

# Schema Evolution in ODMG

Regina Motz  
Instituto de Computación  
Universidad de la República  
Montevideo - Uruguay  
rmotz@fing.edu.uy

## Abstract

Multiple inheritance has been repeatedly identified as an important object-oriented modeling construct for advanced applications. Today most emerging standards such as ODMG object model and UML have some support for multiple inheritance. However, no relevant work has been done on schema evolution of an object model that has multiple inheritance. In this paper, we present a formal representation for ODMG schemas in terms of so-called *schema graphs*, which offers a formal basis for the study of schema evolution. Moreover, we analyze the evolution taxonomy for object oriented models in terms of multiple inheritance. Schema evolution is concerned with the study of schema changes and how they may affect other parts of the database. Thereby, we define the semantics of each schema change by explaining its impact on the rest of the schema, instances and methods. We provide a comprehensive pre-conditions solution in order to maintain schema consistency. In this direction we achieve the notion of a *proper schema graph*, a restricted form of schema graph obtained by introducing certain constraints in order to specify a consistent ODMG schema.

Keywords: Schema Evolution, Object Oriented Data Models, ODMG, Schema Graph.

## 1 Introduction

Multiple inheritance has been repeatedly identified as an important object-oriented modeling construct for advanced applications. Moreover, databases and software systems have complex structures which are likely to continually undergo changes during their lifetime. This is certainly true for OODBs, which have been mainly developed to model highly dynamic applications scenarios where not only the data, but also their structure (i.e. *schema*) is subject to change. Today most emerging standards such as ODMG object model and UML have some support for multiple inheritance. However, no relevant work has been done on schema evolution of an object model that has multiple inheritance. In this paper, we present the schema evolution taxonomy for object oriented models with multiple inheritance.

Schema evolution is usually applied due to two reasons: (i) as the result of a bad schema design involving the removing of anomalies and redundancies in the schema, or (ii) because the domain being modelled is evolving. These reasons produce schema modifications of *structural* nature. Most object-oriented database systems (OODBS) today support some form of re-structuring support via schema evolution [14, 17, 4, 15, 7]. However, this existing support of OODBs is limited to individual types only. More complex changes, such as combining two types or redefining the relationship between two types, are either very difficult or even impossible to achieve with current commercial database technology. In fact, most OODBs would typically require the user to write ad-hoc programs to accomplish such transformations. In the last few years, research has begun to look into this issue of complex changes [1, 3, 9]. Nevertheless, this work is again limited by providing a set of evolution operations that work only on single inheritance hierarchies. To address this limitation, in this paper, we analyse and address the repercussions of considering multiple inheritance on an object model on the existing taxonomy of schema evolution primitives.

Our development is based on the ODMG standard [11] which today is the only source for a reliable basis to develop open OODB applications. The ODMG standard defines an object model which supports the core features of an object data model in terms of the basic concepts such as object, object identity, class and multiple hierarchy.

Peters and Ozsu [16] have introduced a sound axiomatic model to formalize and compare schema evolution modules of OODBs. We utilize their notations with our extensions for the description of our invariants and primitives.

In summary, the contributions of this paper are: (i) We characterize the consistency problems that arise from the use of the existing evolution primitives on ODMG considering multiple inheritance, (ii) We provide a solution based on pre-conditions for maintaining consistency while providing evolution; and (iii) Schema evolution is concerned with the study of schema changes and how they may affect other parts of the database. Thereby, we define the semantics of each ODMG schema change by explaining its impact on the rest of the schema, instances and methods.

The remainder of the paper is organized as follows. Section 2 presents a brief description for ODMG schemas. In Section 3 we give a formalization of ODMG in terms of a *schema graph*. In Section 4 we show the notion of consistent schema evolution for ODMG. Section 5 presents the different structural schema changes and defines the semantics of ODMG schema changes by explaining its impact on the rest of the schema, instances and methods. Section 6 concludes the paper giving some final remarks.

## 2 ODMG Standard: The Object Model

This section briefly presents the main features of ODMG Data Model. The ODMG (Object Database Management Group) Data Model [11] is an extension of the OMG Data Model [6]. This model proposes a standard for object-oriented databases. It provides the basis for product-independent interfaces and thus portable applications.

A complete definition of ODMG, and a discussion of its features, may be found in [10, 11].

