Object Migration

Alberto O. Mendelzon
CSRI, University of Toronto
Toronto, Canada M5S 1A1
mendel@db.toronto.edu

Tova Milo*
CSRI, University of Toronto
Toronto, Canada M5S 1A1
milo@db.toronto.edu

Emmanuel Waller †
LRI, University of Paris-Sud
91405 Orsay, France
Emmanuel.Waller@lri.fr

Abstract

We study a mechanism that supports the migration of objects from one class of an OODB to another, thereby enabling us to model the same object playing different roles throughout its lifetime. Object migration may introduce typing conflicts due to the different typing constraints imposed by the classes. We present a coercion-like adaptation process that automatically resolves these conflicts. The process combines re-classification of objects and modification of attributes. We study the computational complexity of the problem, and show that the adaptation process can be performed efficiently in databases with covariant schemas.

1 Introduction

Two of the dogmas of object-oriented databases are that objects have existence independent of their value (the principle of *object identity*) and that objects are grouped into classes that capture their commonalities. But there is a logical next step that has not been studied in depth yet: why not let objects retain their identity, not only when they change value, but also when they change their class membership? There are many situations where this is very useful in modeling the world in a natural way. Consider for example the dealings between a bank and a person; one and the same person may at different times be viewed by the bank as a customer, a borrower, a creditor, an employee, a supplier, etc. Since class membership is the main mechanism that most object-oriented models provide for assigning roles to objects, changing roles can be modeled by an object *migrating* from one class to another.

*Supported by the Institute for Robotics and Intelligent Systems.

†Part of this work was done while the author was at INRIA-Rocquencourt, France. Work partially supported by Esprit BRA Project Fide2

The general problem we would like to study, then, is what mechanisms are needed to support the same object playing different roles, and hence migrating among different classes, throughout its lifetime. Instead of considering objects that play arbitrary sets of roles that grow and shrink in time, we study first the simpler situation in which an object belongs to a unique class (and its superclasses) at a given time, but may change class membership dynamically. We discuss at the end how our techniques can be extended to the general case. Note that authors who treated the more general case [RS91, ABGO93, SZ89, Fis87] remained mostly at the conceptual and descriptive level. An exception is Su [Su91], whose model allows objects to belong to several classes simultaneously; but the problem he considers is very different from ours, namely the analysis of sequences of migrations produced by a transaction.

The specific problem we address is: how can we adjust the database in the presence of object migration so that all typing constraints imposed by the schema remain valid? The typing constraints we are interested in are of the form "property P of any object of class C must have a value of class C'". For example, "the manager of a department must be an employee." These constraints affect object migration in two ways. First, different classes will have different type constraints. Migrating objects from one class to another requires adaptation of the current state of the object to the new constraints, for example, by changing some references to point somewhere else, or deleting some references, or recursively moving the referenced object to some other class. Second, note that other objects in the database can refer, through their own attributes, to the object being moved. After the move, these references may violate the type constraints of these other objects. Such conflicts must also be resolved.

We can see two approaches to the resolution of the type conflicts caused by object migration. The first one is to ask the user to provide, for each allowable object move, a full specification of the state adaptation process [RS91, ABGO93] and to explicitly resolve every conflict. The second approach is to allow the user to provide only a partial specification, and to resolve other conflicts automatically, using a coercion-like adaptation mechanism.

Having to resolve all the typing conflicts caused by an object migration may be a heavy burden on a designer or application programmer. There might be different ways to resolve each conflict, and different choices can lead to additional new conflicts. For example, to resolve one conflict, it might be necessary to move another object to a new class. This, in turn, can cause other conflicts that need to be solved. A facility that chooses the best conflict resolution policy for each allowable move seems a very desirable alternative. This automatic adaptation mechanism is the subject of this paper. We present in the following several techniques that resolve conflicts by re-classifying objects, or by changing the contents of conflicting attributes.

There are two problems here, a semantic one and a computational one. The semantic one is: given an initially correct database state and an object changing from one class to another, what should the correct database state be (if any) after the move? We address the semantic problem by noting that object migration is a kind of update, and applying the theory of updates proposed in [KM91]. In that paper, correct updates are characterized as those that choose from all acceptable database states the ones that are *closest* to the initial one, under a suitable definition of distance. We introduce a particular notion of closeness by defining a *distance* metric that measures the difference between database states, and use it to determine the nearest consistent state to the original one in which the moved object is in the desired class. (Similar approaches to update/revision of knowledge bases appear in [Bor85, Dal88, Sat88, Win88]).

Now the computational problem appears: can we do this efficiently? In general, we show the problem to be Co-NP-Complete in the size of the database instance. But for an important class of schemas, the *covariant schemas* [AKW90], we give a linear time algorithm. We introduce different *distance* measurements and show how they can be used to control the adaptation process. We study the relationship between adaptation techniques based on different distance measurements. In particular we show that for covariant schemas some of them coincide, thus adaptation can be performed efficiently.

The adaptation process may affect the execution of methods. We show that for consistent method schemas, the technique cannot cause failure in method execution. We also consider possible extensions and applications of the technique. In particular we explain how it can be used to handle schema evolution.

The paper is organized as follows. In Section 2 we briefly introduce the data model. In section 3 we present an adaptation technique based on re-classification of objects. In the next two sections we study the complexity of the technique. General database schemas are considered in Section 4, and covariant schemas in Section 5. A more general adaptation technique is presented in section 6. The effect of adaptation on method execution is studied in section 7. Possible extensions and applications of the techniques are considered in section 8. Finally, conclusions are presented in section 9.

2 Preliminaries

In this section we briefly introduce the data model. We use the data model of [AKW90], extended with set values. The presentation below is rather informal. (For formal definition see [AKW90].) We use in this work a specific data model, but the results can be easily adapted to other object oriented data models.

