Composite Objects Revisted

Won Kim, Elisa Bertino, Jorge F. Garza

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759

ABSTRACT

In object-oriented systems, an object may recursively reference any number of other objects. The references, however, do not capture any special relationships between objects. An important semantic relationship which may be superimposed on a reference is the IS-PART-OF relationship between a pair of objects. A set of objects related by the IS-PART-OF relationship is collectively called a composite object.

An earlier paper [KIM87b] presented a model of composite objects which has been implemented in the ORION object-oriented database system at MCC. Although the composite-object feature has been found quite useful, the model suffers from a number of serious shortcomings, primarily because it overloads a number of orthogonal semantics on the references. In this paper, first we present a more general model of composite objects which does not suffer from these shortcomings. Further, [KIM87b] made an important contribution by exploring the use of composite objects as a unit for versions, physical clustering, and concurrency control. The extended model of composite objects necessitates non-trivial changes to the results of [KIM87b]. This paper describes the new results on the use of composite objects as a unit of not only versions, physical clustering and concurrency control, but also authorization.

1. Introduction

Object-oriented data models [ANDR87, BANE87a, COPE84, FISH87] (and most semantic data models [HAMM81]) allow the definition of arbitrarily complex objects as nested objects of arbitrary depth. An object has a number of attributes; the value of an attribute is itself an object. An object belongs to a class; a class may be a primitive class without any attributes (e.g. integer, string), or may have any number of attributes. An object with an attribute whose value is an object which belongs to a non-primitive class is a nested object. An object may have any number of attributes, and any of the attributes may take values from other non-primitive classes.

The nesting of objects inherent in an object-oriented data model is a powerful concept; however, it does not imply some special relationships between objects which may be important to different classes of applications. One important relationship which should be superimposed on the nested object is the IS-PART-OF relationship, that is, the notion that an object is a part of another object. A set of component objects which form a single logical entity has been called a composite object [KIM87b, STEF86] or a complex object [LORI83, KIM87a]. ORION is perhaps the first database system which supports an object-oriented data model extended with the IS-PART-OF relationship. [LORI83, STON86] report prototyping efforts to extend the relational model of data with complex objects.

ORION provides direct system support for composite objects as a unit for one type of semantic integrity, physical clustering, and locking. [KIM87b] also defines the semantics of versions for composite objects by extending the model of versions for non-nested objects; ORION currently supports versions of normal objects, but not composite objects. [RABI88] presents the framework for a model of authorization, in which an attempt is made to define composite objects as a unit of authorization. Composite objects have been found quite useful for a number of applications of ORION, including some mechanical CAD applications. However, the model and implementation of composite objects, as presented in [KIM87b] and implemented in ORION, suffer from three major shortcomings.

First, the model restricts a composite object to a strict hierarchy of *exclusive* component objects; that is a component object is only part of one composite object. This is certainly the right model for a *physical part hierarchy*, in which an object cannot be part of more than one object. However, this is not acceptable for a *logical part hierarchy*; for example, an identical chapter may be a part of two different books.

Second, the model forces a top-down creation of a composite object; that is, before a component object may be created, its parent object must already exist. This prevents a bottom-up creation of objects by assembling already existing objects.

Third, the model requires that the existence of a component object depends on the existence of the parent object; that is, if an object ceases to exist, all its component objects are also deleted. This feature is sometimes desirable, since it frees the applications from having to search and delete all nested components of a deleted object. Sometimes, however, it impedes reuse of objects in a complex design environment.

The objective of this paper is to describe significant semantic extensions to the model of [KIM87b] which were necessary to eliminate these shortcomings while retaining the advantages of the previous, more limited model. To support the semantic extensions, we needed to extend various aspects of the ORION system implementation. In doing so, we further augment the use of composite objects as a unit of authorization, beyond their use as a unit of concurrency control established in [KIM87b].

The remainder of this paper is organized as follows. Section 2 presents the extended semantics of composite objects. Section 3 describes operations which can be performed on composite objects and specific components of a composite object. Sections

4, 5, and 6 discuss the impacts of the extended semantics of composite objects on schema evolution, versioning, and authorization, respectively. Section 7 describes the use of composite objects as a unit of locking. Section 8 summarizes the paper.

2. Basic Extensions

We will first define the extended semantics of composite objects. Then we will formalize essential aspects of the semantics. Further, we will make the discussion concrete by providing examples of composite objects, and considering basic implementation issues.

2.1 Semantics of References

We say that an object O' has a reference to (or references) another object O, if O' contains the object identifier (UID) of O. We will distinguish two types of reference from one object to another: weak and composite. A composite reference in turn may be of two types: exclusive and shared. A weak reference is the standard reference in object-oriented systems and carries no special semantics. A composite reference is a weak reference augmented with the IS-PART-OF relationship; a composite reference from O' to O means that O is a part of O'. The semantics of a composite reference is further refined on the basis of whether an object is a part of only one object or more than one object. An exclusive composite reference from O' to O means that O is a part of only O'; while a shared composite reference from O' to O means that O is a part of O' and possibly other objects.

We further refine the semantics of a composite reference, either exclusive or shared, on the basis of whether the existence of an object depends on the existence of its parent object; that is, a composite reference may be dependent or independent. A dependent composite reference from O' to O means that the existence of O depends on the existence of O'; while an independent composite reference does not carry this additional semantics. The deletion of an object will trigger recursive deletion of all objects referenced by the object through dependent composite references (both exclusive and shared).

The above discussion reveals the following five types of reference between a pair of objects. The composite reference defined in [KIM87b] is only the dependent exclusive composite reference.

1. weak reference
2. dependent exclusive composite reference
3. independent exclusive composite reference
4. dependent shared composite reference
5. independent shared composite reference

The objects related through composite references form a part hierarchy. The classes to which the objects in the part hierarchy belong are also organized in a hierarchy called a *composite class hierarchy*; each class in the hierarchy is called a *component class*. The root of a composite object is a special object. Under the model of [KIM87b], the root of a composite object never changes (because of the top-down creation of a composite object). Under our extended model, the root of a composite object may change; that is, an object which is the current root of a composite object may become the target of a composite reference from another object. A reference from an object O' to another object O means that the value of an attribute of O' is the identifier of O. The values of an attribute of a class C' are instances of a class C; the class C is the *domain* of the attribute of C'. If an attribute has a composite reference, the attribute is called a *composite attribute*. The meanings of the terms *exclusive composite attribute* and *shared composite attribute* should be obvious.

We distinguish two types of part hierarchy: physical and logical. In a physical part hierarchy, all composite references are exclusive; while a logical part hierarchy may also contain shared composite references. The physical part hierarchy is the composite hierarchy currently implemented in ORION.

2.2 Formalization of the Semantics

We observe that different types of reference partition the set of objects which reference a given object into four different sets of objects. We may formalize a few important aspects of the extended semantics of composite objects in terms of constraints on these sets of objects.

