# Object Versioning in Ode

*R. Agrawal*

IBM Almaden Research Center
San Jose, California 95120

*S. J. Buroff*

AT&T Bell Labs
Summit, New Jersey 07901

*N. H. Gehani*

AT&T Bell Labs
Murray Hill, New Jersey 07974

*D. Shasha*

New York University
New York, NY 10012

## 1. INTRODUCTION

Ode [2-4] is a database system and environment based on the object paradigm. It is an attempt to build a database system that offers one integrated data model for both database and general purpose manipulation. The database is defined, queried, and manipulated using the database programming language O++, which is an upward-compatible extension of the object-oriented programming language C++ [ Stroustrup 1986 ]. O++ extends C++ by providing facilities suitable for database applications, such as facilities for creating persistent and versioned objects, defining and manipulating sets, organizing persistent objects into clusters, iterating over clusters of persistent objects, and associating constraints and triggers with objects.

Providing support for design environments, be it electrical computer-aided design (ECAD), mechanical computer-aided design (MCAD), or software computer-aided design (SCAD), has emerged as the major new application domain for database systems. It may also be said that these applications are the primary force driving the current interest in the object-oriented database systems. At present, there is little agreement on one uniform model for design databases, although some recent papers have articulated major needs [1, 5, 7-9, 13-17, 19-21, 24, 25, 28, 29, 32-34]. Consequently, O++ does not come with a ''hardwired'' design database model; instead, it provides facilities with which users can develop customized models that match their needs.

In designing O++, we have distinguished between primitives and policies. Our design effort has been directed towards coming up with a few but powerful language primitives that allow a wide variety of policies to be implemented, and yet keep the language small. If some functionality could be realized using existing language facilities, we did not add new language primitives. For example, we consciously decided not to introduce new pointer types (such as `own ref` in [12]) to model composite objects [23] with ''local objects'' which are deleted when the composite object is deleted because this can be simulated using C++ destructors. Similarly, we decided against a built-in change notification facility [13] because users can implement such a facility using O++ triggers.

A critical requirement of CAD applications that is addressed by O++ is support for object versioning. Our design, although inspired by many of the earlier proposals [1, 5, 7-9, 13-17, 19-21, 24, 25, 28, 29, 32-34], differs significantly from them in that we focus on introducing a minimum number of concepts with precise semantics and integrating them seamlessly in a programming language. Object versioning in O++ is orthogonal to type, that is, versioning is an object property and not a type property. Versions of an object can be created without requiring any change in the corresponding object type definition, all objects can be versioned, and different objects of the same type can have a different number of versions. O++ supports both dynamic and static bindings to version references. Temporal as well as derived-from relationships between versions are also maintained automatically. Although we introduce few new language constructs, they are powerful enough to make O++ suitable as a platform for implementing a variety of versioning paradigms and application-specific systems. Indeed, we have successfully modeled a design database in

production use in our VLSI laboratory using the O++ versioning facilities. This approach should be contrasted with the approaches where emphasis seems to be on introducing more and more features at the expense of semantics ambiguities and implementation complexities without significantly enhancing the basic functionality. The semantic clarity and feature minimality of our design also yielded a clean implementation.

The rest of the paper describes the versioning facilities provided by O++. Section 2 contains a discussion of our design goals and Section 3 formally defines the semantics of the O++ versioning facilities. In Section 4, we present O++ facilities for creating and manipulating versions, and Section 5 presents our experience of modeling the design database used in our VLSI laboratory using these versioning facilities. In Section 6, we discuss our implementation of O++ versioning facilities. Section 7 contains a discussion of related work, and we conclude with some closing remarks in Section 8.

## 2. O++ VERSIONING DESIGN GOALS

When designing the version facilities in O++ we kept the following design goals in perspective:

- *Version orthogonality*: Object versioning should be orthogonal to type, that is, versioning should be an object property and not a type property. One should be able to create versions of an object without requiring any changes to the type definition of that object. All persistent objects should be able to have versions, and it should be possible to create both versioned and non-versioned objects of the same type. Finally, different objects of the same type should be able to have a different number of versions.

  Version orthogonality allows the decision to create a version of an object can be taken on as needed basis, rather at the time of initial database design. However, most of the current versioning models do not provide version orthogonality. For example, in ORION [13], only objects of types declared to be versionable can be versioned. In IRIS [8], a previously unversioned object can be versioned, but it has to go through a transformation procedure.

- *Generic and specific references*: Depending upon the application, one should be able to access the latest version or a specific version of an object [6, 8, 16, 21]. For example, an address-book object that keeps track of current addresses requires references to the latest versions of person objects to access their latest addresses (generic, dynamic or late binding). On the other hand, a software configuration object can be implemented to reference specific versions of objects representing component modules (static binding). Using a ‘‘logical object id’’, an address-book object can be made to refer to the latest versions of `person` objects. Thus, when the address of a person is changed and a new version of the `person` object created, no change is required to the address-book object. Using ‘‘version ids’’, an object, such as a software configuration object, can be made to refer to specific versions of a component object even though new versions of the objects may have been created.

- *Temporal and derived-from relationships*: Versions of an object should be ordered temporally according to their creation time, which is important for historical databases, such as those used in accounting, legal, and financial applications, that must access the past states of the database [14, 29], and for supporting time in databases [30]. In addition, derived-from relationships reflecting the derivation history of the versions of an object should also be maintained to meet the requirements of the design environments in which several versions of a new design may be derived from the same design by making different changes to it [6, 21, 24]. The derived-from relationship can be used to store versions by storing their ‘‘differences’’ (called deltas [28, 32]). Some current versioning proposals (GemStone [14] and POSTGRES [29], for example) constrain the version relationship of an object to be linear, which is inadequate for design databases.

- *Small changes should have small impact*: Small operations should not cause major changes in the database. Thus, we do not provide version percolation [5, 13, 34] because creating a new version can lead to the automatic creation of a large number of versions of other objects. Users may implement version percolation as a policy by using other O++ facilities.

## 3. FORMAL MODEL

We first present a formal description of the model underlying the versioning facilities of Ode. In the next section, we will present the linguistic constructs provided in O++ that implement this model.

### 3.1 Persistent Object

A persistent object is a non-null set of versions with *temporal* and *derived-from* relationships between the versions.

The temporal relationship is a total ordering on the versions of an object based on the creation time of the versions. It is established when a new version of an object is created and is not affected by updates to the versions. The temporal relationship preserves the history, in discrete steps, of the evolution of an object.

