Multiversion Views: constructing views in a multiversion database

Claudia Bauzer Medeiros

IC - UNICAMP - CP 6176
13081-970 Campinas SP
Brazil

cmbm@dcc.unicamp.br

Marie-Jo Bellosta and Geneviève Jomier

LAMSADE
Université Paris-Dauphine
75775 Paris - France

{bellosta—jomier}@lamsade.dauphine.fr

Abstract

Commercial DBMS offer mechanisms for views and for versions. Research and development efforts in these directions are, however, characterized by concentration on either the one or the other mechanism, very seldom trying to take advantage of their complementary properties. This paper presents the *multiversion view mechanism*, which allows these orthogonal concepts to be managed together, taking advantage of their combined characteristics. Unlike previous efforts to combine views and versions, multiversion views create views over versions of data, thereby offering users coherent logical units of the versioned world. They allow a wide range of (virtual) data reorganization possibilities, which encompass, among others, operations found in temporal databases and OLAP. Multiversion views are illustrated and motivated by needs from a real life large case study of complex configuration management, described at the end of the paper.

Keywords: Multiversion databases, views, OLAP

1 Introduction

Database applications need both views and versions to meet distinct requirements. The main difference between versions and views is that versions are permanently stored in the database, whereas views are virtual. This permanent/virtual status is, in fact, one important difference, but it is not the only one.

Versions are employed when the user wants to keep track of the evolution of the entities modeled in the database. This evolution may be just temporal, following a total order (e.g., when recording changes in personnel in a given enterprise) or may include several other criteria, being organized according to some partial order (e.g., in keeping track of design alternatives in a CAD environment). Nowadays, the use of versions is being extended to several domains, notably in planning activities, where the user wants to organize alternative scenarios for decision purposes.

Whereas versions keep track of evolution, views provide a virtual image of the database, with three main goals [FSS79]:

in the database, hiding unnecessary details;

- restriction for security: the view keeps the user from accessing non-authorized data, hiding sensitive data; and
- restructuration: the view is the means through which data are restructured according to the users' needs, grouping together parts of a database into a virtual work unit, thereby providing another perspective on stored data.

More recently, views have been advocated as means of introducing structure into unstructured or semistructured data (e.g., [AGM+97]). In all cases, as far as the user is concerned, a view is a virtual database on its own.

Thus, both views and versions must be supported for different reasons. However, the standard approach is to deal with each separately, even when data are versioned. Consequently, views are limited to showing one (non-versioned) state. If users want to *see* how data are versioned, they are required to generate distinct views (one per version) and to maintain the relationships among these views. This is obviously a serious limitation: if the data are versioned, why not provide views that reflect this? The main problem is that, since views and versions are always maintained separately, and are conceptually kept apart, combining them presents both design and implementation difficulties. In order to solve this problem, this paper introduces *multiversion views* – a view model which is based on extending a version mechanism which is already available (Cellary and Jomier [CJ90]) with views. This solution has the following main characteristics:

- It preserves versioning of data. Views are treated as virtual databases created on top of the source (base) data. If these base data contain versions, then the views must reflect this to the users, i.e., they constitute *virtual databases with versions*.
- It maintains view functionality. Multiversion views serve, as any other view, to restrict information for ease of use and/or security, and to offer a restructured perspective of the underlying database.

As a result, two very useful data management facilities are offered. First, multiversion views allow users to compare and manipulate multiple states (i.e., the versions) of a given set of database objects, in a single view. The management of inter-relationships between objects and their versions is left to the underlying version mechanism, thus freeing the users from this type of concern. Second, they allow users to reorganize objects according to different versioning criteria, which are not necessarily the same as the original (stored) versioning criterion. As will be shown during the paper, this permits users to manipulate data as in an OLAP context (e.g., [Sho97]) when restricted to single database classes, or reproducing operations in temporal databases (e.g., [Sno95]). Since versions are also common in a design context (e.g., CAD applications), multiversion views contribute to enrich a design environment with multiple perspectives of the data, constituting a means for enhancing cooperative work.

In order to describe multiversion views, we base our work on the *database version model* of Cellary and Jomier [CJ90], in an object-oriented context. We have chosen object-oriented

databases. The database version model of [CJ90] was selected due to the fact that it is based in a formal description, and that it clearly distinguishes between physical implementation issues and logical user-related issues. Thus, physical issues are appropriately handled by the version model adopted.

An informal description of some of the characteristics of multiversion views was given in [MBJ96], motivated by the problem of managing multiple representations in geographic applications. Here we extend and formalize these notions, showing how they can be applied to a more general framework.

