Object Migration in ISA Hierarchies

Tok Wang LING and Pit Koon TEO
Department of Information Systems and Computer Science
National University of Singapore
Kent Ridge, Singapore 0511
lingtw@iscs.nus.sg teopitko@iscs.nus.sg

Abstract

When an object migrates from one class to another in an ISA hierarchy, it acquires or loses membership in respective classes in the ISA hierarchy. Since an object can be an instance of multiple classes in the ISA hierarchy, there is a question of whether to have a single object identifier (OID) or multiple OIDs assigned to an object as it migrates along the ISA hierarchy. We refer to this as the *OID ambiguity* problem. Rules for *meaningful* object migration are first described in order to establish a framework for studying this problem. Then, to address the problem, a number of variables are considered, viz. (1) the storage scheme chosen for the ISA hierarchy, (2) the representation for the OID, (3) the way a message despatched to an object is processed, and (4) the ability to decide if two (or more) OIDs refer to the same real world object. Four approaches are studied, each of which resolves the OID ambiguity problem. One of the approaches is recommended as a superior approach because it retains the desirable OID properties of uniqueness and immutability and has relatively less overheads.

1 Introduction

The support for object identity (OID) [6] is a key characteristic of most object-oriented database systems (OODBMSs). It provides a means to distinguish among objects, even those with similar attribute values. OIDs are system generated, logical, immutable and, when used in inter-object references, allow objects to be shared. An object's OID is durable in the sense that it is independent of changes to (key) attribute values of the object.

Object-oriented systems that support object migration allow an object to migrate from one class to another in an ISA hierarchy. Depending on OID implementation, e.g. typed OIDs, it is possible for an object to adopt another OID when it migrates to another class, thus violating the immutability property of OID. Further, as a consequence of object migration, an object can be an instance of multiple classes in the hierarchy, possibly adopting multiple OIDs. This violates the uniqueness property of OID.

When an object migrates to another class in an ISA hierarchy (possibly becoming an instance of multiple classes in the hierarchy), we can either (1) assign a single OID to the object even though it is an instance of multiple classes, or (2) have multiple OIDs assigned to the object, i.e. one OID for each instance of the object in the hierarchy. We refer to this as an *OID ambiguity* problem. Suppose a single OID is assigned. There is a mismatch between the single OID and the multiple instances of the object in the ISA hierarchy. Given this OID, there is also a question of which class to associate with this OID. Class information is needed to provide the object's structure as well as to process messages despatched to the object.

Suppose multiple OIDs are assigned to instances of the object. In this case, there is a problem of deciding that two (or more) OIDs are in fact referring to the same real world object. We refer to this as an *object identification* problem. This problem exists only when multiple OIDs are assigned to a migrating object, but not when the migrating object has only a single OID assigned.

In the presence of object migration, a message despatched to an object is ambiguous because it can be for any of the classes that the object is an instance of. The classical OO way of processing a message despatched to an object of a class X assumes that X is the most specific class for the object. This assumption may not be true in the presence of object migration. We refer to this as a *message ambiguity* problem.

In this paper, we first describe the rules under which *meaningful* object migration can occur. These rules provide a framework in which to study the OID ambiguity problem, and the associated object identification

Proceedings of the Fourth International Conference on
Database Systems for Advanced Applications (DASFAA'95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995

and message ambiguity problems. Then, to address these problems, a number of variables are considered, viz. the storage scheme for the ISA hierarchy, the representation chosen for the OID, the way a message despatched to an object is processed, and the ability to resolve the object identification problem. In this paper, three possible storage schemes for ISA hierarchies and a number of possible representations for OIDs are discussed. We also describe a method of message processing that always starts from the most specific class associated with an object. Finally, four approaches are described, each of which resolves the OID ambiguity problem, and the associated message ambiguity and object identification problems. These four approaches are:

1. Using typed surrogates[3] to support multiple OIDs for a migrating object.
2. Using untyped surrogates[3] to support multiple OIDs for a migrating object.
3. Using a typed surrogate to support a single OID for a migrating object.
4. Using an untyped surrogate to support a single OID for a migrating object.

