VERSIONS OF SCHEMA FOR OBJECT-ORIENTED DATABASES

Won Kim and Hong-Tai Chou

MCC
3500 West Balcones Center Drive
Austin, Texas 78759

Abstract

Version control is one of the important database requirements for design environments. Various models of versions have been proposed and implemented. However, research in versions has been focused exclusively on versioning single design objects. In a multi-user design environment where the schema (definition) of the design objects may undergo dynamic changes, it is important to be able to version the schema, as well as version the single design objects. In this paper, we propose a model of versions of schema by extending our model of versions of single objects. In particular, we present the semantics of our model of versions of schema for object-oriented databases, explore issues in implementing the model, and examine a few alternatives to our model of versions of schema.

1. INTRODUCTION

In the Advanced Computer Architecture Program at MCC, we have built a prototype object-oriented database system, called ORION. Presently, ORION is being used in supporting the data management needs of PROTEUS, an expert system shell also prototyped in the Advanced Computer Architecture Program at MCC. In ORION we have directly implemented the object-oriented paradigm [GOLD81, GOLD83, BOBR83, SYMB84, BOBR85], added persistence and sharability to objects through transaction support, and provided various advanced functions that applications from the CAD/CAM, AI, and OIS domains require. Advanced functions supported in ORION include versions [CHOU86, CHOU88], composite objects [KIM87], dynamic schema evolution[BANE87b], and multimedia data management [WOEL87].

There is an extensive set of research reports on versions of design objects [ROCH75, TICH82, KATZ84, DITT85, ATWO85, KATZ86]. In a multi-user design environment, it is highly desirable to extend the notion of versions of objects to versions of schema (definition of the objects). One major motivation is to better preserve the history of evolution of objects, so that applications may derive versions of schema, and create and manipulate different sets of objects under different versions of schema. If the schema cannot be versioned, objects that existed before a schema change can in general be irreversibly changed. For example, if an attribute of a class is dropped, the values of the attribute in existing objects (instances) of the class become no longer visible to the application. If, however, a new version of the schema is derived in which the attribute of the class is dropped, the values of the attribute in existing objects continue to be visible to the application under the old version of the schema.

Another important motivation for versioning the schema is to support checkouts and checkins of objects in a federated system of private databases and a public database. A private database system running on an engineering workstation may check a complex design object out of the public database, modify it, and check the modified object into the public database as a new version of the object. If we are to allow the user to change the schema of the checked-out object in the private database, we must require the modified schema to be copied to the public database as a new version of the original schema before a new version of the object may be checked in.

To the best of our knowledge, except for [SKAR86] which provides a brief discussion of versioning a class (rather than the entire schema), the semantics and implementation of versions of schema have not been reported in the literature. In this paper, we make two original contributions to this important but largely unexplored area of research. First, we develop a model of versions of

schema. In doing this, we identify and address a number of semantic issues which arise in versioning the schema. By specifying a user interface, consisting of a surprisingly small set of commands, we show how the users may use our model of versions of schema. Second, we propose fairly detailed implementation techniques for supporting versions of schema in a database system.

The remainder of this paper is organized as follows. In Section 2, we briefly review basic object-oriented concepts. Section 3 provides a review of our model of versions of objects, and the semantics of schema evolution. In Section 4, we identify a number of major semantic issues in versioning the schema, and define our model of versions of schema. In Section 5, we propose implementation techniques for versions of schema, and indicate expected system overhead. Section 6 describes the user interface for our model of versions of schema. Section 7 provides a discussion of alternatives to our approach. The paper is summarized in Section 8.

2. BASIC OBJECT-ORIENTED CONCEPTS

In this section we review basic object-oriented concepts that are relevant to our discussions in the remainder of this paper. This section is extracted from our full paper on the ORION data model in [BANE87a].

objects, attributes (instance variables), methods, and messages

In object-oriented systems, all conceptual entities are modeled as objects. An ordinary integer or string is as much an object as is a complex assembly of parts, such as an aircraft or a submarine. An object consists of some private memory that holds its state. The private memory is made up of the values for a collection of attributes (often called instance variables). The value of an attribute is itself an object, and therefore has its own private memory for its state (i.e., its attributes). A primitive object, such as an integer or a string, has no attributes. It only has a value, which is the object itself. More complex objects contain attributes, through which they reference other objects, which in turn contain attributes.

The behavior of an object is encapsulated in *methods*. Methods consist of code that manipulate or return the state of an object. Methods are a part of the definition of the object. However, methods, as well as attributes, are not visible from outside of the object. Objects can communicate with one another through messages. *Messages* constitute the public interface of an object. For each message understood by an object, there is a corresponding method that executes the message. An object reacts to a message by executing the corresponding method, and returning an object.

classes, class hierarchy, inheritance, and domains

If every object is to carry its own attribute names and its own methods, the amount of information to be specified and stored can become unmanageably large. For this reason, as well as for conceptual simplicity, 'similar' objects are grouped together into a *class*. All objects belonging to the same class are described by the same set of attributes and methods. They all respond to the same messages. Objects that belong to a class are called *instances* of that class. (In this paper, we will use the terms instances and objects interchangeably.) A class describes the form (attributes) of its instances, and the operations (methods) applicable to its instances. Thus, when a message is sent to an instance, the method which implements that message is found in the definition of the class.

Grouping objects into classes helps avoid the specification and storage of much redundant information. The concept of a *class hierarchy* extends this *information hiding* capability one step further. A class hierarchy is a hierarchy of classes in which an edge between a pair of nodes represents the IS-A relationship; that is, the lower level node is a specialization of the higher level node (and conversely, the higher level node is a generalization of the lower level node). The root of a class hierarchy is the system-defined class CLASS. For a pair of classes on a class hierarchy, the higher level class is called a *superclass*, and the lower level class a *subclass*. The attributes and methods (collectively called properties) specified for a class are *inherited* (shared) by all its subclasses. Additional properties may be specified for each of the subclasses. A class inherits properties only from its immediate superclass. Since the latter inherits properties from its own superclass, it follows that a class inherits properties from every class in its *superclass chain*.