The remainder of this paper is organized as follows. Section 2 gives an overview of related work. The subsequent sections of the paper describe the multiversion view mechanism itself. Section 3 introduces multiversion views: Section 3.1 gives a brief overview of the multiversion database model, which is the basis for construction of these views; Sections 3.2 and 3.3 informally present multiversion views, and introduce a short example that is used throughout the text. Sections 4 and 5 formalize the construction of these views. Section 6 describes the architecture for implementing multiversion views. Section 7 presents conclusions and directions for future work. Appendix A describes the real life application from which we extracted the basic example we use to illustrate multiversion views.

Throughout the paper, we sometimes use "code notation" (e.g., when we define database classes or specify queries). We stress that we do not follow any specific language syntax, but rather are providing an intuitive perspective of operations that lie behind multiversion view construction.

2 Related work

Work on views and on versions has usually progressed independently. Exceptions are our own preliminary work on the subject [MBJ96], in the context of geographic applications, the work of Byeon and McLeod [BM93], which tries to provide an integrated framework for working with views and versions, and that of Ra and Rundensteiner [RR97], which uses views to support schema evolution.

Byeon and McLeod's integrated approach tries to unify the concepts of views and versions by considering both to be obtainable from a database by a set of schema and instance transformation operations. These operations are executed in a sequence of $<$ *import, transform* $>$ steps. Views (and versions) are created by means of schema evolution operations and instance definitions, where schema operations receive special attention. Since the goal is to eliminate any distinction between views and versions, this approach has the shortcoming that the concept of data versioning, which users find useful, is lost. The main differences between their views and versions are:

- a view can be built from more than one source database; and
- a version can only be built on top of a single database, by importing its entire schema, and then transforming it.

agree to the restructuring, the view is materialized into a version, and loses all connotation with views and view mechanisms. Examples are the work of [BFK95], where views serve to validate schema evolution, and [Nov95], where views are used to build consistent software configurations from a database of software modules, and then stored as versions of a given software. A related proposal is that of [KR97], based on maintaining versions of relational views, where these versions are created through a view mechanism (constructing views over views by means of updates). The goal is to support multiple versions of a given data set, for decision support applications.

Ra and Rundensteiner [RR97], on the other hand, are interested in schema evolution (e.g., for CAD applications). They have developed a system which maintains multiple schema versions for a given database, allowing "old" and "new" users to share the same data, viewed through distinct schemata. Each version of a schema is handled through mappings from previous to next schema, by means of a schema update mechanism. Eventually, older schema versions may be abandoned.

The remaining literature concentrates either on views or on versions. Byeon and MacLeod [BM93] provide a useful means of analyzing these papers, from an object-oriented perspective, pointing out that the creation of views (or of versions) either concentrates on a single class (i.e., a view schema has just one class, or only a class can be versioned) or on multiple classes. Earlier work concentrates on the first approach, whereas recently authors have considered the issue of multi-class schemata.

Versions are a means of storing different states of a given entity, thereby allowing the control of alternatives and of temporal data evolution. Research has appeared mostly in the context of CASE systems and CAD/CAM projects, often for object-oriented databases (e.g., [KSW86, Kat90, KS92, WR94, TO96]). Versions are also often considered in the context of concurrency control (e.g., [FD96, LST98]) or as the means to support cooperative work (e.g., [DL96]). In several cases, versions of a given data set are created by materializing snapshots of this set. This is usually not seen as a version mechanism, but just as a means of keeping track of data through time. Good surveys on different uses for versioning mechanisms appear in [Kat90] and [Man00].

Views in databases are usually defined as the result of a query. Views may be stored (materialized), but in general it is understood that versions correspond to stable data, whereas though views are generated from stable data, they are usually temporary. Each proposed view mechanism concentrates on a different issue (e.g., operation translation, data restructuration capabilities, or data integration properties). The issue of views in relational databases is understood [FSS79, BLT86], but is still a matter of research in object-oriented databases (e.g., [AB91, MM91, Ber92, SAD94, LDB98]), as well as in deductive databases (e.g., [YPS95]). As remarked in a thorough survey on view mechanisms [MP96], although queries may be enough to build views in the relational world, this is no longer the case for object oriented databases, where the schema must be defined apart. The appearance of data warehouses has provided a new field for work on materialized views (e.g., [Huy98, YKL98]), restricted to relational queries and their optimization, as well as for creating online aggregations of these data – e.g., for OLAP operations, see [BW99].

