VERSIONS AND CHANGE NOTIFICATION IN AN OBJECT-ORIENTED DATABASE SYSTEM

Hong-Tai Chou and Won Kim

MCC
3500 West Balcones Center Drive
Austin, Texas 78759

ABSTRACT

At MCC we have built a prototype object-oriented database system, called ORION to support applications from the CAD/CAM, AI, and OIS domains. Advanced functions supported in ORION include versions, change notification, composite objects, dynamic schema evolution, and multimedia data. The versions and change notification features were based on the model we developed earlier in [CHOU86]. In this paper, we show how we have integrated our model of versions and change notification into the ORION object-oriented data model, and also provide an insight into system overhead that versions and change notification incur.

1. INTRODUCTION

In the Database Program at MCC, we have built a prototype object-oriented database system, called ORION. ORION has been implemented in Common LISP [STEE84], and runs on the Symbolics LISP machine. Presently, it is being used in supporting the data management needs of PROTEUS, an expert system development environment prototyped in the AI / KBS Program at MCC. In ORION we have directly implemented the object-oriented paradigm, 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, change notification, composite objects [BANE87a], multimedia data management [WOEL86], and dynamic schema evolution [BANE87b].

In [CHOU86], we proposed a model of versions and a model of change notification. The model was developed to support versions and change notification in integrated CAD systems, incorporating explicitly the characteristics of CAD environments and CAD databases. An integrated CAD system is viewed as a federated system consisting of a number of engineering workstations and a central server. Each workstation database system will manage the local database of the workstation, and the server database system will serve as a remote mass storage device for workstation databases and coordinate the sharing of data among the workstation database systems. Since the current ORION prototype is a workstation-based system, our implementation has been restricted to only that part of our model which is relevant to versions and change notification in a workstation.

Since ORION is an object-oriented database system, we had to integrate the model of versions and change notification into the object-oriented data model of ORION. There are extensive sets of research reports on versions and change notification [ROCH75, TICH82, NEUM82, KATZ84, DITT85, KATZ86] and object-oriented languages and systems [GOLD81, GOLD83, BOBR83, CURR84, COPE84, AHLS84, SYMB84, IEEE85, STEF86]. However, except for the limited discussions in [AHLS84, ATWO85, ZDON86], integration of versions and change notification into object-oriented systems has not been discussed.

This paper has two objectives (and, we feel, important original contributions). First, we show how models of versions and change notification can be integrated into an object-oriented data model. Second, we provide insight into the overhead incurred in supporting versions and change notification in a database system.

The remainder of this paper is organized as follows. In Section 2, we briefly review the object-oriented concepts and describe messages which the applications can use to work with such concepts. In Section 3, we describe our model and implementation of versions. Section 4 describes our model and implementation of change notification. The description of the implementation, in each section, consists of two parts: extensions to the basic object-oriented messages to integrate our semantics of versions and change notification into the ORION object-oriented data model, and major implementation issues and the system overhead incurred. Section 5 provides a summary of the paper.

2. REVIEW OF OBJECT-ORIENTED CONCEPTS

In this section, we review basic object-oriented concepts that are relevant to our discussions in the remainder of this paper, and then describe messages through which the application can work with the concepts. The way in which we have integrated our models of versions and change notification in ORION will be described in terms of extensions to these basic messages.

2.1 BASIC CONCEPTS

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 instance variables (often called attributes). 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 mes-

sage by executing the corresponding method, and returning an object.

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). 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. If the domain of an attribute of a class C is the class D, an instance of C may take on as the value for that attribute any instance of the class D or a subclass of D. The domain of an attribute may also be a set of instances of a class D and subclasses of D.

2.2 BASIC MESSAGES FOR AN OBJECT-ORIENTED DATABASE

Below we present briefly basic messages for defining, creating, and accessing objects in an object-oriented database. The syntax of the messages, as presented in an abbreviated form in this paper, is similar to that of a number of other object-oriented languages, such as Flavors [LMI85] and LOOPS [BOBR83]. The messages for versions and change notification will be presented later in this paper as extensions to these basic messages.

The following message creates the definition of a new class.