In object-oriented systems, the *domain* (which corresponds to data type in conventional programming languages) of an attribute is a class. The domain of an attribute of a class C may be explicitly bound to a specific class D. Then instances of the class C may take on as values for the attribute instances of the class D as well as instances of subclasses of D.

class lattice, multiple inheritance, and name-conflict resolution

In many object-oriented systems (and in ORION), a class can have more than one superclass, generalizing the class hierarchy to a lattice (or a directed acyclic graph). In a class lattice, a class inherits properties from each of its superclasses. This feature is often referred to as *multiple inheritance* [LMI85, STEF86]. The class lattice simplifies data modeling and often requires fewer classes to be specified than with a class hierarchy. However, it gives rise to conflicts in the names of attributes and methods. One type of conflict is between a class and its superclass (this type of problem also arises in a class hierarchy). Another, which is purely a consequence of multiple inheritance, is among the superclasses of a class.

Name conflicts between a class and its superclasses are resolved in all systems, including ORION, by giving precedence to the definition within the class over that in its superclasses. The approach used in many systems, and in ORION, to resolve name conflicts among superclasses of a given class is the use of *superclass ordering*. If an attribute or a method with the same name appears in more than one superclass of a class C, the one chosen by default is that of the first superclass in the list of (immediate) superclasses of C, which the application will have specified.

3. REVIEW OF VERSIONS AND SCHEMA EVOLUTION

In this section we review the model of versions of objects [CHOU88] and the semantics of schema evolution [BANE87b] which we support in ORION. The model of versions of objects is the basis of our model of versions of schema. We will limit our review to those aspects of our models of versions and schema evolution that are essential to understanding the subsequent sections of this paper.

3.1 VERSIONS

Our model of versions associates different *capabilities* with versions; that is, it distinguishes *transient versions*, which can be updated or deleted at will, from *working versions*, which can be deleted but not updated. A transient version may be created from scratch or *derived* from an existing version. The user may explicitly *promote* a transient version to a working version. A transient version may be implicitly promoted to a working version, if a new transient version is derived from it.

An object is either *versioned* or *non-versioned*. A versioned object is an instance of a class which the application declares to be *versionable*. Since any number of transient versions may be derived at any time from an existing version, a versioned object consists of a hierarchy of versions, called a *version-derivation hierarchy*. We use the term *version instance* to refer to a specific version in the version-derivation hierarchy for a versioned object, and *generic instance* to refer to the abstract versioned object. A generic instance maintains the history of derivation of version instances for a versioned object.

In object-oriented systems, every object is assigned an identifier which uniquely identifies the object in the system. In ORION, both the generic instance and a version instance of the generic instance have object identifiers. An object, either a version instance or a non-versioned object, may reference one or more other objects. If an object references a version instance, the reference may be the object identifier of a generic instance or that of a version instance. If the reference is to a version instance, we say that the object is *statically bound* to the version instance. If the reference is to a generic instance, the object is said to be *dynamically bound* to a *default version instance* of the generic instance. The capability to bind an object dynamically to a default version instance is useful, since transient or working versions that are referenced may be deleted, and new versions created. We allow the user to specify a particular version instance on the version-derivation hierarchy as the default version. In the absence of a user-specified default, the system will select the version instance with the 'most recent' timestamp as the default. The default version instance of a versioned object is recorded in its generic instance.

3.2 SCHEMA EVOLUTION

As we discussed in Section 2, the schema of an object-oriented database is the class lattice; and as such two types of changes to the schema are meaningful: changes to the definitions of a class (contents of a node) in the class lattice, and changes to the structure (edges and nodes) of the class lattice. Changes to the class definitions include adding and deleting attributes and methods. Changes to the class lattice structure include creation and deletion of a class, and alteration of the IS-A relationship between classes (adding and deleting the superclass-subclass relationship between a pair of classes).

Below we outline (but not fully describe) the semantics of some of the schema change operations. Later in this paper, we will illustrate our implementation of versions of schema in terms of the semantics of these operations.

1. *Add an attribute V to a class C*

If the new attribute V causes no name conflicts in the class C or any of its subclasses, V will be inherited by all subclasses of C. If V causes a name conflict with an inherited attribute, V will override the inherited attribute. If the old attribute was also locally defined in C, it is replaced by the new definition. Existing instances of the classes to which V is added receive the nil value.

2. *Drop an attribute V from a class C*

V is dropped from C and subclasses of C that inherited it. Existing instances of these classes lose their values for V. If C or any of its subclasses has other superclasses that have attributes of the same name as that of V, it inherits one of them.

3. *Make a class S a superclass of a class C*

The addition of a new edge from S to C must not introduce a cycle in the class lattice. C and its subclasses inherit attributes and methods from S.

4. *Add a new class C*

All attributes and methods from all superclasses specified for C are inherited, unless there are conflicts. If no superclasses of C are specified, the system-defined

root of the class lattice, CLASS, becomes the superclass of C.

5. *Drop an existing class C*

All edges from C to its subclasses are dropped; which means that the subclasses will lose all the attributes and methods they inherited from C. Next, all edges from the superclasses of C into C are removed. Finally, the definition of C is dropped, and C is removed from the class lattice. The subclasses of C continue to exist.

4. VERSIONS OF SCHEMA

As we observed in Section 1, one of the important database requirements for a multi-user design environment is for the users to be able to view and manipulate different sets of objects under different versions of the schema. We have explored three different approaches to satisfying this requirement. One is to view the entire schema (the entire class lattice) as a versioned object; this is the versions of schema approach. Another is to view each class as a versioned object; this is the versions of class approach. Another is to provide dynamic views, rather than versions, of the schema; this is the views of schema approach. We have selected the versions of schema approach, and developed a model of versions of schema by extending the concepts of version capabilities, version-derivation hierarchies, and default versions from versions of objects to versions of schema. The model also reflects our view that a version of schema is associated with a set of objects created under it. Because a comparison of the three approaches requires a deeper understanding of the semantic and implementation issues involved, we defer such a discussion to Section 7.

