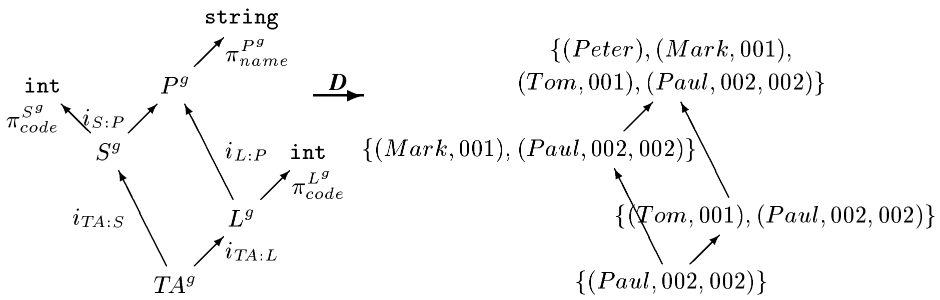

Fig. 3. The database schema $\mathcal{D}_I$ and the content $\mathcal{D}_E$ before the operation *new*.

The invocation of *new* for James with the personal code 007 will have the following effect on the database expressed as a change of D to D' (see Fig. 4):

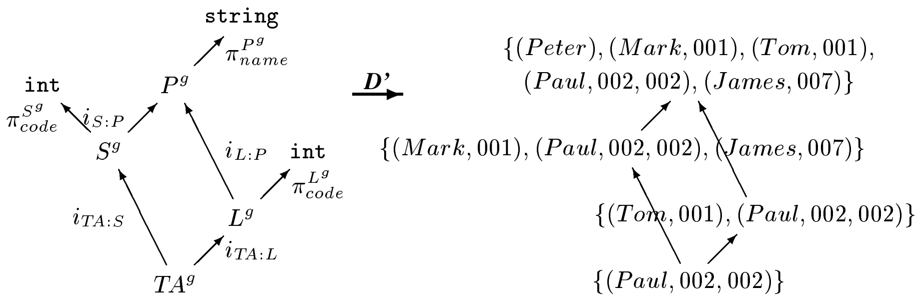

Fig. 4. The database schema $\mathcal{D}_I$ and the content $\mathcal{D}_E$ after the operation *new*.

4 Semantics of Flexible Objects

We have given the formal specification of *new* that creates the objects based on some type existing in the inheritance hierarchy of $\mathcal{D}_I$. In this section we explain how to change the role played by an object; then, we study how it affects the types of objects.

The role-attribute model proposed in [14] defines the conditions on which an object can acquire and discard roles. This definition implies an existence of two corresponding operations. Operation *acquire* extends the type of an object with a new type and *discard* removes the type from an multiple class object. The effect of the first operation, acquiring new type B for an object of type A, is equivalent to the construction of a new object, $A \times B$, with added attributes B. Notice that its type corresponds to the type resulting from the multiple inheritance with the superclasses A and B. However, this class is neither named nor directly accessible. Therefore, we call it *product class*.

On the other hand, the second operation can remove some attributes that are defined as a part of the type that this object discards. Because the structure of objects in such model can change, we call them *flexible objects* or *multiple class objects*. In this section, we define the semantics of these two type of changing operations.

We will define the effect of *acquire* as a change of the database schema $\mathcal{D}_I$ adding the corresponding class defined by the multiple inheritance. However, this is expressed theoretically because the strong typing of the category theory requires an existence of the type corresponding to the structure of the multiple class objects for the sake of consistency. The implementation does not require any change in the database metadata catalogue, and the designer can choose whether the existence of multiple class objects is hidden to the users.

In [2], the authors expressed the type of multiple class objects in terms of its most specific classes. This expression is useful because the system can more easily analyze the effect of *acquire* and *discard* on the structure of objects. In general, *acquire* for an object $obj \in C_1 \times C_2 \times \ldots \times C_n$ and a new role of A takes this object of type $C_1 \times C_2 \times \ldots \times C_n$ and transforms it to an object, obj', of type $C_1 \times C_2 \times \ldots \times C_n \times A$. If we omit the inheritance hierarchy and corresponding commutative conditions (corresponding to the notion of the virtual base class), this operation adds all the attributes of A to the object obj. The additional commutative conditions imposed on the inheritance hierarchy restrict the actual attributes added to the product to those defined as the product of most specific classes of the object.