(**define-class** Classname **:superclasses** ListofSuperclasses
:attributes ListofAttributes
:methods ListofMethodSpecs)

Classname is the name of the new class. All keyword arguments are optional. The ListofSuperclasses associated with the **:superclasses** keyword is a list of the superclasses of the new class. The ListofAttributes associated with the **:attribute** keyword is a list of attribute specifications. An attribute specification is a list consisting of an attribute name and keywords with associated values, as follows:

(AttributeName [**:domain** DomainSpec]
[**:inherit-from** Superclass])

A DomainSpec is either a class or a set of classes. If the keyword **:inherit-from** is specified, the associated value is the name of the superclass from which the attribute will be inherited. Otherwise, the attribute is inherited on the basis of superclass ordering.

The ListofMethodSpecs associated with the **:methods** keyword is a list of pairs (MethodName SuperClass). The MethodName is the name of a method to be inherited from the SuperClass. The SuperClass is a class name. If the keyword **:methods** is not specified, methods are inherited from superclasses, and conflicts are resolved on the basis of superclass ordering.

As a simplistic example, the following class definition may be used to describe a generic structure of cells in a VLSI design:

(**define-class** cell **:attributes** ((cell-name **:domain** string)
....
(input-pins **:domain** pins)
(output-pins **:domain** pins)))

An instance can be created by sending a **create** message to the class to which the instance will belong. Using the above cell example, a cell instance can be created by

(**create** cell :cell-name "AND-gate" ...).

To select all instances (or any one instance) of a class that satisfy a given Boolean query expression, we use a **select** (or **select-any**) message. A *set object* (possibly an empty set) containing these instances is returned. The messages for selection have the following format, where QueryExpression is a Boolean expression of predicates:

(**select** Class QueryExpression)
(**select-any** Class QueryExpression)

To delete all instances of a class that satisfy a given Boolean query expression, a **delete** message is used.

(**delete** Class QueryExpression)

To delete a specific object, a delete-object message is used;

(**delete-object** Object)

where Object is the object identifier.

Similarly, a change message is used to replace the value of an attribute of all instances of a class that satisfy a given Boolean expression.

(**change** Class QueryExpression AttributeName NewValue)

3. VERSIONS

In this section we describe first our model of versions, and then two aspects of our implementation of versions. In Section 3.1, we describe our model of versions. In Section 3.2, we discuss the implementation of versions and provide some insight into the system overhead in supporting versions. Section 3.3 describes our approach to integrating the model of versions into the object-oriented data model of ORION as extensions to the basic messages for creating and manipulating classes and instances.

3.1 MODEL OF VERSIONS

After the initial creation of a design object, new versions of the object can be derived from it, and new versions can in turn be derived from them, forming a *version-derivation hierarchy* for the object. A version-derivation hierarchy captures the evolution of the design and indicates a partial ordering of the versions of the object. The design of a VLSI chip, for example, may involve a number of different versions during the development of the chip. Following the initial design, new versions of the chip design may be derived, say, to reduce the chip size or to reduce the power consumption. In addition, it may be necessary to generate parallel versions to experiment with different alternatives. As the evolution of a chip design progresses, tracking the design history and different design alternatives is an important data management function of an integrated CAD system.

The model of versions we reported in [CHOU86] distinguishes three types of versions on the basis of their *capabilities*: transient, working, and released versions. Intuitively, a transient version is an intermediate design that is been actively worked on and may be subjected to extensive modifications. As such, a transient version may contain a chip design that is functionally incorrect or even incomplete. When the chip design reaches a stable stage, possibly after some initial testing, the designer may promote the version of the design to a working version, ready to be shared by other members of the same team. Finally, as the design matures and is ready for release to the public, the designer can officially check the design into a shared server database as a released version. ORION presently supports only the transient and working versions; released versions are important in a distributed system, to which ORION is being extended. We now describe the properties of different types of versions.

A *transient version* has the following properties.

1. It can be updated by the designer who created it.
2. It can be deleted by the designer who created it.
3. It can be created by a checkout of a released version from the public database, derived from a transient or working version in a private database, or created from scratch.
4. If a new transient version is derived from an existing transient version, the existing transient version is automatically 'promoted' to a working version.
5. It is stored in the private database of a designer who created it.

A *working version*, called an 'effective version' in [KATZ84], has the following properties.

1. It is considered stable and cannot be updated.
2. It can be deleted by the designer who created it.
3. A transient version can be derived from a working version.
4. A transient version can be 'promoted' to a working version. Promotion may be explicit (user-specified) or implicit (system-determined).
5. It is stored in the private database of a designer who created it.