Before we proceed, we define some terms we will use throughout the remainder of this paper.

> A schema version SV-j is called a *descendant schema version* of a schema version SV-i, if SV-j is derived directly or indirectly from SV-i. Conversely, SV-i is called an *ancestor schema version* of SV-j.
>
> A schema version SV-j is called a *child schema version* of a schema version SV-i, if SV-j is derived directly from SV-i. Conversely, SV-i is called a *parent schema version* of SV-j.
>
> The schema version under which the application currently accesses and manipulates objects is called the *current schema version.*
>
> The schema version under which an object was created is called the *creator schema version* of the object.
>
> The *access scope* of a schema version SV is the set of objects which are accessible to SV.
>
> The *direct access scope* of a schema version SV is the set of objects which are created under SV.

The object-oriented paradigm models all logical entities uniformly as objects with unique object identifiers. However, we may distinguish three types of objects for the purposes of version semantics: (instance) objects of a class, class objects, and the schema object. We will show in Section 5 that the implementation techniques we propose for our model of versions of schema make it unnecessary to support versions of class objects. Therefore, throughout this paper, when we talk about manipulating objects under a schema version (read, insert, delete, replace), we mean manipulating instance objects, but not class objects.

Our model of versions of schema may be expressed in terms of seven rules. The first three rules are extensions of the major concepts in our model of versions of objects.

Schema-Version Capability Rule: A schema version may be either a transient schema version or a working schema version. A transient schema version may be updated or deleted; and it may be promoted to a working schema version any time. A working schema version cannot be updated. A working schema version may be deleted or demoted to a transient version, if it has no child schema version.

We note that both a working version of the schema and a working version of an object are non-updatable. However, as we mentioned earlier, a working version of an object may be deleted at any time.

Schema-Version Derivation Rule: Any number of new versions of schema may be derived at any time from any existing schema version, giving rise to a version-derivation hierarchy for the schema. A derived schema version is initially a transient version. If a schema version is derived from a transient schema version, the transient schema version is automatically promoted to a working schema version.

Schema-Version Deletion Rule: A schema version which is a leaf node in the schema-version derivation hierarchy can be deleted, regardless of whether it is a working version or a transient version. A schema version cannot be deleted, if it has any child schema version. When a schema version is deleted, its direct access scope is also deleted.

The schema-version deletion rule makes it clear that a schema version 'owns' the objects created under it, that is, its direct access scope. Two fundamental questions arise concerning the access scope between the creator schema version and a descendant schema version. One is whether the access scope of a schema version SV-i should be inherited into a descendant schema version SV-j. Another is, if the access scope of

SV-i is inherited, whether it should remain updatable (replaced or deleted) under SV-i. The following rule has been developed to address these questions. It will be discussed in detail shortly.

Access-Scope Inheritance Rule: When a schema version SV-j is derived from a schema version SV-i, SV-j by default inherits the access scope of SV-i. However, the user may optionally block the inheritance of the access scope of the parent schema version. Further, once SV-j inherits the access scope of SV-i, the access scope of SV-i becomes by default non-updatable under SV-i. Again, however, the application may optionally leave the access scope of SV-i updatable under SV-i, when inheriting the access scope from SV-i.

The access-scope inheritance rule is the basis of the access-scope rule which defines the objects visible to a schema version in the schema-version derivation hierarchy. Two additional rules explain the update capabilities under a schema version: the direct-access-scope update rule specifies when objects created under a schema version SV can and cannot be updated under SV; and the inherited-access-scope update rule defines what it means to update objects under a schema version SV, when the objects are inherited from an ancestor schema version of SV.

Access-Scope Rule: The access scope of a schema version SV is the set of objects created under SV and those objects in the inherited access scopes of the ancestor schema versions. All objects in the access scope of SV are visible to SV, which means that they may be read or updated (inserted, replaced, deleted) under SV. No other objects are visible to SV.

Direct-Access-Scope Update Rule: The access scope of a schema version SV is non-updatable (no insert, no delete, no replace) under SV, if any schema version SV-k has been derived from SV, unless each SV-k has been derived from SV by blocking the inheritance of SV's access scope or by leaving the access scope of SV updatable under SV.

Inherited-Access-Scope Update Rule: All inherited objects in the access scope of SV may be updated or deleted. However, updates and deletes under SV of the objects inherited into SV are only visible to SV and the descendant schema versions of SV which inherited the access scope from SV.

A basic premise of our model of schema versions is that the access scope of a schema version SV may be inherited into any schema version derived from SV. The default access-scope inheritance rule reflects this. A major advantage of this approach is that by allowing automatic inheritance of the access scope of a schema version SV-i into any new schema version SV-j derived from SV-i, it avoids needless copying of those objects of SV-i which are to be visible to SV-j. This approach is particularly appropriate, if the objective of deriving a new schema version is to experiment with the impacts of a new definition of the schema on existing objects. However, it is inappropriate, if only a relatively small subset of the objects in the access scope of the parent schema version needs to be visible to the derived schema version.

If a schema version SV-j, derived from a schema version SV-i, inherits the access scope of SV-i, it is reasonable to disallow further updates to objects of SV-i under SV-i. If objects of SV-i are updated under SV-i after SV-j has been derived, the creator of SV-j will see different objects of SV-i at different times. However, with this restriction, to update any objects of SV-i, after SV-j has been derived from it, the creator of SV-i will have to derive a new transient schema version SV-k from SV-i, even if there may be no differences between SV-i and SV-k, and update the objects under SV-k.

The discussions above point out that a strict adherence to the default access-scope inheritance rule may not always be desirable. This is the reason for the exceptions to the rule. We allow the application to optionally leave the access scope of a schema version SV-i updatable under SV-i when deriving a new schema version SV-j from SV-i, and even block the inheritance of the access scope altogether. There are then three possible options in deriving a schema version SV-j from a schema version SV-i. We note that the users may dynamically change the option any time after the initial derivation of a schema version (this is further explained in Section 6).