We have an isa hierarchy of classes $\{\mathcal{C}, \leq\}$, where each class represents a set of objects. The isa hierarchy $\leq$ defines a partial order on classes, where $c_1 \leq c_2$ iff c_1 is a subclass of c_2 (every class is a subclass of itself, i.e $c_i \leq c_i$). We assume the existence of a most general class – the *object* class.

Every object o in the database has one most specific class c[1]. For a given database instance d, we denote this class by $C(o, d)$. For brevity, we omit d whenever it is clear from the context, and use $C(o)$.

Every class $c \in \mathcal{C}$ has an associated set of typed attributes $A^c = \{A_1 : t_1, \ldots, A_n : t_n\}$, where t_i is a class name c_i, or a set expression $\{c_i\}$ for some $c_i \in C$ [2]. A class also has a set of methods [3]. Attributes and methods are inherited along the class hierarchy. Attribute and method names can be reused in different parts of the hierarchy, i.e. there is overloading. We assume that no conflicts are caused by multiple inheritance.

The state of an object o is defined by the value of its attributes. The contents of the attributes should obey the type restrictions imposed by the class $C(o)$ to which the object belongs. i.e. an attribute A_i of type c_i (or $\{c_i\}$) can contain an object o_1 iff $C(o_1) \leq c_i$. A database instance where the attributes of all the objects obey the type restrictions is called **consistent**.

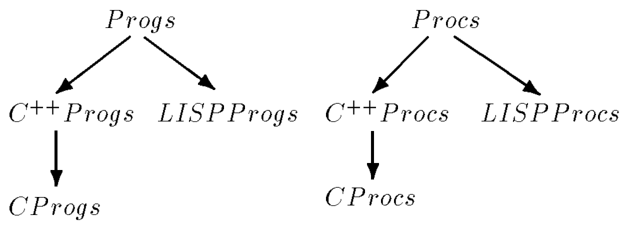
Example 1:
Consider a database designed for software engineering, containing information about programs and procedures. In particular, it records which programs call which

[1] Models where an object can simultaneously belong to several independent classes are considered in Section 8.

[2] We have here only objects. Data models that support (complex) values are considered in section 8.

[3] Attributes correspond to the "base" methods of [AKW90], and methods are the "coded methods" there.

procedures. Part of the class hierarchy is presented below:



Some of the classes are defined below.

$$\begin{array}{ll} Progs & = [Pname : N,\ Calls : \{Procs\}] \\ CProgs & = [Pname : N,\ Calls : \{CProcs\}] \\ C^{++}Progs & = [Pname : N,\ Calls : \{C^{++}Procs\}] \end{array}$$

Assume we want to add a new program to the database, but we do not know in what language it is written. We can phone the *programming languages expert*, and while waiting for the expert add the program to the *Progs* class (with the corresponding procedures in the *Procs* class). The expert can read the program code and decide it is a C program. Thus the object representing the program should be moved to the $CProgs$ sub-class. Note that the type constraint on the *Calls* attribute is different in *Progs* and $CProgs$, and that after the move this type constraint is violated by the moved object, which still points to objects in class *Procs*. To correct the situation, we can move the procedures of the program to the $CProcs$ sub-class of *Procs*.

After reading again the source code of the new procedures, we may discover that one of them actually uses some C^{++} constructs. Thus it must be moved to the class of $C^{++}Procs$. Note that the procedure object is pointed to by the program object via the *Calls* attribute. When the procedure is moved, this attribute is no longer correctly typed, since the *Calls* attribute of $CProgs$ must point to objects in $CProcs$. The inconsistency can be resolved either by removing the procedure from the *Calls* attribute of the program, or by moving the program object to the $C^{++}Progs$ class. The second solution is preferred since it does not lose the information that a certain program calls a certain procedure.

This example illustrates object migration up and down the class hierarchy. A similar scenario may take place when moving an object between two classes that are not hierarchically related to each other (*e.g.* when promoting a junior employee to be a manager.)

Note that, in both cases, the user decides to move an object from one class to other, and the movement of other objects results from that decision. We refer to the first move as *object migration*, and to the other moves as *database adaptation*.

Observe that we implicitly assumed above that the attribute names are significant, that is, if two classes have the same attribute name, then the attribute has the same semantics in both. The attribute *Calls* is used in all the classes to record the procedures called by the program. This justifies preserving the value of the attribute when moving an object from the class *Progs* to $CProgs$, at the cost of moving other objects to new classes. Similarly, the value of the attribute *Pname* in class *Progs* becomes the value of *Pname* in the new class $CProgs$.

Also observe that the new state of the database is "natural," in the sense that we have only moved those objects that it was strictly necessary to move to achieve a consistent state. In general, there may be many ways to fix the violation of the typing constraints caused by the object migration. The most trivial one (though clearly undesirable) is to move the object back to its original class. Other possibilities involve moving other objects, or changing the values of the inconsistent attributes. In general we would like the database to change "as little as possible." To characterize this notion we define various metrics of database change and explain how they can be used to automatically compute the "closest" consistent database state that accomplishes the migration.

3 Adaptation by Re-Classification

Suppose we want to move object o with attribute A to a new class c' that also has an attribute named A. We assume attribute names are meaningful, and hence we would like to retain the old value of A as the new value of A for the moved object. As we stated in the Introduction, this may cause two kinds of conflicts. First, the old value of A may not be compatible with the type of A in the new class. Second, other objects may have attributes pointing to object o, and these attributes may be prohibited by the schema from pointing to an object of type c'.

There are several things we can do to resolve these conflicts:

1. changing the contents of the problematic attributes,
2. moving objects to classes with looser type restrictions, or
3. moving the objects pointed to by problematic attributes to different classes that are compatible with the type restrictions.

A fundamental question is whether we consider changing the value of an attribute to be a more or less drastic change than changing the class membership of some object. The answer is likely to be application dependent. We proceed in two stages: in the rest of this section, we consider a technique for conflict resolution that avoids changing the contents of attributes at all, and only changes class memberships. We call this *adaptation by re-classification*. In section 6 we study a more general model in which a *price specification* is used to indicate the relative "cost" of moving objects between classes and of changing attribute values.