There are two reasons we impose the update restriction on working versions. One is that it is considered stable and thus transient versions can be derived from it. If a working version is to be directly updated, after one or more transient versions have been derived from it, we need a set of careful update algorithms (for insert, delete, update) which will ensure that the derived versions will not see the updates in the working version. However, in view of the fact that all a designer has to do to 'update' a working version is simply to create a new transient version, we have decided that the added complexity of such algorithms is not justified.

Another reason for the update restriction on working versions is that it eliminates the need to support update-mode checkouts. In other words, all checkouts under our model will be read-only, which in turn means that transactions issuing checkout requests will not be blocked.

A *released version* has the following properties.

1. It resides in the public database, and is managed by the database server.
2. It is not updatable.
3. It is not deletable.
4. A transient version can be derived from a released version.
5. A working version can be promoted to a released version.
6. A released version is created by a checkin of a working version from a private database.

3.2 IMPLEMENTATION AND SYSTEM OVERHEAD

An object is either *versionable* or *non-versionable*. Version support applies only to versionable objects. As mentioned earlier, any number of transient versions may be derived at any time from an existing version, a versionable object consists of a hierarchy of versions. In this paper, we use the term *generic object* to refer to the abstract versionable object, and *version instance* to refer to a specific version of the versionable object. As we will show later, a generic object is an object that describes the version-derivation hierarchy of a versionable object.

In object-oriented systems, every object is assigned an identifier which uniquely identifies the object in the system. In ORION, both the generic object and a version instance of the generic object have object identifiers. In addition, a version instance is associated with a system-defined version number, unique within a version-derivation hierarchy. An object, either a version instance or a non-versionable object, may reference one or more other objects. If the object references a versionable object, the reference may be the object identifier of a generic object or 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 object, however, we say that the object is *dynamically bound* to a *default version instance* of the generic object.

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. In other proposals, the default selected is often the 'most recent' version. However, a version-derivation hierarchy, where any number of new versions may be derived from any node on the hierarchy any time, potentially has any number of 'most recent' versions. Therefore, we allow the user to specify a particular version on the version-derivation hierarchy as the default version. In the absence of a user-specified default, the system will select the version with the 'most recent' timestamp as the default.

A versionable object is an instance of a versionable class. When the application creates a versionable object, ORION creates a generic object, along with the first version instance, for the versionable object. A generic object is essentially a data structure for describing the version-derivation hierarchy of a versionable object. A version instance is of course an object, and thus has a unique object identifier. A generic object is a data structure that the system automatically creates and maintains. However, it is also treated as an object, and is assigned an object identifier. A generic object and version instances it describes belong to the same class; however, a generic object has a special flag that distinguishes it from a version instance.

A generic object consists of the following system-defined attributes:

1. version count
2. next-version number
3. default version number
4. user default switch
5. version-derivation hierarchy

The version count is the number of existing versions of the versionable object. The next-version number is the version number to be assigned to the next version instance of the versionable object that will be created. It is incremented after being assigned to the new version instance. The default version number is the version instance to be selected to resolve a dynamically bound reference to the generic object. The user default switch indicates whether the default version was specified by the user.

The version-derivation hierarchy attribute is a tree of version descriptors, one for each version instance on a version-derivation hierarchy. A version descriptor includes

1. the version number of the version instance
2. object identifier of the version instance
3. children

The children attribute is a list of references to the version descriptors of all version instances directly derived from the version instance.

Further, each version instance of a versionable object contains three system-defined attributes, in addition to all user-defined attributes. We allow the application to read, but not to update, any of these attributes. Further, the application may include any of these attributes in the Boolean QueryExpression of a query (discussed in Section 3.1).

1. version number
2. version type
3. object identifier of its generic object

The version number is needed simply to distinguish a version instance of a versionable object from other version instances of the same versionable object. The version type indicates whether the version instance is a transient version or working version. This information is maintained, so that the system may easily reject an attempt to update a working version. The generic object identifier is required, so that, given a version instance, any other version instances of the versionable object may be found efficiently.

From the above discussion, to support versions of objects, we need to keep three system-defined attributes in each version instance. Further, when an instance of a versionable class is created, the system needs to generate a generic object. The generic object must be examined and updated when a new version instance is derived, so that the system may keep track of the most recent version instance and update the version-derivation hierarchy for the generic object. Of course, all this overhead applies only to versionable classes.

