VLDB Journal, 3, 77-106 (1994), Peter Scheuermann, Editor

Versioning and Configuration Management in an Object-Oriented Data Model

Edward Sciore

Received July 29, 1991; revised version received June 11, 1992; accepted June 25, 1993.

Abstract. Many database applications require the storage and manipulation of different versions of data objects. To satisfy the diverse needs of these applications, current database systems support versioning at a very low level. This article demonstrates that application-independent versioning can be supported at a significantly higher level. In particular, we extend the EXTRA data model and EXCESS query language so that configurations can be specified conceptually and non-procedurally. We also show how version sets can be viewed multidimensionally, thereby allowing configurations to be expressed at a higher level of abstraction. The resulting model integrates and generalizes ideas in CAD systems, CASE systems, and temporal databases.

Key Words. EXTRA/EXCESS data models, query language, generic and specific references, semantically based configuration specifications.

1. Introduction

In an object-oriented database system, the objects model entities in the real world. Often it is useful for a database to store information about different aspects of an entity; this information is stored as *versions* of the corresponding object. There are two broad version categories: *system-level* versions and *user-level* versions. System-level versions are created and maintained by the database system. Such versions are used for concurrency control, transaction support (Agrawal and Jagadish, 1989), and redundancy in distributed databases. In contrast, user-level versions are created by applications for specific purposes. Examples of such versions include alternative designs for the object, previous states of the object, and so on.

This article is concerned exclusively with user-level versions. Our contention is that such versions are different aspects of the same conceptual idea, and should

Edward Sciore, Ph.D., is Associate Professor, Computer Science Department, Boston College, Chestnut Hill, MA 02167.

be treated uniformly in a database system. Users should be able to access any subset of versions of an object or to choose a version based on specified properties; users should also be able to configure an object based on specified properties of its components. System-level versions do not fit into this framework, because they have no semantic meaning and are often invisible to user applications. Consequently, in the rest of this article we refer to user-level versions simply as *versions*.

There is a substantial literature on versioning. Unfortunately, most of this research is limited in scope, being focused on a specific application. There are three application domains which have seen a lot of work: historical databases, CASE systems, and CAD databases. We discuss each of them in turn.

A historical database is one in which the information about the entities in the database is a function of time. For example, a database may store the history of its employees: their previous salaries, job titles, awards, and so on. A historical database system should be able to retrieve the state of an employee as of a specified time, or to find the instances when an employee's state satisfied certain properties. Research into historical databases has centered primarily on the relational model (Tansel, 1986; Clifford and Croker, 1987; Snodgrass, 1987; Gadia, 1988), although some work has been done for object-oriented systems (Copeland and Maier, 1984; Caruso and Sciore, 1988; Rose and Segev, 1991; Wu and Dayal, 1992).

Computer-Aided Software Engineering (CASE) systems comprise the second area of versioning research. CASE systems support the development and maintenance of software. A CASE database typically contains information about program modules and their relationships. A module might have several versions, corresponding to previous releases of the module or alternative implementations of it. CASE systems must provide the support necessary to allow a user to configure the modules of a program consistently; examples of such systems are Adele (Belkhatir and Estublier, 1986), DSEE (Leblang and Chase, 1984), Gypsy (Cohen et al., 1988), and Shape (Mahler and Lampen, 1988).

CASE systems typically are language tools. They are not based on an explicit data model. They are developed specifically for software databases, and their configuration languages make sense only for that application. There have been efforts to apply database techniques to CASE systems (Beech and Mahbod, 1988; Hudson and King, 1988), but the treatments of versioning in these systems are much poorer than in specialized CASE systems.

The third area in which versioning is prominent is computer-aided design (CAD). CAD systems support the design of engineering objects. As an object is being designed, it might be revised several times and have different alternative versions created for it. A designer might want to compare the properties of two different versions, or to check a proposed version against a set of design rules. The goal of research in this area has been to develop a general, all-purpose mechanism which can support the production and manipulation of design versions. Early work was based on files (Katz and Lehman, 1984; Katz et al., 1986) or relational systems (Batory and Kim, 1985; Dittrich and Lorie, 1988), but current research is almost

exclusively based on object-oriented database systems (Atwood, 1985; Chou and Kim, 1986, 1988; Klahold et al., 1986; Banerjee et al., 1987; Biliris, 1989; Kim et al., 1989.)

Research on CAD databases has focused on understanding the systems-level requirements of versioning. In particular, questions regarding the support of long transactions and control access to versions have received substantial attention. However, the data modeling and user interface issues have barely been touched. In current systems, for example, the only way to find a version of an object with a particular property is by navigating its version hierarchy, and the language constructs for specifying configurations are primitive in comparison with those of CASE systems.

Each of these three areas has solved different aspects of the versioning problem. However, it is not obvious how these different solutions can be combined, because the assumptions made in each case are incompatible. The many proposals for CAD versioning systems were shown to be very similar, and a unified terminology was given (Katz, 1990). However, this unification extends only partially to CASE systems and not at all to historical databases. In fact, Katz concluded with the statement that the problem of unifying all kinds of versioning is both important and challenging.

We attacked this problem in two previous articles (Sciore, 1991*a*), arguing that there is no fundamental incompatibility among the different kinds of versioning. In the first article, we showed how *annotations* could be used to model versioning in CAD, CASE, and historical databases. The approach of that article was not completely satisfactory, however, because the resulting data model was still low-level and the query language was procedural. In the second article, we showed how the different kinds of versioning result from differences in their level of abstraction, and indicated how a non-procedural query language could be possible. In this article we solidify these ideas and extend them to a concrete data model, namely the EXTRA object-oriented model of Carey et al. (1988).

This article is organized as follows. Section 2 reviews the EXTRA model and EXCESS query language, and shows how to implement the standard CAD versioning ideas in it. Section 3 introduces an extension of EXTRA called EXTRA-V and the EXCESS-V query language. Section 4 examines the notion of *frozen configurations*, and shows how this feature corresponds to views in EXTRA-V. In Section 5 we examine the semantics of versioning in some common applications, and show in each case how the version set of any object can be viewed as a multidimensional space. Section 6 presents our conclusions and some discussion of future research areas.

2. Object-Oriented Versioning

In this section we describe the basics of object-oriented versioning. Because there is no standard object-oriented data model, we have chosen the EXTRA data model (Carey et al., 1988) (as extended in the Pegasus system; Biliris, 1990) as the basis

of our study. EXTRA's simplicity and conceptual elegance allow us to focus on versioning issues and ignore aspects of the data model unrelated to versioning. However, the results of this article should be applicable to other, more "complete" object-oriented models.

2.1 Objects and Types

An *object* is defined to be an instance of a given *type*. A type defines a set of *attributes* for each of its instances, and a set of *operations* on these instances. The set of attributes and operations is called the *scheme* of the type. Each attribute of an object may contain either a *value* or a *reference* to another object. Reference attributes are specified using the *ref* keyword.[1]

The set of types is organized into a *type hierarchy*. If T_2 is a subtype of T_1 in the hierarchy, then T_2 *inherits* the scheme of T_1; that is, the attributes and operations of T_1 can be accessed by instances of T_2 as if they were defined locally. There are two ways in which inheritance can occur: *refinement* and *extension* (Biliris, 1990). Refinement models the common notion of ISA relationship. Objects of type T_2 are "special cases" of T_1-objects, and thus can be used whenever objects of type T_1 are expected. Extension, on the other hand, is related to the idea of prototypes (Sciore, 1991*a*). Each T_2-object has an associated T_1-object; references to attributes or operations that are not defined in T_2 are *delegated* to its associated T_1-object.