1. SV-j inherits the access scope of SV-i, and the access scope of SV-i is made non-updatable under SV-i.
2. SV-j inherits the access scope of SV-i, but the access scope of SV-i remains updatable under SV-i.
3. SV-j does not inherit the access scope of SV-i, and it is immaterial whether the access scope of SV-i is updatable under SV-i.

The access scope of a schema version SV is the set of objects created under SV and those inherited from the ancestor schema versions. For example, if a schema version SV-i is the parent schema version of a schema version SV-j, and SV-j is the parent of a schema version SV-k, and SV-j inherits the access scope of SV-i, and SV-k inherits the access scope of SV-j, the access scope of SV-k is the set of objects created under SV-i, SV-j, and SV-k.

For added flexibility, we will allow the users to further restrict the access scope of a schema version to the set of objects from a range of continuous sequence of ancestor schema versions, from the current schema version to a specified ancestor schema version.

Figure 1 illustrates the access-scope inheritance rule and the access-scope rule. In the figure, three schema

versions are shown with the objects that are visible to them. Under the initial schema version SV-0, two versioned objects, a1 and a2 are created, along with their first version instances a1.v0 and a2.v0, and then a second version instance of a1, a1.v1, is derived. The objects a1 and a2 belong to the same class, and has three attributes, at1, at2, and at3. Then a new schema version, SV-1 is derived from SV-0, by deleting attribute at3 from the class. All objects created under SV-0 are now visible to SV-1, without the deleted attribute at3. Further, a new version instance v1 of a2 is created, and a new versioned object a3 (along with its first version instance a3.v0) is created under SV-1. Then a new schema version SV-2 is derived from SV-1 by adding a new attribute at4 to the class. All objects and their attributes visible to SV-1 are now made visible to SV-2, along with the new attribute at4. A new version instance of a3 is derived under SV-2.

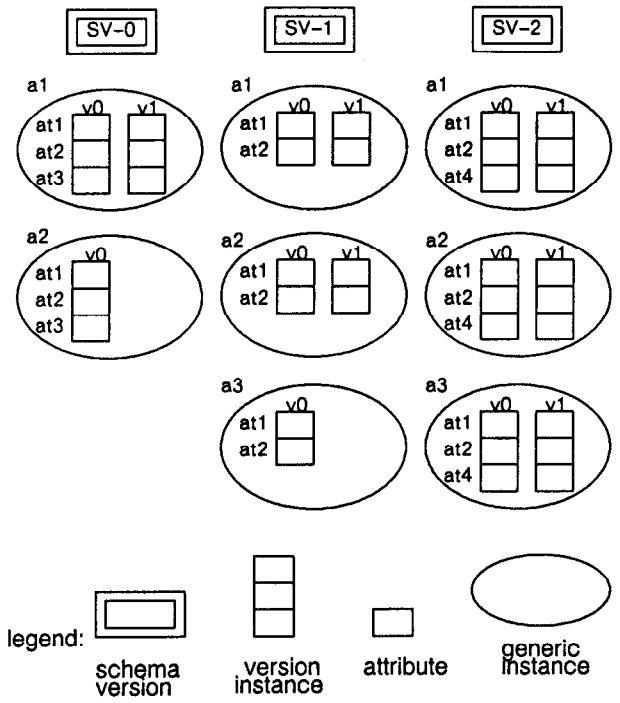

Figure 1. Example Schema Versions

Now, we consider the direct-access-scope update rule. There may be conflicts between different schema versions derived from the same schema version SV with respect to updatability of the access scope of SV. For example, a schema version SV-j may have been derived from SV by leaving the access scope of SV updatable under SV, and a new schema version SV-k is then derived from SV by making the access scope of SV non-updatable under SV. We take the view that when the user derives a schema version SV-j from SV by leaving the access scope of SV updatable under SV, the user in effect does not care whether the access scope of SV is updatable (which means any changes will automatically propagate to SV-j). Therefore, we give preference to the default access-scope inheritance rule over exceptions. In the current example, derivation of the new schema version SV-k will make the access scope of SV non-updatable under SV. By the same token, if SV-k was derived first, the access scope of SV will remain non-updatable under SV, when SV-j is derived next.

Next, we consider the inherited-access-scope update rule. If objects are inherited into SV-j from SV-i, the effects of the updates and deletes made under SV-j are visible only to SV-j; that is, when viewed from SV-i, it is as if the updates and deletes had never taken place. We restrict insertion (creation) of new objects to only the current schema version; that is, newly created objects belong to the direct access scope of the schema version under which they are created.

In the case of an update (replace) of an inherited object under a schema version SV, the new object is made visible to all descendant schema versions of SV which inherit the access scope of SV. The new object persists, even if the old object is deleted under its creator schema version.

Further, when an inherited object is deleted under a schema version SV, the object will not be visible to SV and any descendant schema version of SV which inherits the access scope of SV. However, the object will continue to be accessible to any other schema version which includes the creator schema version SV in its access scope.

5. IMPLEMENTATION ISSUES

In this section, we discuss auxiliary data structures for objects to efficiently support versioning of the schema, detailed algorithms for accessing objects, and storage representation for the schema versions.

5.1 AUXILIARY DATA STRUCTURES FOR OBJECTS

To support object manipulation in the presence of versions of schema, we need to include in every object one system-defined attribute, called the creator attribute, and to associate a data structure, called an anchor instance, with every object. The *creator attribute* of an object indicates the creator schema version of the object. The *anchor instance* of an object is a data structure which describes a set of copies of the object; recall that, in a similar manner, a generic instance of a versioned object describes the set of version instances of the object. Algorithms for object fetch, insert, delete, and replace are presented in the next subsection.

To support delete or replace of an inherited object, the system will create a new copy of the object when the object is first deleted or replaced under a schema version

which is not its creator schema version. An object, when first created, exists without an anchor instance; however, it will carry the identifier of the creator schema version. An anchor instance, and each of the copies of the object it describes, are all identified by the same object identifier of the object; this is necessary so as not to invalidate existing references to the object. The creator attribute in each copy of the object is used to distinguish one copy from any other.