**Objects:** The term *object* comprises both literals and mutable objects. Literals are immutable, i.e. their contents cannot be changed. (Mutable) Objects are identified by an object identifier (*oid*) and their attribute values can be modified. The domain of an object identifier is the database in which the object exists. In the reminder we use *value* as synonym for immutable object and *object* as synonym for mutable object.

**Types:** Types specify structure and behavior of an object. (Same as the abstract data type concept in programming languages.) Structure is defined as a set of *properties*, behavior is defined as a set of *operations*. A type has one *interface* and one or more *implementations*. The interface defines the external interface supported by instances of the type (their properties and the operations). An implementation, defines *data structures* in terms of which instances of the type are physically represented and the *methods* that operates on those data structures to support the externally visible state and behavior defined in the interface. The types are organized in a type lattice. Each type inherits structure and behavior from all its supertypes. The *extension* of a type is the set of all objects of the type. The type of an object is determined at creation time, i.e. each object is a *direct instance* of exactly one type, and at the same type *indirect instance* of all its supertypes. A *key* (attributes) can be defined in a type to specify the unicity of an attribute group in the extension, it provides a surrogate for alternative object identity.

**Classes:** Classes implement a type and have an extension. Although the interfaces are the same for all instances of a type, the implementations may differ. Classes can be organized hierarchically by specialization. Each object created in a class has the type which is implemented by this class. It belongs to the extension of its creation class and to each extension of the creation class' superclasses. (When we don't need to distinguish between the type and a class of a type we will refer generically as *class*.)

**Properties:** Two kinds of property are defined in the model - *attribute* and *relationship*. **Attributes** define the structure (or state) of an object type. Attributes can be marked as keys to provide a surrogate for alternative object identity; keys specify the unicity of an attribute group in the extension. **Relationships** are defined between two mutable object type with the ability to define 1:1, 1:M, and M:N cardinalities for them, guaranteeing referential integrity. Relationships themselves have no names. Instead, named traversal paths are defined for each direction of traversal.

**Operations:** They define object behavior. For each operation, an *operation signature* is included in the object type definition by the type programmer. The signature includes the argument names and types, exceptions potentially raised, and types of the values returned, if any. Operations are always defined on a single object type. Operation names need be unique only within a single type definition. They are implemented in a target language for which an ODMG mapping is defined, e.g. C++.

**Schemas:** Schemas consist of a set of classes, whose extents are stored in a single database, and the relationships between them. The schema defines the contents and structure of the database.

**Inheritance:** Types may be organized into a graph of subtypes and supertypes. A subtype inherits all of the characteristics of its supertypes. It may also define additional characteristics that apply only to its instances. The intention is that an instance of a subtype may be treated like an instance of each of its supertypes. An instance of the subtype supports all of the state and behavior of the supertype as well as new state and/or behavior unique to its more specialized nature. The subtypes may extend or redefine the properties defined by its supertypes. Multiple inheritance is supported. This raises the possibility that a type will inherit characteristics that have the same name (but different semantics) from two different supertypes. This form of name conflict is handled by requiring the inheriting type to redefine the name of one of the inherited characteristics.

### 3 Formalizing ODMG

This section presents a formal representation for ODMG schemas, called *Schema Graph*. We also present a restricted form of it, called *Proper Schema*, obtained by the introduction of additional constraints to a schema graph in order to specify a consistent ODMG-Schema. We conclude this section showing a formal representation for methods and instances.

#### 3.1 Schema Definition

A Schema Graph is a labelled directed graph that captures the inheritance hierarchies and relationships of a schema. It has two kinds of vertices (classes and immutable types) and two kinds of edges (specialization and relationship edges). A specialization edge is unlabelled and represents an inheritance relationship between two classes. Relationship edges are labelled and represent the properties of a class (attribute edges) or a semantic relation between classes.

**Definition 3.1 (Schema Graph)** A Schema Graph is a quadruple given by  $G = (V, E, S, K)$  where

- $V$  is a finite set of vertices  $V = C \cup T_I$ , where  $C$  is a set of mutable types, also called classes, and  $T_I$  is a set of immutable types;
- $E$  is a finite set of relationship and attribute edges,  $E = E_R \cup E_A$ , where  $E_R$  is the set of relationship edges,  $E_R \subseteq C \times L_E \times L_E \times C$ , and  $E_A$  is the set of attribute edges,  $E_A \subseteq C \times L_E \times V$ , with  $L_E$  a finite set of edge labels, each described by a character string.
- $S$  is a finite set of specialization edges,  $S \subseteq C \times C$ .
- $K$  is a function which associates cardinality constraints to relationship edges and attributes edges,  $K = K_R \cup K_A$ , where  $K_R : E_R \rightarrow \{[1, 1], [1, N], [N, 1], [N, N]\}$ , and  $K_A : E_A \rightarrow \{[1], [N]\}$ .