Typically, refinement is implemented by including the scheme of T_1 in each T_2-object, and extension is implemented by storing a pointer to a T_1-object in each T_2-object. In general, a type can have more than one parent in the type hierarchy, and each parent/child relationship can be either by refinement or extension. Implementation details and a proposal for resolving name conflicts were presented by Biliris (1990).

2.2 Versions

The notion of versioning can be modeled using pairs of types, each pair consisting of a *generic type* and a *version type*. Each versioned entity has a single associated generic object and zero or more associated version objects. A generic object contains the information which is common to all of its versions. The versions of a generic object all have the same scheme, so they differ only in the values for their attributes. These different attributes reflect the different *design choices* that caused the version to be created. It is useful for a version object to be able to access the values in its associated generic object directly (Biliris, 1990). This property is modeled by declaring the version type to be an extension of the generic type.

---

1. Actually, there are different forms of reference, based on questions of ownership and sharability. The keywords *own ref* are used for this purpose by Carey et al. (1988), and *exclusive ref* and *dependent ref* are used by Biliris (1990). These issues have already been addressed with respect to versioning (Kim et al., 1989), and so for simplicity we ignore them in this article.

The schema declaration of Figure 1 illustrates the above concepts. Each instance of *Bicycle* models a CAD design project, and has an associated project name, client and due date. The type *BicycleVersion* models versions of a given bicycle design, and contains attributes whose values may differ in different versions. These attributes include the style, number of speeds, frame, and so on. Some attributes (such as *style*) contain primitive values, whereas others (such as *frame*) contain references to other objects. The attribute *designDate* records the date when a version was added to the database, and the attribute *derivedFrom* references the version (if any) that was used to create it.

The semantic connection between these types is modeled by the attributes *versions* and *defaultVersion* in *Bicycle*, and *generic* in *BicycleVersion*. The attribute *generic* associates a version object with its generic object.[2] The attribute *versions* associates a generic object with all of its versions, and the attribute *defaultVersion* associates a generic object with a reference to a single one of these versions. This default version is used in converting generic references to specific ones, as we shall see in the next section. The specification of which version is to be the default is usually based on semantic considerations, such as being the most recent or the current best. We therefore postpone our discussion of default selection until Section 5.3, where we examine the semantics of versioning in more detail.

The declarations of the type-pairs *Frame/FrameVersion* and *Wheel/WheelVersion* are similar to that of *Bicycle/BicycleVersion*. Note that *material* is an attribute of *Frame*, and is intended as a key. That is, the version set of a *Frame*-object consists of all frame designs corresponding to a particular material. Unlike in relational databases, a type definition does not automatically create a corresponding collection. Instead, these collections are created explicitly. In Figure 1 the collections *bikes* and *frames* hold references to objects of type *Bicycle* and *Frame*, respectively. Note that there is no defined collection of *Wheel*-objects.

Each generic and version object in a database should be uniquely identifiable somehow. In this article we assume that each generic object has the attribute *genericId* and each version object has the attribute *versionId*. Version identifiers need only be unique within a version set, so that the combination *(genericId, versionId)* uniquely identifies any version object.

2.3 Data Retrieval

The non-procedural query language for the EXTRA model is called EXCESS. EXCESS is an extension of QUEL in the spirit of GEM (Zaniolo, 1983), DAPLEX (Shipman, 1981), and POSTGRES (Rowe and Stonebraker, 1987). We illustrate the EXTRA syntax by means of example; more formal definitions were made by Carey et al. (1988). The following EXCESS query for Figure 1 retrieves the costs

2. The keyword *proto ref* is taken from Biliris (1990), and indicates the attribute to be used for implementing extension inheritance.

Figure 1. An EXTRA scheme

```
define type Bicycle:
        ( projectName: char[10],
          client: char[20],
          dueDate: Date,
          versions: {ref BicycleVersion},
          defaultVersion: ref BicycleVersion
        )
define type BicycleVersion:
        ( style: char[10],
          numSpeeds: int4,
          frame: ref Frame,
          frontWheel: ref Wheel,
          rearWheel: ref Wheel,
          cost: int4,
          designDate: date,
          derivedFrom: ref BicycleVersion
          generic: proto ref Bicycle,
        ) extends Bicycle
define type Frame:
        ( material: char[15]
          versions: {ref FrameVersion},
          defaultVersion: ref FrameVersion
        )
define type FrameVersion:
        ( color: char[10],
          designDate: date,
          derivedFrom: ref FrameVersion
          generic: proto ref Frame,
        ) extends Frame
define type Wheel:
        ( versions: {ref WheelVersion},
          defaultVersion: ref WheelVersion
        )
define type WheelVersion:
        ( material: char[15],
          size: int4
          designDate: date,
          derivedFrom: ref WheelVersion
          generic: proto ref Wheel,
        ) extends Wheel
create bikes: {ref Bicycle};
create frames: {ref Frame};
```

of all versions of the BMX design project:

```
retrieve V.cost
from B in bikes, V in B.versions
where B.projectName ="BMX"
```

The *from* clause defines a set of bindings for each listed variable. Each possible combination of variable bindings is called a *configuration* of the query. One tuple is returned for each configuration that satisfies the *where* clause.

EXCESS supports the standard aggregation operators. The following query for Figure 1 illustrates the syntax, retrieving the project name of the lowest-cost ten-speed bicycle design.

```
retrieve B.projectName
from B in bikes, V in B.versions
where V.numSpeeds = 10 and
      V.cost = min(retrieve V2.cost
                   from B2 in bikes, V2 in B2.versions
                   where V2.numSpeeds = V.numSpeeds)
```

Because each versioned entity is modeled by objects from two different types, there are two different ways to refer to the entity. A *generic reference* refers to its generic object, and a *specific reference* refers to one of its version objects. In Figure 1, the attributes *frame, frontWheel,* and *rearWheel* of *BicycleVersion* are generic references, whereas the attribute *derivedFrom* is a specific reference. Generic references have a special interpretation—they refer to a default version of the design object, not the design object itself. The attribute *defaultVersion* is used to transform the generic reference into the desired specific reference. For example, the following query retrieves the default color of the default version of those bicycles designed for client "Schwinn":

```
retrieve F.defaultVersion.color
from B in bikes, F in B.defaultVersion.frame
where B.client = "Schwinn"
```

2.4 Updates

There are many ways to create, destroy, and modify objects in EXCESS. In this article we discuss the operations most relevant to database applications, namely the operations to insert, delete, and modify objects in a collection.

A new object is added to a collection with the *copy* command. The following commands on the scheme of Figure 1 create a new *Bicycle* object having a single version:

copy to B (projectName = "BMX",
client = "Schwinn", dueDate = 9/9/99)
from B *in* bikes
copy to B.versions (style = "racing", numSpeeds = 12, frame = F,
..., derivedFrom = nil, generic = B)
from B *in* bikes, F *in* frames
where B.projectName = "BMX" *and* F.material = "alloy"

Additional versions can be added to the version set of *Bikes* similarly.

