A Schema Modification Methodology for an Object-Oriented Database System

Reda AL-HAJJ and M. Erol ARKUN
Bilkent University
Faculty of Engineering and Science
Department of Computer Engineering
and Information Sciences
06533 Ankara, TURKEY

ABSTRACT: *Proper and efficient handling of schema changes is an important aspect of any object-oriented database management system. This paper deals with the definition of a schema modification methodology for ODS; an object-oriented database management system prototype designed at Bilkent University. A brief description of the basic features of ODS is included to facilitate the treatment. The invariants of the class lattice as well as the rules for resolving ambiguities due to schema changes are identified. In addition, schema changes handled by ODS are described within the realm of the specified invariants and rules. The described methodology efficiently handles deletions which are unmanagable in object-oriented systems due to the unidirectional relationship between objects in different classes.*

Keywords : *database systems, object-oriented databases, schema evolution.*

1. Introduction

Object-oriented systems evolved to satisfy the needs of such application areas as Artificial Intelligence, Computer Aided Design/Manufacturing and Office Information Systems [4, 7, 14, 19, 21], where information about the domain is incomplete or becomes available incrementally or even highly subject to change. These application areas require flexibility in changing the database schema. This necessitates that schema changes should be one of the basic characteristics to be considered in judging the power of an object-oriented database system. The flexibility in the underlying data models adds much to the object-oriented database systems. Clearly, it is desirable for an object-oriented database system to satisfy as many schema operations as possible.

W. Kim classifies in [11] schema changes among a number of architectural concepts developed for relational database systems that should be treated within the realm of object-oriented data models. Extensibility is considered in [3] as one of the characteristics that a system must possess in order to be termed an object-oriented database system.

Different data models exist each allowing a wide variety of schema changes. O2 [7] provides two modes for running an application. Schema changes are allowed in the development mode, but not in the execution mode where the schema is frozen and changes to it are forbidden. Schema changes in ORION [4, 5, 6] are based on multiple inheritance while in GemStone [16] simple inheritance based schema changes are treated. In [17, 18] the approach used is based on keeping versions to maintain a consistent view of the type lattice after a schema change.

A basic consideration in schema evolution is how to bring existing objects in line with a modified definition of an existing class. Either all instances of a modified class are instantaneously changed or they are modified only when used, otherwise remain unchanged. ORION [6] and ObServer [19] follow the approach known as screening where the change is delayed and values are either filtered or corrected as they are used. Another approach, known as conversion, is used by GemStone [16] where all instances of a class are modified in accordance with the change. The first approach sounds more sensible as there is no need to do something that will not be used.

How different items of the database are treated by the storage model affects the flexibility of schema changes. In this paper, we address the schema modification methodology defined for ODS; an object-oriented database management system prototype designed at Bilkent University. These schema changes are performed through the user interface part of ODS [21] and are achieved via the flexibility in the storage and indexing models proposed in [1, 2], respectively. Section 2 includes brief descriptions of the basic features of ODS. The invariants and the conflict resolving rules which validate schema modifications of ODS are stated in section 3. The schema evolution functions

allowed by ODS are discussed in section 4. The impact of the storage and the indexing models on schema changes are discussed in section 5. Section 6 includes the conclusions.

2. Basic Features of ODS

The implementation of ODS has been carried out using the C programming language on a UNIX workstation. In ODS, a single entity of the real world is modeled as an object. ODS supports modeling of complex objects and relationships directly so that the properties of objects are not constrained to be simple data values. Each object in ODS is identified by a unique identifier called an object-oriented pointer (OOP). Object-oriented pointers are implemented through surrogates. An object table is used to map the OOP of an object to its physical location in main memory [8].

Objects are grouped into classes. A class is allowed to inherit from more than one class, i.e., ODS supports multiple inheritance. A class contains necessary information to construct and use its instances [8]. Class variables can be added to a class definition. The root of the class lattice is the special class called OBJECT. Users are allowed to define new classes that are direct or indirect subclasses of the OBJECT class.