The derived-from relationship is a partial ordering on the versions of an object that is based upon the derivation history of the versions. A new version (except the initial version) is always based on an existing version; the new version has a derived-from relationship to the version it is based on. The derived-from relationship, like the temporal relationship, is established at version creation time and is not changed by updates to versions. The derived-from relationship between versions of an object can be a tree.[1]

Formally, a persistent object consists of a triple ($V$, $D$, $T$) where $V$ is a non-empty set of versions, $D$ is a set of directed edges on $V$ with the topology of a tree, and $T$ is a 1-1 mapping from $V$ to time. If ($v$, $w$) is in $D$, then we say that $v$ was derived from $w$ or that $w$ is the parent of $v$. Persistent objects and individual versions are identified by object ids and version ids, respectively. Given a version id $v$, function `objectid` yields the corresponding object id, that is, if the object with id $o$ is represented as the triple ($V$, $D$, $T$) and $v \in V$, then `objectid`($v$) = $o$. The id of latest version of $o$, `latest`($o$), is given by

$$\texttt{latest}(o) = \{v \mid v \in V \ /\backslash \ \forall w \in V \text{ if } v \neq w \text{ then } T(w) < T(v)\}$$

### 3.2 Object Creation

A persistent object is created by the operator `pnew` which takes as its arguments the type of the object to be created and, if appropriate, initial values. Semantically, this operator creates a triple ($V$, $D$, $T$) where $V = \{v\}$, $D = \{\}$, and $T = \{(v, \mathit{current\_time}())\}$ and associates a unique object id with it which is returned as the result. Depending upon how the object type is declared, $v$ is either uninitialized, initialized by default, or initialized using the initial values supplied as arguments to `pnew`.

### 3.3 Creating Versions of an Object

Updating an object does not create a new version. New versions must be created explicitly by invoking the function `newversion` which takes as an argument an object id or a version id. If the argument passed is an object id, then the new version is derived from the latest version of the object; otherwise, it is derived from the specified version. In both cases, the creation time associated with the newly created version is the current time. We now define the semantics of `newversion` formally.

Let $c$ = `latest`($o$) before `newversion` is invoked. When object id $o$ of the object ($V$, $D$, $T$) is supplied as an argument to the function `newversion`, then

`newversion`($o$) = $w$ where $w \notin V$.

and the contents of the version $w$ are identical to those of $c$. A side effect of invoking `newversion` is that

---

1. In general, one would like the derived-from relationship to be a directed acyclic graph to model the merging of multiple versions into one version. We decided against acyclic graphs due to the implementation cost and semantic complexity of such generality. If a version has been derived from more than one version, one can identify a primary version, thereby rendering the derived-from relationship to be a tree. It has also been ORION experience that it is not necessary to allow the derived-from relationship to be an acyclic graph; trees are sufficient [personal communication from Won Kim].

   Strictly speaking, the derived-from relationship can become a forest of trees as a result of deleting the root version (see section 3.4). However, as this does not affect the semantics of our model, we will assume, for simplicity of exposition, that the derived-from relationship is a tree.

the object referenced by $o$ is updated to the triple $(V', D', T')$, where $V' = V \cup \{w\}$, $D' = D \cup \{(w, c)\}$, and $T' = T \cup \{(w, current\_time())\}$.

When a version id $v$, where $v$ is a version of $o$, i.e., `objectid(`$v$`)` $= o$, is supplied as an argument to `newversion`, then

`newversion(`$v$`)` $= w$ where $w \notin V$.

and the contents of the version $w$ are identical to those of $v$. A side effect of invoking `newversion` is that the object referenced by $o$ is updated to the triple $(V', D', T')$, where $V' = V \cup \{w\}$, $D' = D \cup \{(w, v)\}$, and $T' = T \cup \{(w, current\_time())\}$.

Note that since $T$ is a 1-1 mapping, only one version can be created at a time.

### 3.4 Deleting Versions of an Object

Operator `pdelete` can be used to delete an entire object or a specific version. When `pdelete` is called with an object id $o$ as the argument the entire object (all of its versions) is deleted. The object specified by the object id $o$ then becomes unavailable. When `pdelete` is called with a version id as the argument, the corresponding version is deleted. The temporal and derived-from relationships are updated appropriately. The derived-from relationship is updated so that the parent of the deleted node becomes the parent of the deleted node's children.[2] Formally, deleting a version with id $v$ where `objectid(`$v$`)`$=o$ and $(V, D, T)$ is the triple associated with $o$, causes the object $o$ to be updated to the triple $(V', D', T')$ where $V' = V - \{v\}$ and

$$D' = D - \{(w, v) | (w, v) \in D\} - \{(v, w) | (v, w) \in D\} \cup \{(w, x) | (w, v) \in D, (v, x) \in D\}$$

and $T' = T - \{(v, t) | (v, t) \in T\}$. If $V$ becomes empty as result of the delete operation, the object id $o$ becomes unavailable.

### 3.5 Accessing Versions of an Object

Function `Tprevious` is used to access the temporally previous version and function `Dprevious` is used to access the version from which a specific version was derived from. The semantics of these functions are now formally defined.

`Tprevious(`$v$`)`, where $v$ is a version id, returns the id of the version created just before $v$. That is, if `objectid(`$v$`)` $= o$, and $o$ references the object denoted by the triple $(V, D, T)$, then `Tprevious(`$v$`)` $= w$ such that $w \in V$, $T(w) < T(v)$, and $\forall u \in V | u \neq w$ and $T(u) < T(v)$, we have $T(u) < T(w)$. If no such version exists, then `Tprevious` returns null.

`Dprevious(`$v$`)`, where $v$ is a version id, returns the id of the version from which $v$ was derived. That is, if `objectid(`$v$`)` $= o$, and $o$ references the object denoted by the triple $(V, D, T)$, then `Dprevious(`$v$`)` $= w$, $w \in V$, such that $(v, w) \in D$. If no such version exists, `Dprevious` returns null.

Calling `Dprevious` or `Tprevious` with an object id $o$ as an argument is equivalent to calling them with `latest(`$o$`)` as the argument.

Function `leaves` is used to traverse the latest versions of all the alternative designs. Given an object id, `leaves` returns a set of version ids that are the leaves of the derived-from relationship graph (which is a tree). Formally, let object id $o$ reference the object denoted by the triple $(V, D, T)$. Then `leaves(`$o$`)` yields a set of version ids, i.e.,

`leaves(`$o$`)` $= \{v | (v, w) \in D \ /\backslash \ \exists u | (u, v) \in D\} \cup V$ if $D$ is empty.

2. Note that deleting the root version will cause the tree of versions to become a forest.

## 4. O++ CONSTRUCTS

We now discuss how the abstract model presented in the previous section has been incorporated in O++. O++ supports both object ids and version ids. However, an object id does not refer to a generic object header as in [6, 8]; rather, it logically refers to the latest version of the object.[3] Therefore, the latest version of an object can be accessed by simply using the object id. Thus, if an object $o_1$ is interested in the latest version of $o_2$, it should save the object id of $o_2$; if $o_1$ is interested in a specific version of $o_2$, it should save the id of that specific version. O++ also provides functions to coerce a version id to the corresponding object id and to coerce an object id to the id of the latest version.

Although O++ provides both object and version ids, it does not provide separate types for them — both are of type pointer to a persistent object. Each id contains information (or points to information) indicating whether the id refers to an object or to a version. Providing separate types for object ids and version ids may enhance the readability of the code, but has the disadvantage that users cannot write a function that can be called, in different invocations, with an object id or with a version id. A function parameter would match either an object id or a version id but not both, thus requiring users to write multiple functions to perform similar operations on objects and versions. Another disadvantage of separate types is that the same type definition cannot be used to define objects, some of which contain references to objects and some to specific object versions.

The specific O++ constructs that we have added to C++ to support versioning facilities are presented now.