Subsequent discussions will show that approach 4 is superior to the other three approaches for resolving the OID ambiguity problem.

Section 2 discusses the motivation behind this paper. Section 3 examines some of the semantics associated with object migration. These semantics describe the rules under which *meaningful* object migration can take place. In Section 4, storage schemes for ISA hierarchies and representations for OIDs are discussed. Then a more reasonable method for processing a despatched message is described. Finally, four approaches are discussed to resolve the OID ambiguity problem. Section 5 concludes the paper.

2 Motivation

Suppose an object O with OID O_x migrates from a class X to a class Y in an ISA hierarchy. Then O becomes an instance of Y and all of Y's superclass(es), including X if Y is a subclass of X. We can either (1) assign a single OID to O even though it is an instance of multiple classes (e.g. X and Y), or (2) have multiple OIDs assigned to O, i.e. one OID for the instance in X and another OID for the instance in Y and so on. The problem of deciding whether to assign a single OID or multiple OIDs to a migrating object is referred to as the *OID ambiguity* problem.

Consider the case in which a single OID is assigned. There are at least two problems. The first problem is to decide whether to retain the original OID, i.e. O_x, or have a new OID, say O_y, assigned to the object. If a new OID is assigned, existing references to O_x are now invalid and must be updated. This update is expensive, if not impossible, because the references may be embedded in complex object structures throughout the database. Once the OID is decided, the second problem is to decide which class to associate with this single OID. Class information is needed to determine the structure of the object and to determine the validity of messages despatched to the object using this OID.

Suppose multiple OIDs are assigned to a migrating object. The immediate problem is that of deciding, given two (or more) OIDs, that they, in fact, refer to the same real world object. This is an *object identification* problem. Resolving the object identification problem is important in a number of instances. For example, the result of an OO query may be a set of OIDs. These OIDs may refer to a set of possibly heterogeneous objects, i.e. objects belonging to distinct classes. It is useful to determine that two or more such OIDs refer to the same real world object.

A real world object that participates in object migration may be an instance of multiple classes in an ISA hierarchy. Any message sent to the object becomes potentially ambiguous, since it can be for any of the classes that the object belongs to. We refer to this as a *message ambiguity* problem. We motivate the message ambiguity problem by looking at two examples:

Example 1 : Suppose MANAGER ISA EMPLOYEE and *bonus()* is a method in both MANAGER and EMPLOYEE such that the implementation of *bonus()* of MANAGER overrides the implementation of *bonus()* of EMPLOYEE. When a *bonus()* message is sent to an instance of EMPLOYEE who also happens to be a manager, it is semantically incorrect to execute the *bonus()* method of EMPLOYEE, since a more specialised *bonus()* method exists for this particular employee who is also a manager. The correct *bonus()* method to execute should be that of MANAGER.

Example 2 : Suppose MANAGER and EMPLOYEE each has a method *print()*. When a *print()* message is sent to an employee who is also a manager, the *print()* method of manager should be activated. The argument is exactly the same as that for activating the *bonus()* method in the example above. It may be argued that it is sometimes desirable to just print the employee details of the employee even though the employee is also a manager. However, we advocate that, in such a case, there should really be two print methods, viz. *empPrint()* and *mgrPrint()* for EMPLOYEE and MANAGER respectively. Defining two distinctly named print methods will not exploit the benefits of polymorphism, but is really necessary, because of the need to print EMPLOYEE and MANAGER instances differently.

3 Semantics of Object Migration

In this section, some interesting semantics associated with object migration are discussed. These semantics characterise the rules for *meaningful* object migration.