To preserve the encapsulation feature, ODS does not allow direct access to values of objects. Object identity is the only thing visible to the outside of an object and message sending is the only means for manipulating the properties of an object [15]. For every message there is a corresponding method. Additionally, ODS supports a user interface that includes a class browser and a programming shell as described in [21]. The storage and indexing models developed for ODS are briefly described next.

In the storage model proposed in [1], the object storage model is treated as a 3-dimensional system: the instance dimension, the nesting dimension and the inheritance dimension. The following three relations are introduced in [1]:

(x R1 y) iff x and y are instances in the same class.
(x R2 y) iff y is an immediate nested chunk of x.
(x R3 y) iff y is an immediate super chunk of x.

Information in an object is distributed between the Segment that contains atomic valued instance variables of all the chunks of one class, the Inheritance Table (IT) that contains information relating class instances to the OOPs of their immediate supers, and the Nesting Table (NT) that contains information related to the immediate nested objects, i.e., objects that form the values for non-atomic valued instance variables. Relations R1, R2 and R3 are applied to objects in the database to build the Segment, NT and IT, respectively [1]. Such a storage structure facilitates the handling of schema modifications as described in this paper and provides a fine degree of granularity [1].

Indexing is required to improve the performance of object-oriented systems. In GemStone [13] the term identity index is used where a predefined path leads to a set of index components one for each term in the path. ORION [12] uses class-hierarchy indexing and nested-attribute indexing for what we call indexing in the inheritance and nesting dimensions, respectively. On the other hand, the key step in the indexing approach proposed in [1, 2] is the recognition that two of the basic object-oriented constructs are inheritance and complex objects. Accordingly, objects in different classes should be related via these two constructs. Two types of indexes are identified; inheritance-nesting index and equality index. The inheritance-nesting index is based on the inverse of the relations R2 and R3 defined via inheritance and complex object constructs.

(y $R2^{-1}$ x) iff an object x exists with an instance variable having object y as its value.
(y $R3^{-1}$ x) iff x is a subobject of y or x inherits from y.

Two types of variable length index records are constructed. The first shows for each object the identities of its subobjects, while the second shows the identities of objects that are referencing an object as the value for their non-atomic valued instance variables. The two types of records are collected in the SubObjects Table (SOT) for the first type and the Referencing Objects Table (ROT) for the second type [1].

Concerning equality index, a B^+ tree is built to relate the value of a particular instance variable to the identities of objects in a class. Using the equality index, the identities of the desired objects are obtained. Using the inheritance-nesting index it is possible to start at any object and reach to any other related object [1, 2].

Example: The aim of this example is to facilitate the understanding of the described constructs as well as to aid in showing the handling of the different schema evolution functions described next in Section 4.

Consider a class Cj with classes Ci1, Ci2, ..., Cin found in the superclass list of Cj and classes Ck1, Ck2, ..., Ckm found at the immediate nesting level of class Cj as shown in Figure 1. All the classes found in a given class lattice are handled by the same way as class Cj, because the described relationships are only applicable at both the immediate superclasses level and immediate nesting class level.

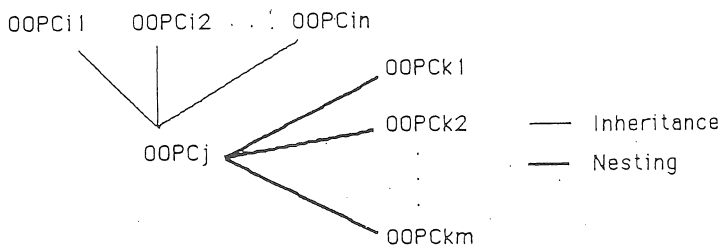

Figure 1: The general scene of a class in the class lattice

Concerning Figure 1, the information which should be included in the IT, NT, SOT and ROT is as follows:

IT(OOPCj,<OOPCi1, OOPCi2, ..., OOPCin>)
NT(OOPCj,<OOPCk1, OOPCk2, ..., OOPCkm>)
SOT(<OOPCi1, OOPCj>, <OOPCi2, OOPCj>, ..., <OOPCin, OOPCj>)
ROT(<OOPCk1, OOPCj>, <OOPCk2, OOPCj>, ..., <OOPCkn, OOPCj>)

3. Constraints of Schema Evolution and Conflicts Resolving Rules

After schema changes the class lattice should preserve some properties in order not to leave the database in an inconsistent state. This section includes the properties of the class lattice that ODS preserves upon schema modifications. In addition to the constraints there are some rules that are used to resolve conflicts resulting from schema changes.