### 4.1 Object Definition

The C++ object facility is called the class. Class declarations consist of two parts: a specification (type) and a body. The class specification can have a private part holding information that can only be used by its implementor, and a public part which is the type's user interface. The *body* consists of the bodies of functions declared in the class specification but whose bodies were not given there. For example, here is a specification of the class `item`:

```
#include "db.h"
class item {
    Name nm;
    double wt; /* in kg */
public:
    item(Name xname, double xwt);
    Name name();
    double weight_lbs();
    double weight_kg();
};
```

As stated, the object definition for versioned and non-versioned objects is identical.

### 4.2 Initial Object Creation

O++ visualizes memory as consisting of two parts: volatile and persistent. *Volatile* objects are allocated in volatile memory and are the same as those created in ordinary programs. *Persistent* objects are allocated in persistent store and they continue to exist after the program that created them has terminated. Each persistent object is identified by a *unique* object identity [22]. The object identity is referred to as a *pointer to a persistent object*.

A persistent object can be created using the persistent storage operator `pnew`. Here is an example:

---

3. We do not discuss concurrency control issues in this paper. For the purposes of this paper, it is sufficient to assume that suitable concurrency control mechanisms exist to ensure that while an uncommitted transaction has created a new version of an object, no other transaction can be creating another version of the same object or have access to this uncommitted version.

```
persistent stockitem *pip;
...
pip = pnew item(initial-values);
```

`pnew` allocates the `item` object in persistent store and returns its id in `pip`. Note that `pip` is a pointer to a persistent `item` object, and *not* a persistent pointer to a `item` object.

Observe that at the time the object was created, the user did not have to decide whether or not versions will be created for this object.

### 4.3 Version Creation

New versions are created by calling function `newversion` which takes as an argument an object id or a version id:

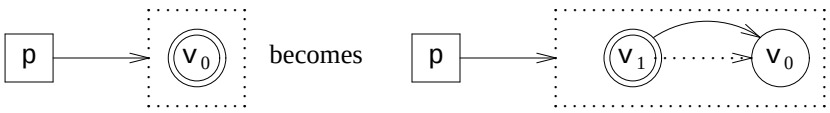
```
newversion(objectid-or-versionid)
```

If the argument passed is an object id, then the new version is derived from the temporally latest version of the object; otherwise, it is derived from the specified version. In either case, the object id starts referencing the new version which then becomes the latest version. `newversion` always returns a version id that refers to the newly created version.

As an example, consider the following code

```
persistent part *p, *vp;
...
p = pnew part;           /* creation of a new object */
...
vp = newversion(p);      /* creation of a version of this object */
```

Assuming that `p` points to the object $v_0$, the effect of the `newversion` call can be described pictorially as

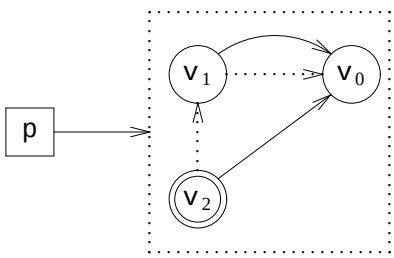



The dotted boxes identify objects and they encapsulate all versions (represented by circles) of the object. Current versions are represented by double circles. Arrows pointing to the boxes refer to objects (and therefore to their latest versions). If an arrow points directly to a circle, then it refers to a specific version. Dotted arrows between object versions represent the temporal relationship between versions, whereas the solid arrows represent the derived-from relationship. Note that $v_1$ can be thought of as a revision [6, 21, 24] of $v_0$. Also, observe that when creating a version, no changes were required in the type definition of this object.

Continuing with the example, we can create another version of $v_0$ with the call

```
newversion(vp0)
```

where `vp0` contains the id of version $v_0$. The effect of the call is to derive a new version $v_2$ based on $v_0$, as shown below:

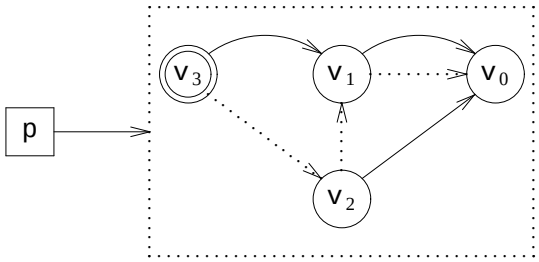


$v_1$ and $v_2$ can be thought of as a variants or alternatives [6, 21, 24], both derived from $v_0$.

Finally, we create another version of $v_1$ with the call

```
newversion(vp1)
```

where `vp1` contains the id of version $v_1$. The `part` object can now be depicted pictorially as follows:




Note that $v_3$, $v_1$, and $v_0$ constitute a version history [13, 21].

### 4.4 Deleting Versions

Given an object id, operator `pdelete` deletes the object and all its versions. Given a version id, `pdelete` deletes the specified version. The temporal and derived-from relationships are updated appropriately as defined in Section 3.

### 4.5 Accessing Versions

O++ provides functions `Dprevious`, `Tprevious`, and `leaves` to access versions of an object. Semantics of these functions have been defined in Section 3. `Dprevious` is used to traverse from a version to the version it was derived from, and `Tprevious` is used to traverse from a version to the version that temporally precedes the specified version. In a design environment, parallel versions derived from the same ancestor are called *alternatives* [6, 21, 24], and each path from the root of the derived-from tree to a leaf represents evolution of an alternative design. Each leaf of the tree represents the most up-to-date version of an alternative design. Function `leaves` is used to visit all up-to-date versions.

### 4.6 Coercion

Given a version id, the function `objectid` yields the id of the object to which the specified version belongs. Given an object id, the function `latest` returns the id of the latest version.

## 5. DMS: A CAD DATABASE EXAMPLE

To illustrate the use of O++ versioning facilities, we will model a CAD design evolution. This example is an abbreviated version of our simulation of the DMS design database system [26] being used in our VLSI design laboratory. We will first create an initial design state and then change this state by adding new versions.