For a database state d, let $C(o, d)$ be the most specific class of the object o in state d.

Definition 3.1 *Let d and d' be two database states defined on the same schema. We say that d'* **preserves** *d, iff d' contains the same objects as d, and for every object o and every attribute A*
- If A is an attribute of both $C(o, d)$ and $C(o, d')$ then the value of the two attributes is identical.
- If A is an attribute of $C(o, d')$ but not of $C(o, d)$, then the value of A is a default value associated with A.

Thus d' agrees with d on the the value of the common attributes of all the objects in the database, but may differ in the classification of the objects.

The difference between two databases is measured using the notion of *distance*. Informally, the *distance* between two databases is the sum of distances between the old and new locations (classes) of all the objects in the database. The distance between two classes measures how much classification information is lost/ gained when an object is moved from class c_1 to class c_2. This distance is defined as the number of classes that are superclasses of one of them but not the other. More formally:

Definition 3.2 *The class hierarchy of a database can be viewed as a directed graph. The* **distance** *between two classes c_1, c_2, is denoted by $dist_c(c_1, c_2)$, and defined as follows:*
Let S_i, $i = 1, 2$ be the set of all class names on paths from the most general object class to c_i. The distance between c_1, c_2 is the size of the symmetric difference between S_1 and S_2, i.e. $dist_c(c_1, c_2) = |(S_1 - S_2) \cup (S_2 - S_1)|$

Definition 3.3 *Let d, d' be two instances of the same database schema, containing the same set of objects O. The distance between d and d', denoted by $dist(d, d')$ is defined as follows*

$$dist(d, d') = \sum_{o \in O} dist_c(C(o, d), C(o, d'))$$

Example 2: Consider the software engineering database of Example 1. Suppose we have a C program $prog1$, represented by object o_1, that calls three C procedures, represented by objects o_1, o_2, o_3. That is, object $o = [Pname : prog1, Calls : \{o_1, o_2, o_3\}]$ where $o \in CProgs$ and $o_1, o_2, o_3 \in CProcs$. Now we discover that procedure o_2 is actually a C^{++} procedure, and we want to move it to the right class. There are several possible databases preserving the original one, where o_2 is in the desired class. These are some of the possibilities:

1 $o \in C^{++}Progs$, $o_2 \in C^{++}Procs$, and $o_1, o_3 \in CProcs$.

2 $o \in C^{++}Progs$, and $o_1, o_2, o_3 \in C^{++}Procs$.

3 $o \in Progs$, $o_2 \in C^{++}Procs$, and $o_1, o_3 \in CProcs$.

The distance from the original database to database (1) is 1 (since only one object was moved one class up in the hierarchy). The distance to database (2) is 3, and to database (3) is 2. In the first case, only essential re-classification was performed. Thus the distance is minimal. We choose database (1) as the resulting state. In general, one would like to resolve the type conflicts by finding a consistent database, with the smallest distance from the original database, and where the moved object is in the new class.

We next introduce an update function *mig* (abbr. for migration), that given a database, an object o and a class c, defines the new state of the database caused by moving o to c. Let DB be a database schema, d a database instance, o an object, and c a class name. Let $\mathcal{DB}$ be the set of all consistent instances of DB that preserve d and where the most specific class of object o is c.

$$mig(d, o, c) = \{d' \in \mathcal{DB} \mid \forall e \in \mathcal{DB}\; dist(d', d) \leq dist(e, d)\}$$

This definition assumes that no constraints other than the type constraints are imposed on the database. If other constraints (e.g. integrity constraints) are imposed, the definition of *mig* is easily adjusted so that only the instances satisfying the constraints are considered.

mig can be used to automatically resolve the typing conflicts caused by object migration. Alternatively, it can be used to suggest to the user a possible solution to the conflicts. The user can either approve the suggestion, or perform a manual adaptation.

Note that there are cases where $mig(d, o, c)$ is empty, i.e. there is no consistent database preserving d, where o is in the desired class.

Example 3: Consider a database schema with two classes c_1 and c_2 both having an attribute A of type c_1. Assume that the database d contains an object $o \in c_1$ where $o = [A : o]$ (i.e. o points to itself). And suppose we want to move o to the class c_2. It is easy to see that there is no consistent database d' preserving d where $o \in c_2$. The result of $mig(d, o, c)$ is therefore empty. This means that the the move cannot be performed without changing the content of some attributes. We shall consider this possibility further in the next sections. □

On the other hand, an object move may result in several possible databases, all having the same distance to d. In this case we can either ask the user to choose the desired database, non-deterministically choose one database, or keep all the resulting databases as possible worlds.

Example 4: Consider a database schema with five classes c_1, c_2, c_3, c_4, c_5 where c_1 is a subclass of c_2, and c_3 is a subclasses of both c_4 and c_5 (multiple inheritance). Let c_3 have an attribute A of type c_1, and c_4, c_5 both have an attribute A of type c_2. Assume that the database d contains an object $o \in c_1$, and an object $o' \in c_3$ where $o' = [A : o]$ (i.e. o' points to o via the attribute A). And suppose we want to move o to the class c_2. The databases d' where $o' \in c_4$, and the databases d'' where $o' \in c_5$ both preserve d and have the same distance from it. The result of $mig(d, o, c)$ is therefore the set $\{d', d''\}$ and not a unique database. □

3.1 Migration as Update

We can provide theoretical justification for our intuitive claim that the migration function only changes what is strictly necessary by using Katsuno and Mendelzon's theory of updates[KM91]. They consider knowledge bases represented by propositional theories; an update operation is a request that a new sentence be inserted into the existing theory. They give a set of postulates that characterize all update operators that cause minimal change in a precise sense. Similar approaches to update/revision of knowledge bases appear in [Bor85, Dal88, Sat88, Win88].