Given a schema graph  $G = (V, E, S, K)$  we use the following notational conventions:  $p \xrightarrow{a} q$  denotes a generic edge  $e \in E$  (this notation will be used in those cases where it does not matter whether  $e$  is a relationship or an attribute edge);  $p \xleftrightarrow{a} q$  with  $a \equiv (\alpha, \beta)$  denotes a relationship edge  $(p, \alpha, \beta, q) \in E_R$ ;  $p \xrightarrow{a} q$  denotes an attribute edge  $(p, a, q) \in E_A$ , and  $p \implies q$  denotes a specialization edge  $s \in S$ , such that  $p$  is a specialization of  $q$ .

The graphical representation use the following conventions: classes are represented by boxes with its name inside of the box, inheritance is represented by arrows. Each arrow connecting two classes starts at the subclass and ends at the superclass. Attributes are specified by arrows from the class to the attribute's domain with the attribute name on the arrow. Relationships are represented by double-directed arrows. The cardinality permitted by the relationship type is indicated by the following arrows:  $\longleftrightarrow$  one-to-one,  $\longleftarrow\!\!\!\!\!\rightarrow$  one-to-many and  $\longleftrightarrow\!\!\!\!\!\rightarrow$  many-to-many.

Not every arbitrary schema graph specifies a valid ODMG schema. The following restrictions are required to hold.

**Definition 3.2 (Proper Schema)** *A schema graph  $G = (V, E, S, K)$  is a proper schema iff it satisfies:*

1. *Uniqueness in the context of a class. The occurrence of a relationship or attribute in the context of a class is unique. In other words, the labels of the edges that incide a class must be unique. That is, for every  $p \in C, q, r \in V$ ,  $p \xrightarrow{a} q$  and  $p \xrightarrow{a} r$  implies  $q \equiv r$ .*

2. *Acyclicity of subtyping. Class specializations are acyclic.*

$$p \implies r_1 \implies \dots r_i \implies \dots r_{i+1} \implies q \text{ implies } q \not\Rightarrow p.$$

3. *Monotonicity of Inheritance. Specialization is preserved along equal attributes/relationships. That is, if  $r$  is a specialization of  $p$ , and on  $p$  and  $r$  holds certain attribute/relationship with  $q$  and  $s$  respectively, then  $s$  is a specialization of  $q$  and the attribute/relationship between  $r$  and  $s$  is at least as specific as the one between  $p$  and  $q$ :*

$$p \xrightarrow{a} q, r \xrightarrow{a} s \text{ and } r \implies p \text{ implies } s \implies q \text{ and } K(r \xrightarrow{a} s) \leq K(p \xrightarrow{a} q)$$

being  $\leq$  the standard ordering between natural numbers.

The two kinds of edges of a schema graph induce two kinds of reachability relations. First, we define an inheritance reachability relation to connect sub- and superclasses in a schema graph.

**Definition 3.3** *Let  $G = (V, E, S, K)$  be a proper schema graph. We denote by  $\xRightarrow{+}$ , the transitive closure of the specialization relation.*

The  $\xRightarrow{+}$  relation can be used to compute, for a given class, the set of its sub- and superclasses.

**Definition 3.4** *Let  $G = (V, E, S, K)$  be a proper schema graph. We denote by  $\xleftrightarrow{+}$ , the transitive closure of the relationship relation.*

**Definition 3.5** *Let  $G = (V, E, S, K)$  be a proper schema graph. Between two classes  $p$  and  $q$  holds an inherited relationship, denoted by  $\Rightarrow\!\!\!\!\!\rightarrow\!\!\!\!\!\leftarrow$ , if there exist  $w, w' \in C$  such that  $p \xRightarrow{+} w$ ,  $w \xleftrightarrow{+} w'$  and  $w' \xleftarrow{+} q$ .*

The concept of a path in a schema graph can now be naturally defined from the above notion of reachability.

**Definition 3.6 (Path)** *Let  $G = (V, E, S, K)$  be a proper schema graph. For every  $u, v \in V$ , we say that there is a path between  $u$  and  $v$ , denoted by  $u \rightsquigarrow v$ , if one of the following cases holds:*

1.  $u, v \in C$  and  $u \xRightarrow{+} v$ .