In [11], the concepts of essential types and exclusionary types are introduced to characterise the properties of types in an ISA hierarchy. An essential type is a type which cannot be lost through object migration. For example, an object of type PERSON still retains that type when the object becomes an EMPLOYEE instance and, later on, a MANAGER instance. PERSON is therefore an essential type for that object. An exclusionary type is a type acquired when the object is created, but can be removed as a result of object migration. Once an exclusionary type is removed, it can never be regained. For example, an object of type CHILD can migrate to a type TEENAGER, thereby losing its CHILD type. The object cannot regain its CHILD type subsequently.

We believe that there are other interesting type properties that should be noted during object migration. As a *general* rule, when an object moves from a class X to another class Y, it becomes an instance of class Y. It retains its membership in class X if Y is a subclass of X or loses its membership in class X if Y is a superclass of X. Consider the ISA hierarchy given in Figure 1. There are five possible scenarios for the migration of an object in the ISA hierarchy:

Case 1: An object can migrate from a class (e.g. class C) to a more specialised class (e.g. class D) which has only one superclass (i.e. class C itself). In this case, the object acquires membership in both A and C.

An illustration of this scenario is the promotion of an employee to a manager post, using the single inheritance situation in Example 1 of Section 2.

Case 2: An object can migrate from a class (e.g. class C) to a more general class (e.g. class Z). In this case, the object loses its membership in class C, while acquiring membership in class Z.

As illustration, consider the case when a manager instance is demoted to become an employee instance, again using the single inheritance situation in Example 1 of Section 2.

Case 3: An object can migrate from a class (e.g.

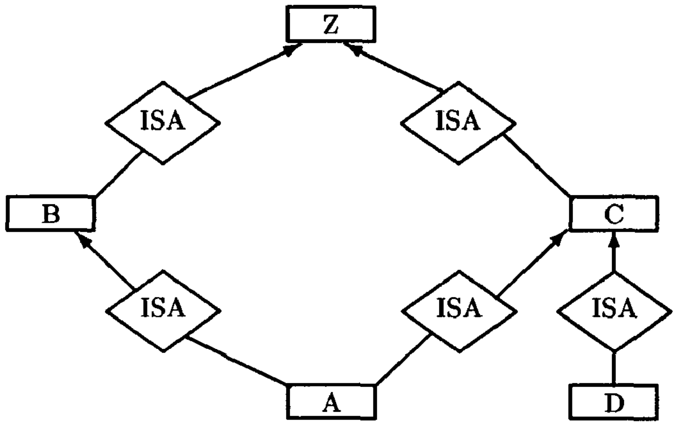

Fig. 1. An Example ISA hierarchy

class C) to a more specialised class (e.g. class A) which has two or more superclasses (e.g. classes B and C). These superclasses must have a common superclass (e.g. class Z). In this case, the object acquires membership in B and A, and retains its membership in C.

For example, consider a multiple inheritance situation such that STUD-EMP is a subclass of both STUDENT and EMPLOYEE, which are subclasses of PERSON. When a STUDENT object migrates to STUD-EMP, it becomes an instance of both STUDENT and EMPLOYEE. Note that it is not possible for distinct instances of two distinct classes to migrate to a common subclass and become a single instance of the common subclass. For example, it is not possible for a STUDENT instance and an EMPLOYEE instance, both of which are unrelated, to migrate to STUD-EMP and become a single instance of STUD-EMP.

Case 4: In a multiple inheritance situation, an instance of a class with multiple superclasses (e.g. class A) can migrate to at most one of its direct superclass(es) (e.g. to class C), but *never* to two or more of its direct superclasses (e.g. to both classes B and C). In moving from class A to C, an instance retains its membership in C (and its superclasses) but loses its membership in A and the superclasses of A that are not superclasses of C (i.e. it loses membership in B but retains membership in Z). If it is desired to migrate to two or more direct superclasses, we can always create a common subclass of these superclasses and migrate the object to this common subclass.

For example, consider the multiple inheritance situation given in Case 3. An instance of STUD-EMP can migrate to either STUDENT or EMPLOYEE, but not to both classes. If it migrates to STUDENT, it loses its membership in EMPLOYEE, and the superclasses of EMPLOYEE that are not also superclasses of STU-

DENT. Similar considerations apply if it migrates to EMPLOYEE.