There are three possibilities to influence the structure of *obj* by A. First, A can be the supertype of some C_i. Then, the structure and the type of this object remains the same. Second, A can be the subtype of some classes C_1, ..., C_n. This case means that these classes can be removed from the product while the class A can be added. Finally, if A is not the subclass or superclass of any C_i, the product can be extended by A.

The semantics of *acquire* can be defined by using the product construction with additional commutative conditions as required by the inheritance hierarchy. An example of the multiple class object *reader-student* is shown in Fig. 5 ($R\times S^p$ has the meaning of $(R\times S)^p$). In this example, R^p represents *readers* and PS^p *postgraduate students*.

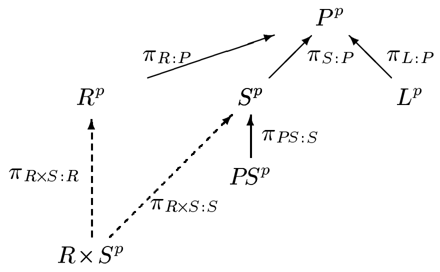

Fig. 5. Multiple class object $R\times S^p$ before invocation of the operation *acquire*.

The three distinct possibilities of classes that can be acquired for $R{\times}S^p$ in the bottom are P^p , PS^p or L^p. Operation *acquire* P does not change the structure of $R\times S$-object because it already contains all the features of P. The other two possibilities of acquiring of PS and L affect the structure of $R\times S$-object (see Fig. 6).

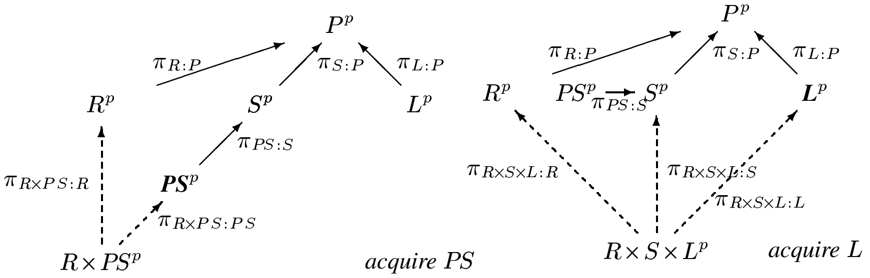

Fig. 6. Illustration of the operation *acquire*.

The operation *discard* works in the reverse order. We can study all distinct cases depending on the relationship between the object type $obj \in C_1 \times C_2 \times \ldots \times C_n$ and discarded type A. If A is not in the supertype or the subtype relationship with any C_i, the structure and the type of the object remain the same. If A occurs in the product $C_1 \times C_2 \times \ldots \times C_n$ as class C_i then obj' has the structure $C_1 \times C_2 \times \ldots \times C_{i-1} \times C_{i+1} \times \ldots \times C_n$.

However, this is not the only possible functionality. For example, we can consider $obj \in R\times S$ and *acquire* type PS getting $obj \in R\times PS$. Now an invocation of *discard* PS would cause the object obj to have the type of R: $obj \in R$ although we expected to get the type it had before applying of *acquire*.

Neither it is also clear what happens if A occurs in the subclass or superclass relationship with some C_i. In this case, the type of obj can remain the same or be restricted. At this time, we prefer removing all the classes C_i that coincide with A or that are in the superclass or subclass relationship with it. We are aware of the fact that the possibility of reversing *acquire* is disabled (we can not return to the preceding state of database).

5 Conflicts and Their Solution

In the previous sections, we have not dealt with the problem of the attribute access for the objects having multiple most specific classes. The present section continues with the use of the category theoretical means as explained in the Sect. 2.

As demonstrated in the previous sections, there is no problem with conflicts concerning the structure of the class state in the category theoretical framework. By using the commutativity conditions, we specify which attributes are essentially the same and which are not.