We can model database states as (very simple) propositional theories by having a propositional letter for each fact of the form "object o belongs to class c" and one for each fact of the form "attribute A of object o points to object o' of class c'." We add appropriate integrity constraints to make sure that each object belongs to a unique class and that the typing restrictions are satisfied.

An object migration operation becomes the insertion of a sentence of the first form, giving the desired new class membership for some object. To make the correspondence to update theory cleaner, we can generalize our notion of migration to allow insertion of an arbitrary "migration sentence," that is, an arbitrary propositional combination of atomic class membership statements. For example, we could say something like "move employee John *out* of the class of full-time employees *or* move John's project into the class of suspended projects." Furthermore, we can specify transactions that move several objects simultaneously. This mapping induces a notion of update that falls within the class of minimal change operators defined in [KM91] (proof is omitted for lack of space). The subject is further discussed in Section 8.

4 Complexity

In this section we study the complexity of computing $mig(d, o, c)$ as a function of the size of the database. The problem is a special case of the problem of updates in knowledge bases, studied in [EG92]. Our hope was that, although updates in knowledge bases are in general intractable, the relative simplicity of the constrains imposed on the object migration (only typing constrains) will make the problem tractable. [4] As we show below, it turns out that the tractability of object migration depends on the properties of the database schema.

We start by identifying the objects that may be affected by the moving an object o from one class to another.

Definition 4.1 *Let d be a database, and let o be an object in d. We say that an object o_1 is reachable from o iff one of the following hold:*
1) $o_1 = o$.
2) o points to o_1 in one of its attributes, or is pointed to by one of o_1's attributes.
3) o_1 is reachable from some object o_2 that is reachable from o.

From the definition of mig it is easy to see that database objects that are not reachable from o are not affected by moving o from one class to another. In particular

Proposition 1 *If o_1 is not reachable from o, then in all the databases in $mig(d, o, c)$, o_1 belongs to the same class as in d.*

An algorithm for computing $mig(d, o, c)$, based on this observation, is informally sketched below:

Algorithm 1
Let O_r be the set of objects reachable from o. For every possible partition P of the database objects among classes that differs from the original one only in the location of objects in O_r, check if the most specific class of o is c, and if the resulting database is consistent. If both conditions are satisfied, compute the distance from the original database d. Then choose the databases with minimal distance. □

Note that the number of possible object partitions is $|C|^{|O_r|}$.[5] In the worst case, the algorithm is exponential in the size of the database. We show in the next section that for an important class of database schemas – the covariant schemas – a linear algorithm exists.

[4] Some restricted classes of tractable updates were presented in [EG92]. They do not include, however, the object migration problem.

[5] If the database has to satisfy integrity constraints other than typing constraints, then objects other than those reachable from o may be affected by the move. In this case, we have to consider all possible partitions of all the objects.

The following theorem states that a polynomial time algorithm does not exist for general schemas, unless P = Co-NP.

Given a database d, an object o, and a class c, we call the problem of checking whether $mig(d, o, c)$ is empty, the **object migration** problem.

Theorem 4.1 *The* object migration *problem is Co-NP-Complete.*

Proof: (Sketch) It is easy to check in NP that a given database is not at minimal distance, by guessing a database with smaller distance. Thus the problem is in Co-NP. The hardness is by reduction from 3-CNF unsatisfiability, and is given in the Appendix. □

It should be noted that, in the database used in the proof, the number of classes is fixed and does not depend on the length of the 3-CNF formula being simulated. Thus the problem is Co-NP-Complete even if the class hierarchy is not considered part of the input.

5 Covariance

In this subsection, we show how covariance simplifies the object migration problem. A database schema is *covariant* iff for every two classes c_1, c_2 both having an attribute A, where A is of type c'_1 ($\{c'_1\}$) in c_1, and of type c'_2 ($\{c'_2\}$) in c_2, we have that $c_1 \leq c_2$ implies $c'_1 \leq c'_2$. This is a natural restriction: it says that the attributes of a subclass should specialize the attributes of the superclass. For example, the schema of the software engineering database presented earlier is covariant. In fact, in some object-oriented database systems such as O2[O2T94], only covariant schemas are definable.

Moving objects in databases where the schema is covariant turns out to be computationally easy. In particular we show that

Theorem 5.1 *For covariant schemas with no multiple inheritance,* **mig** *can be computed in time linear in the size of the database instance*[6].

Here is a sketch of the algorithm.

Algorithm 2
First the object o is moved to the required class. This may cause some of the attributes of the moved object to be incorrectly typed. The situation is fixed by moving the objects pointed to by these attributes to the required classes. This process is repeated until all the attributes of the moved objects are correctly typed, but in such a way that, after an object is moved for the first time, it is restricted to move only *down* the hierarchy from then on. In the next phase, we handle objects pointing to the moved objects. To fix objects that violate a type restriction by pointing to one of the objects moved during the first phase, the pointing objects are moved *up* the class hierarchy to suitable classes. In this step, only objects that were not moved in the first stage are allowed to move. If these two restrictions on object movement cannot be satisfied, the algorithm concludes that $mid(d, o, c)$ is empty. □

[6] Where the constant factor depends on the depth of the class hierarchy

The linearity of the algorithm follows from the fact that (i) in the first phase of the algorithm, except for the first move, objects move only down the class hierarchy, and (ii) in the second phase, objects move only up the class hierarchy. Thus, each object is moved at most h times, where h is the depth of the class hierarchy. To prove the correctness of the algorithm we show that the resulting database is the closest to the original database. Since the algorithm is deterministic, it follows that object migration in covariant schemas results in a **unique** database.