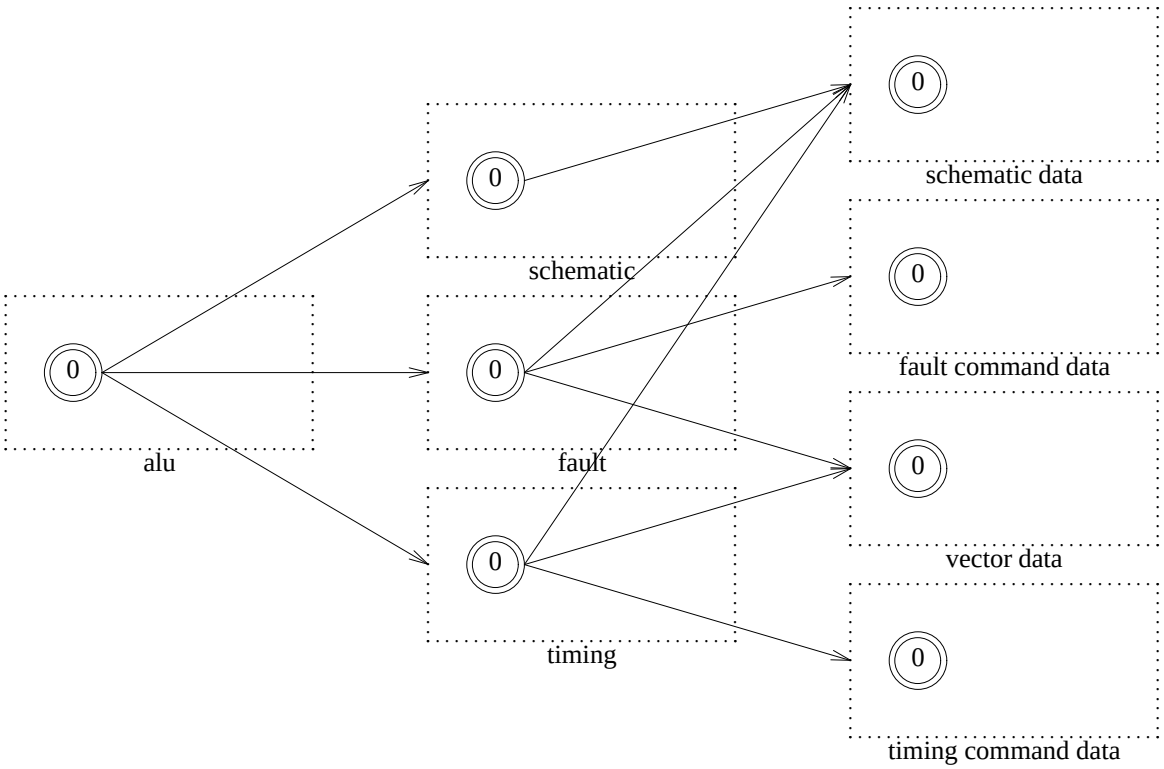
We will design an ALU chip that has several representations [6, 21, 24] of which we will only consider three in this example: schematic, fault and timing. Each representation consists of a set of data objects. The schematic representation only consists of the schematic data. The fault representation consists of the schematic data (same as the one in the schematic representation), vectors, and fault commands. The timing representation consists of the schematic data (same as the one in the schematic representation), vectors (same as the one in the fault representation), and timing commands. Thus, the ALU and its representations are complex objects [6, 21, 24] that can be modeled using C++ class definition facility.

Here are some class definitions that we use:

**File:** dms.h

```
#define MAX 16

class part {
public:
    persistent representation *schematic,
                    *fault, *timing;
    …
};

class representation {
public:
    persistent dataobject *schematic,*cmd,*vector;
    persistent part *components<MAX>;
    persistent config *parent;
    …
};
```

The initial design state is pictorially shown below:




In the figures in this section, temporal relationships between versions of an object are implied by the version numbers. Derived-from relationships between the versions are implied by the spatial placement of versions inside a dotted box (which surrounds a logical object). A double circle, as before, represents the current version, an arrow pointing to a box refers to a logical object, and an arrow pointing directly to a circle refers to a specific version.

The above design state can be created by using the following instructions:

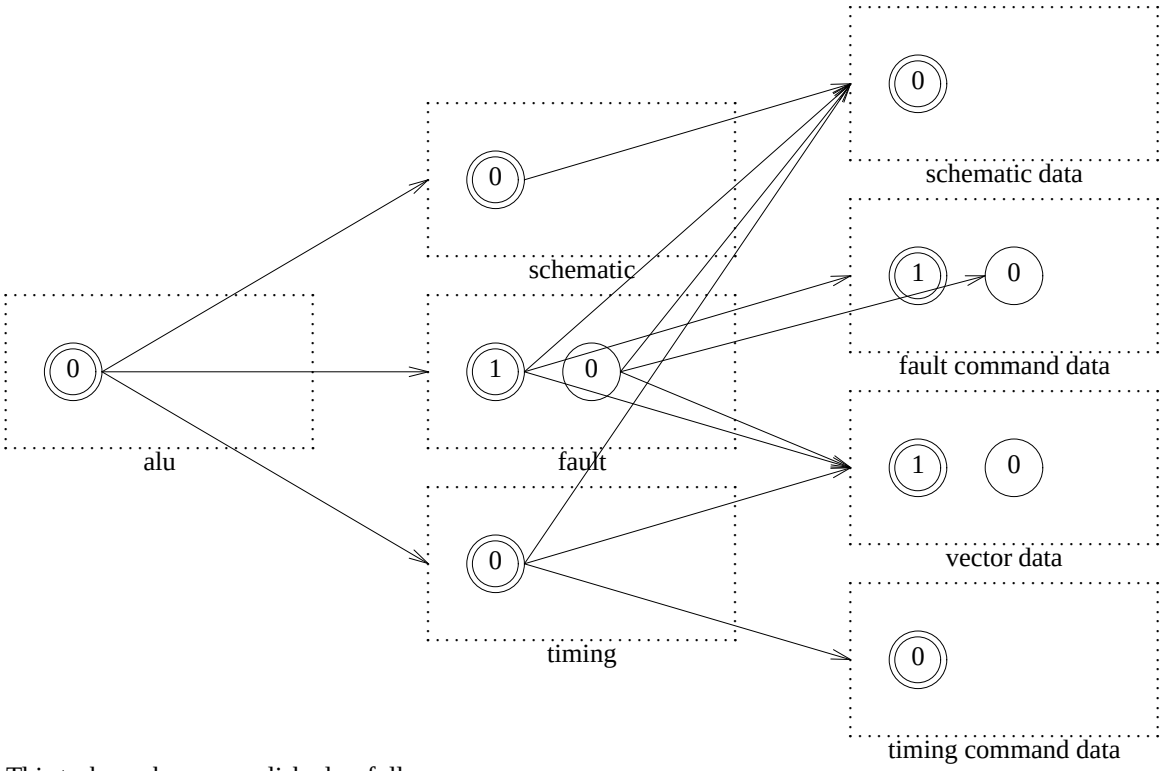
```
#include "dms.h"
persistent part *alu;

alu = pnew part;  /*creating an alu object*/

/*creating the schematic representation of the alu*/
 alu->schematic  = pnew representation;

 alu->schematic->schematic=pnew dataobject("alu/schematic0");

/*creating the fault representation of the alu*/
 alu->fault = pnew representation;

 alu->fault->schematic = alu->schematic->schematic;
 alu->fault->vector=pnew dataobject("alu/fault/vectors0");
 alu->fault->cmd=pnew dataobject("alu/fault/cmd0");

/*creating the timing representation of the alu*/
 alu->timing = pnew representation;

 alu->timing->schematic = alu->schematic->schematic;
 alu->timing->vector = alu->fault->vector;
 alu->timing->cmd=pnew dataobject("alu/timing/cmd0");
```

Now the ALU designer creates a new version of vectors as follows:

```
newversion(alu->fault->vector);
/* modify the new version of the vectors */
...
```

Anyone using the timing or fault representation will now see this new version of vectors because these representations contain the object id of the vector object and hence always refer to its latest version.