In object-oriented databases, the construction of the view schema should be kept apart

defined which cannot be specified by queries. However, none of the proposed mechanisms separates schema definition from extension definition, and the issues of schema construction are thus blurred with those of extension specification and *oid* (object identifier) definition. One of the first papers to consider views in terms of schema restructuration was [TYI88], who considers view creation in terms of schema definition (by operations on the inheritance graph), implemented in Smalltalk. Different sets of schema structuration operations were proposed by [KC88] (in the context of the ORION project), [Run93] (for CAD data integration and customization), [BFK95] and [LDB97, LDB98] (formalizing a data model for views in object oriented database systems) and [FR97] (for constructing object oriented views on top of relational databases).

Multiversion views, as we show next, differ from all other proposals which try to combine views and versions by allowing users to "see" multiple consistent versioned states of the world throughout views. These versioned states can either reflect the way data are organized, but just showing parts of the database (akin to the "restrict" views of Section 1) or reorganize the underlying database by schema or versioning changes (akin to the "restructure"). Existing methods cannot perform the same tasks. First, no other study allows defining a view with simultaneous access to multiple versions in a database while at the same time ensuring version consistency. Furthermore, multiversion views also allow changing versioning criteria *virtually*, which we call changing version semantics. This, again, is not available elsewhere.

3 The Concept of Views over Multiversion Databases

The previous section briefly considered work on views and versions, showing that most of this work treats these issues separately. We, on the other hand, propose a mechanism that encompasses both, but yet allows them to be treated separately. *Multiversion views* are views which are built on top of a multiversion database, constructed using the database version approach of [CJ90].

This section presents these views informally, and the subsequent sections formalize the definition. The first part of the section gives an outline of the database version approach, and the second part gives a high level description of the multiversion view definition, introducing a simplified example which we use throughout the paper. The example is based on a real life case study described in Appendix A.

3.1 The database version approach

Our paper concerns building views for a database with versions. Several version mechanisms have been proposed in the literature (e.g., [CK86, Kat90, Bla91, Sci91]). In this section, we briefly present the *database version approach* [CJ90, GJ94], which we will use as the version model and mechanism for our work. The advantages of this approach have been discussed elsewhere, and do not concern this paper.

Database version A conventional monoversion database (i.e., a database where versions are not considered) represents *one* state of a modeled part of the world. In the database

modeled part of the world. Each state is called a *database version*. A database version, denoted d, has an identifier, denoted d_i, and contains a version of each object stored in the database. A database version is defined by the couple <*schema, logical extension version*>. A logical extension version reflects one (versioned) state of the modeled world, and roughly corresponds to the extension of a monoversion database.

A multiversion database contains multiversion objects, i.e., each object therein is composed of several *logical object versions*. A multiversion object is denoted by mo. A logical object version is similar to an object in a monoversion database: it has an identifier and a value. A logical extension version contains a logical object version of each object stored in the database. Given a multiversion object mo, and a set of database versions $\{d_1, d_2 \dots d_n\}$, the identifier of the logical version of mo contained in some database version d_k is given by $< mo, d_k >$. More generally, the identifier of a logical version of a multiversion object contained in a database version is a couple <*multiversion object identifier, database version identifier*>, denoting the fact that a given multiversion object may have a different value for each database version.

Since, formally, all the objects that exist in the multiversion database appear logically at each database version, a special value $\perp$, meaning *does not exist*, is used to express object non–existence in a particular database version.

Physical object versions. Since different database versions usually differ only partially from one another, logical versions of an object often have identical values. To avoid redundancy, they are mapped to a *physical object version*, which can be shared by several database versions.

Derivation operation. Database versions are created by *derivation*. A derivation operation is addressed to a specific database version, which becomes the derivation's *parent* database version, and it derives a *child* database version, which, just after the derivation, is a logical copy of the parent. A database version may have as many children as desired. Once created, a database version evolves automonously, according to transactions addressed to it. For optimization reasons, the trace of database version derivation is kept by the database management system.

Example 1. Figure 1 presents an example of a multiversion database which has three logical database versions $d1, d2, d3$, for hardware components. The database contains two objects each of which has a name (resp., DISK1, DRIVER2) and a price (represented by $$ in the figure). $d2$ and $d3$ present alternatives for an overall price reduction, given the initial scenario in $d1$. For instance, the price of DISK1 is $$$ in $d1$, and $ in $d2$ (DRIVER2 was not affected), or much larger in $d3$ (DRIVER2 was discarded). The fact that DRIVER2 was discarded is represented in $d3$ by symbol $\perp$, indicating that DRIVER2 does not exist in that specific database version. If mo_1 and mo_2 are the multiversion object identifiers for respectively DISK1 and DRIVER2, then $< mo_1, d3 >$ stands for DISK1's state at database version $d3$ (i.e., a logical version of DISK1). Database versions $d2$ and $d3$ are both derived from the database version $d1$, as two alternatives. DISK1's price is different in each database

DRIVER2 had no changes from $d1$ to $d2$, and therefore its logical versions share the same physical version.