Definition 1:
With respect to an object O in a composite object,

IX(O) = {O.i | O.i has an independent exclusive composite reference to O}
DX(O) = {O.i | O.i has a dependent exclusive composite reference to O}
IS(O) = {O.i | O.i has an independent shared composite reference to O}
DS(O) = {O.i | O.i has a dependent shared composite reference to O}.

We formalize the deletion of an object O', denoted by del(O'), with respect to the four types of composite reference from O' to another object O.

1) Independent exclusive composite reference from O' to O
del(O') $\not\Rightarrow$ del(O).
2) Dependent exclusive composite reference from O' to O
del(O') => del(O).
3) Independent shared composite reference from O' to O
del(O') $\not\Rightarrow$ del(O).
4) Dependent shared composite reference from O' to O
del(O') => del(O) only if DS(O) = {O'}.
That is, the deletion of O' causes the deletion of O only if all other dependent shared composite references to O have been removed; otherwise DS(O) = DS(O) - O'.

In the remainder of this paper, we will use the terms *parent set* and *ancestor set* of an object O as the set of objects which have direct or indirect composite references to O, respectively. The term *component set* of an object O will denote all objects directly or indirectly referenced from O via composite references. Further, given two objects O and O', we say that O is a *level n component* of O' if the shortest path between O and O' has n composite references.

The five types of reference we have defined may be combined to generate arbitrarily complex "object topologies". The following set of topology rules represents constraints on the object topologies based on the semantics of different types of reference. In the following, card(S) denotes the cardinality of a set S.

Topology Rule 1:
card (IX(O)) $\leq$ 1
card (DX(O)) $\leq$ 1

Topology Rule 2:
card (IX(O)) = 1 => card (DX(O)) = 0
card (DX(O)) = 1 => card (IX(O)) = 0

If an object O has an independent exclusive composite reference to it, then it cannot have a dependent exclusive composite reference from another object; and vice versa.

Topology Rule 3:
(card (IX(O)) = 1 or card(DX(O)) = 1)
=> (card(IS(O)) = 0 and card(DS(O)) = 0)
(card (IS(O)) > 0 or card(DS(O)) > 0
=> (card(IX(O)) = 0 and card(DX(O)) = 0)

If object O has an exclusive (either independent or

dependent) composite reference from an object, then it cannot have shared (either independent or dependent) composite references from other objects; and vice versa.

Topology Rule 4:
An object O can have any number of weak references to it, even when it has composite references to it.

Suppose that we wish to make an object O a part of an object O' through an attribute A of O'. The following are the conditions which must be satisfied.

Make-Component Rule:

1. If A is an exclusive composite attribute, O must not already have any composite reference to it (exclusive or shared).
2. If A is a shared composite attribute, O must not already have an exclusive composite reference.

The following summarizes the semantics of the deletion of an object in a composite object.

Deletion Rule:
Suppose that O' is the root of a composite object and that O is a component of O'.
del(O') = > del(O) if any of following three conditions holds:

1. O' has a dependent exclusive reference to O.
2. O' has a dependent shared reference to O and DS(O) = {O'}
3. An object O" exists such that
   del(O') = > del(O") and either (3.a) or (3.b) olds:
   3.a O" has a dependent exclusive composite reference to O
   3.b O" has a dependent shared composite reference to O and DS(O) = {O"}.

2.3 Syntax Extensions

class definition

To support the extended semantics of a composite reference, we extend the ORION syntax for attribute specification in the class definition [BANE87a] as follows:

```
(AttributeName   [:init InitialValue]
                 [:domain DomainSpec]
                 [:inherit-from SuperClass]
                 [:document Documentation]
                 [:share SharedValue]
                 [:composite TrueOrNil]
                 [:exclusive TrueOrNil]
                 [:dependent TrueOrNil]
```

The keyword **composite** when set to True specifies that the reference is a composite reference. If the keyword **exclusive** is also True, the composite reference is an exclusive composite reference. If the keywords **composite** and **dependent** are both True, the reference is a dependent composite reference. The default value for both the **exclusive** and **dependent** keywords is True (to be compatible with the semantics of composite objects currently supported in ORION).

Below we provide two examples. The first is a physical part hierarchy; and the second a logical part hierarchy.

Example 1:

Let us consider a Vehicle composite hierarchy. We require that a vehicle part may be used for only one vehicle at any point in time; however, vehicle parts may be re-used for other vehicles. The definition for the class Vehicle is as follows (for simplicity we omit the definitions of all domain classes):

```
(make-class 'Vehicle :superclasses nil
    :attributes '(
        (Manufacturer   :domain Company)
        (Body           :domain AutoBody
                        :composite true
                        :exclusive true
                        :dependent nil)
        (Drivetrain     :domain AutoDrivetrain
                        :composite true
                        :exclusive true
                        :dependent nil)
        (Tires          :domain (set-of AutoTires)
                        :composite true
                        :exclusive true
                        :dependent nil)
        (Color          :domain String)))
```

Since all composite attributes in the class Vehicle are exclusive references, a set of vehicle components (Body, Drivetrain, and Tires) may be used for only one vehicle. However, since the exclusive references are independent, the components can be re-used for other vehicles, if the vehicle which they constitute is dismantled later. The vehicle components may exist even if they are not part of any vehicle.

Example 2:

Let us now consider electronic documents. Suppose that a document consists of a title, authors and a number of sections. A sections in turn is composed of paragraphs. A document may share entire sections or section paragraphs with other documents. Annotations may be added to documents; however, they are not shared among different documents. Further, documents may contain images that are extracted from files. The following are the definitions of two of the classes involved, Document and Section:

```
(make-class 'Document
    :superclasses nil
    :attribute '(
        (Title          :domain string)
        (Author         :domain (set-of string))
        (Content        :domain(set-of Section)
                        :composite true
                        :exclusive nil
                        :dependent true)
        (Figures        :domain (set-of Image)
                        :composite true
                        :exclusive nil
                        :dependent nil)
        (Annotations    :domain (set-of Paragraph))
                        :composite true
                        :exclusive true
                        :dependent true))

(make-class 'Section
    :superclasses nil
    :attribute '(
        (Content        :domain (set-of Paragraph)
                        :composite true
                        :exclusive nil
                        :dependent true))
```

The attribute Content, defined as a set, is a shared composite reference. Other documents may share any element in this set (i.e., any section). A section exists, if it belongs to at least one document. Similarly, the class Section has a Paragraph shared composite reference. A paragraph may be shared among different sections (of possibly different documents). For a paragraph to exist, there must be at least one section containing it and thus a document containing it. In the case of Annotations we assume that a given annotation is used in only one document, thus the reference is exclusive. The attribute Figures is defined as an independent composite reference, since the existence of images does not depend on the documents containing them.

instance creation

The following message is used to make an instance a part of one or more composite objects at the time of creation of that instance:

```
(make Class
    :parent  ((ParentObject.1   ParentAttributeName.1)
              (ParentObject.2   ParentAttributeName.2)
                        .....
              (ParentObject.M   ParentAttributeName.M))
    :Attribute.1   value.1
    :Attribute.2   value.2
              .....
    :Attribute.N   value.N).
```

The keyword **parent** is associated with one or more pair (ParentObject.i ParentAttributeName.i), where ParentObject.i with the attribute ParentAttributeName.i is to reference the instance being created. If ParentAttributeName.i is a composite attribute, the new instance becomes part of ParentObject.i. Further, if ParentAttributeName.i is a dependent composite attribute, then the existence of the new object will depend on ParentObject.i.