Class Lattice Invariant

The class lattice of ODS is defined as a directed acyclic graph with a single root, the OBJECT class. It is connected, i.e., there is at least a path from each class to the root. Any schema change to the class lattice should maintain this property.

Non-empty Class Invariant

A user-defined class whose instance and class variables and methods are deleted is considered empty. An empty class only connects its subclasses to its superclasses without adding any property to the connection and hence its presence is meaningless. The class lattice should not contain any empty user-defined class. If a schema change results in an empty class, this class is automatically dropped.

Distinct Names Invariant

No two classes can have the same name. Furhter, no two methods defined within the same class

can have the same name. In addition, the instance variables within the same class should have distinct names. Name conflicts resulting from the inheritance of objects and methods are resolved according to the following two rules:

Name Conflict Resolving Rule
On name conflicts priority is given to the target class and the priority decreases while going away from the class towards the OBJECT class. For classes that have the same priority according to this rule, i.e., classes found in the same superclass list of the target class the conflict is resolved according to the order in the list. The class found at the head of the list is given the highest priority while that at the tail of the list is given the lowest priority. The OBJECT class has the least priority.

Inheritance Priority Rule
The addition of instance variables (methods) to a class and the deletion of instance variables (methods) from a class may violate conflicts that are resolved according to the above priority rule. An instance variable (method) in name conflict with another one may be inherited on deleting the latter. Further, currently inherited instance variables (methods) may cease to be considered after the addition of an instance variable (method) with the same name to a class with a higher priority.

Full Inheritance Invariant
A class inherits all the instance variables and methods from its superclass(es). No selection is done but name conflicts are resolved according to the name conflict resolving rule and the inheritance priority rule.

Homogeneous Domain Invariant
The domain for each instance variable should be bound to a specific class in the class lattice. When a class is specified as a domain, all its direct and indirect subclasses may be used in the same context because a class includes instances of its all direct and indirect subclasses.

Class Addition/Deletion Invariant
A new superclass A of an existing class B should be added as the last superclass in the order of superclasses of B. Moreover, on deleting B from the class lattice its supers should replace it as immediate superclasses of the immediate subclasses of class B. Superclasses of B are added at the end of the list because all such classes have lower priority than those classes in the same list with B. The OBJECT class should be the superclass of a newly added class unless the supers of the added class are explicitly specified.

Data Integrity Invariant
On deleting an instance from a class it is necessary to consider and drop the references to the deleted instance. No dangling reference should be kept.

Object Deletion Rule
The policy to be followed for the deletion operation is that an object can not be deleted unless it is a root object. The term root object means an object which is not an instance of any subclass; a root object has no entry in the SOT.

4. Schema Evolution Functions Allowed in ODS

In this section we described persistent schema changes allowed in our system. The description depends on the storage and indexing models of our system.

Dropping References to a Deleted Chunk
The introduction of the IT, NT, ROT and SOT facilitates much to the maintenance of database integrity. These tables help in dropping all the references to a deleted chunk in accordance with the data integrity invariant and without violating the object deletion rule. Via the ROT and SOT references to a deleted chunk can be efficiently dropped without any trouble. For instance on deleting Ci2 in Figure 1 from the class lattice, the reference from Cj to Ci2 is dropped inside the IT after being reached using the information in the SOT.

Adding an Instance to a Class
An instance is added to an existing class by adjusting the IT, NT, ROT, and SOT to reflect the

relation between the new instance and other related instances existing in the database. The new instance uses all the facilities available to other instances in its class.