Our experiences with versions of objects have led us to conclude that versioning incurs fairly low processing overhead. However, it is obvious from our examination of the storage overhead that only fairly large objects should be versioned, so that the storage overhead may be amortized.

Evaluation of queries against a versionable class presents a somewhat interesting problem. The version number attribute may be used in the QueryExpression to find all or any specific version instances of a versionable class, or to find only the user-default or the most recent version instances of a versionable class that satisfy the QueryExpression. The first type of queries presents no difficulties; either all version instances of the specified versionable class are sequentially scanned or a subset of them may be searched using a secondary index on some attribute of the class. It is the second type of queries which presents implementation and performance difficulties, because information about the user-default or the most recent version instance of a versionable object is stored in the generic object, rather than in individual version instances. Presently, ORION does not support this type of queries. The following is our proposal for supporting it.

If the entire class must be scanned (either because there is no secondary index that may be used or because the query includes no predicates that can reduce the logical search space), we may fetch each generic object, find the identifier of the user default or the most recent version instance, whichever is necessary, and then fetch the version instance. If, however, there is an appropriate secondary index to quickly reduce the search space to a small set (smaller than the set of all generic objects for the versionable class) of qualifying version instances, to isolate only the default or most recent version instances from the set, we need to do the following for each version instance in the set:

1. fetch the version instance
2. find the version number of the version instance
3. fetch the generic object of the version instance (using the generic object identifier stored in the version instance)
4. find the default or most recent version, whichever is necessary, of the versionable object
5. compare the version number found on step 2 with that found on step 4, and if they match, output the version instance fetched on step 1.

A straightforward approach to eliminate this inefficiency is of course to keep information about the default or most recent version instance of a versionable object in the version instances. However, it will require updating this information in all version instances of a versionable object, each time a new version instance is created or when the user changes the default version instance. Of course, if new version instances are not created frequently and the number of version instances for a versionable object is very small, this approach would be acceptable. Otherwise, the only reasonable approach we have been able to find is to reduce the disk I/O penalty by physically clustering version instances with their generic object.

3.3 MESSAGES FOR VERSIONS

Our approach to integrate our model of versions and change notification into the object-oriented data model of ORION was to augment the basic set of messages with limited extensions to some of the basic messages and with a set of additional messages.

First, the **define-class** message was extended with an additional keyword argument, versionable, as follows.

(**define-class** Classname **:versionable** TrueOrNil)

The keyword **:versionable** can have a value true or nil, indicating whether versions can be created for instances of the class. Thus, the definition of the example class cell can be extended to make cell instances versionable.

(**define-class** cell **:attributes** ((cell-name **:domain** string)
....
:versionable True))

When the user issues a **create** message to a versionable class, ORION creates a generic object, as well as the first version instance of the versionable object. The new version instance is a transient version, and becomes the root of the version-derivation hierarchy for the versionable object. The optional keyword arguments of a **define-class** message supply attribute names and values for the version instance.

To derive a new version from an existing version, a **derive-version** message is sent to a VersionedObject, as follows.

(**derive-version** VersionedObject)

If VersionedObject is a transient version, ORION first promotes it to a working version. The message also causes a copy to be made of the VersionedObject. The copy becomes a new transient version, and is assigned a new version number and an object identifier. If VersionedObject is a generic object, the message is re-directed to the default version.

The application may explicitly promote a transient version to a working version by sending the **promote** message to a VersionedObject. If the VersionedObject is already a working version, no action is taken. Conversely, a working version can be demoted

to a transient version by a **demote** message. If VersionedObject is already a transient version, no action is taken. If VersionedObject is a working version, and there exist other versions that have been derived from this working version, no action is taken. In both the promote and demote messages, if the VersionedObject specified is a generic object, the message is re-directed to the default version.

The default version number of a generic object can be changed with a **change-default-version** message.

(**change-default-version** VersionedObject [NewDefault])

If the argument NewDefault is specified, the default version number becomes fixed, and will not change even if new versions are created. It can be changed only with another **change-default-version** message. If the NewDefault is not specified, the most recently created version number is used as the default.