When more than one pair (ParentObject.i ParentAttributeName.i) is specified such that ParentAttributeName.i is a composite attribute, then the instance being created is simultaneously made a part of all the specified objects. However, because of topology rule 3, these attributes must be shared composite attributes. The topology rules are enforced using the schema definitions for classes of ParentObject.1, ParentObject.2, ..., ParentObject.M.

Next, if an already existing object is made a part of a composite object through an exclusive reference, the system must check if there are no other composite references to that object. Similarly, if the object is made a part through a shared reference, the system has to ensure that there is no exclusive reference to that object.

As discussed in [BANE87a], the **parent** keyword in the **make** statement is used also for clustering purposes. If several objects are specified, then the newly created object is clustered with the first specified parent, that is, with ParentObject.1. (However, clustering is only performed if the classes of the two objects are stored in the same physical segment.)

2.4 Implementation

Given a component of a composite object, the user often finds it necessary to determine its parents or ancestors. Further, the system needs to determine efficiently the parents or the roots of a given component of a composite object to efficiently support locking, versions, and authorization of composite objects. We need to maintain in each component of a composite object a list of *reverse composite references*, that is, object identifiers of the parent objects. Although it is often useful for the system to be able to determine directly the roots of a composite object for a given component object, we have decided not to keep such information in each component; the reason is that the roots of a composite object may change when composite objects are created in a bottom-up fashion. Further, we have decided to keep the reverse pointers in each component object, rather than in a separate data structure. This approach allows us to avoid a level of indirection in accessing the parents of a given component, and simplifies deletion and migration of objects; however, it causes the object size to increase.

The number of reverse composite references in a component object is equal to the number of parent objects. A reverse composite reference actually consists of a couple of flags in addition to the object identifier of a parent. One flag (D) indicates whether the object is a dependent component of the parent; while the other flag (X) indicates whether the object is an exclusive component of the parent.

The following algorithm for making an existing object O a part of another object O' through an attribute A illustrates the use of reverse composite references.

1. Access Object O.
2. If (A is a shared composite attribute and the X flag in a reverse composite reference in O is set)
   or
   (A is an exclusive composite attribute and O has any reverse composite reference)
   then return (error).
3. Insert in O a reverse composite reference to O' with
   the D flag set if A is a dependent attribute
   the X flag set if A is an exclusive attribute.

3. Operations on Composite Objects

We now discuss operations on composite objects. These operations are used to determine the components, children, parents, and ancestors of an object. Most of these operations are rather obvious. However, it is important to realize that the purpose of modeling a composite object is above all to define operations which directly make use of the semantics of composite objects.

3.1. Determining ancestors and components of an object

To determine the components of an object, the message **components-of** is used.

```
(components-of Object    [ListofClasses]   [Level]
                         [Exclusive]       [Shared])
```

If the argument ListofClasses is specified, the message returns only components that belong to the specified classes. Otherwise all components of the object are returned. This message can have another optional integer argument, Level. This is used to return components of a given object up to the specified Level. If Exclusive is True, only the exclusive components of Object are retrieved; and if Shared is True, only shared components are retrieved. If both Exclusive and Shared are Nil, all components are retrieved.

Similarly, to determine the parents or ancestors of a given object, the messages **parents-of** and **ancestors-of** are used.

```
(parents-of Object     [ListofClasses]   [Exclusive]
                       [Shared])
(ancestors-of Object   [ListofClasses]   [Exclusive]
                       [Shared])
```

If the argument ListofClasses is specified, the message returns all parents or ancestors that belong to the specified classes; otherwise, all parents or ancestors are retrieved.

3.2. Predicates on component classes and instances

In [BANE87a] messages that act as predicates with respect to class objects and instance objects have been presented. Among others, the message **compositep** is used to determine if an attribute of a class has a composite reference property.

(**compositep** Class [AttributeName])

If the argument AttributeName is not supplied, the message returns True if the class has at least one attribute with such property.

Similar messages are defined to determine whether a class has composite attributes with exclusive or shared references, and further the references are dependent.

```
(exclusive-compositep     Class   [AttributeName])
(dependent-compositep     Class   [AttributeName]).
```

To determine if an object Object1 is a part of another object Object2, the following message is used:

(component-of Object1 Object2).

This message returns True if Object1 is a direct or indirect component of Object2.

To determine if Object1 is a direct component of Object2 the following message is used:

(child-of Object1 Object2).

This message returns True if Object1 is a direct component of Object2.

We note that to determine if Object1 is a part of Object2, the message **components-of** can be used to retrieve the components of Object2, followed by a scan of the returned set of objects to check if Object1 is in this set. Therefore the message **component-of** can be seen as a shorthand. Similar messages may be defined for determining whether Object1 is an exclusive or shared component of Object2.

(exclusive-component-of Object1 Object2).

This message returns True if Object1 is an exclusive component of Object2. It returns Nil if either Object1 is not a component of Object2, or it is a shared component.

(shared-component-of Object1 Object2).

This message returns True if Object1 is a shared component of Object2. It returns Nil if either Object1 is not a component of Object2, or it is an exclusive component. We note that sending the **component-of** and **exclusive-component-of** messages in sequence has the same effect as the message **shared-component-of**. In fact, if **(component-of** Object1 Object2) returns True and a subsequent execution of **(exclusive-component-of** Object1 Object2), in the same transaction, returns Nil, Object2 is a shared component of Object1.

We note that there is no need to define a message for determining if an Object1 belongs to the ancestor set of an Object2, since in this case the message **component-of** can be used, by passing to it Object2 as the first argument and Object1 as second. Similarly, we can use the message **child-of** to determine if Object1 belongs to the parent set of Object2.

4. Schema Evolution

Schema evolution is the specification of a set of dynamic changes to the database schema and the semantics of each of the changes. [BANE87b] presents a framework for schema evolution for ORION. The set of schema changes considered consists largely of changes to the class definitions, such as adding and deleting attributes and methods; and changes to the structure of the class lattice, such as creation and deletion of a class, and modifications to the IS-A relationship between classes. The framework is extended somewhat in [KIM87b] to consider composite objects in greater detail. The extended model of composite objects we presented earlier in this paper requires changes to the semantics of several types of schema changes defined in [BANE87b, KIM87b], and also gives rise to new types of meaningful schema changes.

4.1 Deletion of a Composite Attribute

The following is a subset of the schema evolution taxonomy presented in [BANE87b]. The schema change operations in this subset are the ones whose semantics must be altered to account for the extended semantics of composite references we presented earlier in this paper. These operations involve deletion of a composite attribute. The model of composite objects in [KIM87b] causes all objects referenced through a composite attribute to be deleted if the attribute is removed; however, the extended model requires only those objects which are referenced through dependent composite attributes to be dropped when the attributes are dropped.