Versions are modified using the *replace* command. For example, the following command increases the cost of all racing bicycle versions of the BMX project:

replace V (cost = V.cost + 100)
from B *in* bikes, V *in* B.versions
where B.projectName = "BMX" *and* V.style = "racing"

Versions are deleted using the *delete* command. The following command deletes all sufficiently old BMX versions:

delete V
from B *in* bikes, V *in* B.versions
where B.projectName = "BMX" *and* V.designDate < 12/31/89

3. A Versioning Data Model

Section 2 showed how a versioned entity can be implemented using two types: its generic type and its version type. Although versioned information can be stored in this way, it is important to note that the model knows nothing about versioning. That is, the conceptual notion of a "versioned entity" is lost, and conceptual operations on versioned entities must be translated into operations on the underlying implementation.

Previous research has recognized this problem, but has focused primarily on facilitating the creation and manipulation of versions. For example, ORION provides a *new-version* operation, which derives a new version from the current default, checks it out of the project database, and initializes its variables appropriately. Such operations are useful, but are only part of what is needed. In this section we show how a fully conceptual view of versioning can be built into the data model, and how it can be mapped automatically into the structures of Section 2.

3.1 Data Definition

Standard data models such as EXTRA are what we call *non-versioning*. In a non-versioning model, attributes that are modified lose their previous values. To avoid

losing information, a user must explicitly create new version objects containing the new values.

In contrast, an object in a *versioning model* can have two kinds of attributes: *versioned attributes* and *unversioned attributes*. Unversioned attributes are version-independent. That is, changes to unversioned attributes will be done in place, and all versions of the object see the same values for these attributes. Values for versioned attributes are stored with the versions of the object. Changes to any of these attributes cause a new version to be created.

We now define a versioning model called EXTRA-V. Syntactically, EXTRA-V is identical to EXTRA, except that it contains additional keywords. One of these keywords is *versioned*. Attributes appearing after this keyword in a type declaration are versioned, and those appearing before the keyword are unversioned. If the keyword does not appear in a type declaration, then all attributes are unversioned. For clarity in comparison with EXTRA, we call types declared in EXTRA-V *conceptual types*, and their instances *conceptual objects*.

Figure 2 shows how the scheme of Figure 1 can be defined in EXTRA-V. Each conceptual *Bicycle*-object corresponds to a different design project. The attributes *projectName* and *dueDate* are unversioned because they contain information about the design project as a whole; that is, if the due date of a *Bicycle*-object is changed, then all of its versions should see the change. Each *Bicycle*-object can have several versions, each one corresponding to a possible design. Each version of the conceptual object has its own values of the versioned attributes.

EXTRA-V schemes can be mapped to EXTRA schemes, using the type-pair strategy of Section 2. In particular, each conceptual EXTRA-V type has a corresponding EXTRA generic type and version type. The generic type has the same name as the conceptual type name, and contains all unversioned attributes, as well as the attributes *genericId, versions* and *defaultVersion*. The version type name is the concatenation of the string "Version" to the conceptual type name; this type contains the versioned attributes as well as the attributes *versionId* and *generic*. Note how this technique maps the declarations of Figure 2 to those of Figure 1.

Subtyping in EXTRA-V is handled as follows. Let T_1 and T_2 be two conceptual types such that T_2 is a subtype of T_1. Then, in the mapping, the generic type of T_2 will be a subtype of the generic type of T_1 and the version type of T_2 will be a subtype of the version type of T_1. We note two things about this definition. First, a type can inherit versioned attributes, even though it defines no versioned attributes itself. For example, suppose in Figure 2 that we declare the type *childBicycle* to be a subtype of *Bicycle*, having only the unversioned attribute *intendedAge*. Then this type inherits the versioned attributes of *Bicycle*. Second, because attributes in a supertype can be overridden by redefining them in a subtype, an unversioned attribute can be declared to be versioned in a subtype (or vice versa).

Figure 2. A conceptual scheme

```
define type Bicycle:
        ( projectName: char[10],
          client: char[20],
          dueDate: Date
        versioned
          style: char[10],
          numSpeeds: int4,
          frame: ref Frame,
          frontWheel: ref Wheel,
          rearWheel: ref Wheel,
          cost: int4,
          designDate: Date,
          derivedFrom: specific ref Bicycle
        )
define type Frame:
        ( material: char[15]
        versioned
          color: char[10],
          designDate: Date,
          derivedFrom: specific ref Frame
        )
define type Wheel:
        ( versioned
          material: char[15],
          size: int4,
          designDate: Date,
          derivedFrom: specific ref Wheel
        )
create bikes: {ref Bicycle};
create frames: {ref Frame};
```

Our mapping from EXTRA-V schemes to EXTRA schemes requires that each conceptual type map to both a generic and a version type. In particular, if a conceptual type T has no versioned attributes then all versions of an instance of T will have exactly the same information, namely the values of the unversioned attributes. Consequently, EXTRA-V is an extension of EXTRA. Types that do not involve versioning can be declared in EXTRA-V exactly the same way as in EXTRA, and will have exactly the same semantics.[3]

3. Our mapping from EXTRA-V schemes to EXTRA defines the semantics of EXTRA-V schemes. It also simplifies many of the mappings in the rest of this article. A real implementation of EXTRA-V could, of course, employ more efficient mappings.

Figure 3. Historical database scheme

```
define type Person:
        ( name: char[20]
        versioned
           address: char[20],
           occurredAt: Date
        )
define type Employee:
        ( employeeNum: int4
        versioned
           salary: int4,
           position: char[10],
           worksFor: ref Company
        ) refines Person
define type Company:
        ( name: char[10],
           industry: char[10]
        versioned
           sales: int4,
           ceo: ref Employee,
           occurredAt: Date
        )
create employees: {ref Employee};
create companies: {ref Company};
```

Our bicycle-design scheme is an example of a CAD application. Figure 3 illustrates an historical database in EXTRA-V. This scheme has three conceptual types: *Person, Employee,* and *Company*. A version of a conceptual object denotes a previous or current state of the object. Each time a versioned attribute changes, a new version is created corresponding to the new state. The attribute *occurredAt* records the time at which the change logically took place.

3.2 Generic and Specific References

Another keyword unique to EXTRA-V is *specific ref*, which is needed to distinguish generic references from specific ones. Conceptually, a generic reference points to an object and all of its versions, whereas a specific reference points to a single version only. In EXTRA-V, the keyword *ref* denotes a generic reference and the keyword *specific ref* denotes a specific reference.

Generic references are often more appropriate than specific references. For example, consider the generic references *worksFor* and *ceo* in Figure 3. Each *Employee* version contains the information about an employee during some time interval. If the employee worked for some company (say, IBM) during that interval, then *worksFor* would refer to the complete history of IBM, not just the value of IBM during that interval. There are two advantages to this approach. First, a

new version of the employee does not have to be created each time a new version of the company is created, which would be the case if *worksFor* were a specific reference. Second, it is possible to bind the generic reference to any version of the company, so that information about an employee's current company at a previous time (or current information about an employee's previous company) can be easily requested.

Note in Figure 2 that *Frame* is one of the versioned attributes of *Bicycle*. Consequently, different versions of a *Bicycle*-object can have frames made of different materials. Had this attribute been declared as unversioned, then two versions of a bicycle project could not be configured with frames made of different materials.

3.3 Data Manipulation

We now define the language EXCESS-V, which extends EXCESS to versioned databases. Our principle is that versioning should be as transparent as possible, so that users can interact with versioned data as if it were unversioned. That is, we want a standard EXCESS query on a versioned scheme to behave as if each object had only one version. Which version of each referenced object should be used? In an unversioned database, the most recent "version" of the object is always used, because it is the only one available. In a versioned database we use the default version when answering queries.