The anchor instance of an object consists of the following system-defined attributes.

1. a list of terminator schema versions
2. a list of copies of the object

A *terminator schema version* is the schema version under which the object was deleted. We note that terminator schema version can only be a descendant of the creator schema version; if an object is deleted or replaced under its creator schema version, it is physically deleted or replaced, respectively.

Figure 2 illustrates the way in which an anchor instance is created and manipulated. In the figure, the anchor instance is shown with a terminator, and each copy of the object includes the creator schema version. The anchor instance and copies of the object are created in the following sequence. The object initially exists as copy-0, and SV-0 is its creator schema version. The anchor instance is created when the object is updated under schema version SV-4, and a new copy of the object, copy-1, is created with SV-4 as the creator schema version. The anchor instance describes two copies of the object. Then copy-0, which is still accessible to schema version SV-1, is deleted under SV-1, causing schema version SV-1 to be recorded in the terminator attribute of the anchor instance.

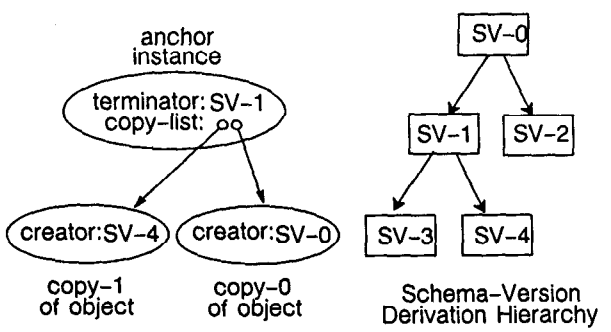

Figure 2. An Anchor Instance and Object Copies

5.2 OBJECT ACCESS ALGORITHMS

(This section may be skipped without loss of continuity.) The following algorithms precisely describe the way in which an object is fetched, inserted, replaced, and deleted. They use the augmented object structure and the anchor instance discussed earlier. In the algorithms, the parameter sv denotes the current schema version. We assume the existence of two boolean functions: (1) Ancestor-of, which returns TRUE when its first argument is an ancestor schema version whose access scope is inherited by the schema version specified in the second argument; and (2) Frozen, which returns True when the argument specifies a schema version whose direct access scope is non-updatable under the schema version. For convenience, we define another boolean function

```
Ancestor-Of* (sv1, sv2) =
        Ancestor-Of (sv1, sv2) OR sv1 = sv2.
```

```
find_closest_copy (copy-list, sv)
/* routine for finding the ancestor copy that is closest to
sv */
  for each c in copy-list
    if ancestor-of* (c.creator, sv) return c;
          /* an ancestor copy found */
  end-for;
  return nil;
end find_closest_copy;

copy_blocked (terminators, sv, copy-sv)
/* routine for checking if a copy is visible under sv */
  for each t in terminators
    if (ancestor-of* (t, sv)) and (ancestor-of (copy-sv, t))
      return TRUE; /* copy-sv is blocked from sv by t */
  end-for;
  return FALSE; /* sv is not blocked by any terminator */
end copy_blocked;

ALGORITHM_SV_FETCH (object-id, sv)
  object <- locate_object (object-id);
  if (object is an anchor instance)
    do anchor-instance <- object;
       object <- find_closest_copy
                    (anchor-instance.copy-list, sv);
       if (object = nil) or
          copy_blocked(anchor-instance.terminators,
                                  sv, object.creator)
         error; /* no copy is visible under sv */
    end-do;
  else if not (ancestor-of*(object.creator, sv))
          error; /* the only existing copy is not visible */
  return object;
end algorithm_sv_fetch;

ALGORITHM_SV_INSERT (object, sv)
  if frozen (sv) error;
  object.creator <- sv;
  allocate an id for the object and return the id;
end algorithm_sv_insert;

ALGORITHM_SV_DELETE (object-id, sv)
  if frozen (sv) error;
  object <- locate_object (object-id);
  if (object is an anchor instance)
```

```
  do anchor-instance <- object;
     object <- find_closest_copy
                  (anchor-instance.copy-list, sv);
     if (object = nil) or
        copy_blocked(anchor-instance.terminators,
                                 sv, object.creator)
          error; /* no copy is visible under sv */
     else if (object.creator = sv) /* delete by creator */
           do remove object from
                    anchor-instance.copy-list;
            if (anchor-instance.copy-list = nil)
                  remove anchor-instance;
             else add sv to anchor-instance.terminators;
            end-do;
           else add sv to anchor-instance.terminators;
  end-do;
 else /* there is no anchor instance */
  if (object.creator = sv) remove object;
  else if (ancestor-of (object.creator, sv))
        create an anchor instance for object-id
        with sv and (object) as the values of its
        terminators and copy-list attributes,
        respectively;
       else error; /* the only existing copy is not visible
                  under sv */
end algorithm_sv_delete;


ALGORITHM_SV_REPLACE (object-id, new-object, sv)
  if frozen (sv) error;
  object <- locate_object (object-id);
  if (object is an anchor instance)
    do anchor-instance <- object;
       object <- find_closest_copy
                  (anchor-instance.copy-list, sv);
       if (object = nil) or
          copy_blocked(anchor-instance.terminators,
                                 sv, object.creator)
         error; /* no copy is visible under sv */
       else if (object.creator = sv)
                 object.data <- new-object.data;
                /* update in place */
            else do new-object.creator <- sv;
                    add new-object to
                    anchor-instance.copy-list
                    as the first element;
                 end-do;
    end-do;
  else /* there is no anchor instance */
   if (object.creator = sv) object.data <- new-object.data;
         /* update in place */
   else if (ancestor-of (object.creator, sv))
         create an anchor instance for object-id with nil
         and (new-object object) as the values of its
         terminators and copy-list attributes,
         respectively;
        else error; /* the only existing copy is not visible
                  under sv */
end algorithm_sv_replace;
```

It is clear that the boolean function Ancestor-Of is an important factor in the performance of the algorithms. To minimize this overhead, the ancestor relationship can be encoded in bit vectors, one for each schema version. The bit vector associated with the root schema version contains all zero's. When a new schema version SV-j is derived and inherits objects from SV-k, a new bit vector BV-j is created and initialized as follows:

BV-j[i] = 1 for i = k, and BV-j[i] = BV-k[i] for all other i's, where BV-k is the bit vector of SV-k. If SV-j does not inherit objects from SV-k, BV-j is initialized to all zeros. It is easy to prove by induction that BV-j[i] is 1 if and only if SV-j inherits instances from SV-i. Thus, we have reduced the check for ancestor relationship to a vector access.