1. Drop an attribute A from a class C.
   This operation causes all instances of the class C to lose their values for attribute A. If A is a composite attribute, objects that are referenced through A are deleted in accordance with the Deletion Rule (Section 2.2). The attribute must also be dropped from all subclasses that inherit it.
2. Change the inheritance (parent) of an attribute (inherit another attribute with the same name).
   Depending on the origin of the old and new attribute [BANE87b], this operation may cause the old attribute to be dropped. If the attribute dropped is a composite attribute, this operation is identical to (1) above.
3. Remove a class S as superclass of a class C.
   If this operation causes class C to lose a composite attribute A, objects (of other classes) that are recursively referenced by instances of C and its subclasses through A are deleted according to (1).
4. Drop an existing class C.
   If the class C has one or more composite attributes, objects referenced through the attributes are dropped in accordance with the Deletion Rule (Section 2.2). All subclasses of C become immediate subclasses of the superclasses of C.

4.2 Changes to the Attribute Type

Let us now explore meaningful changes from one type of attribute to a different one in the context of the extended model of composite objects. The changes may be one of two types with respect to implementation: state-independent and state-dependent changes. Roughly, a *state-independent change* is a change which removes a constraint from a composite reference, while an *state-dependent change* adds a constraint to a reference. A state-dependent change requires verification of the X (exclusivity) flag in the reverse composite references to the objects involved in the change.

The following are the state-independent changes to the attribute type which are meaningful under our extended semantics of composite references.

I1. Change a composite attribute to a non-composite attribute. This operation is also defined in [BANE87b].

I2. Change an exclusive composite attribute to a shared composite attribute.

I3. Change a dependent composite attribute (either exclusive or shared) to an independent composite attribute.

I4. Change an independent composite attribute (either exclusive or shared) to a dependent composite attribute.

State-dependent changes to the composite attributes are as follows. Let us suppose that the class C is the domain of an attribute A of a class C', and that A is to be changed.

D1. Change a non-composite attribute to an exclusive composite attribute.
   There must be no composite (either shared or exclusive) references to instances of the class C which are referenced by instances of the class C'.

D2. Change a non-composite attribute to a shared composite attribute.
   Topology Rule 3 (Section 2.2) must be verified to ensure that there are no exclusive composite references to instances of the class C which are referenced by instances of the class C'.

D3. Change a shared composite attribute to an exclusive composite attribute.
   Topology Rule 3 must be verified to guarantee that there

is at most one shared reference to instances of the class C which are referenced by instances of the class C'.

4.3 Implementation

Implementation of changes to the attribute type involves accessing referenced objects and updating the D and X flags in the reverse composite references in them. The validity of a state-dependent change depends on the consistency of these flags; that is, if the flags are not in a state consistent with the semantic requirements of the change, the change is rejected. For this reason, state-dependent changes require 'immediate' verification of the flags. However, state-independent changes simply require updates to these flags; as such, the changes may be made 'immediately' or 'deferred' until the objects actually need to be accessed.

Let us first discuss the two options for implementing state-independent changes. The 'immediate' implementation of the changes is as follows. Suppose that a class C' has a composite attribute A whose domain is a class C.

I1. Change a composite attribute to a non-composite attribute.
This is implemented by accessing all instances of the class C and dropping reverse composite references to instances of the class C'.

I2. Change an exclusive composite attribute to a shared composite attribute.
This is implemented by accessing all instances of the class C and turning off the X flag in the reverse composite references to instances of the class C'.

I3. Change a dependent composite attribute (either exclusive or shared) to an independent composite attribute.
This is implemented by accessing all instances of the class C and turning off the D flag in the reverse composite references to instances of the class C'.

I4. Change an independent composite attribute (either exclusive or shared) to a dependent composite attribute.
This is implemented by accessing all instances of the class C and turning on the D flag in the reverse composite references to instances of the class C'.

The 'deferred' implementation of state-independent changes involves keeping an *operation log* of changes to the attribute types in a class. A class has n operation logs, one for each attribute of which the class is the domain. An operation log for a class C maintains, for each change, the change type and change count (CC), as well as the identifier of the class of whose attribute C is the domain. Initially, CC is zero and is incremented by one each time the type of attribute in a class C is changed.

The CC is also an system-defined attribute of the class C; that is, each instance of C carries a value for CC, although the value may not be up to date. When an instance of C is accessed, the CC of the instance is checked against the CC in the operation log associated with the class: if CC(instance) < CC(class), then the flags in the reverse composite reference in the instance must be modified. The changes that must be made are the ones with a CC which is greater than the CC of the instance. Once the changes have been applied, the CC in the instance is set to the highest CC in the operation log. When a new instance of the class C is created, the CC of the instance is set to the current value of the CC of the class, since the changes issued before the creation of the instance need not be applied to this instance.

Now we discuss the implementation of state-dependent changes to attribute types. Operation D2 (changing a weak reference to a shared composite reference) is implemented as follows (operation D1 is similar). Again, we suppose that a class C' has an attribute A whose domain is a class C.

1. Access all instances of the class C' and determine all instances of the class C which are referenced through A.
2. For all instances of the class C determined above, verify that they have no exclusive references.
3 If any instance has an exclusive reference to it, reject the change. Otherwise, add reverse composite references to the instances of C.

Step 2 above may be very expensive, since there is no reverse reference corresponding to a weak reference.

Operation D3 (changing a shared composite reference to an exclusive composite reference) is implemented in a similar way.

1. Access all instances of the class C.
2. Reject the change if an instance O exists such that O has more than one reverse composite reference, and at least one of the reverse composite references is from an instance of the class C'. Otherwise, turn on the X flag in all reverse composite references to instances of the class C'.

5. Versions of Composite Objects

The extended semantics of composite references also necessitate changes to the model of versions of composite objects described in [KIM87b]. In this section, we present the extended model of versions of composite objects.

5.1 Review of the ORION Version Model

Let us begin with a brief review of the model of versions of objects implemented in ORION [CHOU86, CHOU88]. ORION allows the user to optionally declare a class to be *versionable*, in which case an instance of the class is a versionable object. A versionable object is in essence a logical collection of version instances in which one version instance has been derived from another version instance. Any number of new version instances may be derived from any version instance in a version-derivation hierarchy, giving rise to a hierarchy of version instances called a *version-derivation hierarchy*. The history of derivation of version instances for a versionable object is maintained in a *generic instance*.

If an object O is a versionable object, an object O' may be bound to (i.e., reference) O in one of two ways: statically or dynamically. O' is said to be *statically bound* to O, if O' references directly a specific version instance of O. If O' references the generic instance of O, O' is said to be *dynamically bound* to O. If an object O' is dynamically bound to another object O, the system determines the *default version instance* of O and binds O' to it. The user may specify the default version instance for any given versionable object; in the absence of a user-specified default, the system determines the system default on the basis of a timestamp ordering of the creation of the version instances.