The binding of default versions to variables occurs in the *from* clause. In EXCESS, the expression *"X in S"* binds variable *X* to each member of the set *S*. We modify this definition in EXCESS-V so that *X* is bound to the default version of each member of *S*. Note that if the objects in *S* are unversioned, then EXCESS-V behaves exactly like EXCESS; thus EXCESS-V is an extension of EXCESS.

For example, consider a database for Figure 3 where the default version of each object is defined to be the most current one. Then the following query retrieves the name of all employees who currently make more than their current CEO currently does.

retrieve E1.name
from E1 *in* employees, C *in* E1.worksFor, E2 *in* C.ceo
where E1.salary > E2.salary

EXCESS (as well as many other languages) provides an extended dot notation to reduce the number of explicit variable declarations in a query. Using this notation, the above query can be written equivalently as follows:

retrieve E1.name
from E1 *in* employees
where E1.salary > E1.worksFor.ceo.salary

The system interprets extended dot expressions by generating implicit variable declarations in the *from* clause of the query, using the keyword *in*. Consequently,

using such expressions on versioned databases causes the default versions of the generic references to be chosen.

We use the new keyword *inall* in EXCESS-V to access non-default versions of objects. The expression *"X inall S"* binds *X* to *all* versions of each member of *S*. For example, the following query using Figure 3 retrieves the names of all current salespersons and their highest salaries:

```
retrieve E1.name, E1.salary
from   E1 in employees
where  E1.position = "salesperson" and
       E1.salary = max(retrieve E2.salary
                       from E2 inall employees
                       where E2.name = E1.name)
```

The expression *"E2.name = E1.name"* in the aggregation portion of the above query guarantees that *E2* will be bound only to versions belonging to the same version set as *E1*. The expression *"E2.genericId = E1.genericId"* would have had the same effect.

Given an EXCESS-V query *Q*, there is a straightforward translation of *Q* into an equivalent EXCESS query Q'. Each expression *"X in S"* of *Q* is translated into *"X in S.default"* in Q', and each expression *"X inall S"* of *Q* is translated into *"X in S.versions"* in Q'.

3.4 Updates

In Section 2.4 we described the EXCESS update commands *copy*, *delete*, and *replace*. As with retrievals, the increased understanding of version semantics in EXTRA-V also allows update operations to be much simpler.

The *copy* command creates a new object and inserts it into a collection. In our versioning model, each conceptual object has both a generic and versioned part. Thus, this command is interpreted in EXCESS-V as creating a new generic object having a single version. For example, in Figure 2 the following EXCESS-V command creates a new *Bicycle*-object:

```
copy to B (projectName="BMX2", client="Fuji",
           dueDate=6/6/96, style = "racing",
           ..., designDate = 7/13/92 )
from B in bikes
```

The *delete* command removes an object from a collection, including all of its versions. For example, in Figure 2 the command:

```
delete B
from B in bikes
where B.style = "mountain"
```

deletes any project in *bikes* whose default version is a mountain bike. Had the keyword *in* been *inall* in this command, then all projects having any mountain bike version would be deleted.

The above behavior of delete is necessary for the command to be an extension of deletion on unversioned databases. In an unversioned database, executing a command such as "Delete the BMX project" would cause all information about the project to disappear (or at least cause it to be unavailable). Similar behavior is necessary in EXCESS-V. Thus, for example, it would not be appropriate for the above command to delete only the current default version of the project, because other versions would then become visible.

The notion of deleting individual versions of a conceptual object is somewhat out of place in a versioning model, because versions are usually considered to be archival information. However, there are situations when version deletion is necessary, and we introduce the command *delete version* to this end. For example, the command

delete version B
from B *inall* bikes
where B.style = "mountain"

deletes all mountain-bike versions from all projects.

The *replace* command is used in EXCESS to modify objects. We extend this meaning in EXCESS-V as follows. Updates to the unversioned attributes of an object are done in place and are seen by all of its versions. Updates to versioned attributes, however, cause the creation of a new version. For example, in Figure 3 the following command conceptually changes the salary and job of Joe Smith:

replace E (salary = 30, worksFor = C, occurredAt = 3/3/93)
from E *in* employees, C *in* companies
where C.name = "IBM" *and* E.name = "Joe Smith"

Formally, this command selects the object in *employees* corresponding to Mr. Smith, and chooses a default version for it. It then creates a new version derived from this default; that is, the new version contains the specified values for *salary, worksFor* and *occurredAt*, and the values of the default version for the other attributes.

For another example, the following command using Figure 2 increases the cost of all racing bike versions by $100:

replace B (cost = B.cost + 100)
from B *inall* bikes
where B.style = "racing"

Formally, this command is evaluated as follows. First, there is a binding of variable *B* for each racing bike version of any project in *bikes*. Second, a new version is created for each binding, whose attribute values are derived from the corresponding version. Finally, the *cost* attribute of these new versions are incremented by 100.

Occasionally it is necessary for a user to modify a versioned attribute in place, without creating a new version. Consequently, we introduce the *replace version* command. The meaning of this command is the same as *replace*, except that new versions are not created.

3.5 Relationship to Configuration Management

In Section 2 we defined a query as constructing a configuration of the database objects. Our use of the term *configuration* is meant to suggest that EXCESS-V is a configuration management language when applied to versioned databases. That is, the body of an EXCESS-V query configures a specified set of objects by determining which versions are associated. This view is quite different from previous approaches to configuration management in CAD and CASE applications. In these systems, a configuration is specified by giving a list of version objects. Whenever a generic reference needs to be resolved, the appropriate version object from the specified list is used.

The chief difference between these other systems is the way in which this list of version objects is specified. In ORION (Banerjee et al., 1987), the list is explicitly given, and each version is specified by its unique version number. Gypsy also requires an explicit list of versions (Cohen et al., 1988), but a version can be specified in several ways: by its version number, by giving a predicate on its attributes, or by invoking system-defined rules (such as "choose the most recent version"). In Shape (Mahler and Lampen, 1988), the list is specified implicitly using a sequence of design rules. Each design rule defines a predicate, and the first version of a generic object satisfying a design rule is chosen. Finally, Adele (Belkhatir and Estublier, 1986) allows both implicit and explicit specification of versions, using a specification language that allows complex predicates to be defined.

There are several problem areas caused by these approaches to configuration management. The first area concerns expressive power. The ways in which selection predicates are expressed in these languages are *ad hoc*, and are less expressive than EXCESS-V. For example, EXCESS-V allows arbitrary Boolean predicates (as opposed to Gypsy, which supports positive conjunctions only), and nested queries involving grouping and aggregation. Thus the query "Find the lowest-cost configuration of each style of bicycle, and choose the configuration having the largest number of speeds" is impossible to express in the above systems.

The second area concerns the fact that constraints on configurations cannot involve multiple objects. This liability stems from the use of version-lists to specify configurations. Each version object is specified independently, so there is no way to make coordinated choices.[4] For example, the following EXCESS-V query selects all bicycle configurations so that the front and rear wheels have the same size:

4. Except for synchronization, but that is another issue. See Section 5.3.

retrieve B
from B *inall* bikes, FW *inall* B.frontWheel,
RW *inall* B.rearWheel
where FW.size = RW.size

Again, such a configuration specification is impossible to write in CAD and CASE systems.