The **delete-object** message is used to delete a version instance or a generic object. If the message is sent to a generic object, the entire version-derivation hierarchy is deleted. In other words, all version instances of the versionable object, as well as the generic object, are deleted. If a **delete-object** message is sent to a version instance, the version instance is deleted. If the version instance is a transient version, or a working version from which no other versions have been derived, the history (or version descriptor) of the version instance is deleted as well. (The history of a version instance is maintained within the generic object of the version instance, as discussed in Section 3.2.) If the version instance is the only version instance of the versionable object, the generic object is also deleted. If the **delete-object** message is sent to a working version that has other derived versions, however, the history of the version instance is not deleted.

To fetch, update, or delete version instances of a versionable class based on a QueryExpression, the **select**, **change**, and **delete** messages shown in Section 3.1 can be used, without any changes in their syntax or semantics. These messages cause all version instances of the specified class, irrespective of their version numbers, to be examined. As we discussed in Section 3.2, the application can include these system attributes in the QueryExpression.

We also provide a number of messages which relate a generic object and version instances. The user can obtain the parent version of a VersionedObject by sending a **parent-version** message to the VersionedObject. Similarly, the child versions of a given version can be obtained with a **child-versions** message to a VersionedObject. If the user has access to a VersionedObject, the user may obtain the generic object by sending a **generic-object** message to the VersionedObject. The default version number can be obtained by sending a **default-version** message to a generic object.

4. CHANGE NOTIFICATION

In this section, we discuss first our model of change notification, and then describe our implementation and application interface as extensions to the basic messages corresponding to the object-oriented concepts.

4.1 MODEL OF CHANGE NOTIFICATION

A complex object recursively references other 'lower level' objects. An object may be referenced (shared) by any number of objects, and may in turn reference any number of other objects. When an object is updated or deleted, or a new version of the object is created, some or all of the objects that have referenced it may become invalid, and thus need to be notified of the change.

In a distributed CAD environment, two types of notification techniques must be supported: message-based and flag-based. In the *message-based* approach, the system sends messages to notify (human) users of potentially affected objects. The message-based approach is further distinguished as *immediate* or *deferred*, depending on whether the affected users are notified immediately after the changes to an object are committed or at some later time that the users may have specified.

In the *flag-based* approach, the system simply updates data structures that it maintains, so that affected users will become aware of changes in an object only when they explicitly access the object. The flag-based approach is necessarily a deferred notification strategy.

We see that an object has a number of change-notification options at its disposal: message vs. flag-based, immediate vs. deferred (in the case of message-based notification), and types of changes to post notification (update, delete, creation of a new version of an object). When the application defines an object, it must specify these options with respect to the objects it references. However, it is impractical to require the user to specify a possibly different set of options for each of the references in an object, since an object may reference a large number of other objects. We believe that a more sensible approach is to have a single set of options specified for an object, and apply it across all objects the object references.

4.2 IMPLEMENTATION

In the flag-based approach, each object that participates in change notification has two distinct timestamps. One timestamp, called the *change-notification timestamp*, indicates the time the object was created or the last time it was updated. The other, called *change-approval timestamp*, indicates the last time at which the owner of the object approved all changes to the objects it references.

Let V.CA and V.CN denote the change-approval and change-notification timestamps of an object V. Let R be the set of objects that are referenced by object V. If no object in R has a change-notification timestamp that exceeds the change-approval timestamp of V (i.e, for all X in R, X.CN <= V.CA), then V is *reference-consistent*. V is *reference-inconsistent* if there are one or more objects in R that have been updated, but the effects of these updates on V have not been determined.

To make V reference-consistent, the effects of the updated objects in R must be determined, and, if necessary, V must be updated. If the updates to objects in R have no effect on V, only V.CA needs to be changed to the current time. Otherwise, V.CN (and possibly V.CA if the changes are approved) will need to be set to the current time.

ORION provides three system-defined attributes for a class for which the notify option is on.

1. change-notification timestamp
2. change-approval timestamp
3. a set of change-notification events

As mentioned earlier, he change-notification timestamp associated with an object indicates the last time at which a change has been made to the object. The change-approval timestamp indicates the last time the user approved changes to all objects referenced through all notification-sensitive attributes. Changes are notified either on an update or on a delete operation, or both. 'Delete' is the default, and 'update' subsumes 'delete'.