Adding an Instance Variable to a Class

The addition of a new instance variable to a class can be done in two different ways each depending on whether the instance variable is atomic or non-atomic. In the first method, applicable when the new instance variable has an atomic value, the segment of the class to which the new instance variable is to be added, is accessed. The segment's contents are changed by adding the value of the new instance variable to each chunk in the segment. In the second method, applicable when the new instance variable has a non-atomic value, the NT is adjusted to reflect the relationship between instances of a class and the domain of the new added instance variable. According to the distinct names invariant, the new instance variable should have a name distinct from all variables in its class. Further, the addition of a new instance variable may affect the inheritance priority according to the name conflict resolving rule and the inheritance priority rule.

Changing the Domain of an Instance Variable

The domain of an existing instance variable can be changed as one of the following four choices: 1) from atomic domain to atomic domain; 2) from atomic domain to non-atomic domain; 3) from non-atomic domain to atomic domain; 4) from non-atomic domain to non-atomic domain. The first choice does not cause any trouble; but for the other three choices which contain a non-atomic domain the relationships are maintained in the NT and ROT to reflect the change.

Changing the Name of an Instance Variable

The name of an existing instance variable may be changed without any trouble. This is because data encapsulation is one of the distinguishing features of ODS and instance variables are treated by message passing. Such a schema change may affect the inheritance priority according to the inheritance priority rule and the name conflicting rule. Furthermore, the distinct names invariant should be satisfied.

Dropping an Instance Variable from a Class

It is possible to drop an existing instance variable from a class. The deletion of an instance variable depends on whether the instance variable has a chunk (or a collection) as its value or just an atomic value. An instance variable which has a chunk (or a collection) as its value can be deleted from the NT by deleting the information that represent the value of the instance variable from the record of the chunk. The ROT is adjusted to reflect the change. The deletion of atomic valued instance variables can be done within the related segment. Inheritance priority may change according to the name conflict resolving rule and the inheritance priority rule. Further, the non-empty class invariant should continue to hold after the deletion.

Adding a Class to the Superclass List of an Existing Class

An existing class may be added to the superclass list of another class without violating the class addition/deletion invariant. The addition is done at the tail of a superclass list. For instance an existing class, say Cir, may be appended at the tail of the superclass list of class Cj in Figure 1. Class Cir will have the lowest priority in the list.

Deleting a Class from the Superclass List of an Existing Class

A class C may be dropped from the superclass list of another class D. The super(s) of class C will replace it in the superclass(list) of its subclass D as immediate super(s). The SOT is adjusted to drop the reference(s) that existed prior to the deletion of class C. For instance on deleting Ci2 from the superclass list of Cj in Figure 1, classes found in the superclass list of Ci2 replace Ci2 in the superclass list of Cj.

Changing the Priority of the Superclasses of an Existing Class

The order of the superclasses of a certain class may be rearranged within the IT that shows for each class its immediate superclasses. For instance classes Ci2 and Cin may interchange their places within the superclass list of class Cj and hence the priority of Cin becomes higher than the priority of Ci2. Modifying the superclass list helps in changing the priority of the superclasses found at the same level with respect to a target class. These schema changes affect the inheritance priority maintained by both the name conflict resolving rule and the inheritance priority rule.

The Addition of a New Class

A new class may either have the OBJECT class as a superclass or else other existing classes in its superclass list. Furthermore, a new class may have zero,one or more subclasses. The non-empty class constraint and class lattice invariant should be maintained. According to the distinct names invariant, the name of the new class should not match with any of the existing classes. Chunks in the new class, being related by R1, are put in a new Segment. According to the full inheritance invariant, the new class inherits all instance variables and methods from its superclass(es). Instances in the new class are related to instances in other classes either by the relation R2 or R3 and hence $R2^{-1}$ and $R3^{-1}$. These relations are reflected by adding entries to the IT, NT, ROT and SOT to reflect the position of the instances of the new class with respect to existing instances of other classes. For instance considering class Cj in Figure 1 as an addition of a new class with the specified classes along both the inheritance and nesting dimensions as shown in Figure 1; the information which should be reflected into each of the IT, NT, ROT and SOT is the same information described in the example given in section 2.

Dropping a Class

The instances, instance variables and methods of an existing class C may be dropped. If all the definition and contents of class C are dropped, then class C should be deleted as an empty class to preserve the non-empty class invariant. The immediate supers of class C replace it in the inheritance mechanism for not to leave its subclasses dangling and violate the class lattice invariant. After the deletion, the inheritance priority may change according to the inheritance priority rule and the name conflict resolving rule.