The third area also stems from the use of version-lists, and from the fact that all generic references to an object must be resolved in the same way. In our CAD example application, this restriction implies that there is no way to define a configuration for *bikes* in which the front and rear wheels are different versions of the same *Wheel*-object. In CASE applications, this restriction implies that programs cannot use more than one implementation of a module generically. So for example, a program that needs different kinds of sorting routines cannot use generic references to the *sort* module, but must use specific references to the individual versions—which, of course, means that the program will not be able to automatically take advantage of new versions of the sorting routines.

Finally, all configuration-management systems known to us return a single configuration only. That is, there is no construct analogous to our *inall* keyword that allows a query to return multiple configurations satisfying a given condition.

4. Views and Freezing

One feature missing in EXTRA is the support for view definitions. Views in object-oriented systems are more complex than in the relational model, primarily because of problems with the class hierarchy and object identity. A discussion of these issues and a proposed view definition language for O_2 appear in Abiteboul and Bonner (1991). For the purposes of this article, we only need views that define subsets of objects from a collection. We adopt the following syntax:

define view tenspeeds *as*
retrieve B
from B *in* bikes
where B.numSpeeds = 10

This definition defines a virtual collection named *tenspeeds*, containing references to those objects in *bikes* whose default version has ten speeds. Because *tenspeeds* is defined as a view, its contents may change as new versions are added or other default mechanisms are used.

Queries involving views can be evaluated by query substitution, so that no new objects need be materialized. That is, the query

retrieve T.cost
from T *in* tenspeeds
where T.frame.color = "red"

is equivalent to

retrieve B.cost
from B *in* bikes
where B.frame.color = "red" *and* B.numSpeeds = 10

View definitions are important for versioning systems, because they can be used to *freeze* objects in particular configurations. Traditionally, a frozen object has a single configuration that never changes. Frozen objects allow important configurations to be saved for future reference, such as a public release of a program, a proposed bicycle design sent to a client, or the state of a particular company in 1984. For example, in the following view definition

define view proposedBike *as*
retrieve B
from B *inall* bicycles, F *inall* B.frame,
RW *inall* B.rearWheel, FW *inall* B.frontWheel
where B.versionNum = 3 *and*
F.versionNum = 1 *and*
RW.versionNum = 20 *and*
FW.versionNum = 20 *and*
B.projectName = "BMX"

the single virtual object in *proposedBike* has a single frozen configuration.

In other configuration management systems, freezing is performed by converting all generic references reachable from an object into specific references, and consequently is an expensive operation. View definitions eliminate the need to materialize the frozen configuration. Instead, the configuration is constructed when it is needed.

An important additional benefit of view definitions is that objects can be *partially frozen*. A partially frozen object is constrained, but the constraint may not determine a unique configuration or the specified configuration may change over time. For example, in a CASE application we might want to freeze a module according to a specific time, but allow other considerations (such as which operating system version) to be unfrozen. In such a case, the partially frozen object encodes all of the alternative design versions that were most recent as of this date. Similarly, a module might be frozen with respect to a particular design decision; in this case it would encode the history of revisions for that design decision.

In our bicycle database, we might define a view that specifies the current most lightweight bicycle configuration. This view defines a partially frozen configuration, which changes as new frames or wheels are designed.

Although a view definition may currently determine a unique configuration, future events might cause that configuration to change or new configurations to be added. The only way to completely freeze an object in a specific configuration is to restrict each generic reference on an immutable key, such as a version number.

The above view definition is such an example. However, it is not necessary for the user to explicitly write such restrictions. Instead, the system should provide a way in which a given configuration specification can be "precompiled" into a frozen version of this specification. We introduce the keyword *frozen* for this purpose. For example, suppose that at this moment the following view happens to define the same configuration as the previous view definition:

define view bike2 *as*
retrieve B
from B *in* bikes
where B.projectName = "BMX" *and*
B.frame.material = "alloy" *and*
B.frontWheel.size $\geq$ 27 *and*
B.rearWheel = B.frontWheel

If the first line of this definition is changed to begin *define frozen view*, then the system will precompile it, automatically creating the previous view definition. A similar feature exists in DSEE (Leblang and Chase, 1984).

5. Multi-Dimensional Versioning

5.1 Version Semantics

In the previous sections we have treated the version set of an object as being conceptually unstructured. However, the semantics of a type usually imposes a particular logical structure on the version sets of its member objects. In this section we examine this idea in the context of two different application domains.

5.1.1 CAD and CASE Databases Each conceptual object in a CAD or CASE application corresponds to a design project whose versions are the various designs for that project. There are two reasons why new versions are added to a version set (Katz, 1990). The new version might be the result of a bug fix, in which case it is a *revision* of a previous version. Or it might be the result of trying out a different implementation strategy, in which case it is an *alternative* version.

Each alternative implementation strategy is defined by the values of some set of attributes, which we call *alternative attributes*. All versions of an object having the same values for these attributes are considered to belong to the same *design alternative*. For example, in a CASE application we might declare the attribute *opSys* to be an alternative attribute; this choice would specify a design alternative for each possible target operating system. In Figure 2, the attributes *style* and *numSpeeds* might be used as alternative attributes for the type *Bicycle*. Under this assumption, the version set of Figure 4 has three design alternatives: 10-speed racing bikes, 3-speed racing bikes, and 10-speed mountain bikes.

Figure 4. Version set for a *Bicycle* Object

	style	numSpeeds	designDate	derivedFrom
v1	racing	10	1981	*nil*
v2	mountain	10	1983	v1
v3	racing	3	1985	v1
v4	racing	10	1987	v3
v5	mountain	10	1989	v2

If the semantics of alternative versions is to partition the version set into its different design alternatives, then the semantics of revisions is to organize the versions within a particular design alternative. In particular, a version is considered to be a revision of another if they both belong to the same design alternative and the first was created after the second. The attribute *designDate* is used for this purpose in Figure 2. Thus, in Figure 4 version *v4* is a revision of *v1*, and *v5* is a revision of *v2*.

Note that this definition of revision is a logical concept, and is unrelated to the information kept in the attribute *derivedFrom*. For example, in Figure 4 version *v4* is a revision of *v1* but is derived from *v3*. Presumably in this example, the designer of *v4* decided to redo version *v3* for a different design alternative; the result is a revision of *v1*.

The attribute *derivedFrom* structures the version set of an object into what is called a *version hierarchy*. Version hierarchies have a different semantics from revisions. In particular, the former structures a version set according to how its versions were created; the latter structures a version set according to the values of its versions. All versioning systems that we know of (with the exception of Dittrich and Lorie, 1988) support only the semantics of version hierarchies. The concept of revision is then shoehorned into this narrow context, which makes certain configurations difficult to express.

The advantage of our definition of revision is that it allows versions to be accessed simply by mentioning the desired design alternative and effective revision date. In our bicycle-design example, a user might ask for the most recent ten-speed touring bikes as of 1986. Such a specification can be expressed naturally as *bikes(10, touring, 1986)*. Intuitively, the semantics of alternatives and revisions organizes a version set into a multidimensional space; versions can be accessed by giving the desired coordinates in this space.

5.1.2 Historical Databases. Historical databases form the second application domain in which we examine version semantics. In Figure 3, we saw how each version of *Employee* records a change to its associated object, using the attribute *occurredAt* to store the time at which the change took place. This attribute can be thought of

Figure 5. Temporal version set

	salary	position	occurredAt	recordedAt
v1	20	clerk	1981	1981
v2	25	clerk	1982	1982
v3	22	clerk	1981	1983
v4	35	vp	1985	1984
v5	99	ceo	1988	1988

as defining a one-dimensional time line, and allows the version set to be viewed as a function from times to versions. For example, the expression *employees(1985)* is a natural way of specifying the most current version of each member of *employees* as of 1985.