However, the core of the conflict lies in the naming problem caused by the flat naming space for the classes defined by the multiple inheritance. However, in the case of product classes (used to the model multiple class objects), we choose another policy for the naming because we suppose that the user has not direct knowledge about attributes of such classes. If he wants to access objects with the multiple most specific classes he should use a reference defined only for some existing base class. This context for attribute access gives as natural means for conflict solution.

Before dealing with the conflict resolution, we briefly explain how objects can be accessed by using references in LDM. Conceptually, objects are referenced through their identifiers which are themselves not accessible by a user. Therefore, the objects can be directly denoted by variables or indirectly by expressions containing the attribute access or the method invocations starting from an object-denoting variable. Since the category theory is a strongly typed language we have an unique type associated with each object reference (see [2]). In this section, we show how to use this type to determine the context in which attributes are accessed and, thus, solve possible conflicts.

Recall that the attributes of the product classes $A \times B$ composed from A and B are accessible only through the generalized classes A^g and B^g. For A, it is defined as a disjoint union $A^g = A^p \cup A \times B^p$ and, similarly, for B. Therefore, it remains to define the labeling for the product classes of $A \times B$. Then, we can prove that there is no conflict concerning the attribute access.

The problem arises from $A{::}attr$ and $B{::}attr$, which were brought to the product class through the multiple inheritance. The type of the generalized class used to access the multiple class object (in our case, A^g or B^g) determines the *context*, and we must choose the right projections to compose $\pi^{A^g}_{attr}$. There are two possibilities – $\pi^{A^g}_{attr} = \pi^{A^p}_{attr} + \pi^{A \times B^p}_{A::attr}$ or $\pi^{A^g}_{attr} = \pi^{A^p}_{attr} + \pi^{A \times B^p}_{B::attr}$. The first, and the second term defines, how $\pi^{A^g}_{attr}$ is translated for A and $A \times B$-objects, respectively. For example, naming scheme for generalized classes of the university database discussed in the previous sections and the product class $S \times L$ can be computed as follows:

$$\pi^{S^g}_{name} = \pi^{S^p}_{name} + \pi^{S \times L^p}_{name}$$
$$\pi^{P^g}_{name} = \pi^{S^p}_{name} + \pi^{L^p}_{name} + \pi^{P^p}_{name} + \pi^{S \times L^p}_{name} = \pi^{S^g}_{name} + \pi^{L^g}_{name} + \pi^{P^p}_{name}$$

It means that $\pi^{P^g}_{name}$ is for the objects of type S translated as $\pi^{S^g}_{name}$, for the L-objects as $\pi^{L^g}_{name}$ and for the object of type P as $\pi^{P^p}_{name}$. The projections $\pi^{S \times L^g}_{name}$, $\pi^{S \times L^g}_{S:code}$ and $\pi^{S \times L^g}_{L:code}$ are not defined in the database because the system is not aware of the class $S \times L$. This class serves only for the sake of typing consistency in the model category $\mathcal{D}_E$. Nevertheless, we define these projections and labels as shown later on to give the semantics for the context dependent solution of the name conflicts.

The attribute *code* that causes potential conflict is more interesting. It is not defined for the class P. For the other classes, the projections are

$$\pi^{S^g}_{code} = \pi^{S^p}_{code} + \pi^{S \times L^p}_{S::code} \qquad \pi^{L^g}_{code} = \pi^{L^p}_{code} + \pi^{S \times L^p}_{L::code}$$

It means that $\pi^{S^g}_{code}$ is for the objects belonging to S equal to $\pi^{S^p}_{code}$ (see Fig. 7) while for the objects belonging to $S \times L^p$ as $\pi^{S \times L^p}_{S::code}$. Similarly, we define sums of morphisms for L. Notice the definition of the sum of morphisms $\pi^{S^g}_{code}$, which determines that in the context of S we always access the attribute S::*code* for both kinds of objects belonging to the class S or $S \times L$. Similarly, the attribute *code* in the context L means L::*code* for L and $S \times L$-objects.