Case 5: An object can migrate from a class (e.g. class C) to another class (e.g. class B) only if classes C and B have a common superclass. This is logically equivalent to case 2 (or case 4) followed by case 1 (or case 3). To be semantically meaningful, the common superclass cannot be an all-encompassing superclass such as ROOT or OBJECT, which is the superclass of all classes in the system. The object loses its membership in class C and acquires membership in class B.

For all cases, an object is constrained to migrate within the ISA hierarchy, as defined in a schema. This is not a restrictive condition. It is not semantically reasonable for an object to migrate arbitrarily from a class X to another class Y such that there is no ISA relationship between X, Y or their superclasses, if any. For example, it is not possible for a SUPPLIER object to migrate to a PART class, but it is possible for a TECHNICIAN object to migrate to become an ENGINEER object, as long as TECHNICIAN and ENGINEER have a common superclass, say EMPLOYEE.

Our approach is different from that in [9], which uses the concept of *aspects* to model a real world entity that plays different roles or adopts multiple facets. An aspect extends an object with a new state and behaviour while still retaining the object's OID. An object of a class X may acquire many aspects which are not ISA-related to the class X. An object of another class Y can acquire aspects meant for class X as long as the type of Y conforms[9] to the type of X. While flexible, it may lead to some semantically meaningless aspects being acquired. For example, a person object with only name and age attributes may have an aspect EMPLOYEE defined for it. A bird object, also with only name and age attributes, can acquire the EMPLOYEE aspect, since it conforms to the type for the person object.

Note that it is difficult to implement object migration using a deletion followed by an insertion. Each deletion means that references to the deleted object must be nullified. Identifying these dangling references is difficult, if not impossible, since they can be embedded in complex object structures throughout the database.

4 Resolving Object Migration Problems

To resolve the OID ambiguity problem, a number of variables must be considered, viz. the storage scheme that is used to store ISA hierarchies, the representation chosen for OID and the ability to handle the associated problems of message ambiguity and object identification. We discuss these variables in detail in subsequent subsections. The selection of a particular storage scheme for ISA hierarchies will *not* solve the OID ambiguity problem, but it determines whether it is more amenable (or less troublesome) to support single OID or multiple OIDs for a migrating object. The representation chosen for an OID (e.g. physical/structured addresses, typed or untyped surrogates[3]) determines the methods used to solve the message ambiguity and object identification problems.

4.1 Hierarchy Storage Schemes

Three possible storage schemes for ISA hierarchies are described below, using the single inheritance situation in Example 1 of Section 2 to illustrate each of these schemes.

Scheme 1 : Use a monolithic structure to store the entire hierarchy, thus achieving a flattened table structure for the hierarchy.

Under this scheme, the hierarchy of Example 1 in Section 2 can be stored by using the MANAGER record structure and storing all MANAGER and EMPLOYEE instances in this structure. This is not a satisfactory approach because most employees are not managers and therefore there are likely to be many null values within this structure. We will not consider scheme 1 further in this paper.

Scheme 2 : For an instance of a subclass, only the values of attributes defined or specified in the subclass and either inherited key attributes or OIDs of corresponding superclass instances, or both, are stored; the values of its other inherited attributes are not stored with the subclass, but with the respective superclass(es).

Consider again the hierarchy of Example 1 (Section 2). For all employees (including employees who are managers), store the values of all employee-related attributes within an EMPLOYEE structure, and store only the values of manager-related attributes within a MANAGER structure. In order to access the EMPLOYEE information of a MANAGER object, we have to store, with the MANAGER object, either (1) a pointer (e.g. the OID of the EMPLOYEE instance, distinct from the OID of the MANAGER instance) or (2) the values of the key attributes of the EMPLOYEE instance, as in the relational approach, or (3) both pointer and key attribute values. Values of attributes inherited from EMPLOYEE, possibly other than the key attributes, are not stored in the MANAGER structure.