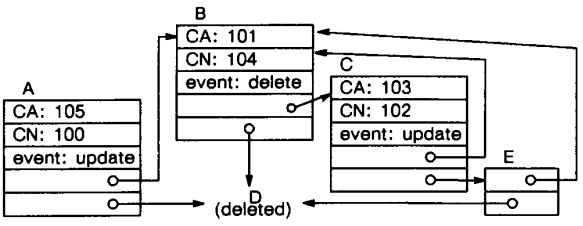
If a class D is the domain of a notification sensitive attribute of a class C, then each instance of D, as well as D's subclasses, contains a change-notification timestamp. Whenever a new instance d of D (or a subclass of D) is created, or an existing in-

stance is changed, its change-notification timestamp captures the time of that event. At a later time, when an instance c of C makes a reference to the instance d through a notification-sensitive attribute, the user or application can compare the change-approval timestamp of c with the change-notification timestamp of d, and decide whether the change to d needs approval. When instance d is deleted, instance c will have a dangling reference. The dangling reference is itself an indication of reference-inconsistency, if the reference is made through a notification-sensitive attribute. The system maintains the information necessary to determine if an instance is reference-consistent. However, it is the user's or application's responsibility to determine reference-consistency, and then take measures to make the instance reference-consistent, if necessary.

To assist the user or application in detecting inconsistent references in a complex object, such as a complete chip design, ORION provides a utility program that checks the reference-consistency of an object. The check is performed recursively to span all notification-sensitive objects that are referenced either directly or indirectly by the given object. After the check is completed, the utility returns a list of warnings. Each warning is a triplet of the form <from-object, to-object, event> which means that the reference from from-object to to-object is potentially inconsistent as a result of a certain event.

Starting from the root of a complex object, the utility program traverses through notification-sensitive attributes to all objects that are reachable from the root object. While traversing, the consistency of a reference from one object to another (through a notification-sensitive attribute) is checked according to the sensitivity of the referencing object. For a delete-sensitive object (i.e. an object whose notification event is delete), the existence of every object that it references is examined. For an update-sensitive object, its change-approval timestamp is compared against the change-notification timestamp of every object referenced by it, as described in Section 4.1.

In the example shown in Figure 1, there are five objects, one of which has been deleted. Each object has two user-defined attributes that contain references to other objects. Let's assume that all the attributes in objects A, B, and C are notification-sensitive. Consequently, objects A, B, and C are notification-sensitive, and hence have three additional system-defined attributes. If we apply the consistency check to object A, three warnings will be generated. The two references from A and B to the deleted object D are obviously invalid. The reference from C to B is inconsistent because the change-approval timestamp of C is smaller than the change-notification timestamp of B. Since B is sensitive to deletion only, the reference from B to C is considered consistent although the timestamps indicate otherwise. References stemming from object E are not checked as E is not notification-sensitive.



Warning: (<A,D,delete>, <B,D,delete>, <C,B,update>)

Figure 1. Example of Reference-Consistency Check

In order for the utility program to function properly, there are few more details to consider. First, since there may exist circular references among objects, the utility program uses a hash table to avoid repeated checks of the same object, and more importantly, to avoid a potential infinite loop. Second, as ORION supports set as the domain of an attribute, it is possible that an attribute contains a set of object references as its value. Under this circumstance, every member of the set must be subjected to consistency check. Details of the algorithm are shown in Figure 2.

```
check-consistency(obj)
  if obj is not notification-sensitive or has been traverse, return;
  else register obj in the hash table as been traversed
       for every notification-sensitive attribute a of obj
          if a contains a reference call check-reference (obj, a);
          else if a contains a set of references
                  for every object reference b in the set
                   call check-reference(obj, b);
end-check-consistency;
check-reference(from-object, to-object)
  if to-object is deleted
    add <from-object, to-object, delete> to the list of warnings;
  else if from-object is update-sensitive and
                        from-object.CA < to-object.CN
        add <from-object, to-object, update> to list of warnings;
      call check-consistency(to-object);
end-check-reference;
```

Figure 2. Algorithm for Reference-Consistency Check

To support message-based notification, we will need to maintain, for each object V, an *inverted reference list* of objects which reference V and which require notification of changes to V. When a new reference to V is created, the name of the object that references V is appended to the inverted reference list of V. For each reference, the event type (a combination of update/deletion/creation of objects) and the timing of notification (immediate or deferred) are also recorded. When an object with a non-empty inverted reference list is changed, the list is scanned for those objects with a matching event type, and messages are sent to the owners of those objects.

4.3 MESSAGES FOR CHANGE NOTIFICATION

To support change notification, we further extend the **define-class** message with a **notify** argument, as follows.