$$\begin{tikzcd}
S^p \arrow[r, "i_1"] \arrow[d, "\pi^{S^p}_{code}"'] & S^g = S^p + S \times L^p \arrow[d, "\pi^{S^g}_{code} = \pi^{S^p}_{code} + \pi^{S \times L^p}_{S::code}"] & S \times L^p \arrow[l, "i_2"'] \arrow[d, "\pi^{S \times L^p}_{S::code}"] \\
\mathtt{int} \arrow[r, "i_1"'] & \mathtt{int} + \mathtt{int} & \mathtt{int} \arrow[l, "i_2"]
\end{tikzcd}$$

Fig. 7. Composing the sum of morphisms for the product classes.

Therefore, the property of the sum of morphisms offer us how to distinguish attributes S::*code* and L::*code* by using the context in which the object is accessed through the reference. The different attributes *code* of $S \times L$-objects will be accessed depending on the context in which the object is referred. By using a variable of the type S^g, access to the attribute *code* will be translated through the sum of morphisms $\pi^{S^g}_{code}$ as $\pi^{S \times L^p}_{S::code}$ while through the morphism $\pi^{L^g}_{code}$ as $\pi^{S \times L^p}_{L::code}$. Generally, we extend the naming scheme for the generalized classes to cover the product classes by the following step:

Algorithm 2 (**Context Dependent Access**)

We generate labels for product class $P \times B^p$ by using the *naming algorithm* (see Algorithm 1). **For each** attribute $name$ in B satisfying one of the following conditions **do**:

1. **if** $name$ was inherited to B and does not cause any conflict, and

   if the attribute $name$ does not exist in P, we extend both sums of morphisms $\pi^{B^g}_{name}$ and $\pi^{B^g}_{X::name}$ by adding the morphism $\pi^{P \times B^p}_{B::name}$,

   otherwise, we extend both sums of morphisms $\pi^{B^g}_{name}$ and $\pi^{B^g}_{X::name}$ by the morphism $\pi^{P \times B^p}_{S::name}$ (S is the source class of the attribute $name$ in B);
2. **if** $name$ causes a conflict in B and was distinguished by using the prefix $S::name$, we extend the sum of morphisms $\pi^{B^g}_{S::name}$ by adding the morphism $\pi^{P \times B^p}_{S::name}$ (S is the source class for that attribute $name$);
3. **otherwise**, $name$ was defined in B for the first time (i.e. it was not inherited and therefore has no prefix in B), we extend the sum of morphisms $\pi^{B^g}_{name}$ by $\pi^{P \times B^p}_{B::name}$.

□

Proposition 1 Let the product class $P \times B$ be a subclass of B (there exists the projection $\pi_{P \times B:B} : P \times B^p \to B^p$). Then, each access to the attribute $name$ of the $P \times B^p$-object, defined in the context of the base class B^g as the sum morphism $\pi^{B^g}_{name}$ or $\pi^{B^g}_{X::name}$ (with respect to the Algorithm 2), is unique. □

This proposition follows from the Algorithm 1 and 2. We show that for each attribute access in the context of B there exists an unique projection in $P\times B$ denoting the same attribute. The core of the problem is to find the corresponding prefixes of the attributes in the subclasses. We have four possibilities how the prefixes can change. If the attribute $name$ was inherited from X to B and does not cause any conflict in B, we must find out what prefix this attribute will have in $P\times B^p$. If $name$ does not cause any conflict in $P\times B^p$, we know that the prefix B can be used to distinguish this attribute. Therefore, $name$ can be accessed by using $\pi^{P\times B^p}_{B::name}$. Otherwise, we should use $\pi^{P\times B^p}_{S::name}$, where S is the source class for this attribute, because every conflicting attribute is distinguished by the first step of the naming algorithm using its unique source class..

If there was any conflict caused by $name$ in B, $name$ is distinguished using the prefix denoting the source class of this attribute. This prefix will never change in the subclasses. Therefore, $name$ can be accessed in the context of B by using the sum of morphisms $\pi^{B^g}_{S::name}$ only and we add the projection $\pi^{P\times B^p}_{S::name}$. Finally, if the attribute $name$ was defined in B, B is the unique source for this attribute. This means that we can always find the projection $\pi^{P\times B^p}_{B::name}$ in $P\times B$.