5.2 Model of Versions of Composite Objects

One difficulty in extending the model of versions of objects to a model of versions of composite objects is extending the semantics of a composite reference between a pair of instance objects to that between a pair of generic instances and between an instance object and a generic instance. [KIM87b] takes the view that the semantics of a composite reference apply to a pair of generic instances, and that they transfer to one pair of version instances of the respective generic instances. The following set of rules captures the semantics of versions of composite objects corresponding to the extended model of composite objects. ([KIM87b] uses five rules, but we have consolidated them into four rules; the reader should recall that [KIM87b] restricts the composite objects to those defined through dependent exclusive composite references.) Let us consider a pair of classes C and D, and suppose C has a composite attribute A whose domain is D, and both C and D are versionable classes.

Rule CV-1X: The existence of a composite reference from a generic instance g-c of the class C to a generic instance g-d of the

class D means that any number of version instances of g-c may have the same composite reference to g-d.

Rule CV-2X: A version instance may have at most one composite reference to it, if the reference is exclusive; or any number of composite references to it, if they are all shared references. A generic instance may have more than one exclusive composite reference to it, only if all references are from objects that belong to the same version-derivation hierarchy. However, it may have any number of shared composite references to it.

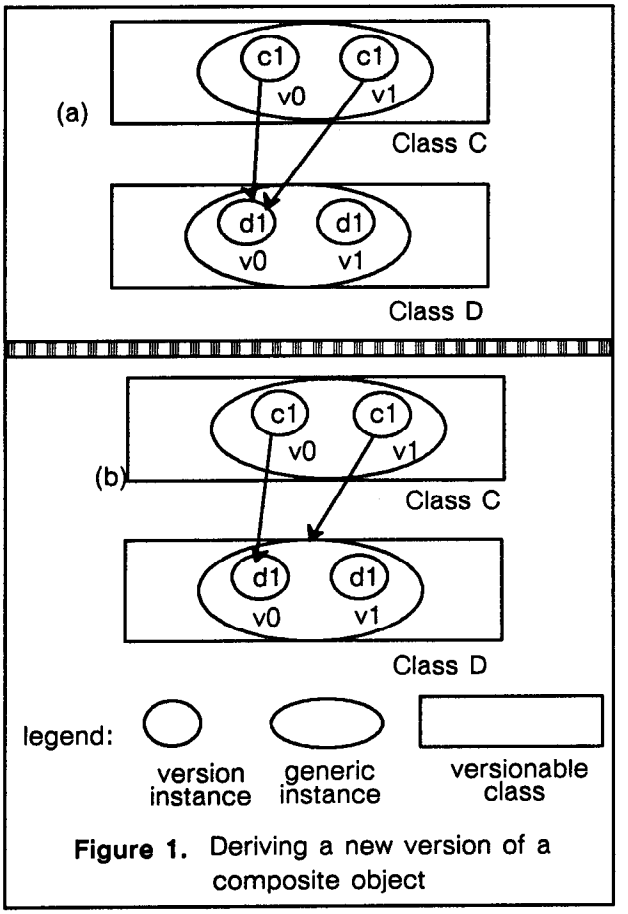

Figure 1. Deriving a new version of a composite object

As a consequence of Rule CV-2X, when a version instance c-i of the class C is copied to derive a new version instance c-j, and c-i has an exclusive reference to a version instance d-k (rather than a generic instance g-d) of the class D, the new version instance will have the same exclusive reference that the initial version instance has (as shown in Figure 1a). The reference in the new copy is set to the generic instance g-d of the referenced version instance (as shown in Figure 1b). However, if the reference is a dependent composite reference, it is set to Nil.

Rules CV-1X and CV-2X together imply that different version instances of the same generic instance g-c may have composite references to different version instances of the generic instance g-d, as long as each version instance of g-d is referenced through one exclusive composite reference or only shared composite references. Figure 2 illustrates this.

Rule CV-3X: The existence of a composite reference from a version instance c-i of a generic instance c-g to a version instance d-j of a generic instance d-g implies a composite reference from c-g to d-g.

Rules CV-2X and CV-3X together prevent version instances of different versionable objects, say, O' and O", from having exclusive composite references to different version instances of the same versionable object O.

Rule CV-4X: When a generic instance g-c is deleted, all generic instances to which it has exclusive references are recursively deleted. Further, if a generic instance is deleted, all its version instances are deleted; and if the last remaining version instance of a generic instance is deleted, the generic instance is also deleted.

Rules CV-2X and CV-4X together imply that the deletion of a version instance causes a recursive deletion of all version instances statically bound to it through dependent references. If the deleted version instance c-i is the only version of the object O, its generic instance g-c is also deleted along with all generic instances which have composite dependent references from g-c.

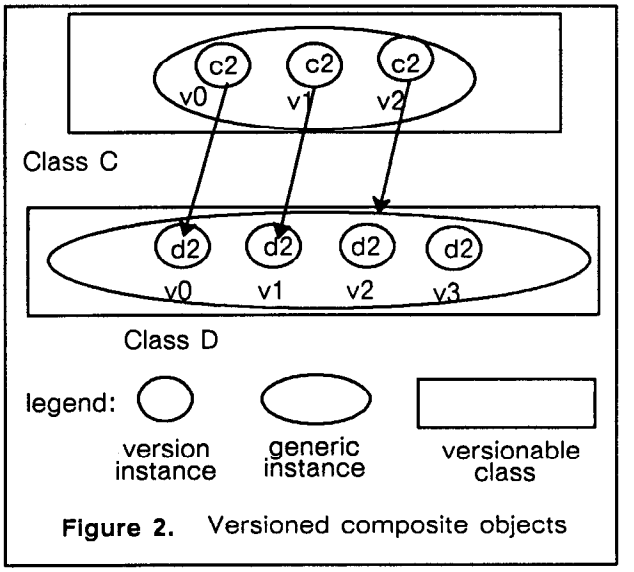

Figure 2. Versioned composite objects

5.3 Implementation

The major issue in implementing versions of composite objects is the storage of reverse composite references. Let us consider a composite reference from an object O' to a versionable object O.

1. If the reference is to a version instance v of O,
   - a reverse composite reference to O' is stored in the version instance v.
   - if O' is not a versionable object, a reverse composite reference to O' is also stored in the generic instance g of O.
   - if O' is a versionable object, a reverse composite reference to the generic instance g' of O' is stored in the generic instance g of O.
2. If the reference is to the generic instance g of O,
   - if O' is not a versionable object, a reverse composite reference to O' is stored in the generic instance g of O.
   - if O' is a versionable object, a reverse composite reference to the generic instance g' of O' is stored in the generic instance g of O

A reverse composite reference from g of O to g' of O', which we will call a *reverse composite generic reference*, has associated with it a counter, called **ref_count**, which keeps track of the number of composite references from version instances of O' to version instances of O. The ref_count is used to determine when a reverse composite generic reference must be removed.

The reason we replicate the reverse composite reference, in case 1 above, in a generic instance is as follows. Let us assume that there is an exclusive composite reference from an object O'

to a version instance v of O, and that there is another composite reference from an object O" to the generic instance g of O. In order to check the legality of this latter type of reference, if the generic instance g does not have reverse composite references, the system must access all version instances of O to verify that either they have no exclusive references from other objects or, if there are such references, they are from version instances which belong to O".