(**define-class** Classname **:notify** ListofAttributeNames)

Whether versionable or not, instances of a class C may be notified when changes are made to the values of selected attributes. The **:notify** keyword causes the system to maintain timestamps for instances of the class. If the ListofAttributeNames is not specified, instances of the class will not be notified of changes. We will call the attributes specified in the ListofAttributeNames *notification-sensitive attributes*. The classes that are the domains of the notification-sensitive attributes must also be defined with the **:notify** keyword.

Once the user creates an object, the user may specify the change-notification events with respect to the objects it references. As discussed earlier, we allow the user to specify a single option for an object, and apply it across all objects the object references.

The **set-notification-events** message can be used to assign a set of notification events to an instance of a notifiable class. The format of the message is

(**set-notification-events** Object ListOfEvents)

where ListOfEvents is a list of event names, where the permitted event names are the symbols **update** and **delete**.

We provide messages to obtain the values of the timestamps and the notification events; **change-notification-timestamp**, **change-approval-timestamp**, and **notification-events**. Further, to approve changes, and thus modify the change-approval timestamp of an instance of a notifiable class, an **approve-change** message may be sent to an object.

5. SUMMARY

In our earlier paper [CHOU86], we proposed a model of versions and change notification for integrated CAD/CAM systems. The model explicitly takes into account the characteristics of CAD/CAM system architecture, the way in which designers interact with the system and among themselves, and the nature of the design database. Since then, we have prototyped an object-oriented database system called ORION to support applications from the CAD/CAM, AI, and OIS domains. The first ORION prototype is a workstation-based system, and as such, we have only implemented the part of the model that is relevant in a single-user, multiple-transaction environment. The next ORION prototype currently being built is a fully distributed system and will include the complete model of versions and change notification described in this paper.

One problem we had to address was the integration of our model of versions and change notification into the object-oriented data model of ORION. Our approach to this problem was to first define a basic set of messages for an object-oriented database, and then to augment this set of messages for version support and change notification. The basic set of messages include those for defining classes, creating, fetching, updating, and deleting instances. For versions and change notification, we made limited extensions to some of the basic messages and also introduced an additional set of messages.

In this paper, by describing our extended set of messages, we provided an insight into integrating versions and change notification into an object-oriented database system. Further, we gave an insight into the system overhead incurred in implementing versions and change notification in a database system. This latter will be a valuable guide to developers of other models of versions and the implementors of such models even in non-object-oriented database systems.

ACKNOWLEDGEMENT

The syntax of the messages we presented here was specified mostly by Jay Banerjee.

REFERENCES

[AHLS84] Ahlsen, M., et al. "An Architecture for Object Management in OIS," in *ACM Trans. on Office Information Systems*, vol. 2, no. 3, July 1984.

[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," in *ACM Trans. on Office Information Systems*, vol. 5, no. 3, January 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*, June 1987.

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

[BOBR85] Bobrow, D.G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. *CommonLoops: Merging Common Lisp and Object-Oriented Programming*, Intelligent Systems Laboratory Series ISL-85-8, Xerox PARC,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.

[COPE84] Copeland, G. and D. Maier. "Making Smalltalk a Database System," *ACM SIGMOD*, June 1984.

[CURR84] Curry, G.A. and R.M. Ayers. "Experience with Traits in the Xerox Star Workstation," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 5, September 1984.

[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.

[IEEE85] *Database Engineering*, IEEE Computer Society, vol. 8, no. 4, December 1985 special issue on Object-Oriented Systems (edited by F. Lochovsky).

[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.

[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*, 1986.

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

[NEUM82] Neumann, T., and C. Hornung. "Consistency and Transactions in CAD Databases," *Proc. Intl Conf. on Very Large Data Bases*, Sept. 82, Mexico City, Mexico.

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

[STEE84] Steele, G.L. *Common Lisp: The Language*, Digital Press, 1984.

[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.

[WOEL86] Woelk, D., W. Kim, and W. Luther. "An Object-Oriented Approach to Multimedia Databases," in *Proc. ACM SIGMOD Conf. on the Management of Data*, Washington D.C., May 1986.

[ZDON86] Zdonik, S.B. and P. Wegner. "Language and Methodology for Object-Oriented Database Environments," in *Proc. Hawaii Intl. Conf. on System Sciences*, January 1986.