The designer now wants to create a new version of the fault representation that refers to a new version of the fault commands. At the same time, the designer wants to make the old fault representation refer to the old fault commands regardless of future changes to the fault commands. The desired design state is pictorially illustrated below:




This task can be accomplished as follows:

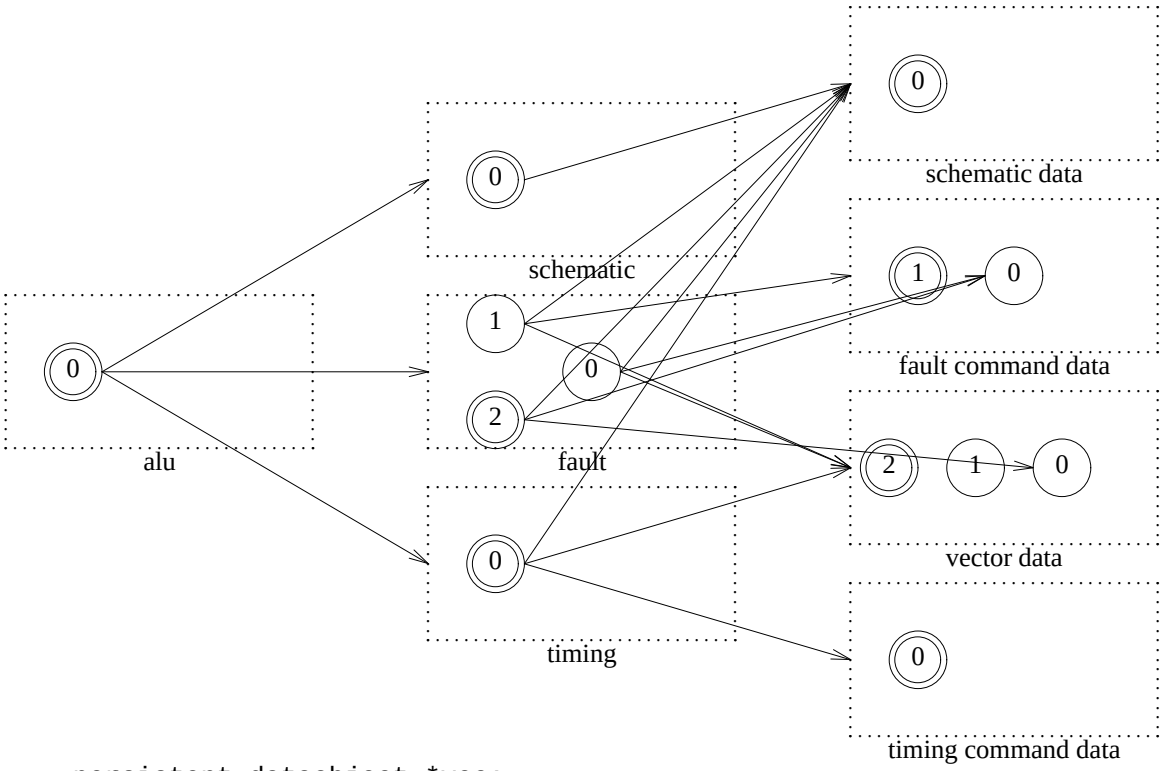
```
newversion(alu->fault);  /*create a new fault representation*/
                         /*alu->fault now refers to the new */
                         /*representation*/
Tprevious(alu->fault)->cmd = latest(alu->fault->cmd);
                  /*freeze the fault command object referenced by the*/
                  /*old version of the fault representation.*/
newversion(alu->fault->cmd);  /*a new version of the fault command object*/
/* modify the new version of fault commands */
...
```

Continuing with our design evolution, the ALU designer now creates another version of vectors:

```
newversion(alu->fault->vector);
/* modify the new version of the vectors */
...
```

Note that the timing representation and both versions of the fault representation will now refer to this new version of vectors.

Finally, the designer creates an alternative version of the fault representation which is also derived from the very first representation. This representation refers to the latest version of the schematic data but it refers to the original versions of the vectors and fault commands. The desired design state is shown below:




```
persistent dataobject *vec;
…
/* create a new fault representation */
newversion(Tprevious(alu->fault));
/* alu->fault now refers to the new representation */
/* access the original version of vectors */
for (vec=alu->fault->vector;Tprevious(vec)!=NULL;vec=Tprevious(vec))
      ;
/* the new fault representation refers to this version of vectors */
alu->fault->vector = vec;
```

There was no need to update references to the fault command object because the original fault representation object points to the appropriate fault command object. Note that the macros can be defined, using the existing C++ macro definition facility, to simplify the task of accessing a specific version in a version history.

Each representation can be thought of as a configuration [8, 9, 13, 16, 21, 33]. This example amply illustrates how configurations can be created using the facilities provided in O++. In a similar manner, contexts [5, 8, 13, 16, 21] may also be created to specify default versions.

## 6. IMPLEMENTATION

We are implementing an O++ compiler which translates O++ programs to C++. The resulting C++ code uses a persistence library [10], which contains classes to support, amongst other things, persistent objects and object versions.

C++ supports inheritance, including multiple inheritance [31], which is used for object specialization. The specialized object types inherit the properties of the ‘‘base’’ object type, i.e., the data and functions, of the base object type. We use the inheritance property in the implementation of versions. In order for objects of a class *T* to have versions, the O++ compiler must generate a class *T’* which is the same as *T* but with an instantiation of the generic class `Version` as one of its base classes. Generic class `Version` is instantiated with the class of the objects to be versioned (i.e., *T’*). This transformation is done by the O++

compiler (and not by the user).

The set of versions of an object is represented by an object of the generic class `VersionSet`. The versions in a set are connected in two ways. First, they are kept on a temporal list so that the latest version is always accessible. In addition, a tree of versions is maintained. The place of a version in the tree is determined when the new version is created.

We now discuss the classes used for supporting versioning. C++ specifications of these classes are given in the Appendix.

### 6.1 Class `Version`

This generic class must be a base class of any class whose objects are to be versioned. It has protected constructors so that objects of class `Version` cannot be created.

A `Version` object contains pointer members that link the object into the temporal list and the version tree. It contains member functions to traverse the list and the tree and also to create a new version and to delete itself. The new version is initialized to be a copy of the creating version and is inserted into the tree as the creating version's newest (rightmost) child. The new version is also placed on the front of the temporal chain. A `Version` object also contains a member function that returns a pointer to the `VersionSet` object of which the versioned object is a member.

### 6.2 Class `VersionSet`

A `VersionSet` object represents a set of versions of a particular object. It contains pointers to two objects, the most recent version on the temporal chain and the root version of the version tree. There are member functions for obtaining each of these values. The `VersionSet` class is a generic class like `Version` and is instantiated with the same type; namely, the type of the objects in the version set.

### 6.3 Class `VersionIter`

The generic class `VersionIter` provides an iterator that traverses the temporal list from most recent to oldest version.

### 6.4 Class `VersionLeafIter`

The generic class `VersionLeafIter` provides an iterator that traverses the leaves of the version tree from left to right (oldest to newest).

### 6.5 Class `VersionPtr`

The `VersionPtr` generic class represents a pointer to a set of versions of an object. It contains a type conversion operator that returns a pointer to an object of the type being versioned[4]. A `VersionPtr` object can be in one of three states:

1. A null pointer. In this case, the type conversion operator returns null.
2. Pointing to a `VersionSet` object. In this case, the type conversion operator returns a pointer to the latest version of the object.
3. Pointing to a particular object in the version set. In this case, the type conversion operator returns a pointer to the object.