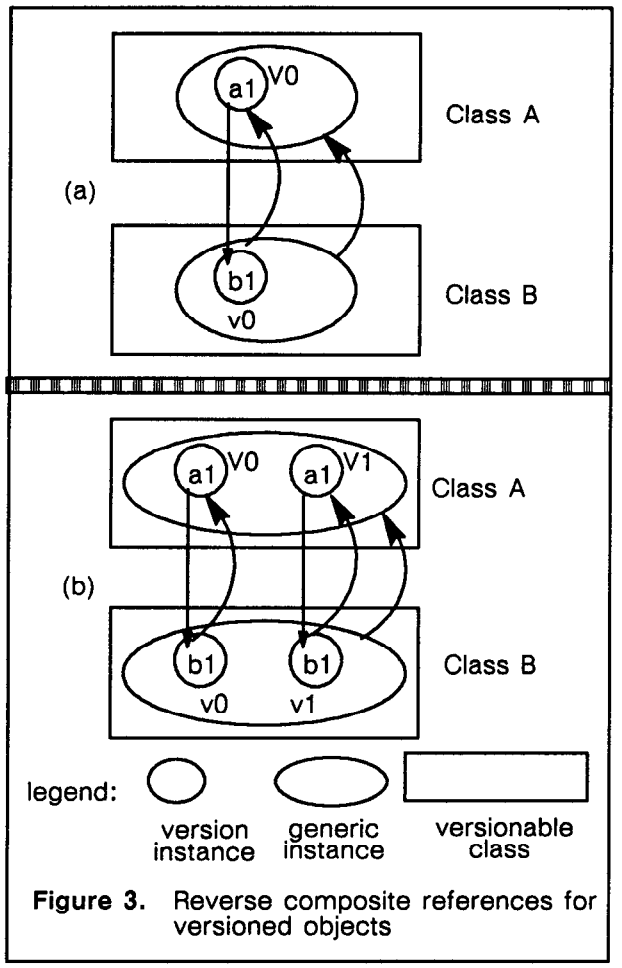

Figure 3. Reverse composite references for versioned objects

Figure 3 illustrates the storage of reverse composite references and reverse composite generic references in version instances and generic instances of versionable objects. For example, the ref_count associated with the reverse composite generic reference from object b1 to object a1 will have a value of 1 or 2 in Figure 3.a and 3.b, respectively. Let us suppose that in Figure 3.b the reference from a1.v0 to b1.v0 is removed. This will cause the reverse composite reference from b1.v0 to a1.v0 to be removed. However, the reverse composite generic reference from b1 to a1 is not removed; only the ref_count is decremented by one. Now if the composite reference from a1.v1 to b1.v1 is removed, then the reverse composite reference from b1.v1 to a1.v1 is removed. This time the reverse composite generic reference from b1 to a1 is also removed, since decrementing ref_count by one will set it to zero. We also note that if the operation **parents_of** is applied on the generic instance b1 in Figure 3.b, the result would be the instance a1, even if all composite references are statically bound.

6. Authorization

The ORION authorization model is based on three fundamental concepts, discussed in detail in [RABI88]. The first is the concept of implicit authorization, that minimizes the amount of storage needed to represent all authorizations in a system by having the system to deduce authorizations from explicitly stored authorizations. The second concept is the positive and negative authorizations which differentiate between prohibition and absence of an authorization. The third is the notion of strong and weak authorizations. A weak authorization can be overridden by other authorizations, while a strong authorization and all authorizations implied by it cannot be overridden. As discussed in [RABI88], the combination of these concepts provides a basis for a powerful authorization mechanism.

In relational database systems, the smallest unit of authorization is a relation or an attribute (column) of a relation. For object-oriented applications which deal with a relatively small number of large objects (such as documents, design files), it is often useful to be able to authorize access to single objects. Further, in systems which support composite objects, it is desirable to include composite objects as a unit of authorization, regardless of whether a composite object is a collection of a small number of large objects or a large number of small objects. If a composite object is a unit of authorization, the user (who created the composite object or who has the grant authorization on it) needs to grant authorization on the composite object as a single unit, rather than on each of the component objects. Further, when a composite object is accessed, the system needs to check only one authorization (for the entire composite object), rather than authorizations on all component objects.

One novel contribution of [KIM87b] is that it shows the use of composite objects as a unit of concurrency; others had proposed the use of composite objects as a unit of physical clustering and retrieval [LORI83, LORI85]. In this section, we further augment the utility of composite objects by introducing their use as a unit of authorization. To do this we extend the notion of implicit authorization to composite classes and composite objects. An authorization on a composite class C implies the same authorization on all instances of C and on all objects which are components of the instances of C. For example, let us consider a composite class hierarchy consisting of the classes Vehicle, Autobody, and AutoDrivetrain. If a user is granted a Read authorization on the class Vehicle, the user implicitly receives the same authorization on all instances of Vehicle, and all instances of Autobody and AutoDrivetrain which are components of the instances of Vehicle. We note, however, that the authorization on Vehicle does not imply the same authorization on all instances of Autobody and AutoDrivetrain, since not all instances of Autobody and AutoDrivetrain may be components of Vehicle. Further, because of negative authorizations, a new authorization issued on a component class may conflict with an authorization on the class which is implied by a previously granted authorization. In this case, the authorization subsystem must reject the new authorization.

Similarly, an authorization on a composite object implies the same authorization on each component of the composite object. For example, if a user is granted a Read authorization on the root of the composite object in Figure 4, the user implicitly receives a Read authorization on each of the component objects, Instance[j], Instance[k], Instance[m], Instance[n], and Instance[o]. Again, if a new authorization issued conflicts with an existing authorization, the new authorization is rejected.

If an instance is a component of more than one composite object, a user can receive more than one implicit authorization on that instance. For example, let us consider the composite objects in Figure 5. If a user receives a Read authorization on the composite object rooted at Instance[j], then the user implicitly receives a Read authorization on Instance[o']. If the user is later granted a Read authorization on the composite object rooted at Instance[k], the user again receives an implicit authorization on Instance[o'].

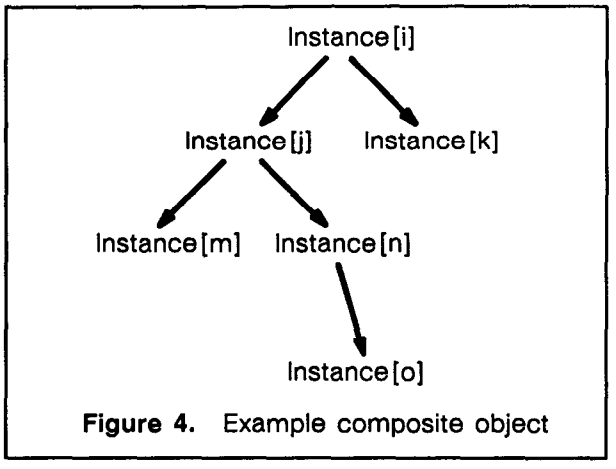

Figure 4. Example composite object

Negative authorizations give rise to conflicts among implicit authorizations on objects which are components of more than one composite object. For example, if the user is granted a negative Read authorization on the composite object rooted at Instance[k] in Figure 5, it will conflict with the (positive) implicit authorization the user received from Instance[j]. In this case, the negative authorization can be granted only if the Read authorization on the composite object rooted at Instance[j] is a weak authorization (and therefore it can be overridden).