2.  $u, v \in C$  and  $u \xleftrightarrow{+} v$ .
3.  $u, v \in C$  and  $u \nrightarrow v$ .
4.  $u, v \in V$ , and  $\exists p \in C$  such that  $p \rightarrow u$  and  $p \leadsto v$ .
5.  $u, v \in V$ , and  $\exists p \in C$  such that  $u \leadsto p$  and  $p \rightarrow v$ .
6.  $u \in C$ ,  $v \in V$ , and  $u \rightarrow v$
7.  $u \in V$ ,  $v \in C$  and  $v \rightarrow u$

Note, however, that for two given vertices there may be more than one path.

### 3.2 Instance Definition

The instance of a schema consists of a set of interrelated *objects*, which are collected into *classes*. Consequently, we model instance by a triple  $I = (O, Eo, M)$ , where  $O$  is a set of objects,  $Eo$  is a set of object-relationships and  $M : O \rightarrow C$  is a mapping which partitions the set of objects into classes.

**Definition 3.7 (Extension of a class)** *The extension of a class  $p$  is the set of objects which belong to all the subclasses of  $p$  (included  $p$ ).*

$$Ext(p) := \{o \in O \mid M(o) \xRightarrow{*} p\}$$

**Definition 3.8 (Direct Extension of a class)** *The direct extension of a class  $p$  is the set of objects which belong to  $p$ .*

$$DExt(p) := \{o \in O \mid M(o) = p\}$$

**Lemma 3.1** *For every  $o \in O$  and  $p, q \in C$ , if  $M(o) = p$  and  $o \in Ext(q)$  then  $p \xRightarrow{*} q$ .*

As next we formalize the relationship between instance and schema (we use similar abbreviations and notational conventions to the ones used at the schema level).

**Definition 3.9 (Instance satisfying a Schema)** *Let  $G = (V, E, S, K)$  be a proper schema,  $O$  a set of objects and  $E_O \subseteq O \times L_E \times O$  a set of object-relationships. An instance  $I = (O, E_O, M)$  satisfies the schema  $G$  iff*

#### 1. Relationship for each object-relationship

(a)  $\forall o_p, o_q \in O$ ,  $o_p \xrightarrow{a} o_q$  implies  $\exists p, q \in C$ , such that  $o_p \in Ext(p)$  and  $o_q \in Ext(q)$  and  $p \xrightarrow{a} q$ .

(b)  $\forall p, q \in C$ ,  $p \xrightarrow{a} q$ ,  $\forall o_p, o_q \in O$ ,  $o_p \in Ext(p)$  and  $o_q \in Ext(q)$  implies  $o_p \xrightarrow{a} o_q$ .

2. **Cardinality constraint**  $\forall o_p, o_q, o_{q'} \in O$ ,  $\forall p, q \in C$ ,  $p \xrightarrow{a} q$  with  $K(p \xrightarrow{a} q) \leq [N, 1]$ , if  $o_p \xrightarrow{a} o_q$  and  $o_p \xrightarrow{a} o_{q'}$ , then  $o_q \equiv o_{q'}$ .

## 4 Schema Consistency

The result of a schema change should be a new schema without inconsistencies. However, the evolution of a schema may produce various inconsistencies in other parts of the database. There are two fundamental problems to consider, namely structural and behavioral consistency [5]. *Structural consistency* guarantees the correctness of the performed schema changes and reflects their impact on the instances of the database. *Behavioral consistency* refers to the impact of schema changes on existing programs, i.e. each method must continue to respect its signature and its code must not result in run-time errors or unexpected

results. Since behavioral consistency is beyond the scope of this work, we will provide only a restricted support for operation modifications, namely only changes in name and signature. Even though, we can identify the operations affected by a schema modification and generate hints for code that should be modified by hand. Those operations that may produce unexpected results after a schema modification shall be marked as potentially invalid and the user warned about this.

The traditional approach to the structural consistency problem is to define a number of constraints, called *invariants*, that need to be preserved by the schema modifications in order to ensure the consistency of the resulting schema. Following [16], a consistent schema is a directed acyclic graph class structure that satisfies the uniqueness of property definitions in the context of a class (acyclicity invariant, distinct name invariant and domain compatibility invariant, in other words, monotonicity of inheritance extended to attributes). The inheritance process is constrained by a set of invariants (same origin invariant and multiple inheritance invariant). In addition, all attributes, relationships and methods must make reference to classes which actually exist (referential integrity invariant).