### 6.6 A Sample of Generated Code

We now show a sample of code that is generated for implementing versions and which uses the above classes.

---

4. The same as the type it was instantiated with.

Consider the following O++ program segment:

```
class employee { … };
persistent employee *p, *pv;
…
/* Create a new employee object */
p = pnew employee;
…
/* Create a new version of employee */
pv = newversion(p);
…
/* Create another version from the one we just created */
pv = newversion(pv);
…
/* Now create a new version from the parent of the current version */
pv = newversion(Dprevious(pv));
…
```

This program segment is transformed to the following C++ code:

```
class employee: public Version(employee) { … };
VersionPtr(employee)  p, pv;
…
p = new employee;
…
pv = p->newVersion();
pv = pv->newVersion();
pv = pv->parentVer()->newVersion();
```

Note that the translated version of `employee` has as its base class an instantiation of the generic class `Version`. The instantiation argument is the class `employee` itself. The pointers to persistent objects `p` and `pv` in the original program segment are declared as objects of type `VersionPtr` instantiated with the class `employee`.

Now suppose the user wants to manipulate the versions of the object pointed to by `p` as follows:

```
/* traverse temporal chain */
for (pv = latest(p); pv != NULL; pv = Tprevious(pv)) { … }
…
/* traverse a derivation history */
for (pv = latest(p); pv != NULL; pv = Dprevious(pv)) { … }
```

The generated code for the above program fragment is as follows:

```
p = pv->versionSet(); // pointer to version set containing this object
for(VersionIter(employee) iter(p); pv = iter(); ){ …}
                      // traversing temporal chain
for(pv=p->versionSet()->currentVer();pv!=NULL;pv=pv->parentVer()){…}
                     // traversing derivation history
```

### 6.7 Experiences with Using C++ as the Implementation Language

We now give a brief summary of some of our experiences in using C++ to implement the object manager (the persistence library). Multiple inheritance, as supported by C++, is essential for our implementation. We translate classes in O++ to classes in C++, which amongst other things, have class `Version` as a base class. Without multiple inheritance we would not have been able to make `Version` a base class for derived classes.

C++ currently does not support generic classes, i.e., classes that can be parameterized with classes and functions. We needed generic classes for our implementation. So we simulated them using the

preprocessor `#define` macros. These generic classes are somewhat difficult to use, even more difficult to implement, and extremely difficult to debug (because the C++ compiler does not see them; it only sees the expanded instantiations). These problems will hopefully go away when C++ is extended with the ''template'' generic facility as has been proposed [18].

We also found operator overloading, which is allowed by C++, to be very convenient. By overloading the definitions of the `->` and `*` operators we were able to define class `VersionPtr` in such a way that its objects could be manipulated just like normal pointers.

## 7. RELATED WORK

The literature abounds with models for version management [1, 5, 7-9, 13-17, 19-21, 24, 25, 28, 29, 32-34]. O++ culls out kernel features from these proposals and provides primitives within the framework of an object-oriented language for implementing a variety of versioning models and application-specific systems. In this section, we highlight the major features of some of the recent versioning models, and indicate how they can be implemented in O++.

Version histories (instances over time), configurations (compositions of specific versions of component objects of a complex object), and equivalences (different views of an object) have been proposed in [21] as a framework for organizing design databases that evolve over time. Version histories are organized as trees. Versions in a configuration can be bound statically or dynamically. This framework can easily be implemented by using the facilities provided in O++ as illustrated by the design example in Section 5.

A version management model based on the concept of version environments has been proposed in [24]. A version environment offers mechanisms for ordering versions by various relationships (time, derived-from, etc.) and partitioning versions according to specific properties (valid, invalid, in-progress, alternative, effective, etc.). A version is inserted in the database explicitly by the user who is also responsible for maintaining relationship graphs by using the operations provided for creating and deleting edges between versions and for assigning versions to partitions. We felt that the temporal and derived-from relationships are common enough to be automatically maintained by the system. The persistence model of O++ [2] does not require explicit insertion of persistent objects in a database; such objects automatically persist across program invocations.

A comprehensive versioning model for public/private distributed architecture of CAD systems has been developed as part of the ORION project [13]. Versions can be transient, working, or released depending upon their location in public, project, or private databases. Versions can be created by checkout and checkin, derivation, and promotion. Only objects of classes declared to be versionable can be versioned. Both static and dynamic bindings are supported, and the user may define a number of contexts to specify default versions for a particular configuration of a complex object. A version derivation hierarchy of every versioned object is automatically maintained. The other features of the model include version change notification and version percolation. Some of the features of this model are directly supported in O++, and others can be built using the facilities provided in O++.

The IRIS version model [8] shares many of the features of the ORION version model. In addition, versioning is orthogonal to type, and a previously unversioned object can be versioned, but it has to go through a transformation procedure, which is not required in O++.

GemStone [14] and POSTGRES [29] allow versioning of objects to capture the history of database states. The version relationship of an object is constrained to be linear. The dynamic binding of version references and the automatic maintenance of temporal relationship between object versions make O++ suitable for developing historical databases.

Version control in ENCORE [19] is realized by introducing two new types: History-Bearing-Entity (HBE) and Version-Set. To create a versioned object, its corresponding type must inherit the properties defined by these two types. Properties defined by HBE include next-version and previous-version. The next-version property can be multivalued, thereby allowing the version graph to be a tree. Version-Set is used to collect all of the versions of an object. It provides an insert operation that allows new versions to be added at the end of a version sequence or as an alternative to an existing version.

The EXODUS storage manager [11] provides a general mechanism for implementing a variety of versioning schemes. When an object is updated with the versioning option on, a new version is created. Versions of large objects share common pages. However, at present E programmers [27] cannot make use of the versioning capability provided by the storage manager.

## 8. CONCLUDING REMARKS

The major consideration in designing the versioning facility in Ode was to introduce a few but semantically sound and powerful concepts that allow implementation of a wide variety of versioning paradigms, and to integrate them seamlessly in Ode's database programming language, O++. The following are some of the salient features of our versioning facilities:

- Object versioning is orthogonal to type, that is, versioning is an object property and not a type property. This feature allows versions of an object to be created without requiring any change to the type definition of that object. Thus, the decision to create a version of an object can be taken on as needed basis, rather at the time of initial database design. All persistent objects can have versions, and both versioned and non-versioned objects of the same type can be created. And different objects of the same type can have a different number of versions.
- Reference to an object can be bound statically to a specific version of the object or dynamically to whatever is its latest version. Static binding is useful for configuration objects that refer to specific versions of component objects. Dynamic references are useful for objects that must track latest versions of constituent objects. An object may contain both static and dynamic references to other objects.
- Both temporal as well as derived-from relationships between versions of an object are maintained automatically. Temporal relationship reflecting the creation history of the versions of an object is important for historical databases and for supporting time in databases. Derived-from relationship reflecting the derivation history of the versions of an object meets the requirements of the design environments in which several versions of a new design may be derived from the same design by making different changes to it. The derived-from relationship can also be used to store versions by storing their ''differences''.