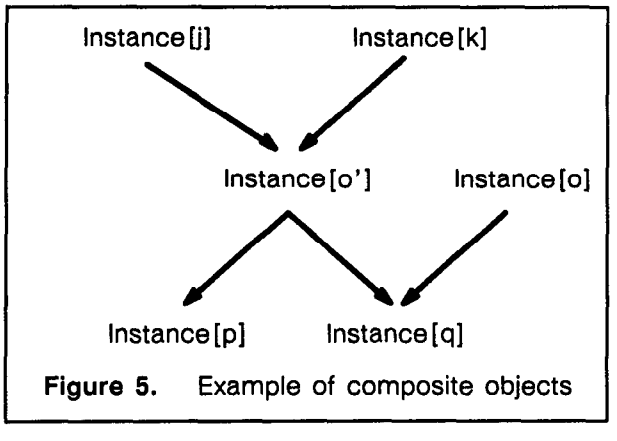

Figure 5. Example of composite objects

The matrix in Figure 6 summarizes conflicts in authorization implied by explicit authorizations on two composite objects rooted at Instance[j] and Instance[k] in Figure 5. The rows indicate all possible authorizations on the composite object rooted at Instance[j] and the columns authorizations on the composite object rooted at Instance[k]. The [i,j]-th element of the matrix contains the resulting authorizations on Instance[o']; the symbol 'Conflict' denotes that a conflict arises on the implicit authorizations on Instance[o']. We consider positive and negative (denoted by ¬), and strong (s) and weak (w) forms of two authorization types, Read (R) and Write (W). For example, s ¬W stands for a strong negative Write authorization. We note that a (positive) W authorization implies a (positive) R authorization; and a negative R authorization implies a negative W authorization [RABI88].

If an object O is a component of n composite objects, and an authorization is granted on one of the composite objects, the authorization subsystem must ensure that the new authorization does not conflict with any of the authorizations on O which are implied by the current authorizations on the composite objects. If there is no conflict, the resulting authorization on O is the

Instance[j] \ Instance[k]	s R	s W	s¬R	s¬W	w R	w W	w¬R	w¬W
s R	s R	s W	Conflict	s R s¬W	s R	s R w W	s R	s R w¬W
s W	s W	s W	Conflict	Conflict	s W	s W	s W	s W
s¬R	Conflic	Conflict	s¬R	s¬R	s¬R	s¬R	s¬R	s¬R
s¬W	s R s¬W	Conflict	s¬R	s¬W	w R s¬W	s¬W	w¬R s¬W	s¬W
w R	s R	s W	s¬R	w R s¬W	w R	w W	Conflict	w R w¬W
wW	s R w W	s W	s¬R	s¬W	w W	w W	Conflict	Conflict
w¬R	s R	s W	s¬R	w¬R	Conflict	Conflict	w¬R	w¬R
w¬W	s R w¬W	s W	s¬R	s¬W	w R w¬W	Conflict	w¬R	w¬W

Figure 6. Implicit authorization on a component object

strongest of all the implied authorizations on O. For example, if a user receives a strong R authorization from Instance[j] and a strong W authorization from Instance[k], the authorization implied on Instance[o'] is a strong W authorization, which in turn implies a strong R authorization. Similarly, if a user receives a strong ¬R authorization from Instance[j] and a strong ¬W authorization from Instance[k], the authorization implied on Instance[o'] is a strong ¬R authorization, which implies a strong ¬W authorization.

When an authorization is granted on a composite object, the authorization component of a database system must ensure that there are no conflicts between the authorization being granted and authorizations (either explicit or implicit) already on any of the component objects. As an example, if a user receives a strong ¬R authorization from Instance[j], a later attempt to grant the user a strong W authorization on Instance[k] will fail. This is because a ¬R implies a ¬W, which contradicts the positive strong W being granted.

7. Locking

[KIM87b, GARZ88] present a locking protocol which recognizes a composite object as a single lockable granule. The protocol is applicable to composite objects consisting of exclusive composite references (i.e., physical part hierarchies). The protocol introduces three non-conventional lock modes: ISO, IXO, SIXO, corresponding to the well-known lock modes IS, IX, and SIX[GRAY78], repectively. These locks are introduced to prevent a transaction from updating/reading a component object of a composite object O while another transaction is reading/updating the entire composite object O. The compatibility semantics of these locks are described in Figure 7. As shown in the compatibility matrix, the main point is that, while IS and IX modes do not conflict, the ISO mode conflicts with IX mode, and IXO and SIXO modes conflict with both IS and IX modes.

To lock an entire composite object using this protocol, the root object is locked in S or X mode, and the root class is locked in IS, IX, S, SIX, or X mode. Further, the component classes of the composite class hierarchy are locked in ISO, IXO, S, SIXO, or X

current mode \ requested mode	IS	IX	S	SIX	X	ISO	IXO	SIXO
IS	✓	✓	✓	✓	No	✓	No	No
IX	✓	✓	No	No	No	No	No	No
S	✓	No	✓	No	No	✓	No	No
SIX	✓	No	No	No	No	No	No	No
X	No	No	No	No	No	No	No	No
ISO	✓	No	✓	No	No	✓	✓	✓
IXO	No	No	No	No	No	✓	✓	No
SIXO	No	No	No	No	No	✓	No	No

Figure 7. Compatibility matrix for granularity locking and exclusive composite object locking

mode, respectively. The following examples illustrate the protocol.

1. Access the vehicle composite object Vi
   a. lock vehicle class object in IS mode
   b. lock the vehicle composite instance Vi in S mode
   c. lock the component class objects in ISO mode.
2. Update the vehicle Vi or its components
   a. lock vehicle class object in IX mode
   b. lock the vehicle composite instance Vi in X mode
   c. lock the component class objects in IXO mode.

This protocol allows multiple users to read and update different composite objects that share the same composite class hierarchy, as long as they update different composite objects.

This protocol is only applicable to composite objects consisting of exclusive composite references; as such, it must be extended for the shared composite references. Information needs to be maintained about the component classes of a composite class hierarchy, and the nature of the references to the component classes. A component class of exclusive references is locked as before. However, three new lock modes are introduced for the component class of shared references: ISOS (intention shared object-shared), IXOS (intention exclusive object-shared), and SIXOS (shared intention exclusive object-shared), which correspond to the ISO, IXO, and SIXO for component classes with exclusive references. Figure 8 shows the compatibility matrix for the expanded set of lock modes.