Changing the Name of an Existing Class

The name of an existing class may be changed, but without violating the distinct name invariant. So, the new name assigned to the class should be different from those of other existing classes.

Adding a Method to a Class

It is possible to add a method to the definition of an existing class. But distinct names invariant should be preserved by forcing the name of the new method to be distinct from those existing in the same class. Moreover, inheritance priority may change following both the inheritance priority rule and the name conflict resolving rule.

Dropping a Method from a Class

An existing method may be dropped from the definition of an existing class. A method may be deleted after becoming irrelevant due to schema changes. A deleted method may be replaced by a new method added to the class definition or an existing method in another class according to both the inheritance priority rule and the name conflict resolving rule.

Finally, it is important to state that the algorithms used to handle the described schema modifications are found in [1].

5. The Impact of the Storage and Indexing Models on Schema Evolution Functions

An important advantage due to the separation of non-atomic references through IT, NT, ROT and SOT is realized here, whereby instance independence, nesting independence and inheritance independence are achieved. The IT, NT, ROT and SOT contain all the associations between chunks which are instances in different classes. Changes to an edge in the class lattice can be performed solely via the IT without any need to access individual chunks. Making a class a superclass of another class can be done solely via the IT that contains information related to the change. Removing a class from the superclass list of a class can again be done solely via the IT that contains information related to the change by dropping references to instances of the deleted class. Changing the order of the superclasses of a class can also be accomplished solely within the IT. By this way, inheritance independence is achieved. On the other hand, a schema change that involves a non-atomic valued instance variable is done inside the NT and the ROT is adjusted to reflect the change and so, nesting independence is achieved.

A new atomic valued instance variable may be added to a class definition requiring that the values of the new instance variable be added to the existing chunks found in the segment representing the

class. So the instance indepndence is achieved.

5.1 Schema Changes and Indexing

It is important to consider the relationship between schema changes and the indexing model. We claim that schema changes result only in minor changes to existing indexes, namely to the records of the classes that are directly involved in the change. Queries that use these indexes are not affected because a query that has recognized a change is instantiated in a forward and backward fashion. How and to what extent this claim is achieved is discussed next.

The identity index introduces SOT and ROT that show the immediate nesting and the immediate referencing objects for all objects found in the database. The effect of a schema change does not propagate further than the immediate neighbors of the changed item. A schema change will at most affect the immediate neighbor objects of the object that undergoes the change. For instance the addition of a new class between classes Cj and Ci2 in Figure 1, only affect entries related to Cj and Ci2 in the IT. The same thing is true for the addition of a class along the nesting dimension. So the addition of a class to the class lattice affects only the immediate subclasses, immediate nested classes, and immediate referencing classes and nothing more.

The identity indexes will not be affected by the addition of new instances to a class or the deletion of existing instances from a class where entries need to be added to or deleted from the SOT, ROT, IT and NT. For instance, suppose that instances in class 'student' reference instances in class 'person' and instances in class 'teacher' along the inheritance and nesting dimensions respectively. An identity index is automatically set up for instances of classes 'student', 'teacher', and 'person'. When the class 'course' is added between the classes 'student' and 'teacher', the identity index built before will be automatically extended to include instances of 'course' class. From the ROT only entries corresponding to the class 'teacher' are adjusted to show the course each 'teacher' instance is offerring instead of showing the students taking his course and each of the new entries that correspond to the 'course' class will refer to the students attending the course.

6. Conclusions

ODS supports the basic object-oriented concepts of object identity, class structure, inheritance, complex objects, encapsulation and schema modification, in addition to composite objects which is considered in [11] among the non-basic constructs. A schema modification methodology for ODS is described. The schema evolution functions are efficiently effected due to the flexibility that ODS gained from its particular storage and indexing models. The performance of the query processor is improved by physically dropping all references to a deleted chunk according to the object deletion rule, thus eliminating many instantiations otherwise required to recognize the absence of referenced chunks. Finally, because all the component chunks of a composite object are treated as a single chunk, we are currently addressing the problem of establishing a methodology for handling the rules and invariants for schema changes of such objects.