The attribute *occurredAt* holds what is known as *logical time*—that is, the time at which the changes took place in the real world. Another form of time is known as *physical time*, which models the time at which the changes took place in the database. A system which supports both logical and physical time is called a *temporal database system* (Snodgrass, 1987). Physical time can be supported by the addition of the attribute *recordedAt* to the types in Figure 3. Figure 5 shows a version set using this revised scheme for an example employee *sue*. Versions *v1* and *v2* assert that Sue was hired as a clerk in 1981 and given a pay raise a year later. In 1983 her starting salary was changed retroactively, probably due to a clerical error. In 1984 it was announced (and recorded in the database) that she would be promoted to VP in 1985. Finally, she was promoted to CEO in 1988, at which time the information was also recorded.

Logical and physical time are orthogonal concepts, and define a two-dimensional version space. That is, a particular version can be identified by giving two coordinates: the logical time of the change and the physical time of the change. A user can specify a particular version of an object simply by giving its (logical-time, physical-time) coordinates. Using Figure 5 as an example, *sue(1981, 1982)* specifies *v1*, *sue(1981, 1986)* specifies *v3*, and *sue(1988, 1987)* specifies *v4*.

5.2 Dimension Types

The previous section showed that the semantics of both CAD and historical databases imposes a multidimensional structure on version sets. In this section we examine how these ideas translate to our versioning data model. We discuss the following issues:

- How the dimensions for a given type can be declared in the scheme;

Figure 6. Some dimension types

```
define type BikeStyle:
  dimension style = #
  (versioned
     style: char[10]
  )
define type BikeSpeeds:
  dimension numSpeeds = #
  (versioned
     numSpeeds: int4
  )
define type PhysicalTime:
  dimension max(recordedAt ≤ #)
  (versioned
     recordedAt: Date
  )
```

- How queries can access versions by specifying coordinates;
- How the system can interpret such queries.

5.2.1 Dimension Declaration. We define a *dimension predicate* to be a Boolean expression which uses the designated symbol # as a constant, and which may be surrounded by an optional aggregation operator. A dimension predicate can be declared for an EXTRA-V type by using the new keyword *dimension*. A type containing a dimension predicate is called a *dimension type.*

Figure 6 contains definitions for the dimension types *BikeStyle, BikeSpeeds,* and *PhysicalTime*. Each dimension predicate encodes the semantics of its dimension. Intuitively, the designated symbol represents an unspecified coordinate value of the dimension; the predicate tells what versions to return given that coordinate. For example, the predicate for *BikeStyle* says to choose the versions having the given *style*-value. The predicate for *PhysicalTime* says to choose the versions having the highest value of *recordedAt* which is not greater than the given value.

Dimension types can be inherited. Although this inheritance can be declared using standard EXTRA syntax, it seems more appropriate to separate inherited dimension types from other inherited types. We therefore declare inherited dimension types after the *versioned* keyword. Figure 7 shows a revised definition of *Bicycle* from Figure 2. Note that under this new definition, instances of *Bicycle* have exactly the same attributes as before (except that *designDate* has been renamed to *recordedAt*). The difference is that three of the attributes have been designated as defining a three-dimensional version space.

One of the advantages of our approach is that each type can independently declare its version dimensions. For example, in Figure 3 the types *Employee* and

Figure 7. Defining a 3-dimensional version space

```
define type Bicycle:
  ( projectName: char[10],
    dueDate: Date
  versioned by BikeSpeeds, BikeStyle, PhysicalTime
    frame: ref Frame,
    frontWheel: ref Wheel,
    rearWheel: ref Wheel,
    cost: int4
  )
```

Company could be specified as two-dimensional temporal databases by including "*versioned by LogicalTime, PhysicalTime*" in their type declarations. Alternatively, *Employee* could be versioned according to *LogicalTime* only; in this case, the previous values of the database for each logical time would be kept for *Company* but not *Employee*. It is even possible for *Employee* to be versioned according to *LogicalTime* only, and *Company* to be versioned according to *PhysicalTime* only. This flexibility also allows different kinds of versioning to be used for different parts of a database. That is, there need not be just "design databases" or "historical databases"; any mixture is possible. Moreover, dimensions are not hard-coded into the system, so new dimensions can be declared by an application as needed.

The choice of dimensions in Figure 7 was totally arbitrary on our part. We could just as easily have declared *Bicycle* to have fewer (or more) dimensions, or changed the semantics of the dimension types in Figure 6. Our model provides a powerful and flexible way for a database designer to specify the logical structure of version sets; it is up to the designer to ensure that it corresponds to her intuition of the application.

5.2.2 Multidimensional Coordinates. The existence of dimension specifications does not change the meaning of the queries and updates of Section 3, because the conceptual scheme of each type has not changed. However, their existence does provide added semantics that can lead to significantly shorter and more natural queries. In particular, a desired version of an object can be specified by giving its coordinates in the multidimensional space defined by its type.

Coordinate specification occurs in the *from* clause. For example, the following query returns the cost of the most recent 5-speed racing bicycle for the BMX project as of 1986:

```
retrieve B.cost
from B in bikes(5, "racing", 1986)
where B.projectName = "BMX"
```

Note that the order of the coordinate values is determined by the order of their declaration in *Bicycle*. If a type has many dimensions, this positional coordinate notation can become difficult to use. We therefore also support coordinate spec-

ification by giving the name of the dimension attribute. For example, the above coordinates could be equivalently specified by the expression

bikes(numSpeeds=5, style="racing", recordedAt=1986)

In addition to avoiding the need to know the position of each dimension, this notation allows a user to specify only a portion of the dimensions. The unspecified dimensions are given default values, as described in Section 5.3.

For another example, suppose that the scheme of Figure 3 has been declared so that both types are versioned along the single dimension *LogicalTime*. Then the following query returns the 1989 salaries of all employees who worked for IBM in 1983:

retrieve E.salary
from E *in* employees(1989), E′ *in* employees(1983)
where E.genericId = E′.genericId *and*
E′.worksFor.name = "IBM"

The following query returns the names of all employees who were the CEO in 1989 of the same company that they worked for in 1983:

retrieve E.name
from E *in* employees(1983), C *in* E.worksFor(1989)
where C.ceo.genericId = E.genericId

5.2.3 Coordinate Interpretation We now describe how coordinate specifications are interpreted by the system. We do this by showing how a query containing coordinate specifications can be translated into an equivalent query with the syntax of Section 3.

The query translation process is best understood by considering some specific examples first. Consider the above query involving the bicycle-design scheme. The expression *"B in bikes(5, racing, 1986)"* should be translated into *"B inall bikes"* together with some additional predicates in the *where* clause specifying the required bindings for B. In particular, the following three predicates are needed:

B.numSpeeds = 5 *and*
B.style = "racing" *and*
B.recordedAt = max(*retrieve* B2.recordedAt
from B2 *inall* bikes
where B2.genericId = B.genericId *and*
B2.numSpeeds = B.numSpeeds *and*
B2.style = B.style *and*
B2.recordedAt $\leq$ 1986)

Each predicate corresponds to a dimension of *B*. The predicates for the first two dimensions are straightforward. The third dimension's predicate involves aggrega-

tion; the versions to be aggregated are those versions which belong to the same version set as B and are 5-speed racing bikes.