Proposition 2 *If d is an instance of a covariant database schema with no multiple inheritance, then $mig(d, o, c)$ contains at most one database instance.*

6 General Adaptation

The adaptation technique presented above avoids changing the values of attributes, and resolves conflicts solely by re-classifying objects. There are cases where the association between objects and classes is important and should not be changed when resolving conflicts. In particular, membership in certain classes may be crucial, while membership in others may not. Similarly, the values of specific attributes may be important and unchangeable, while others may be "sacrificed." In this section we introduce a cost structure by which membership in different classes can be assigned different strengths, which in turn can be compared with the strength of attribute values. We first apply this cost structure in 6.1 to give a more flexible reclassification method, and then generalize this in 6.2 to the case where attribute values can be changed.

The relative importance of classes and attributes can be described using a **price specification**. A price specification assigns a weight w_c to each class c, and a weight $w_{A,c}$ to each attribute A in c. (We assume that all weights are positive.) We explain below how price specification can be used to handle object migration.

An instance d of a database schema DB can be viewed as a directed graph $G_d = (V, E)$. The nodes V represent the classes and objects in a database. There are two kinds of edges in E: **Class edges** represent the membership of objects in classes. If the most specific class of o is c, then we have an edge from o to c and to all its super-classes. **Attribute edges** are labeled edges representing the content of attributes in objects. If the attribute A of the object o contains (points to) o', then the graph contains a labeled edge $o \xrightarrow{A} o'$. If an attribute A is of set type, then o may have several outgoing edges labeled with A.

The price specification associated with the database schema induces weights on the edges of the graph. The weight of an edge from an object o to a class c is w_c. The weight of an attribute edge $o \xrightarrow{A} o'$ is $w_{A,c}$, where c is the most specific class of o.

An update to a database d can be described in terms of modifications to the corresponding graph G_d. To measure the difference between two databases, we again use the notion of **distance**. The distance is defined in terms of the difference between their corresponding graphs.

Definition 6.1 *Let d, d' be two databases, containing the same set of objects. Let $G_d = (V, E)$, $G_{d'} = (V, E')$ be their corresponding priced graphs. We denote by E_c, E'_c the class edges in G_d, G'_d resp. and by E_a, E'_a the attribute edges, and use w_e to denote the weight of an edge e.*
The **class distance** *between d, d', denoted by $Cdist(d, d')$, is defined as follows,*

$$Cdist(d, d') = \sum_{e \in (E_c - E'_c) \cup (E'_c - E_c)} w_e$$

The **attribute distance** *between d, d', denoted by $Adist(d, d')$, is defined as follows,*

$$Adist(d, d') = \sum_{e \in (E_a - E'_a) \cup (E'_a - E_a)} w_e$$

The difference in *class* edges measures the distance between the classes to which objects belong in the two databases. If an object o is moved form class c to class c' then the number of class edges removed/added because of the move is exactly the distance between those two classes (see definition 3.2). It follows that if the weight of all class edges is 1 then $Cdist(d, d') = dist(d, d')$.

The attribute distance measures the difference in the content of attributes. The attribute edges that belong to the graph G_d but not to $G_{d'}$ are exactly the "lost" attributes. The attribute edges that belong to $G_{d'}$ but not to G_d are new attribute values.

6.1 Adaptation by re-classification – Revisited

The $Cdist$ function can be used to refine the adaptation process presented in the previous section. Instead of minimizing amounts of re-classification, we minimize the overall cost of the re-classification.

The refined $mig_p(d, o, c)$ migration function computes the new state of the database caused by moving o to c, based on the price specification. The definition of $mig_p(d, o, c)$ is the same as that of $mig(d, o, c)$ except the it uses the priced class distance function $Cdist$ instead of the non priced function $dist$.

Prices can be used to control the re-classification of objects. We may want a person to become an employee, or a manager, or a student, but we don't want a person to become a car. Consider a subtree of the class hierarchy rooted in class c. Suppose we want objects to move only within the bounds of the classes in the subtree. This can be accomplished by giving a very high weight (infinite) to c. To move an object to a class outside this group one has to disconnect an edge from the object to the class c (recall that an object has class edges to all its super classes). The resulting database has infinite distance from the original database. We can modify the migration function $mig_p(d, o, c)$ to reject solutions with infinite distance. The refined function will therefore consider only solutions where objects stay within the required set of classes.

It is interesting to note that although prices allow more flexibility, the process of choosing the desired database state does not become more complex. For general database schemas, $mig_p(d, o, c)$ can be computed using an algorithm similar to Algorithm 1 presented in section 4. Moreover, for covariant schemas we can slightly change Algorithm 2 presented in section 5 and get a linear time algorithm computing $mig_p(d, o, c)$. The only modification needed is to add another restriction on objects moves: Objects belonging to a class that has super class c with infinite weight may not be moved to a class that is not a subclass of c.

Theorem 6.1 *For covariant schemas with no multiple inheritance, mig_p can be computed in time linear in the size of the database instance.*

6.2 A Method for General Adaptation

We now consider the general case where attribute values can be modified to avoid re-classification of objects. To simplify the presentation, we assume that the only choice is whether a value of an attribute is preserved or reset to a default value. In Section 8 we discuss the extension that allows the user to specify values for such attributes.

Definition 6.2 *Let d, d' be two instances of the same database schema. We say that d'* **partially preserves** *d, iff d' contains the same set of objects as d, and for every attribute A and every object o in d', either the value of $o.A$ is the same at that in d, or it is the default value of A.*

Let DB be a database schema, d a database instance, o an object, and c a class name. Let $\mathcal{DB}$ be the set of all consistent instances of DB partially preserving d, and where the most specific class of o is c.

We define the general priced migration function $general\!-\!mig_p$ as follows.

$$general\!-\!mig_p(d,o,c) = \{\ d' \in \mathcal{DB} \mid \forall e \in \mathcal{DB}$$
$$Cdist(d',d) + Adist(d',d) \leq Cdist(e,d) + Adist(e,d)\}$$

The function $general\!-\!mig_p$ finds the closest consistent database where c is in class o. To do that, it may re-classify objects or change the values of attributes. As explained above, weights can be used to control the range of classes. Similarly, they can be used to control attribute modification. For example,assigning a very high (infinite) weight to certain attributes assures that their content is preserved if possible.