When a schema version is derived, the user can choose not to inherit any instances from the ancestor schema versions. However, class objects are always inherited by the new schema version, at least at the time of derivation. Thus we need another bit vector for each schema version to guide the retrieval and updates of class objects.

5.3 STORAGE REPRESENTATION FOR THE SCHEMA

In this subsection, first we present the schema representation in ORION, and then describe our proposal for representing schema versions and the schema-version derivation hierarchy.

representation for a single schema

In ORION, we represent the schema (without versioning it) as a set of class objects, where a class object is represented as a set of instances of several system-defined classes. These classes are analogous to system catalogs in conventional database systems [IBM81]. Three of these classes are shown, in a simplified form, in Figure 3. For each class, attribute, and method defined, there is a corresponding instance in the class Class, Attribute, and Method, respectively.

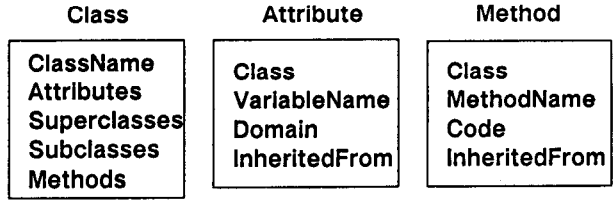

Figure 3. Classes for the Schema

The class Class contains attributes ClassName, Attributes, Superclasses, Subclasses, and Methods. ClassName is the name of the class. Attributes is the set of all attributes defined for or inherited into the class. The

attributes Superclasses and Subclasses are sets of superclasses and subclasses of the class, respectively. Methods is a set of methods defined for or inherited into the class. We emphasize that the Attributes and Methods attributes for a class hold values for not only the attributes and methods defined for the class, but also those inherited from all superclasses. This technique is known as 'flattening' of the class lattice [ZARA85], and is used often to speed up access to the schema.

The class Attribute (Method) has an instance for every attribute (method) defined for or inherited into each class. The class Attribute has attributes Class, VariableName, Domain, and InheritedFrom. The attribute Class references the class to which the attribute belongs. Domain specifies the class to which the value of the attribute is bound. InheritedFrom refers to an instance of the class Attribute, and it indicates the attribute of the superclass from which the attribute is inherited.

representation for schema versions

We now address the issue of maintaining schema versions in the database. Since one full copy of the schema can require significant storage space, our objective is to not maintain a physically separate copy of the entire schema for each version of the schema. Our solution is quite simple. We note that all changes to the schema are either changes to the definition of a class or changes to the relationship between classes, and that the relationships between classes are encoded in the class objects. We continue to maintain instances of the system-defined classes Class, Attribute, and Method as non-versioned objects, but apply the anchor instance structure to class objects to support updates to a class object under different schema versions. We need no changes to the basic representation for a single schema described above. When the definition of a class is changed, a copy of the class object for the class is created. When the relationship between a pair of classes changes (e.g., when a new subclass S of a class C is created, a class S is made a new superclass of a class C, etc.), a copy of the class object for each of the classes is created.

We illustrate our storage representation for schema versions using Figure 4, in which the class lattice is constructed and modified under five schema versions. Next to the nodes and edges of the class lattice, we indicate the schema version under which they are created or manipulated. Each of the five anchor instance structures represents each of the five class objects in the class lattice. The creator schema version is indicated immediately below each copy of a class object, while the terminator is indicated below the anchor instance. The anchor instances are in the final state that results from the following sequence of five schema changes. First, under schema version SV-0, classes C-a, C-b, and C-c are defined. Second, class C-d is created as a subclass of

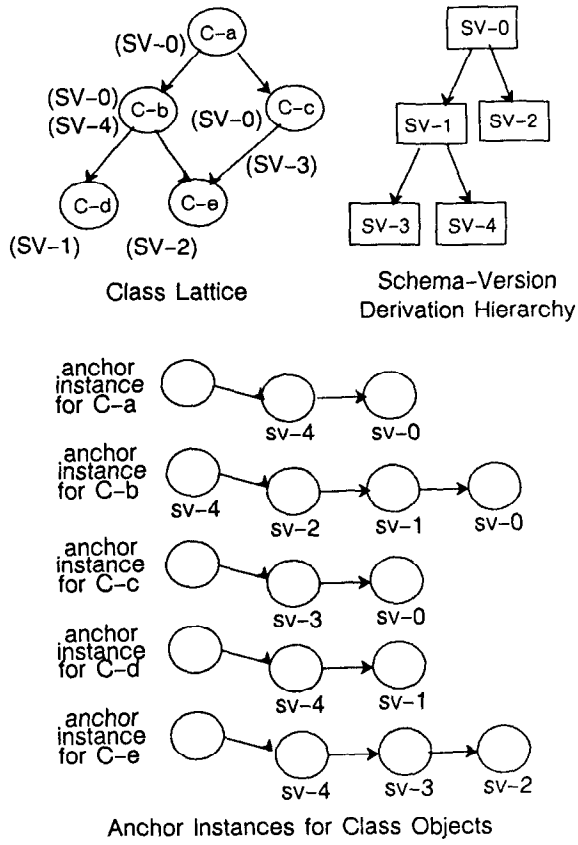

Figure 4. Representation of Schema Versions

C-b under schema version SV-1. This requires an automatic creation, under SV-1, of a new copy of the class object for C-b, since it must reference the class object for C-d as a subclass. Third, under schema version SV-2, class C-e is defined as a subclass of class C-b. Again, this causes the automatic creation of a copy of the class object for C-b under SV-2. Fourth, under schema version SV-3, the class C-c is made a superclass of C-e. A new copy of the class object for C-c, and a new copy of the class object for C-e are automatically generated, so that the former will reference the latter as a subclass, and the latter references the former as a superclass. Fifth, under schema version SV-4, the class C-b is deleted, making C-a the immediate superclass of C-d and C-e. This makes the class object for C-b inaccessible under SV-4, and causes the creation of a new copy of the class object for C-a, C-d, and C-e. We leave it to the reader to verify the correctness of the final state of the anchor instance structure for each of the class objects.