This approach facilitates retrieval of all employee records, via a scan of the EMPLOYEE structure. However, to retrieve all manager records, it requires a retrieval of all manager objects from the MANAGER structure, and for each of these manager objects, the matching employee object in the EMPLOYEE structure must also be retrieved.

Scheme 2 seems to be particularly amenable to supporting multiple OIDs for an object as it migrates from one class to another along the ISA hierarchy. To see why this is so, consider the situation when an employee is promoted to manager. Its manager-related information and, say, the employee OID are inserted into the MANAGER structure as a manager object. A manager OID (distinct from the employee OID) is assigned to the manager object and inserted into a hash or (physical) address lookup table. Suppose, on the contrary, that a manager is demoted to become an employee again. In this situation, the manager object is deleted and its OID is cleaned up from the hash or lookup table. Existing references to the manager object are no longer valid and can rightly be highlighted as such.

Scheme 2 is probably not suitable for supporting a single OID for a migrating object. The OID must be mapped to a single storage location, but scheme 2 is such that an object which is an instance of several classes in the ISA hierarchy has several storage locations associated with it. It is necessary to chain these storage locations using, say, address pointers.

Scheme 3 : An object that is an instance of multiple classes in the ISA hierarchy is *only* stored in the structure of its most specific class, which includes the set of attributes either defined or inherited by its most specific class in the ISA hierarchy.

In Example 1 (Section 2), store information of employees who are not managers in an EMPLOYEE structure. Store information of managers in a MANAGER structure, which also holds the values of employee-related attributes inherited from EMPLOYEE.

Such a storage scheme facilitates retrieving all manager-related information, but requires a scan of two structures in order to retrieve all employee records. Consider a request to retrieve n manager records from the database. Using scheme 3, it requires only n accesses to the MANAGER structure. In scheme 2, it requires $2n$ accesses, i.e. n accesses to the MANAGER structure and another n accesses to the EMPLOYEE structure. Note that a retrieval of n' employee records using either scheme 2 or scheme 3 requires the same number of accesses to the database.

Scheme 3 is suitable for supporting a single OID for an object as it migrates along the ISA hierarchy. There is only one storage structure associated with an object, even though it is an instance of several classes in the ISA hierarchy. The single OID assigned to the object can therefore be mapped unambiguously to the location of this storage structure. In contrast, scheme 3 is clearly not suitable for supporting multiple OIDs for a migrating object.

4.2 Representation for OID

As observed in [1], an OID is often defined in terms of implementation (e.g. as surrogates or pointers). There are at least four possible representations for OID[3]:-

- Using the physical (disk) address of an object. This is rarely used because it does not allow an object to be moved. Note that during the runtime of an OO program, the OID of a *non-persistent* object is typically the main memory (virtual) address at which the object is located.

- Using a structured address, comprising a physical component (e.g. segment and page number) and a logical offset (e.g. into a physical page). This approach allows an object to be moved within a page, or to another page by using a forwarding technique, but the OID is still dependent on the physical location of the object. Examples of OODBMSs that use this method include ONTOS and Objectivity/DB. In O2[4], record identifiers generated by the underlying storage manager are used as OIDs for objects. Such record identifiers are typically structured addresses. ObjectStore[7] uses the virtual memory address of the object in cache as its OID. While the original intention of OID is that it should also be independent of the (virtual) address of the location that the object was stored in[6], the use of structured addresses as OIDs means that OIDs are no longer logical.

- Using untyped surrogates. The OID is purely logical and must be mapped to the physical address of an object, via a lookup or hash table. The hash table is likely to be huge and therefore cannot be placed in memory for performance. Furthermore, the updating of this hash table becomes a performance issue when objects are moved on disk as a result of database reorganisation, or when objects are deleted/created. Generally, a smaller in-memory hash table is maintained that maps the OIDs of in-memory objects to its physical address. When an object is needed whose OID is not in the hash table, an object fault occurs; the object is then brought into memory and an entry made in the hash table. Untyped surrogate OIDs often have poorer retrieval performance[3]. POSTGRES[10], IRIS[5] and GEMSTONE[2] are examples of OODBMSs using untyped surrogates.