Schema invariants must hold in every state of the database. Rather than checking schema consistency after performing a schema change, a set of preconditions should be defined for each modification in order to ensure schema consistency preservation.

One could expect that the semantics of structural modifications should be independent of the chosen implementation method. Unfortunately, for much of the work in the area of schema evolution this is not the case, since there is no unified semantics for modifications. As mentioned in [8], there are, in general, many possible interpretations of a particular schema modification. For example, in an object-oriented data model supporting optional attributes, suppose that we change an attribute of a class from being optional to become required. There are a number of ways in which such schema modification may be reflected at the instance level: we could insert a default value for the attribute wherever it is omitted, or we could simply delete any object from the class for which the attribute is missing. The set of invariants that define a consistent schema thus provides a basis for a semantics of schema specifications. This shows that in addition to having a set of invariants for defining when a schema is consistent, one needs a precise definition of the semantics of each basic schema change at the three levels: schema, method, and instance level.

At the instance level, instances populating the extension of a class have to correspond to the actual class definition. Hence, whenever a schema modification occurs, the instances in the database have to reflect such change. There are four instance operations associated with schema evolution: **creation**, **derivation**, **modification** and **deletion** [9]. New instances can be created or derived from old ones. Derivation requires a source, a destination, and a derivation rule. The source identifies the class location of the old instance, while the destination the class location of the new instance. The derivation rule is a function to be applied to the old values to compute the new ones. Modification is similar to derivation, but source and destination are the same and the instances preserve their *object identity* (*Oid*). In most cases of schema evolution, instances must retain their identity. This is necessary, e.g., to preserve relationship integrity. But there are also changes in which the user might decide that identity preservation no longer applies. For instance, in the partition of a class into two new classes, each instance must be split into two, neither of them retaining the same identity as the original one, but both derived from it by applying different derivation rules.

## 5 ODMG Structural Schema Modifications

In general, structural modifications of an object-oriented database are grouped into two types: (i) changes to a class, and (ii) changes to the class lattice. Changes to a class consist of changes to its properties, such as changing the name or domain of an attribute, adding or dropping an attribute or method, etc. Changes to the class lattice includes e.g. adding or dropping a class, and changing the superclass/subclass relationship between a pair of classes.

Structural modifications can also be categorized according to their results [12, 2]. When changes imply a re-building of the schema due to the use of different constructs to represent the same information, we talk about *pure structural changes* or *capacity preserving changes*. For example, when an attribute is moved to another class or a class is renamed. On the other hand, changes may imply loss or addition

---

## 1. Changes to a class.

- (a) Rename a class, its attributes, relationships or operations. *(P)*
- (b) Add a new attribute, relationship or operation. *(A)*
- (c) Remove an existing attribute, relationship or operation. *(R)*

## 2. Changes to the class lattice.

- (a) Add a new class. *(A)*
- (b) Remove an existing class. *(R)*
- (c) Add a specialization edge. *(P)*
- (d) Remove a specialization edge. *(P)*

Figure 1: A Taxonomy of Schema Changes.

---

of information capacity, e.g. when removing an attribute from a class or adding a new one, respectively. When modifications imply a loss of information capacity we talk about *capacity reducing changes*, whereas in the opposite case we talk about *capacity augmenting changes*. In Figure 1, we give an account of a comprehensive set of schema changes for ODMG related with their action on information capacity where (A) indicates a capacity augmenting change, (R) indicates a capacity reducing change and (P) indicates a capacity preserving change.

Due to space limitations, in the following we only present some examples of ODMG Schema Evolution. The specification of the complete structural schema changes can be found in [13].

**Add a specialization edge** This modification adds an inheritance edge between two classes. It is annotated as an information augmenting change.

**Preconditions:** The involved classes must already exist. The new specialization edge must not induce a cycle in the inheritance graph (Def. 3.2 Acyclicity of subtyping).