The next query in Section 5.2.2 involved the employee-company scheme, and is translated as follows:

retrieve E.salary
from E, E′ *inall* employees
where E.genericId = E′.genericId *and*
E′.worksFor.name = "IBM" *and*
E.occurredAt = max(*retrieve* E2.occurredAt
from E2 *inall* employees
where E2.genericId = E.genericId *and*
E2.occurredAt $\leq$ 1989) *and*
E′.occurredAt = max(*retrieve* E3.occurredAt
from E3 *inall* employees
where E3.genericId = E′.genericId *and*
E3.occurredAt $\leq$ 1983)

Note how the predicates generated for each variable are independent of each other.

One must be careful in translating coordinates having multiple aggregation dimensions, because the order in which aggregation is performed is important. For an example, suppose that Figure 3 is a temporal database; that is, the types *Employee* and *Company* are dimensioned according to both *LogicalTime* and *PhysicalTime*. Then the query

retrieve E.salary
from E *in* employees(1986,1988)

returns the salary of all employees in 1986 as they were known in 1988. The proper translation of this query is as follows:

retrieve E.salary
from E *inall* employees
where E.occurredAt = max(*retrieve* E2.occurredAt
from E2 *inall* employees
where E2.genericId = E.genericId *and*
E2.occurredAt $\leq$ 1986 *and*
E2.recordedAt $\leq$ 1988) *and*
E.recordedAt = max(*retrieve* E3.recordedAt
from E3 *inall* employees
where E3.genericId = E.genericId *and*
E3.occurredAt = E.occurredAt *and*
E3.recordedAt $\leq$ 1988)

The first aggregation chooses the versions with the latest possible logical date. If there are several such versions, then the second aggregation chooses the one

with the latest possible physical date. Note how the second aggregation depends on the binding generated by the first aggregation; these two predicates are thus not independent of each other.

To define the general translation algorithm, we use the following notation. Let D be a dimension type and V be a variable. Then:

- *att(D, V)* is the attribute declared in D qualified by V;
- *cond(D, x, V)* is the condition part of the predicate declared in D, where the designated symbol # is replaced by x and each expression is qualified by V; and
- *aggOp(D)* is the aggregation operator declared in the predicate of D (or *null* if no aggregation was declared).

A query is translated as follows. Let the expression "V *in* $e(x_1, \ldots, x_n)$" be specified in the *from* clause of a query, where e is an expression of type T, and T inherits the dimension types $D_1, \ldots, D_n$. Then this specification becomes "V *inall* e" in the new *from* clause, and n predicates $\{P_1, \ldots, P_n\}$ are added to the *where* clause. Each P_i is defined as follows:

(a) If $aggOp(D_i)$ is *null*, then $P_i = cond(D_i, x_i, V)$;

(b) if $aggOp(D_i)$ is not *null*, then P_i is

$$
\begin{aligned}
att(D_i, \mathrm{V}) = aggOp(D_i)\ (&\textit{retrieve } V' \\
&\textit{from } V' \textit{ inall } e \\
&\textit{where } V'.genericId = V.genericId \textit{ and} \\
&\qquad att(D_1, V') = att(D_1, V) \textit{ and} \\
&\qquad \cdots \textit{ and} \\
&\qquad att(D_{i-1}, V') = att(D_{i-1}, V) \textit{ and} \\
&\qquad cond(D_i, x_i, V') \textit{ and} \\
&\qquad \cdots \quad \textit{and} \\
&\qquad cond\, D_n, x_n, V')\ )
\end{aligned}
$$

The aggregation expression for a P_i has $n + 1$ subpredicates within it. The first subpredicate restricts the aggregation so that only versions from the same version set as V are used. The next i - 1 subpredicates require that the aggregated versions respect the variable bindings established by P_1 through P_{i-1}. The remaining subpredicates constrain the remaining dimensions of the version space appropriately.

5.3 Contexts

We now deal with the issue of how default versions of objects are chosen. We assume that there is a global variable associated with each dimension type; by convention,

this variable has the same name as the type. The values of all of these variables is called the current *context*.

When a query or update command is issued, the current context is used to choose the necessary default versions. Recall that defaults are specified in the *from* clause via the *in* keyword. Consider the declaration *"V in e"*, where e is an expression of type T. If T has dimensions $\{D_1, \ldots, D_n\}$, then the n global variables corresponding to each D_i determines a coordinate in the version space of T. This coordinate is used to choose the default version. In other words, the above declaration is equivalent to *"V in* $e(x_1, \ldots, x_n)$*"*, where each x_i is the value of the appropriate global variable.

It is often useful to be able to evaluate a query in a particular context. Instead of requiring the user to manually change the appropriate context variables, execute the query, and restore the old context values, we introduce new syntax. The *incontext* clause in EXCESS-V specifies a context, to be effective only for the duration of the query. For example, suppose that the types of Figure 3 have the two dimensions *(LogicalTime, PhysicalTime)*, and that both context variables have the value 1990. Then the query

retrieve E.salary
from E *in* employees, C *in* E.worksFor
where C.name = "IBM"
incontext LogicalTime = 1976

returns the 1976 salary of all employees working for IBM in 1976, as it was known in 1990.

Contexts provide the way to synchronize the way defaults are chosen for multiple objects. In the above example, every generic reference was evaluated using the coordinates (1976, 1990). In general, if two objects share a common dimension, then the system will use the same coordinate value when choosing defaults. The need for this form of synchronization is prominent in CASE systems. For example, one could configure a program as it existed in 1986 for the UNIX system with the clause

incontext PhysicalTime = 1986, OpSys = "UNIX".

The use of contexts provides a synchronous way of accessing previous versions of objects. It is also possible to create a "partially synchronous" configuration by combining contexts and explicit references to versions. For example, in a CASE system one could configure a program in a particular context, but substitute a test version of some module. In our historical database of Figure 3, this ability allows the following query to be expressed:

retrieve E.salary, C.sales, C.ceo.salary
from E *in* employees, C *in* E.worksFor(1980, 1990)
where C.name = "IBM"

incontext LogicalTime = 1976

Here the generic references *employees* and *C.ceo* are evaluated in the context 1976, whereas the generic reference *E.worksFor* is evaluated in the context 1980. Thus the query returns the 1976 salary of all employees who worked for IBM in 1976, the 1980 sales of IBM, and the 1976 salary of whoever was the CEO of IBM in 1980.

We note that the specification of default versions in other systems is significantly less expressive than what we have described here. CAD systems typically provide a fixed set of choice heuristics, such as "most recent" and "latest release". In ORION, for example, this set is built-in to the system and is not extendible. Moreover, these heuristics are always applied to the current date; it is not possible to specify a configuration as of a previous date. Biliris has proposed additions to the ORION model that would allow such a possibility (Biliris, 1990); however, his approach is somewhat contorted and only allows time-based dimensions to be synchronized. The TQuel temporal query language uses defaults in a very limited way: Only the current time can be used as a context value.

Our use of global variables for synchronization is similar to the way compiler switches are used in CASE systems. The results of this section show how these CASE features can be added to a data model and be made application-independent.

6. Conclusions

We have shown how versioning and configuration management can be added to the EXTRA/EXCESS data model. The resulting versioning model, EXTRA-V/EXCESS-V, has the following features:

- A small number of new constructs are added, which can be ignored by people who do not need or want to know about versioning.
- The constructs are all conceptual; that is, they do not depend on how versions are implemented at the physical level.
- The constructs are application-independent. Constructs which are specific to a particular application domain (such as CASE) have either been generalized (e.g. synchronization and context), or omitted.
- The constructs are high-level, allowing users to access versioned data in exactly the same non-procedural way as unversioned data.