Let us consider the composite objects in Figure 9, where component object Instance[c] and Instance[c'] belong to the same class C; Instance[w] and Instance[w'] belong to a class W; and Instance[i], Instance[j], and Instance[k] belong to classes I, J, and K, respectively. The following examples illustrate the revised locking protocol.

1. Update the composite object rooted at Instance[i]
   a. Lock class I in IX mode.
   b. Lock composite object Instance[i] in X mode.
   c. Lock class C in IXO mode.
2. Access the composite object rooted at Instance[k]
   a. Lock class K in IS mode.
   b. Lock composite object Instance[k] in S mode.
   c. Lock class C in ISOS mode.
   d. Lock class W in ISO mode.

current mode \ requested mode →	IS	IX	S	SIX	X	ISO	IXO	SIXO	ISOS	IXOS	SIXOS
IS	✓	✓	✓	✓	No	✓	No	No	✓	No	No
IX	✓	✓	No	No	No	No	No	No	No	No	No
S	✓	No	✓	No	No	✓	No	No	✓	No	No
SIX	✓	No	No	No	No	No	No	No	No	No	No
X	No	No	No	No	No	No	No	No	No	No	No
ISO	✓	No	✓	No	No	✓	✓	✓	✓	✓	✓
IXO	No	No	No	No	No	✓	✓	No	✓	✓	No
SIXO	No	No	No	No	No	✓	No	No	✓	No	No
ISOS	✓	No	✓	No	No	✓	✓	✓	✓	No	No
IXOS	No	No	No	No	No	✓	✓	No	No	No	No
SIXOS	No	No	No	No	No	✓	No	No	No	No	No

Figure 8. Compatibility matrix for granularity locking and shared/exclusive composite object locking

3. Update the composite object rooted at Instance[j]
   a. Lock class J in IX mode.
   b. Lock composite object Instance[j] in X mode.
   c. Lock class C in IXOS mode.
   d. Lock class W in IXO mode.

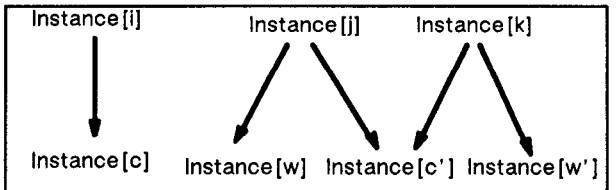

Figure 9. Example composite objects

This protocol allows us to have several readers and writers on a component class of exclusive references, and several readers and one writer on a component class of shared references. Therefore, examples 1 and 2 are compatible, while example 3 is incompatible with both 1 and 2. This protocol, as is the case with the protocol of [KIM87b], also suffers from the restriction that if there is even one reader (writer) via the composite class hierarchy, there cannot be any direct readers or writers via the instances of component classes, or vice versa.

[GARZA88] also describes a locking algorithm which makes use the object identifier of the root of a composite object. The algorithm sets a lock on the root of a composite object when a component object is directly accessed. The algorithm cannot be used for shared composite references. As an example, let us consider the composite objects in Figure 5. Suppose that a transaction T1 requests an S lock on Instance[o']. The algorithm will set locks on the roots of Instance[o'], namely, Instance[j] and Instance[k]. This will cause all components of the composite objects rooted at Instance[j] and Instance[k] to be implicitly

locked. Suppose now that another transaction T2 requests an X lock on Instance[o]. The algorithm will grant T2 the X lock, and implicitly locks Instance[q] in X mode, which of course conflicts with the implicit S lock which T1 holds on the instance.

Both the original protocol of [KIM87b] and the extended protocol just presented are appropriate largely for conventional short transactions. Unfortunately, they may not be suitable for long-duration transactions. For long-duration transactions, it may be better to lock individual component objects as needed. An appropriate locking protocol for long-duration transactions is still a research issue.

8. Summary

In this paper, first we presented a new model of composite objects by cleanly separating out a number of different semantics which the model of composite objects developed in [KIM87b] overloaded on the reference between a pair of objects. The new model distinguishes four types of composite reference, that is, a reference on which the IS-PART-OF relationship between a pair of objects is superimposed. They include independent exclusive, dependent exclusive, independent shared, and dependent shared composite references.

Next, we explored the consequences of the new model of composite objects on the semantics of schema evolution, versioning, authorization, and concurrency control on composite objects. [KIM87b] first described the use of composite objects as a unit of concurrency control, augmenting the use of composite objects as a unit of physical clustering and retrieval proposed in the literature. In this paper, we further enhanced the utility of composite objects by showing their use as a unit of authorization.

The model of composite objects presented in [KIM87b] has largely been implemented in the ORION object-oriented database system at MCC. We are currently re-implementing the composite object feature in ORION to reflect the new model we have developed.

REFERENCES

[ANDR87] Andrews, T., and C. Harris "Combined Language and Database Advances in an Object-Oriented Development Environment," in *Proc. 2nd Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications,* Orlando, Florida, Oct. 1987.

[BANE87a] Banerjee, J., H. T Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, H. J. Kim "Data Model Issues for Object-Oriented Applications," *ACM Trans. on Office Information Systems,* Vol.5, N.1, 1987.

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

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

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

[COPE84] Copeland, G., and D. Maier "Making Smalltalk a Database Systems," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data,* Boston, Mass., June 1984.

[FISH87] Fishman, D., et al. "IRIS: an Object-Oriented Database Management System," *ACM Trans. on Office Information Systems,* Vol.5, N.1, 1987.

[GARZ88] Garza, J. F., and W. Kim "Transaction Management in an Object-Oriented Database System," in *Proc. ACM-SIGMOD Intl. Conf. on Management of Data,* Chicago, May 1988.

[GRAY78] Gray, J.N. *Notes on Data Base Operating Systems,* IBM Research Report: RJ2188, IBM Research, San Jose, Calif. 1978.

[HAMM81] Hammer, M., and D. McLeod "Database Description with SDM: A Semantic Data Model," *ACM Trans. on Database Systems,* Vol.6, N.3, 1981.

[KIM87a] Kim, W., H. T Chou, and J. Banerjee "Operations and Implementation of Complex Objects," in *Proc. Data Engineering Conference,* Los Angeles, Calif., Feb. 1987.

[KIM87b] Kim, W., J. Banerjee, H.T Chou, J. F. Garza, D. Woelk "Composite Object Support in an Object-Oriented Database Systems," in *Proc. 2nd Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications,* Orlando, Florida, Oct. 1987.

[LORI83] Lorie, R., and W. Plouffe "Complex Objects and Their Use in Design Transactions," in *Proc. Databases for Engineering Applications,* Database Week 1983 (ACM), San Jose, Calif., May 1983.

[LORI85] Lorie, R., W. Kim, D.McNabb, W.Plouffe, A.Meier "Supporting Complex Objects in a Relational System for Engineering Databases" 1985.

[RABI88] Rabitti, F., D. Woelk, W. Kim "A Model of Authorization for Object-Oriented and Semantic Databases," in *Proc. Intl. Conf. on Extending Database Technology,* Venice, Italy, March 1988.

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

[STON86] Stonebraker, M., and L.Rowe, "The Design of POSTGRES," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data,* Washington D.C., May 1986.