**Schema:** The properties provided by the new superclass (either inherited or locally defined) are propagated to all its subclasses; the same redefinitions errors as those presented for the addition of properties may arise here. For example, by the creation of an specialization edge from a class  $p$  to a superclass  $q$ , if the class  $p$  has an attribute whose name already exists in  $q$ , then the domain compatibility invariant would be violated unless the attribute corresponds to a specialization of the inherited one. Moreover, name conflicts may occur. A name conflict occurs when a subclass already inherits a property with the same name but coming from a different superclass. This is illustrated in Figure 2. The initial schema shows a class `MobilHome` that inherits the attributes `stay` and `circulate` from the class `CarExc`. The addition of the specialization edge from `MobilHome` to the class `BicycleExc` produces a multiple inheritance conflict between the two attributes `BicycleExc.stay` and `CarExc.stay`. By the monotonicity of inheritance (Def. 3.2.3) the class `MobilHome` inherits both attributes `stay` from each of its superclasses (`CarExc` and `BicycleExc`), but, in order to maintain the structural consistency of the schema, the user is induced to include a redefinition of one of the conflicting attributes, e.g. “`MobilHome.hotel` redefines `BicycleExc.stay`”.

The attributes `circulate` from classes `BicycleExc` and `CarExc` do not present any conflict as both attributes have a common origin, namely `Excursion.circulate`. So, in this case the most specialized attribute is inherited, i.e. `BicycleExc.circulate`.

**Instances:** At the instance level the impact of adding a specialization edge is reduced to the addition of a set of properties, which leads to an instance modification operation. Default values must be provided by

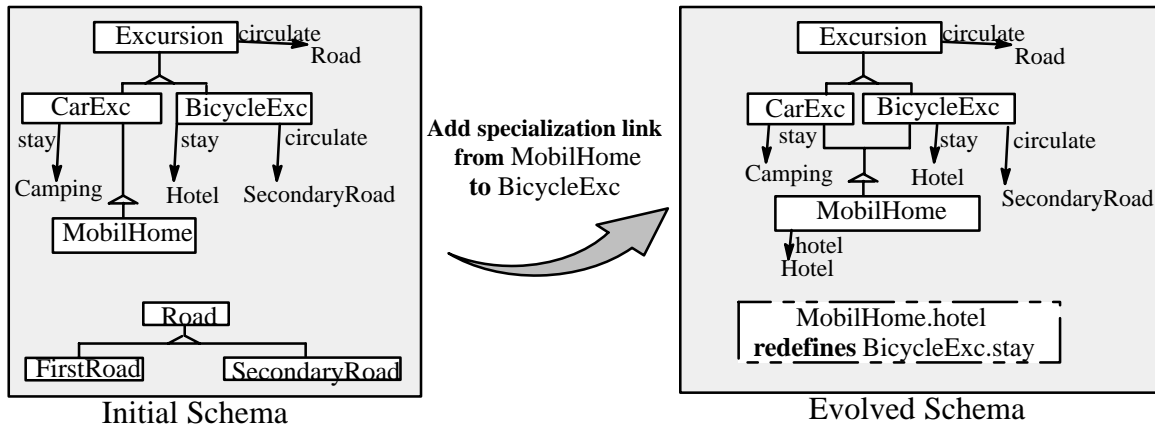


Figure 2: Addition of an specialization edge.

the user for the inherited attributes which are not redefined in the subclass. The propagation is stopped in case of locally redefinition of a property.

**Methods:** At the method level, this modification may induce a change of behavior in inherited methods that use redefined attributes. These methods are given to the user for manual modification.

**Remove a specialization edge** This modification removes the inheritance edge between two classes. Since some inherited attributes may be lost from the specialized subclass it is annotated as an information reducing change.

**Preconditions:** The specialization edge must exist.

**Schema:** Removing a superclass reference from a class  $p$  might lead to inconsistencies in the schema since somewhere in the schema the class  $p$  could be used as redefinition domain of some attribute. This kind of error is illustrated in Figure 3. Suppose that the specialization edge between `SecondaryRoad` and `Road` is removed. This leads to a domain compatibility invariant violation in the class `BicycleExc` since the redefinition of the `circulate` attribute is no longer valid. By default the system will remove such attributes.

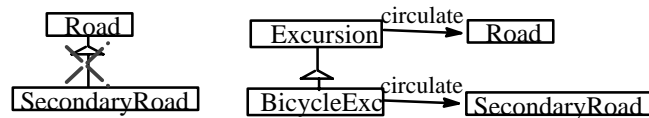


Figure 3: Conflict when removing a specialization edge.

**Instances:** At the instance level the impact of removing a specialization edge is reduced to the removal of a set of properties. The propagation is stopped in case of locally redefinition of a property or because they are still inherited through an alternative path.

**Methods:** This modification induces the removal of methods that use attributes that were inherited through the removed specialization edge.