References

[1] Al-Hajj R., E. Arkun, A Model for Storage Management in Object-Oriented Database Management Systems, Proceeding of the Fifth International Symposium on Computer and Information Sciences, October 1990, Cappadocia , Turkey.

[2] Al-Hajj R., E. Arkun, A Model for Indexing in Object-Oriented Database Management Systems, Proceeding of the Fifth International Symposium on Computer and Information Sciences, October 1990, Cappadocia , Turkey.

[3] Atkinson M., et al., The Object-Oriented Database System Manifesto, Proceeding of the International Conference on Deductive Object-Oriented Databases, Kyoto, Japan, December, 1989.

[4] Banerjee J., et al., Data Model Issues for Object-Oriented Applications, ACM Transactions on Office Information Systems, vol.5, no.1, Jan.1987, pp.3-26.

[5] Banerjee J., et al., Schema Evolution in Object-Oriented Persistent Databases, Proc. of the 6th Advanced Database Symposium, Information Processing Society of Japan's Special Interest Group on Database Systems, Tokyo, Japan , August 1986, pp.23-31.

[6] Banerjee J., et al., Semantics and Implementation of Schema Evolution in Object-Oriented Databases, Proc. of ACM/SIGMOD International Conference on Management of Data, San Francisco, California, May 1987.

[7] Deux O., et al., The Story of O2, IEEE Transactions on Knowledge and Data Engineering, vol.2, no.1, March 1990, pp.91-108.

[8] Kesim N., Object Memory for an Object-Oriented Database Management System, M.S. Thesis, Bilkent University, Ankara, July 1988.

[9] Khoshafian S.N., and G.P. Copeland, Object Identity, Proceeding of ACM International Conference on Object-Oriented Programming Systems, Languages and Applicatoins, Sept. 1986.

[10] Kim W., H. Chou, and, J. Banerjee, Operations and Implementations of Complex Objects, IEEE Transactions on Software Engineering, Vol. 14, No. 7, July 1988.

[11] Kim W, Object-Oriented Databases: Definition and Research Directions, IEEE Transactions on Knowledge and Data Engineering, Vol.2, No.3, September 1990, pp.327-341.

[12] Kim W., et al., Indexing Techniques for Object-Oriented Databases, Object-Oriented Concepts, Applications, and Databases, W. Kim and F. Lochovsky, Eds., Addison-Wesley, 1989.

[13] Maier D., and J. Stein, Indexing in an Object-Oriented DBMS, Proc. of the Workshop on Object-Oriented Database Systems, September 1986.

[14] Maier D., A. Otis, and A. Purdy, Object-Oriented Database Development at Servio Logic, Database Engineering, IEEE, vol.8, no.4, December 1985.

[15] Özelçi M.S., Message Passing in an Object-Oriented Database Management System, M.S. Thesis, Bilkent University, Ankara, July 1988.

[16] Penney D.J., J. Stein, Class Modification in the GemStone Object-Oriented Database Management Systems, Proc. of Second International Conference on Object-Oriented Programming Systems, Languages and Applications, Orlando, FL, Oct. 1987.

[17] Skarra A.H., and S.B. Zdonik, The Management of Changing Types in an Object-Oriented Database, Proc. of ACM International Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, OR, September 1986, pp.483-495.

[18] Skarra A.H., and S.B. Zdonik, Type Evolution in an Object-Oriented Database, Research Directions in Object-Oriented Programming, ed. B. Shiver, and P. Wenger, MIT Press Series in computer Systems, 1987, pp.393-415.

[19] Skarra A., et al., An Object Server for an Object-Oriented Database, Proceeding of ACM/IEEE International Workshop on Object-Oriented Database Systems, 1986.

[20] Stefik M., and D.G. Bobrow, Object-Oriented Programming: Themes and Variations, AI Magazine, January 1986, pp.40-62.

[21] Türkmen S., C. Yengül, E. Arkun, An Object-Oriented Database System Prototype, Proceeding of the Fourth International Symposium on Computer and Information Sciences, October 1989, Cesme, Turkey.