In OODBMSs that use either physical addresses, untyped surrogates or structured addresses to represent OIDs, the ID of the class which an object belongs to is kept as a separate system defined attribute of the object. When a message is sent to an object, the system must fetch the object, retrieve the class ID and then access the class definition to check for message validity. Invalid messages can cause needless object accesses and type checking is expensive.

- Using typed surrogates, each of which is a pair $< C, I >$ where C is the identifier of the class to which the object belongs, and I is the identifier of the instance either within the class or within the whole database. Again, a hash table is maintained that maps the OID of an in-memory object to its physical address. An advantage of using typed surrogate is that the validity of a message sent to an object can be checked without fetching the object. However, this representation makes the migration of an object from one class to another very difficult, since the OID, and any references to it, must be updated. This approach is therefore expensive and really violates the idea of having immutable OIDs.

OIDs can be generated using any algorithm that is guaranteed to produce unique IDs (e.g. using date/time, or a monotonically increasing function). GEMSTONE uses its STONE monitor to dynamically allocate OIDs, while POSTGRES has a software routine to handle OID assignments. Invariably, the routine that generates OIDs becomes a bottleneck. To overcome this problem, GEMSTONE allocates OIDs in blocks.

Using physical or structured addresses for OIDs is not true to the spirit of the original proposal for using OIDs[6]. As hardware becomes faster and memory cheaper, the performance gains from using physical addresses for OIDs will become insignificant. Therefore, only typed and untyped surrogates offer reasonable representations for OIDs.

4.3 Semantics for Message Processing

Under the classical OO approach to message processing, if a message is despatched to an object of a class X, the definition for class X is searched to see if there is a method defined for X that matches the signature of the message. If found, the method is executed. Otherwise, a search up the class hierarchy is initiated until either a method is found or the search returns a failure. The underlying assumption here is that the class associated with the object is its most specific class. This assumption may not be true in the presence of object migration, since a real world object may be an instance of multiple classes, and there may be a more specific instance of the object which has to be considered to process the message correctly, as example 1 in Section 2 shows.

To resolve this problem, a message despatched to an object should be processed starting from the most specific class that the object is a member of. Only if the message definition is not found in the most specific class should its superclasses be searched. Note that subclasses, if any, of the most specific class associated with an object need not be searched, because the object is not an instance of any of these subclasses. For example, if an employee is not a manager, and a message is sent to the employee to invoke the *bonus()* method, then clearly, the search for the *bonus()* method implementation should start and end at the EMPLOYEE class.

Note that the search for a superclass for which the message applies should only yield at most one superclass; If two or more superclasses have this method, an inheritance conflict is present which we assume has been resolved by using the method proposed in [8].

4.4 Approaches to Resolve the OID Ambiguity Problem

There are four possible approaches for resolving the OID ambiguity problem:

Approach 1 : Using multiple OIDs, with OIDs represented by typed surrogates. In this implementation, every object has associated with it a typed surrogate as its OID. When an object migrates to a new class, it acquires an additional OID. An object that migrates may therefore end up with multiple OIDs.

This approach faces the twin problems of object identification and message ambiguity. One way to resolve these problems is to link an instance to its superclass and subclass instances via OIDs, as discussed below.