We also considered the semantics of versioning applications, and saw that version sets often form a multidimensional space. We then showed how such semantics can be declared using dimension types, and accessed by giving the desired coordinates in that space. This approach led to semantically-based configuration specification and a framework for specifying how default versions are chosen.

There are several issues that require further exploration. Query optimization strategies have not yet been examined. For example, configuration specification can involve a substantial amount of aggregation, and a straightforward evaluation of such queries may not be efficient. Can the standard relational techniques (such as magic sets) be used, or are other strategies necessary?

We also need to examine the physical strategies used to store versions. CASE systems store versions as differential files based on the version hierarchy; historical database systems use techniques based on the semantics of time intervals; and CAD systems use still other techniques. A more careful study of these implementation strategies is necessary. In particular, we need to understand better when a given storage strategy is useful and the extent to which different strategies can be combined. The possibility of using indexes also needs consideration. Ultimately, we would like a database designer to be able to declare storage strategies for versioned types similarly to the way it is done with relations.

This article has been concerned with encoding the basic ideas of versioning and configuration management into a data model. There are many features related to versioning that we have not had the time to examine. For example, CAD systems provide change notification, schema evolution, and checkin/checkout; CASE systems provide incremental compilation; and historical systems provide operators on time intervals. Each of these features needs to be reexamined in the light of our model. Are the ideas application-independent, or do they involve some special properties of the application? Are there general concepts that are missing from our current model? For example, do schema versions have special requirements that are not handled by dimension types? For that matter, what would it mean for a dimension type to be versioned?

References

Abiteboul, S. and Bonner, A. Objects and views. *Proceedings of the ACM-SIGMOD Conference,* Denver, 1991.

Agrawal, R. and Jagadish, H. On correctly configuring versioned objects. *Proceedings of the Fifth VLDB Conference,* Amsterdam, 1989.

Atwood, T. An object-oriented DBMS for design support applications. *Proceedings of the IEEE Computer Aided Technologies Conference,* 1985.

Banerjee, J., Chou, H., Garza, J., Kim, W., Woelk, D., Ballou, N., and Kim, H. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems* 5(1):3-26, 1987.

Batory, D. and Kim, W. Modelling concepts for VLSI CAD objects. *ACM Transactions on Database Systems* 10(3):322-346, 1985.

Beech, D. and Mahbod, B. Generalized version control in an object-oriented database. *Proceedings of the Fourth IEEE Data Engineering Conference,* (Los Angeles CA, Feb. 1988), pp. 14-22.

Belkhatir, N. and Estublier, J. Experience with a data base of programs. *Proceedings of the ACM SIGSOFT-SIGPLAN Symposium on Practical Software Development Environments,* 1986.

Biliris, A. Database support for evolving design objects. *Proceedings of the ACM/IEEE Design Automation Conference,* 1989.

Biliris, A. Modeling design object relationships in PEGASUS. *Proceedings of the Sixth IEEE Data Engineering Conference,* Los Angeles, 1990.

Carey, M., DeWitt, D., and Vandenberg, S. A data model and query language for EXODUS. *Proceedings of the ACM-SIGMOD Conference,* Chicago, 1988.

Caruso, M. and Sciore, E. Meta-functions and contexts in an object-oriented database language. *Proceedings of the ACM-SIGMOD Conference,* Chicago, 1988.

Chou, H.T. and Kim, W. A unifying framework for versions in a CAD environment. *Proceedings of the Twelfth VLDB Conference,* Kyoto, Japan, 1986.

Chou, H.T. and Kim, W. Versions and change notification in an object-oriented database system. *Proceedings of the ACM/IEEE Design Automation Conference,* 1988.

Clifford, J. and Croker, A. The historical relation data model (HRDM) and algebra based on lifespans. *Proceedings of the Thirteenth IEEE Data Engineering Conference,* Los Angeles, 1987.

Cohen, E., Soni, D., Gluecker, R., Hasling, W., Schwanke, R., and Wagner, M. Version management in Gypsy. *Proceedings of the ACM SIGSOFT-SIGPLAN Symposium on Practical Software Development Environments,* Boston, 1988.

Copeland, G. and Maier, D. Making smalltalk a database system. *Proceedings of the ACM-SIGMOD Conference,* Boston, 1984.

Dittrich, K. and Lorie, R. Version support for engineering database systems. *IEEE Transactions on Software Engineering,* 14(4):429-437, 1988.

Gadia, S. A homogeneous relational model and query languages for temporal databases. *ACM Transactions on Database Systems,* 13(4):418-448, 1988.

Hudson, S. and King, R. The cactis project: Database support for software environments. *IEEE Transactions on Software Engineering,* 14(6):709-719, 1988.

Katz, R. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys,* 22(4):375-408, 1990.

Katz, R. and Lehman, T. Database support for versions and alternatives of large design files. *Proceedings of the IEEE Transactions on Software Engineering,* 10(2):191-200, 1984.

Katz, R., Chang, E. and Bhateja, R. Version modelling concepts for computer-aided design databases. *Proceedings of the ACM-SIGMOD Conference,* Washington D.C., 1986.

Kim, W., Bertino, E. and Garza, J. Composite Objects Revisited. *Proceedings of the ACM-SIGMOD Conference,* Portland, OR, 1989.

Klahold, P., Schlageter, G. and Wilkes, W. A general model for version management in databases. *Proceedings of the Twelfth VLDB Conference,* Kyoto, Japan, 1986.

Leblang, D. and Chase, R. Computer-aided software engineering in a distributed workstation environment. *Proceedings of the ACM SIGSOFT-SIGPLAN Symposium on Practical Software Development Environments,* Pittsburgh, PA, 1984.

Mahler, A. and Lampen, A. An integrated toolset for engineering software configurations. *Proceedings of the ACM SIGSOFT-SIGPLAN Symposium on Practical Software Development Environments,* Boston, MA, 1988.

Rose, E. and Segev, A. TOODM—A temporal object-oriented data model with temporal constraints. *Proceedings of the E/R Institute Conference on the Entity Relationship Approach,* San Mateo, CA, 1991.

Rowe, L. and Stonebraker, M. The POSTGRES Data Model. *Proceedings of the Thirteenth VLDB Conference,* Brighton, England, 1987.

Sciore, E. Using annotations to support multiple kinds of versioning in an object-oriented database system. *ACM Transactions on Database Systems* 16(3):417-438, 1991*a*.

Sciore, E. Multidimensional versioning for object-oriented databases. *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases,* Munich, 1991*b*.

Shipman, D. The functional data model and the data langauge DAPLEX. *ACM Transactions on Database Systems,* 6(1):140-173, 1981.

Snodgrass, R. The temporal query language TQUEL. *ACM Transactions on Database Systems,* 12(2):247-298, 1987.

Tansel, A. Adding time dimension to relational model and extending relational algebra. *Information Systems,* 11(4):343-355, 1986.

Wu, G. and Dayal, U. A uniform model for temporal object-oriented databases. *Proceedings of the Eighteenth IEEE Data Engineering Conference* Kyoto, Japan, 1992.

Zaniolo, C. The database language GEM. *Proceedings of the ACM-SIGMOD Conference,* San Jose, CA, 1983.