The above facilities have been incorporated seamlessly in O++. Although we introduce few new language constructs, they are powerful enough to make O++ a suitable platform for implementing a variety of versioning paradigms and application-specific systems. Using these facilities, we found it was relatively straightforward to model a CAD database. This experience reinforced our belief that these versioning facilities provide the right functionality. The semantic clarity and feature minimality of our design also yielded a clean implementation.

We are currently building an Ode prototype. We are also working on a transaction model that exploits the versioning features in a novel way. We hope to report on these experiences in near future.

## 9. ACKNOWLEDGMENTS

We wish to thank Anoop Singhal for explaining to us the features of the DMS design system and Shaul Dar, Joel Richardson, and Jags Srinivasan for their comments.

## 10. APPENDIX: SPECIFICATIONS OF THE LIBRARIES

We give in this appendix important portions of specifications of the classes used to support versioning of persistent objects. We assume that the reader is familiar with C++ [ Stroustrup 1986 ].

**File:** `version.h`

```
class Version {
        /*  Constructors and destructors.  They are protected since we
        **  don't want users creating Version objects.  Version should
        **  be used only as a base class.
        */

protected:
                        Version(void); // Our constructor.

        virtual         ~Version(void); // Our destructor.

public:
        /*  Functions for navigating around the version set */

        Type*           prevVer(void);
                                // Return a pointer to the previous
                                // version of this object.  That is,
                                // the one that was created immediately
                                // before this one in time.  Returns
                                // NULL if this is the oldest version.

        Type*           nextVer(void);
                                // Return a pointer to the next version
                                // of this object.  That is, the one
                                // that was created immediately after
                                // this one in time.  Returns NULL if
                                // this is the newest version.

        Type*           parentVer(void);
                                // Return a pointer to the parent
                                // version of this object or NULL if
                                // this object is the root of the
                                // version tree.

        Type*           childVer(void);
                                // Return a pointer to the leftmost
                                // child version of this object or NULL
                                // if this object has no children.  The
                                // leftmost child is the first child
                                // created.

        Type*           prevSibVer(void);
                                // Return a pointer to the previous
                                // sibling version of this object or
                                // NULL if this is the leftmost (first)
                                // child of its parent.

        Type*           nextSibVer(void);
                                // Return a pointer to the next sibling
                                // version of this object or NULL if
                                // this is the rightmost (last) child
                                // of its parent.

        /*  Functions for creating and deleting versions */

        Type*           newVersion(void);
                                // Create a new version of this object.
                                // This object will be the parent of the
                                // new version object.

        void            delVersion(void);
                                // Delete this version of the object.
                                // You cannot delete the root version
                                // of the set.

        /*  Functions involving the VersionSet */
```

```
        VersionSet(Type)*
                        versionSet(void);
                                // Return a pointer to the VersionSet
                                // containing this object.

        /*  Our public data. */

public:
        static Objection rootVersion;
                                // Objection raised when an attempt is
                                // made to delete the root version of
                                // the set.
};
```

**File:** versset.h

```
class VersionSet {

        /*  Our constructors and destructors.  Make the constructors
        **  private since only Version should be creating a VersionSet.
        */

private:
                        VersionSet(void);
                                // Constructor use in our readObj
                                // function to construct an object
                                // without initialization since we
                                // are about to read it in.

                        VersionSet(Type* op);
                                // Our constructor.  Argument is pointer
                                // to the first (root) version of the
                                // object.
public:
        Type*           currentVer(void);
                                // Returns a pointer to the current
                                // (latest) version in the set.

        Type*           rootVer(void);
                                // Returns a pointer to the root of the
                                // tree of versions.
};
```

**File:** versiter.h

```
class VersionIter {
public:
                        VersionIter(Type* op);
                                // A constructor.  Initialize it to
                                // iterate over the version set
                                // containing the argument object.

                        VersionIter(VersionSet(Type)* sp);
                                // A constructor.  Initialize it to
                                // iterate over indicated version set.

        Type*           operator()(void);
                                // Get next version object in set.

        void            reset(void);
                                // Reset the iterator to start over at
                                // the beginning of the set.
};
```

**File:** versleafiter.h

```
class VersionLeafIter {
public:
                        VersionLeafIter(Type* op);
                                // A constructor.  Initialize it to
                                // iterate over the version set
                                // containing the argument object.

                        VersionLeafIter(VersionSet(Type)* sp);
                                // A constructor.  Initialize it to
                                // iterate over the indicated version
                                // set.

        Type*           operator()(void);
                                // Get the next version object in the
                                // set.

        void            reset(void);
                                // Reset the iterator to start over at
                                // the beginning of the set.
};
```

**File:** versptr.h

```
class VersionPtr {
public:
        /*  Our constructors.
        */

                        VersionPtr(void);
                                // Initialize to a NULL pointer.

                        VersionPtr(Type* op);
                                // Initialize to point to a particular
                                // object.

                        VersionPtr(VersionSet<Type>* sp);
                                // Initialize to point to a set.

                        VersionPtr(const VersionPtr<Type>& vp);
                                // The copy constructor.

        /*  Our assignment operators */

        VersionPtr(Type)& operator=(Type* op);
                                // Assign a pointer to a particular
                                // version.

        VersionPtr(Type)& operator=(VersionSet(Type)* sp);
                                // Assign a pointer to a version set.

        VersionPtr(Type)& operator=(const VersionPtr(Type)& vp);
                                // Assign a VersionPtr to another
                                // VersionPtr.

        /*  Other operators */

        Type&           operator*(void);
                                // Dereference the pointer.

        Type*           operator->(void);
                                // Member selection.

        /*  The type conversion  */

                        operator Type*(void);
                                // Return a pointer to an object.

        /*  Our public data.
        */

public:
        static Objection wrongSet;
                                // Objection raised when assigning an
                                // object pointer to a version pointer
                                // and the object is not in the set to
                                // which the version pointer refers.

private:
        /*  Our private data.
        */

        SafePersPtr(VersionSet(Type)) setP;
                                // A pointer to the version set we are
                                // pointing to.

        SafePersPtr(Type) objP;
                                // A pointer to the object we are
                                // pointing at.
};
```

**Remarks**

The class type of the object being versioned must observe the following:

1. The class type must have no constructors or must have a default constructor as one of its public constructors. A default constructor is one that takes no arguments.
2. The class type must have an `operator=` defined. This is used to initialize the new version object from its parent. The default assignment operator suffices.

# Object Versioning in Ode

*R. Agrawal*

IBM Almaden Research Center
San Jose, California 95120

*S. J. Buroff*

AT&T Bell Labs
Summit, New Jersey 07901

*N. H. Gehani*

AT&T Bell Labs
Murray Hill, New Jersey 07974

*D. Shasha*

New York University
New York, NY 10012

*ABSTRACT*

The design of the versioning facilities in the Ode object-oriented database system is based on a few powerful concepts that allow implementation of a wide variety of versioning paradigms. These facilities have been cleanly integrated into Ode's database programming language O++. Object versioning in Ode is orthogonal to type, that is, versioning is an object property and not a type property. Versions of an object can be created without requiring any change in the corresponding object type definition, all objects can be versioned, and different objects of the same type can have a different number of versions. Both dynamic and static bindings to version references are supported. Temporal as well as derived-from relationships between versions are maintained automatically.