We make use of the following concepts. The fan-in of a class F_i is the number of superclasses of the class. The fan-out F_o of a class is the number of subclasses of the class. Since an object is also an instance of its superclasses, if any, the object is linked to its superclass instances via the OIDs of these superclass instances. There are at most F_i such OIDs. An object of a class X may have a more specialised instance of itself in a subclass of X. Links are setup between the object and all its specialised instances in the subclasses by using the OIDs of these specialised instances. There are at most F_o such OIDs. When an object O migrates from a class

C to its subclass C_s, an instance is created in C_s, say O_s, which still retains its link to O. The object O also has its link to O_s updated simultaneously. Existing references to O remain valid, since O (and its OID) still exists. Therefore, given an object, it is possible to traverse the ISA hierarchy, retrieving more generalised instances of the object from the superclasses, if any, and more specialised instances of the object from the subclasses, if any. The overhead associated with keeping the subclass and superclass information with an object is a constant and is $F_i + F_o$ OIDs per object instance. The problem of object identification is solved using this structure, because instances of an object are now linked via OIDs.

The message ambiguity problem is solved since the most specific class associated with the object can be retrieved by traversing the links.

As noted in Section 4.1, storage scheme 2 is more suitable for storing the ISA hierarchies for Approach 1. Using Approach 1 means that an object's OID is not unique and not immutable when it migrates from one class to another.

Approach 2 : Using multiple OIDs, with OIDs represented by untyped surrogates. Again, the immediate problems with this approach are those of object identification and message ambiguity. To resolve these problems, the same data structure described in Approach 1 can be used. The methods to solve these problems are exactly the same as in Approach 1.

As discussed in Section 4.1, storage scheme 2 is more suitable for storing the ISA hierarchies for Approach 2. Using this approach, an object's OID is not unique and not immutable when it migrates from one class to another.

Approach 3 : Using single OID, with the OID represented by a typed surrogate. Suppose that a migrating object is assigned a single typed surrogate OID. The OID must be updated to reflect the most specific class associated with the object. For example, suppose an employee is represented by the OID $< E, I_1 >$, where E is the identifier associated with the EMPLOYEE class and I_1 is an instance identifier. If the employee becomes a manager, its OID should be updated to become $< M, I_2 >$, where M is the identifier associated with the MANAGER class and I_2 is an instance identifier. Any references to $< E, I_1 >$ in the database must be updated to $< M, I_2 >$, since the former no longer exists. The updating of existing references can be expensive, if not impossible. Suppose the manager is demoted to be an employee. A new OID should be assigned to the demoted employee, say $< E, I_3 >$. It is meaningless and really without much benefit to reuse $< E, I_1 >$ for the demoted employee. To do so would mean that an object can have multiple typed surrogates as its OIDs (i.e. similar to Approach 1).

Using this approach, there is no problem of object identification, since an object has only one OID. The message ambiguity problem is solved since the object's OID is associated with the most specific class of the object.

Storage scheme 3 is more suitable for storing the ISA hierarchies for this approach, as discussed in Section 4.1. Using this approach, a migrating object's OID is not immutable but is unique in the database.

Approach 4 : Using single OID, with the OID represented by an untyped surrogate. In this implementation, every object is assigned an untyped surrogate OID. The OID is used to hash into an auxiliary table which stores the most specific class associated with the object, as well as the physical address of the object on disk. For example, suppose an object is an instance of MANAGER, EMPLOYEE and PERSON. Then, in the auxiliary table, the class information associated with this object is MANAGER. When the object migrates, the OID is retained, but the class information in the auxiliary table is updated to the most specific classs of the object.

The message ambiguity problem is resolved since the most specific class (and the physical address) for an object can be retrieved via a lookup of the auxiliary table. Message processing up the ISA hierarchy, starting from the most specific class, can be done using schema information, if necessary.

Storage scheme 3 is more suitable for storing the ISA hierarchies for this approach (see Section 4.1). Using this approach, a migrating object's OID is immutable and unique.

From the above discussions, we can see that approach 4, together with storage scheme 3, offers the best solution to the OID ambiguity problem. First, it is truer to the spirit of using OID, since each real world object has only one unique OID, regardless of its possible instances in the classes of the ISA hierarchy. There is no problem of object identification. Second, it does not require the setup of links between instances along the ISA hierarchy. It only requires a hash or lookup table, which is needed anyway to map an OID to its physical address. Approach 1 is better than approach 2 because class information is part of the OID and, unlike approach 2, needs not be separately looked up. Approach 3 is the worst of the four approaches, since an object's OID must be changed for every migration and existing references to the object must be updated.