Recall that any attribute name conflict is caused only by the need of the consistent flat naming scheme for the classes in the inheritance hierarchy. It can be resolved in the compile time through the static typing. In the case of product classes the naming algorithm is applied to determine which attribute is meant in different contexts, but the names themselves cannot be used to access attributes directly through product classes.

6 Conclusion and Further Investigation

This paper has discussed an introduction of flexible objects into OODB model as required by the real world situations. It has presented an approach to the formal object-oriented models based on the *category theory* that provides theoretical tools to model the semantics of operations *new*, *acquire* and *discard*.

In terms of LDM, we have investigated issues concerning the name conflicts arising from the multiple class membership of objects. Two algorithms, which are essential to the definition of an object with the multiple most specific classes, were presented.

The *naming algorithm* was used to distinguish attributes of the same name in the classes defined by the multiple inheritance. We have defined the semantics of the operation *new*. The operations *acquire* and *discard* were proposed as the basic means for changing the roles of multiple class objects. Most importantly, we showed that structural conflicts concerning these objects can be resolved from the context determined by the static typing.

However, only the structural components of the objects were studied; a reformulation of the naming algorithm and product classes for methods represents an open investigation area. This can be easily done because methods are in LDM treated in a similar way as attributes; however, the problem of method dispatching makes this case more complicated. It seems that neither preferred class approach nor the argument specificity approach [2] satisfy the real needs of the database systems because the semantics of the dispatching itself still remains to be an open problem.

Acknowledgements

The ideas presented in this paper have been influenced by the discussions with Michal Máčel. The authors would also like to thank Alexander Meduna for his useful comments.

References

1. BARR, M., WELLS, C.: Category Theory for Computing Science. Prentice Hall, 1995.
2. BERTINO, E., GUERRINI, G.: Objects with Multiple Most Specific Classes. In Proceedings of 9th European Conference ECOOP'95 on Object-Oriented Programming, W. G. Olthoff, Ed., vol. 952 of LNCS, Springer, 1995, pp. 102–126.
3. CADIS, B., DISKIN, Z.: Algebraic Graph-Oriented = Category Theory Based. Manifesto of categorizing database theory. Tech. Rep. 9406, Frame Inform Systems, Riga, Latvia, 1994.
4. CADIS, B., DISKIN, Z.: Databases as graphical algebras: Algebraic graph-based approach to data modeling and database design. Tech. rep., Frame Inform Systems, Riga, Latvia, 1996.
5. CADIS, B., DISKIN, Z.: 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.
6. GOGUEN, J.: A Categorical Manifesto. Mathematical Structures in Computer Science, 1(1):49–67, 1991.
7. 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.
8. KOLENČÍK, P.: Classes in Object-Oriented Model as Categorical Objects. In Sborník prací studentů a doktorandů, FEECS TU of Brno, 1996, pp. 75–77.
9. KOLENČÍK, P.: Methods in The Object-Oriented Model as Sums of Morphisms. In Proceeding of International Conference MOSIS'97, vol. 2, ISBN 80–85988–17–8, 1997, pp. 83–88.
10. 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.
11. PIESSENS, F., STEEGMANS, E.: Categorical data-specifications. Theory and Applications of Categories, 1:156–173, 1995. Available at `http://www.tac.mta.ca/tac/`.
12. STROUSTRUP, E.: The annotated C++ reference manual. Addison Wesley, 1992.
13. TUIJN, C.: Data Modeling from a Categorical Perspective. PhD thesis, Antwerpen University, 1994.
14. VELHO, A. V., CARAPUCA, R.: From Entity-Relationship Models to Role-Attribute Models. In Proceedings of 12th International Conference ER'93 on the Entity-Relationship Approach, R. A. Elmasri, V. Kouramajian, and B. Thalheim, Eds., vol. 823 of LNCS, Springer, 1994, pp. 257–270.