representation for the schema-version derivation hierarchy

We also need a data structure to describe the derivation hierarchy of schema versions. A simple solution is to

create a system-defined class SCHEMA for schema versions. This class will have only one instance, which is versionable. Each schema version is a version instance of this versioned object, and the schema-version derivation hierarchy is maintained in the generic instance associated with the versioned object. The class SCHEMA has two attributes, Update-Instances and Ancestor-SVs. Update-Instances is a boolean variable which is False when the access scope of the schema version is made non-updatable by a child schema version. Ancestor-SVs contains the list of ancestor schema versions from which the schema version inherits the access scope.

6. USER INTERFACE

In this section, we specify messages the user (application) can send to use our model of schema versions. The set of messages is surprisingly small, despite the rich semantics that the model captures.

(**derive-schema-version** parent-SV, highest-ancestor-SV, update-instances)
This command is used to return a new schema version derived from an existing schema version, specified in the parent-SV parameter. The schema version specified in the parent-SV parameter, if it is presently a transient schema version, is automatically promoted to a working schema version.

The highest-ancestor-SV parameter specifies the highest ancestor schema version whose access scope is to be inherited into the new schema version. The default is **root**, the root of the schema-version derivation hierarchy, consistent with the access-scope inheritance rule. If the value of the parameter is **self**, the access scope of the parent schema version is not inherited into the new schema version. Otherwise, the access scope of the schema version being derived is the set of objects in the direct access scope of each of the schema versions from parent-SV to highest-ancestor-SV.

The default value of the update-instances parameter is False by the access-scope inheritance rule. Then the objects in the access scope inherited into the schema version being derived become non-updatable under their respective creator schema versions. If update-instances is True, derivation of the present schema version has no impact on the updatability of the inherited instances under their respective creator schema versions.

(**change-inheritance** highest-ancestor-SV, update-instances)
This command is used to allow changes to the highest-ancestor-SV, and the update-instances parameters, specified when deriving a schema version, after the schema version has been derived.

(**delete-schema-version** SV)
This command is used to delete a schema version SV, along with all version instances created under it. If the specified schema version has at least one child schema version, the command is rejected.

(**promote-schema-version** SV)
This command is used to upgrade the status of a transient schema version SV to a working schema version. If SV is already a working schema version, the command has no effect.

(**set-current-schema-version** current-SV)
This command is used to switch the schema to the schema version specified in the current-SV parameter. *All database operations are performed under the current schema version, including changes to the schema, updates to the access scope of the schema, and creation of new objects.*

(**current-schema-version**)
This command is used to return the current schema version.

7. ALTERNATIVE APPROACHES

In this section, we explore a couple of alternative approaches to versions of schema. One is to treat each class object, rather than the entire schema, as a versionable object. Another is to support views, rather than versions, of the database schema.

7.1 VERSIONS OF CLASSES

It is clear that our model allows versioning of simple instance objects, and, of course, the schema. The question is whether we should support versioning of class objects, either instead of versioning the schema or in conjunction with versioning of the schema. The proposal briefly outlined in [SKAR86] is to treat the class objects as versionable objects, and not the schema. If the schema is not versioned, a 'virtual' version of the schema is constructed as a lattice of versioned class objects; of course, only one version instance of a class object will be included in any 'virtual' version of the schema.

Versioning the class objects has a few problems. When the schema is not explicitly versioned, the user must nevertheless manage the 'virtual' versions of schema by keeping track of which versions of class objects belong to which 'virtual' versions of schema. In the representation of the class objects, since the class objects are versioned, dynamic binding may be used for references from one class to its superclasses and subclasses. The system will resolve any reference to a generic class object to a default version of the class object. The user then must maintain, for each 'virtual' schema version, a list of default versions for all references to generic instances of the class objects! Further, the use of dynamic binding of references to class objects implies that the 'virtual' schema version cannot be flattened for efficient access!

Even if only static binding is used for references to versions of the class objects, the situation is not much

better. For example, suppose a new version v-j of a class C is derived from version v-i; and remember that the different versions of a class object have different object identifiers. Since each class object has the SuperClasses and SubClasses attributes, all subclasses of the class C must now reference the new version of C under the new 'virtual' schema version. This will mean that a new version must be created for each of the subclasses of C. Similarly, a new version must be derived for each of the superclasses of C. In other words, if we are to use versions of the class objects to support 'virtual' schema versions, we will end up generating a new version of the entire class lattice for each 'virtual' schema version!

The above problems also arise even if the schema is also explicitly versioned, as long as the class objects are also versioned. Our model does not support versioning of the class objects; instead, we use copies of the class objects to support updates of objects inherited from ancestor schema versions. In our approach, a schema version is a lattice of class objects, where only one copy of any class object is selected. The copies of a class object roughly correspond to versions of a class object in the class versioning approach. The only difference between the two approaches is that our approach allows only one copy (version) of each class object within any schema version. As such, the users cannot experiment with alternative definitions of a class within one schema version; of course, however, they may derive alternative schema versions, each with a different definition of the class, and experiment with the database by inheriting the access scope of the original schema version. On the other hand, the flexibility available (which we discussed above) in the class-version approach creates more opportunities for mistakes. For example, by allowing any two schema versions to share the same default of a class object, changes to the class object under one schema version may generate surprises in the other schema version.