To compute $general\!-\!mig_p(d,o,c)$, one must not only consider the partition of objects among classes, but also the content of the attributes. It is easy to see that only objects that are reachable from o are affected by moving o from one place to another. In particular, proposition 1 still holds for $general\!-\!mig_p(d,o,c)$. The values of the attributes of the reachable objects are either the same as in the original database, or ar set to default values. More precisely,

Proposition 3 *Let d, d' be two databases s.t. $d' \in general\!-\!mig_p(d,o,c)$. Let $G_d = (V,E), G_{d'} = (V,E')$ be their graph representations, and let E_a, E'_a be the attribute edges in the two graphs resp.*
(i) If $o_1 \xrightarrow{A} o_2 \in G_d$ and $C(o_2,d') \leq C(o_1,d').A$ then $o_1 \xrightarrow{A} o_2 \in G'_d$.
(ii) If $o_1 \xrightarrow{A} o_2 \in E'_a - E_a$ then o_2 is the default value of the attribute A in the class $C(o_1,d')$.

Based on this observation, an algorithm for computing $general - mig_p(d,o,c)$ can be designed along lines similar to Algorithm 1.

Algorithm 3
Let O_r be the set of objects reachable from o. For every partition P of the database objects among classes, that differs from the original one only in the location of objects in O_r, check if the most specific class of o is c. If so, create a graph whose nodes are the classes and objects of the database, and where the *class* edges correspond to the partition P. Next, add to the graph all the attribute edges of G_d satisfying the typing constraints. All the other attributes are assigned default values. Finally, compute the distance from the original database d, and choose databases with minimal distance. □

The algorithm runs in $|C|^{|O_r|}$ time. We show next that a polynomial algorithm does not exist unless P = Co-NP. The complexity of $mig(d,o,c)$ was studied using the *object migration* problem (checking if the result of $mig(d,o,c)$ is empty). This problem cannot be used for studying $general\!-\!mig_p(d,o,c)$ since its output is never empty [7]. Instead, we use the *membership testing* problem. This is the problem of checking whether an object o' belongs to a class c' in one of the databases in $general\!-\!mig_p(d,o,c)$.

Theorem 6.2 *The membership testing problem is Co-NP-Complete*

Proof: (Sketch) The Co-NP algorithm is similar to the one in the proof of Theorem 4.1. The hardness is again proved by reduction from 3-CNF unsatisfiability. The construction is the same as in the proof of Theorem 4.1, except that now we have weights for classes and attributes. We also have in every class c an additional object o_c that serves as default value for that class. □

It is open whether the problem remains Co-NP-Complete for the covariant case. We conjecture it does.

7 Methods

So far we only considered the structural component of objects, that is, the attributes. The methods of an object use its internal state (and other methods), and depend on the class of the object. Since objects are moved from one class to another, they may have different methods than before, or different implementations for the same method names. Moreover, even if an object still has the same methods with the same implementation, the value of its attributes may change, thus the methods can have different behaviour. The question is whether moving an object may cause the execution of some methods associated with the object to fail.

The notion of *consistent method schema* was defined in [AKW90]. Informally, a database schema DB that includes methods is *consistent* iff for every consistent instance (i.e. an instance where all the attributes are correctly typed) none of the methods fail. (for formal definition of consistency and failure, see [AKW90]). It follows that

Proposition 4 *If DB is a consistent schema, then for every database instance d, object o, and a class c, none of the methods of DB fail when executed on objects in $d' \in general - mig_p(d,o,c)$, $d' \in mig_p(d,o,c)$ or $d' \in mig(d,o,c)$.*

A consistent method schema assures that after the object migration not only the attributes are correctly typed, but also all the methods can be executed successfully. If the schema is not consistent, then such failures can occur.

[7] A consistent state (though very far from the original state) can always be achieved by "nullifying" the values of all the attributes, and setting them to default values.

8 Extensions

We next show how the techniques described in the previous sections can be extended in various ways by small modifications to definitions 3.1 (database preservation) and 6.2 (partial preservation), and to the adaptation algorithms. Some of the extensions are:

Supplying new values for attributes: The user may want to provide new values for some attributes as part of the object migration. (e.g. "move this employee to the Managers class, and change her car from Subaru to Mercedes"). Definitions 3.1 and 6.2 can be adjusted accordingly – the new values must be preserved. Algorithms 1-3 should be modified so that the database is first updated with the new values, and the adaptation process handles the updated database. At the end of the process, instead of assigning default values to attributes, the user-supplied values are assigned.

Transactions and arbitrary formulas: For both theoretical and practical reasons, it is useful to allow not just a single migration operation, but the insertion of arbitrary "class membership formulas." For example, we may want to support transactions that move several objects simultaneously, or we may want to move an object into a set of classes without caring exactly which one, or to get an object *out* of a certain class.

Note that database adaptation for transactions that move several objects cannot be simulated by moving each object separately and performing adaptation after each such move. This is because the adaptation process of one object may move other objects out of their target classes. To move several objects simultaneously, the adaptation procedure should take into account only databases where all the moved objects are in the required classes. When inserting arbitrary "class membership formulas", the adaptation process should take into account only databases where the inserted formulas are satisfied. The definitions and algorithms should be adjusted accordingly.

Values: The data model considered above supports objects and classes. A similar adaptation technique can be used for models that support (complex) values as well (e.g. [AK89]).

Consider the graph representation of a database. In addition to object and class nodes, we now also have value and type nodes. Each occurrence of a (complex) value in the database is represented by a node (different occurrences of the same value are represented by distinct nodes). There are *attribute/member* edges from tuple/set values to their attributes/members, and *type* edges from values to their type (and to all the super-types). Moving a value from one type to another is interpreted as type coercion. One has to specify for each atomic type the set of types eligible for coercion, and supply a coercion algorithm for them. Coercion for complex values can then be defined in terms of their components.