This paper presents the versioning facilities in Ode, gives their formal semantics, illustrates their power by modeling a design database in production use, and discusses their implementation. Modeling the design database reinforces our belief that the versioning facilities provided in Ode offer the right functionality. The dynamic binding of version references and the automatic maintenance of temporal relationships between object versions also make Ode suitable for developing historical databases and for supporting time in databases.

**REFERENCES**

[1] H. D. Afsarmanesh, D. McLeod, D. Knapp and A. Parker, ''An Extensible Object-Oriented Approach to Databases for VLSI/CAD'', *Proc. of the 11th Int'l Conf. on Very Large Databases*, Stockholm, Sweden, Aug. 1985.

[2] R. Agrawal and N. H. Gehani, ''Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++'', *2nd Int'l Workshop on Database Programming Languages*, Portland, OR, June 1989.

[3] R. Agrawal and N. H. Gehani, ''Ode (Object Database and Environment): The Language and the Data Model'', *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, Portland, Oregon, May-June 1989, 36-45.

[4] R. Agrawal, N. H. Gehani and J. Srinivasan, ''OdeView: The Graphical Interface to Ode'', *Proc. ACM-SIGMOD 1990 Int'l Conf. on Management of Data*, 1990, 34-43.

[5] T. M. Atwood, ''An Object-Oriented DBMS for Design Support Applications'', *Proc. IEEE 1st Int'l Conf. Computer-Aided Technologies*, Montreal, Canada, Sept. 1985, 299-307.

[6] J. Banerjee, H. T. Chou, J. F. Garza, W. Kim, D. Woelk and N. Ballou, ‘‘Data Model Issues for Object-Oriented Applications’’, *ACM Trans. Office Information Systems 5*, 1 (Jan. 1987), 3-26.

[7] D. Batory and W. Kim, ‘‘Modeling Concepts for VLSI CAD Objects’’, *ACM Trans. Database Syst. 10*, 3 (Sept. 1985), .

[8] D. Beech and B. Mahbod, ‘‘Generalized Version Control in an Object-Oriented Database’’, *Proc. IEEE 4th Int’l Conf. Data Engineering*, Los Angeles, California, Feb. 1988, 14-22.

[9] P. A. Bernstein, ‘‘Database Support for Software Engineering’’, *Proc. 9th IEEE Int’l Conf. Software Eng.*, March 1987, 166-178.

[10] S. J. Buroff and D. Shasha, ‘‘A Persistence Library for C++’’, AT&T Bell Laboratories, Murray Hill, New Jersey, 1989.

[11] M. J. Carey, D. J. DeWitt, J. E. Richardson and E. J. Shekita, ‘‘Object and File Management in the EXODUS Extensible Database System’’, *Proc. 12th Int’l Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986, 91-100.

[12] M. J. Carey, D. J. DeWitt and S. L. Vandenberg, ‘‘A Data Model and Query Language for EXODUS’’, *Proc. ACM-SIGMOD 1988 Int’l Conf. on Management of Data*, Chicago, Illinois, June 1988, 413-423.

[13] H. T. Chou and W. Kim, ‘‘A Unifying Framework for Version Control in a CAD Environment’’, *Proc. 12th Int’l Conf. on Very Large Databases*, Kyoto, Japan, Aug. 1986, 336-344.

[14] G. Copeland and D. Maier, ‘‘Making Smalltalk a Database System’’, *Proceedings of the 1984 ACM SIGMOD Intl. Conf. on Management of Data*, Boston, Massachusetts, June 1984, 316-325.

[15] J. Diederich and J. Milton, ‘‘ODDESSY: An Object-Oriented Database Design System’’, *Proc. IEEE 3rd Int’l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 235-244.

[16] K. Dittrich and R. Lorie, ‘‘Version Support for Engineering Database Systems’’, Rep. RJ4769, IBM Research Lab., San Jose, California, July 1985.

[17] K. R. Dittrich, W. Gotthard and P. C. Lockemann, ‘‘DAMOKLES – A Database System for Software Engineering Environments’’, LNCS 244, 1987.

[18] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[19] M. F. Hornick and S. B. Zdonik, ‘‘A Shared Segmented Memory System for an Object-Oriented Database’’, *ACM Trans. Office Information Systems 5*, 1 (Jan. 1987), 70-95.

[20] S. E. Hudson and R. King, ‘‘Object-Oriented Database Support for Software Environments’’, *Proc. ACM-SIGMOD 1987 Int’l Conf. on Management of Data*, San Fransisco, California, May 1987, 491-503.

[21] R. Katz, E. Chang and E. Bhateja, ‘‘Version Modeling Concepts for Computer-Aided Design Databases’’, *Proc. ACM-SIGMOD 1986 Int’l Conf. on Management of Data*, Washington D.C., May 1986.

[22] S. N. Khoshafian and G. P. Copeland, ‘‘Object Identity’’, *Proc. OOPSLA ’86*, Portland, Oregon, Sept. 1986, 406-416.

[23] W. Kim, E. Bertino and J. Garza, ‘‘Composite Objects Revisited’’, *Proc. ACM-SIGMOD 1989 Int’l Conf. Management of Data*, Portland, Oregon, May-June 1989.

[24] P. Klahold, G. Schlageter and W. Wilkes, ‘‘A General Model for Version Management in Databases’’, *Proc. 12th Int’l Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986, 319-327.

[25] D. McLeod, K. Narayanaswamy and K. B. Rao, ‘‘An Approach to Information Management for CAD/VLSI Applications’’, *Proc. Databases for Engineering Applications, ACM-SIGMOD Database Week 1983*, San Jose, California, May 1983, 115-121.

[26] Z. Mehmood, D. Singer, A. Singhal and K. W. Wu, ‘‘A Data Management System for CAD’’, *Proc. of ICCAD Conference*, 1987.

[27] J. E. Richardson and M. J. Carey, ‘‘Persistence in the E Language: Issues and Implementation’’, *Software—Practice & Experience 19*, 12 (Dec. 1989), 1115-1150.

[28] M. Rochkind, ‘‘The Source Code Control System’’, *IEEE Trans. Software Eng. SE-1*, 4 (Dec. 1975), 364-370.

[29] L. A. Rowe and M. R. Stonebraker, ‘‘The POSTGRES Data Model’’, *Proc. 13th Int’l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, 83-96.

[30] R. Snodgrass and I. Ahn, ‘‘A Taxonomy of Time in Databases’’, *Proc. ACM-SIGMOD 1985 Int’l Conf. on Management of Data*, Austin, Texas, May 1985.

[31] B. Stroustrup, ‘‘Multiple Inheritance for C++’’, *Proc. European UNIX User’s Group*, Helsinki, May 1987, 189-208.

[32] W. Tichy, ‘‘RCS: A System for Version Control’’, *Software Practice and Experience 15*, 7 (July 1986), 637-654.

[33] W. Tichy, ‘‘Tools for Software Configuration Management’’, *Int’l Workshop Software Version and Configuration Control*, Grassau, FRG, Jan. 1988.

[34] S. B. Zdonik, ‘‘Version Management in an Object-Oriented Database’’, *Proc. Int’l Workshop on Advanced Programming Environments*, Trondheim, Norway, June 1986.