**Add a new attribute, relationship or operation** The addition of a new attribute, relationship or operation  $x$  to an already existent class  $p$ , is an information augmenting change.

**Preconditions:** Since the same rules apply for the addition of both an attribute and a relationship we use here the generic term “property” to refer to both. Obviously, the property or operation to be added must have a distinct name, since we must preserve the uniqueness of properties and operations (Def. 3.2.1). Moreover, in case that the new property redefines an inherited one, the monotonicity of inheritance (Def. 3.2.3) needs to be maintained. This means that the new property must be a specialization of the inherited one. The same invariant must be maintained in all subclasses of the class where the new property is being added, i.e., properties locally defined must be specializations of the new one. (A relationship is more specialized than another if its cardinality is more restrictive than the one of the other). Figure 4 illustrates cases of valid and invalid additions of an attribute.

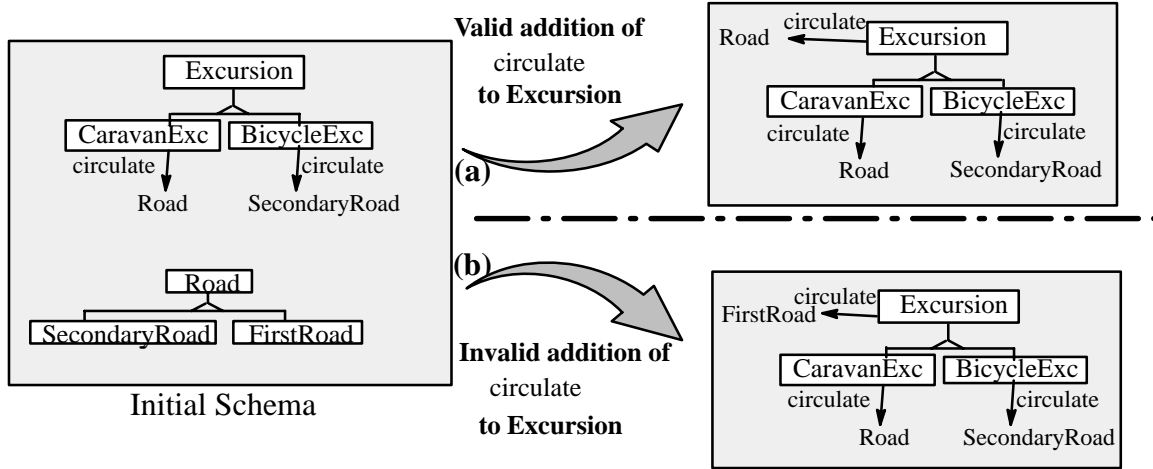


Figure 4: Addition of a new property.

The initial schema of Figure 4 shows a categorization of Excursions into CaravanExc and BicycleExc. Both have an attribute *circulate* which indicates the kind of road they can circulate. Roads in turn are categorized into SecondaryRoad, and FirstRoad. CaravanExc can circulate by all kind of roads whereas BicycleExc can circulate only by SecondaryRoad. The addition of an attribute *circulate* with domain Road to the class Excursion (Fig. 4.a) satisfies the domain compatibility invariant since it is a generalization of both attributes *circulate* from classes CaravanExc and BicycleExc. On the contrary, the addition of an attribute *circulate* with domain FirstRoad to the class Excursion (Fig. 4.b) leads to a downward redefinition error (since the FirstRoad class is neither a superclass of Road or SecondaryRoad). The same situation may occur by the addition of a relationship in relation to its cardinality. The addition of a new attribute, relationship or operation must verify uniqueness of name in context of a class and existence of its parameter types (referential integrity).

**Schema:** This modification has no impact on the rest of the schema.

**Instances:** This modification leads to an instance modification operation. Adding an attribute to a class implies its addition to all instances of the class and its propagation to all subclasses. Such propagation is stopped in case of a locally redefinition of the attribute because a local property overrides inherited ones. Because null values are not allowed in ODMG, the addition of an attribute to a class must be accompanied by a default value for that attribute. The addition of a relationship or an operation has not impact on the instance level.

**Methods:** The addition of a new class, property or operation to a class presents no impact on methods when there is no inherited property with the same name in that class. Otherwise, all methods referring to the inherited property are marked in order to warn the developer since the modification may have changed the behavior of the method. However, there may be also some other kind of methods, for example a method that “*prints all attributes of a class*”, which need to be modified. One possible solution is to provide the user with a list of methods associated to the modified class and let her/him decide which methods require manual adaptation.