A similar approach can also be used for databases that supports complex types other than tuples and sets, e.g. bags, list, and arbitrary abstract data types[BM92].

Multiple Roles: Objects may have simultaneously multiple independent roles. For example, a person can be simultaneously a student, an employee, a member of the national bridge team, etc. There are several proposals to allow an object to have multiple *aspects* [RS91] or play multiple *roles* [ABGO93, Fis87, SZ89]. Our techniques can be used to support automatic adaptation in such systems.

Consider the graph representation of a database. The class nodes represent the different aspects/roles. Each object node has *class* edges to all the aspects/roles of the object. An object has different attribute edges for each of its aspects/roles. (i.e. for each role we have attribute edges corresponding to the value of the attribute in that role.) The edges are labeled with aspect/role names.

As before, changing the set of aspects/roles of an object can be modeled by modifying the graph representation. To handle inconsistencies caused by changing aspects/roles of object we look for a consistent database with smallest distance to the original one, and where the object has the desired set of aspects/roles.

Schema Evolution Object migration can be used to handle schema evolution, by modeling changes in the schema with object moves. For example, to merge two classes we can move all the objects from one class to the other, a new class can be populated by moving objects into the classes (or adding new objects), to delete a class we can move all its members to one of its super classes (or any other desired class).

Modifying the definition of some class c (change/add attributes or methods) is more difficult, but can still be modeled with object migration. We can (i) add to the schema a new class c' with the desired definition s.t. c' a subclass of c and a super class of the subclasses of c, (ii) move all the objects in c to c', and (iii) delete c, and rename c' to c. Note that the adaptation procedure must be slightly modified so that it avoids moving objects into the old class.

9 Conclusions and Open Problems

In this paper we studied the problem of objects that migrate from one class to another. Due to typing constraints, object migration may result in an inconsistent database state. We studied adaptation techniques that resolve conflicts by re-classifying objects, or by changing the values of conflicting attributes. We showed that the problem is in general computationally difficult, but can be solved efficiently for the important case of covariant schemas.

The techniques presented in the paper can be used to automatically adjust the database state. Alternatively, they can be used to suggest to the user possible solutions to the typing conflicts caused by the migration. The user can then either approve the suggestions, or perform a manual adaptation.

The adaptation process may result in several consistent databases all having the same distance from the original one. An interesting open question is whether for a given database schema it is possible to characterize those moves that result in a unique database. Another possibility is to refine the adaptation process so that a unique database is chosen. Various techniques have been developed for choosing a unique model for a datalog program with negation. These techniques might be used here by considering an object-oriented variant of datalog with negation, featuring overriding [ALUW93] and object identity [AK89]. The migration of an object could then be specified in such a language, and minimal distance can correspond to minimal model. Whether the minimal model obtained would also make sense with respect to object migration, appears a very interesting issue.

On the same issue, an unambiguous semantics to a view update is obtained in [BS81] by specifying the "constant complement" of the view. Can this approach be adapted to specify the part of the database which contains information considered as certain; that is, the "constant" part that should not be affected by the migration, thus yielding a unique database?

On the other hand, when conflicts are solved using re-classification only, a consistent database preserving the original database may not exist. Is it possible to characterize the set of all "safe" moves, that is, those where a consistent database always exists?

Acknowledgments: Thanks to Victor Vianu for suggesting several of the further research directions listed at the end of the paper. We also thank Anthony Kosky and Leonid Libkin for their helpful comments on an earlier version of this paper.

References

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *18th Conf. on Very Large Databases, VLDB, Dublin, Ireland*, pages 39–51, 1993.

[AK89] S. Abiteboul and P. Kanellakis. Identity as a query language primitive. In *Proc. SIGMOD, Portland, Oregon*, pages 159–173, 1989.

[AKW90] S. Abiteboul, P. Kanellakis, and E. Waller. Method schemas. In *Proc. 9th Symp. on Principles of Database Systems - PODS*, pages 16–27, 1990.

[ALUW93] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and rules. In *Proc. SIGMOD, Washington D.C.*, pages 32–41, 1993.

[BM92] C. Beeri and T. Milo. Functional and predicative programming in oodb's. In *Proc. 11th Symp. on Principles of Database Systems - PODS, San-Diego*, pages 176–190, 1992.

[Bor85] A. Borgida. Language features for flexible handling of exceptions in information system. *ACM Transactions on Database Systems*, 10(4):563–603, 1985.

[BS81] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. on Database Systems*, 6(1):557–575, 1981.

[Dal88] M. Dalal. Investigations into a theory of knowledge base revision. In *Proc. 7th National Conf. on Artificial Intelligence*, pages 475–479, 1988.

[EG92] T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates, and counterfactuals. In *Proc. 11th Symp. on Principles of Database Systems - PODS, San Diego*, pages 261–273, 1992.

[Fis87] D.H. Fishman et al. Iris: An object oriented database management system. *ACM Trans. on Office Information Systems*, 5(1):46–69, 1987.

[KM91] H. Katsuno and A. O. Mendelzon. On the difference between updating a knowledge base and revising it. In *Proc. 2nd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 387–394, 1991.

[O2T94] O2Technology. *The O2 User's Manual Version 4.3.1*, 1994.

[RS91] J. Richardson and P. Schwartz. Aspects: Extending objects to support multiple independent roles. In *Proc. of the Int. conf on Management of Data, SIGMOD, Denver, Colorado*, pages 298–307, 1991.

[Sat88] K. Satoh. Nonmonotonic reasoning by minimal belief revision. In *Proc. of the Int. Conf. on 5th Generation Computer Systems*, pages 455–462, 1988.

[Su91] Jianwen Su. Dynamic constraints and object migration. In *17th Conf. on Very Large Databases, VLDB, Barcelona, Spain*, pages 233–242, 1991.