7.2 VIEWS OF THE SCHEMA

We have been able to identify two difficulties with our model of versions of schema. One is the system overhead in supporting updates of objects inherited from an ancestor schema version. We feel that the solution we presented in Section 5 is reasonably good; however, it still requires a non-trivial system overhead. Another difficulty is that updates of inherited objects may cause confusion to the users. Suppose, for example, that a user, operating under a schema version SV-k, deletes an object inherited from a schema version SV-j, which in turn inherited the object from a schema version SV-i. If the user then operates under SV-j, the object will be visible again. Of course, this is exactly the desired effect; however, it may nonetheless be somewhat confusing to the user.

The cause of these difficulties is that each schema version is associated with an access scope, and the access scope of a schema version is updated from a descendant schema version. One way to alleviate these difficulties maybe to simply provide dynamic *views* of a single underlying schema. In this model, as in our model, any number of views may be derived from any schema view, giving rise to a derivation hierarchy of schema views. However, this model does not admit the notion of inheriting objects from an ancestor schema view (remember that inheritance of objects and updates of inherited objects are the cause of the problem we are attempting to address). Instead, all objects are associated with the single underlying schema, and the access scope of each schema view is the subset of all objects in the database whose attributes are defined by the schema view. In other words, all updates to objects under one schema view become visible to all schema views which include the definition of the attributes of the objects. This means that the schema view approach does not associate with any view a snapshot of the state of the database at some point in time. This contrasts sharply with our model, in which the set of objects manipulated under a schema version SV may be preserved with respect to schema versions derived from SV.

8. SUMMARY

In a multi-user design environment, versioning of design objects and versioning of the schema for these objects are important requirements. There exists an extensive set of research reports on versions of design objects, and some systems have even implemented version control. However, there has been virtually no formal proposal to define the semantics of versioning the schema, although many professionals in the design and database communities have talked of the importance of research into versions of schema. In this paper, we described a model of versions of an object-oriented schema, using as its basis the model of versions of objects which we have implemented in the ORION prototype object-oriented database system. In particular, the model includes such notions as capabilities, derivation hierarchy for versions of schema; and incorporates the view that each version of the schema captures the state of the database at some point in time and that the state may be inherited into any derived schema version for read and update.

Next we presented the data structure for representing objects and the schema, and algorithms for manipulating objects under our model of versions of schema. Then we discussed two possible alternative approaches to achieving some of the objectives which we we tried to meet with our model of versions of schema. These alternatives include versioning of the class objects and dynamic views of a single schema.

REFERENCES

[ATWO85] Atwood, T.M. "An Object-Oriented DBMS for Design Support Applications," *Proc. IEEE COMPINT 85*, Montreal, Canada,pp. 299-307.

[BANE87a] Banerjee, J., et al. "Data Model Issues for Object-Oriented Applications," *ACM Trans. on Office Information Systems*, April 1987.

[BANE87b] Banerjee, J., W. Kim, H.J. Kim, and H.F. Korth. "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," in *Proc. ACM SIGMOD Conference on Management of Data*, San Francisco, CA., May 1987.

[BOBR83] Bobrow, D.G.. and M. Stefik. *The LOOPS Manual*, Xerox PARC, Palo Alto, CA., 1983.

[BOBR85] Bobrow, D.G. et al. *CommonLoops: Merging Common Lisp and Object-Oriented Programming*, Intelligent Systems Laboratory Series ISL-85-8, Xerox PARC, Palo Alto, CA., 1985.

[CHOU86] Chou, H.T., and W. Kim. "A Unifying Framework for Versions in a CAD Environment," in *Proc. Intl Conf. on Very Large Data Bases*, August 1986, Kyoto, Japan.

[CHOU88] Chou, H.T., and W. Kim. "Versions and Change Notification in an Object-Oriented Database System," to appear in *Proc. 25th Design Automation Conference*, June 1988.

[DITT85] Dittrich K. and R. Lorie. "Version Support for Engineering Database Systems," *IBM Research Report*: RJ4769, IBM Research, Calif., July 1985.

[GOLD81] Goldberg, A. "Introducing the Smalltalk-80 System," *Byte*, vol. 6, no. 8, August 1981, pp. 14-26.

[GOLD83] Goldberg, A. and D. Robson. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA 1983.

[IBM81] SQL/Data System: Concepts and Facilities. GH24-5013-0, File No. S370-50, IBM Corporation, Jan. 1981.

[KATZ84] Katz, R. and T. Lehman. "Database Support for Versions and Alternatives of Large Design Files," *IEEE Trans. on Software Engineering, vol. SE-10, no. 2*, March 1984, pp. 191-200.

[KATZ86] Katz R., E. Chang, and R. Bhateja. "Version Modeling Concepts for Computer-Aided Design Databases," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Washington, D.C., May 1986.

[KIM87] Kim, W., et al. "Composite Object Support in an Object-Oriented Database System," in *Proc. Object-Oriented Programming Systems, Languages, and Applications*, Oct. 1987, Orlando, Florida.

[LMI85] *ObjectLISP User Manual*, LMI, Cambridge, MA, 1985.

[ROCH75] Rochkind M. "The Source Code Control System," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, December 1975, pp. 364-370.

[SKAR86] Skarra, A.H., and S. Zdonik. "The Management of Changing Types in an Object-Oriented Database," in *Proc. Object-Oriented Programming Systems, Languages, and Applications*, Oct. 1986, Portland, Oregon.

[STEF86] Stefik, M., and D.G. Bobrow. "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, January 1986, pp. 40-62.

[SYMB84] *FLAV Objects, Message Passing, and Flavors*, Symbolics, Inc., Cambridge, MA, 1984.

[TICH82] Tichy W. "Design, Implementation, and Evaluation of a Revision Control System," *IEEE 6th International Conference on Software Engineering*, September 1982.

[WOEL87] Woelk, D., and W. Kim. "Multimedia Information Management in an Object-Oriented Database system," in *Proc. Intl Conf. on Very Large Data Bases*, Sept. 1987, Brighton, England.

[ZARA85] Zara, R.V. and D.R. Henke. "Building a Layered Database for Design Automation," in *Proc. 22nd Design Automation Conf.*, 1985, pp. 645-651.