**Remove an existing property or operation** This modification deletes a property or operation from a class. Obviously, it is annotated as an information reducing change.

**Preconditions:** The remove of a property or an operation  $x$  from a class  $p$ , may only be performed on the class that define them:  $p \in C$  and  $\exists q \bullet p \xrightarrow{x} q$  or  $x \in Op(p)$

**Schema:** This modification may produce a rename of a redefined inherited property in cases of multiple inheritance conflicts.

**Instances:** Removing an attribute from a class implies an instance modification operation. The removed attribute must be deleted from all instances of the class. The remove operation is propagated to all subclasses until a subclass has the same attribute locally defined. Removing a relationship implies the deletion of all object-relationships from it. Removing an operation has not impact on the instance level.

**Methods:** All methods referring directly or indirectly to the removed property or operation are also removed. If there is an inherited property replacing the removed one, then the modification may only produce a change in the behavior of the method. In this case the validity of the method must be confirmed by the user.

## 6 Final Remarks

In this work we have addressed the issue of schema evolution in case of multiple inheritance. Within this context we have analyzed the most relevant effects of schema evolution operations on other elements of the object model. We propose a schema graph formalization of ODMG in order to specify schema consistency of schema evolution operations in terms of pre-conditions. We have already developed a prototype that automatically transforms from ODL schemas to Schema Graphs and vice-versa. We are currently working in an implementation of the taxonomy of schema evolution operations considering the multiple inheritance with pre-conditions as proposed in this work.

## References

- [1] Rundensteiner E. A., Claypool K. T., Li M., Chen L., Zhang X., Natarajan C., Jin J., De Lima S., and Weiner S. SERF: ODMG-Based Generic Re-structuring Facility. In *Demo Session Proceedings of SIGMOD 99*, 1999.
- [2] Paul L. Bergstein. Maintenance of Object-Oriented Systems during Schema Evolution. *Theory and Practice of Object Systems*, 3(3):1–28, 1997.
- [3] Philippe Bréche. Advanced Primitives for Changing Schemas of Object Databases. In *CAiSE'96*, Heraklion, Crete, May 1996. Springer Verlag.

- [4] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In , editor, *Object Oriented Concepts, Databases and Applications*. ACM Press, 1989.
- [5] Christine Delcourt and Roberto Zicari. The Design of an Integrity Constraint Checker (ICC) for an Object-Oriented Database System. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, number 512 in Lecture Notes in Computer Science, Geneve, Switzerland, July 1991. Springer.
- [6] Object Management Group. The Common Object Request Broker: Architecture and Specification. Document number 91.12.1, OMG, 1991.
- [7] Itasca Systems Inc. OODBMS Feature Checklist. Rev 1.1. Technical report tm-92-001, Itasca System, december 1993.
- [8] Anthony Kosky, Susan Davidson, and Peter Bunemann. Semantics of Database Transformations. Technical report, MS-CIS-95-25, University of Pennsylvania, USA, 1995.
- [9] Barbara Lerner. A Model for Compound Type Changes. Technical report, 95-095, University of Massachusetts at Amherst, october 1995.
- [10] M. Loomis, T. Atwood, R. Catell, J. Duhl, G. Ferran, and D. Wade. The ODMG Object Model. *Journal of Object-Oriented Programming*, 10(6), June 1993.
- [11] M. Loomis, T. Atwood, R. Catell, J. Duhl, G. Ferran, and D. Wade. *Object Databases: The ODMG-93 Standard*. Morgan Kaufmann, 1993.
- [12] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The use of Information Capacity in Schema Integration and Translation. In *Proc. of the 19th. VLDB Conf. Dublin, Ireland.*, 1993.
- [13] R. Motz. Semantics of Schema Changes in ODMG. Technical Report 17-02, PeDeCiBa - Uruguay, 2002.
- [14] O2 Technology. *O2 Reference Manual, Version 4.5* , 1994. Release 2.0.
- [15] Object Design, Inc. *ObjectStore User Guide, Chapter 9 Schema Evolution*, 1993. Release 2.0.
- [16] Randal J. Peters and M. Tamer Özsu. Axiomatization of Dynamic Schema Evolution in Objectbases. In *Int. Conf. on Data Engineering (ICDE-11)*, Taipei, Taiwan, March 1995. IEEE.
- [17] Versant Object Technology. Versant user manual. Technical report, Versant Object Technology, 1992.