[SZ89] L.A. Stein and S.B. Zdonik. Clovers: The dynamic behavior of type and instances. Technical report, Brown University, Technical report no. CS-89-42, 1989.

[Win88] M. Winslett. Reasoning about action using a possible-model approach. In *Proc. 7th National Conf. on Artificial Intelligence*, pages 89–93, 1988.

Appendix

Theorem 4.1 The *object migration* problem is Co-NP-Complete.

Proof: It is easy to check in NP that a given database is not at minimal distance, by guessing a database with smaller distance. Thus the problem is in Co-NP.

The hardness result is by reduction from the problem of checking if a 3-CNF formula is unsatisfiable, known to be Co-NP-complete. We show below that for every 3-CNF formula ϕ, there exists a database schema DB, an instance d, an object o, and a class c, such that $|d| = \mathcal{O}(|\phi|)$, and $mig(d, o, c) = \emptyset$ iff ϕ is unsatisfiable. The instance for the migration problem is constructed in time polynomial in the size of ϕ.

The database schema DB has classes representing variables, clauses, and formulas.
There are three classes for variables: U, T, and F. The class U represents variables whose truth value is unknown, T represents truth assignments, and F false assignments. Objects in U, T and F have no attributes. The classes for clauses correspond to the different patterns of clauses in a 3-CNF formula. Let v denote a variable. There are 8 possible patterns:

$$(v \vee v \vee v)\ ,\ (\neg v \vee v \vee v)\ ,\ (v \vee \neg v \vee v)\ ,\ \ldots\ ,\ (\neg v \vee \neg v \vee \neg v)$$

For each pattern $i = 1 \ldots 8$, we have 5 classes $C_i^u, C_i^t, C_i^{t_1}, C_i^{t_2}, C_i^{t_3}$, as described below.

The class C_i^u represents the case where the truth value of the clause of the i^{th} pattern is unknown (since the value of the literals in the clause is unknown). The structure of objects in class C_i^u is $C_i^u = [v_1 : U, v_2 : U, v_3 : U]$

The class C_i^t represents the case where a clause of the i^{th} pattern is known to be true because **all** its literals are true. The structure of C_i^t depends on the pattern it represents.

$$\begin{array}{lcl} C_1^t & = & [v_1 : T, v_2 : T, v_3 : T] \\ C_2^t & = & [v_1 : F, v_2 : T, v_3 : T] \\ C_3^t & = & [v_1 : T, v_2 : F, v_3 : T] \\ & & \ldots \\ C_8^t & = & [v_1 : F, v_2 : F, v_3 : F] \end{array}$$

The classes $C_i^{t_j}$ $(j = 1 \ldots 3)$ represent the case where a clause of the i^{th} pattern is known to be true because its j^{th} literal is true (the values of the other literals is unknown). For brevity, we only show in the following the structure of the classes for the second pattern $(\neg v \vee v \vee v)$. The other classes are constructed similarly.

$$\begin{array}{lcl} C_2^{t_1} & = & [v_1 : F, v_2 : U, v_3 : U] \\ C_2^{t_2} & = & [v_1 : U, v_2 : T, v_3 : U] \\ C_2^{t_3} & = & [v_1 : U, v_2 : U, v_3 : T] \end{array}$$

Finally, we have two classes ϕ^u and ϕ^t. The class ϕ^u represents the case where the truth value of ϕ is unknown (since the truth value of the clauses is unknown). The class ϕ^t represents the case where the formula is known to be true (where all the clauses are known to be true). ϕ^u and ϕ^t have one attribute for each clause in ϕ. The type of the attribute corresponds to the pattern of the clause.

For example, consider the formula

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_4)$$

The structure of the corresponding classes is

$$\begin{array}{lcl} \phi^u & = & [Clause_1 : C_2^u, Clause_2 : C_1^u, Clause_3 : C_8^u] \\ \phi^t & = & [Clause_1 : C_2^t, Clause_2 : C_1^t, Clause_3 : C_8^t] \end{array}$$

The class hierarchy is the following: $T \le U$, $F \le U$, $\phi^t \le \phi^u$, $C_i^t \le C_i^u$, $C_i^{t_j} \le C_i^t$ for $i = 1 \ldots 8$, $j = 1 \ldots 3$.

Given a 3-CNF formula ϕ, we build a database instance d that represents the fact that the truth value of ϕ is unknown. d contains objects representing the formula, its clauses, and its variables. The database contains an object $o \in \phi^u$ corresponding to the whole formula. For each clause c_i in the formula, we have an object o_{c_i}. If c_i is of pattern j, than $o_{c_i} \in C_j^u$. We also have an object o_{x_j} for every variable x_j in the formula. All these objects belong to class U. The attributes of the formula object o point to objects representing clauses of the formula. The attributes of the clause objects point to objects representing the variables of the clause.

Consider for example the formula ϕ presented above. The database instance is the following:

$$\begin{array}{l} o = [Clause_1 : o_{c_1}, Clause_2 : o_{c_2}, Clause_3 : o_{c_3}] \\ o_{c_1} = [v_1 : o_{x_1}, v_2 : o_{x_2}, v_3 : o_{x_3}] \\ o_{c_2} = [v_1 : o_{x_1}, v_2 : o_{x_2}, v_3 : o_{x_3}] \\ o_{c_3} = [v_1 : o_{x_1}, v_2 : o_{x_2}, v_3 : o_{x_4}] \end{array}$$

Where $o \in \phi^u$, $o_{c_1} \in C_2^u$, $o_{c_2} \in C_1^u$, $o_{c_3} \in C_8^u$, and $o_{x_1}, \ldots, o_{x_4} \in U$.

Now assume that we move o to the class ϕ_t. A careful examination of the above construction shows that a consistent database preserving d and where o is in class ϕ_t exists iff ϕ is satisfiable. Every such database corresponds to an assignment satisfying ϕ. Thus $mig(d, o, \phi_t)$ is empty iff the formula ϕ is unsatisfiable.