5 Conclusion

In this paper, we provided semantics for meaningful object migration along an ISA hierarchy. This defined clearly the situations under which meaningful object migration can take place.

We addressed the problems associated with object migration. The primary concern for object migration is that it allows an object to be an instance of multiple classes in an ISA hierarchy, and therefore the question of whether it should have a single or multiple OIDs arises. We referred to the problem of deciding whether to have a single or multiple OIDs for a migrating object as the OID ambiguity problem.

The OID ambiguity problem was resolved by considering a number of variables, viz. the storage scheme for storing ISA hierarchies, the representations chosen for OIDs, and the ability to resolve two other associated problems, viz. the message ambiguity and object identification problems.

We examined three possible storage schemes for ISA hierarchies. Some storage schemes are suitable for supporting a single OID for a migrating object, while other schemes are more amenable to supporting multiple OIDs for a migrating object. For the representations of OIDs, typed and untyped surrogates are viable representations for OIDs and truer to the spirit of using OIDs. Two examples were given to illustrate and motivate the message ambiguity problem. A solution to this problem was described, based on a re-examination of the classical OO way of handling a message that is despatched to an object. The object identification problem exists only when multiple OIDs are assigned to a migrating object. A data structure was described that resolves the object identification problem.

Four possible approaches were then considered to resolve the OID ambiguity problem. The first approach used typed surrogates to support multiple OIDs for an object during migration. The second approach used untyped surrogates to support multiple OIDs for an object during migration. The third approach used a typed surrogate to support a single OID for a migrating object. The fourth approach used an untyped surrogate to support a single OID for a migrating object. Each of these approaches works well with one of the three storage schemes discussed in this paper. We recommended the fourth approach, with an associated storage scheme, as a superior approach to solving the OID ambiguity problem because it retains the desirable OID properties of uniqueness and immutability and has less overheads compared to the other approaches.

References

[1] F. Bancilhon, *A Logic Programming/Object-Oriented Cocktail*, ACM Sigmod Record, Vol. 15, No. 3, Sep 1986.

[2] P. Butterworth, A. Otis, J. Stein, *The Gemstone Object Database Management System*, Communications of the ACM, Vol. 34 No. 10, Oct 91.

[3] R. Cattell, *Object Data Management: Object-Oriented and Extended Relational Database Systems*, Addison Wesley, 1991.

[4] O. Deux et. al., *The Story of O2*, Communications of the ACM, Vol. 34 No. 10, Oct 91.

[5] D. Fishman, D. Beech, H. Cate, E. Chow, T. Connors, J. Davis, N. Derrett, C. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. Neimatt, T. Ryan, M. Shan, *Iris: An object-oriented database management system*, ACM Trans. Office Information Syst., Vol. 5, pp 48-69, Jan 1987.

[6] S. Khoshafian, G. Copeland, *Object Identity*, Proc. OOPSLA, Portland, Oregon, Sep 1986.

[7] C. Lamb, G. Landis, J. Orenstein, D. Weinreb, *The ObjectStore Database System*, Communications of the ACM, Vol. 34 No. 10, Oct 91.

[8] T.W. Ling, P.K. Teo, *Inheritance conflicts in object-oriented systems*, Proc. Database and Expert Systems Applications, Prague, Czech Rep., Sep 1993.

[9] J. Richardson, P. Schwarz, *Aspects: Extending objects to support multiple, independent roles*, Proc. ACM Sigmod 91, Denver, Colorado, May 29-31, 1991.

[10] M. Stonebraker, L. Rowe, M. Hirohama, *The Implementation of POSTGRES*, IEEE Trans. Knowledge and Data Engineering, Vol. 2, No. 1, Mar 1990.

[11] S. Zdonik, *Object-oriented type evolution*, Advances in database programming languages, F. Bancilhon, P. Buneman eds., ACM Press, 1990.