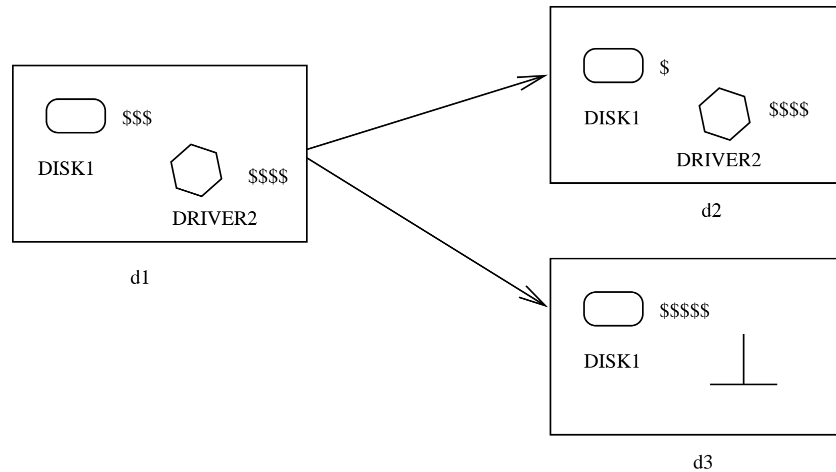

Figure 1: Example of a multiversion database and its derivation graph

Updating a logical object version. The key operation in the model is the update of a logical version of an object, i.e., updating the object's value within a particular database version. To update a logical object version contained in a database version d, first, the corresponding physical object version is identified. If it is unshared, it can be simply updated. If it is shared by several logical object versions, then a new physical object version is created, in which changes are introduced. As a result, the *shallow equality* between two objects becomes, in a multiversion database, the *identity* between the physical versions of two logical object versions, assuming that it is never the case that two distinct physical versions of one object have the same value.

To delete a logical object version, i.e., to delete an object in a particular database version, it is sufficient to update it with the $\perp$ value. To create a new multiversion object, first, its $\perp$ version is created and implicitly associated with all the database versions. Next, it may be updated in selected database versions, as required.

Complex object versioning. The database version approach is particularly advantageous in the management of complex objects, i.e., objects referencing other objects, because it provides orthogonality between composition and versioning. Just as a complex object points at its components, a version of a complex object contains multiversion object identifiers as references to components.

3.2 Virtual Multiversion Databases

Paraphrasing [AGM+97], for the user, the view is a "stand-alone" database created from the original database. Consequently, if we start from a multiversion database, then it is natural

We call this original database the *source* database for view construction, which itself may contain views – i.e., a view may be built on top of other views. Thus, the key concept of the view definition presented in this paper is that a view is a *virtual multiversion* database constructed from some *source multiversion* database.

A multiversion view is defined by the tuple

```
< Virtual intension; Virtual extension >
```

where both intension and extension are built from the source database. In standard database theory, the *intension* of a database corresponds to its schema. Here, we extend this notion, in order to let it encompass versioning criteria as well, i.e.,

```
Virtual intension = < Definition of the virtual schema ;
                      Definition of the virtual version semantics >
Virtual extension = set of queries over the source database
```

Since the view is a multiversion database, it is composed of virtual database versions. We recall that a database version d is defined by the couple <logical schema, logical extension>. Thus, a virtual database version is defined by the couple <*virtual logical schema, virtual logical extension* >.

Having defined these concepts, we now state the basics of multiversion view construction. Multiversion views are constructed according to two steps:

1. Intension definition (formalized at section 4)
    - specification of view schema; and
    - specification of version semantics
2. Extension definition, by querying the database (described at section 5).

If neither schema nor version criteria are modified, our views indeed correspond to standard views (always assuming that standard views allow data versioning, which is not normally the case). However, if at least one of these is modified, then new issues are raised.

Furthermore, version restructuring to our knowledge is not available by any existing method, unless the whole database is re-created. Schema restructuring, on the other hand, is available within certain limits, see section 2.

3.3 Basic intension definition

Traditionally, a database view may either just reflect part of an underlying database – i.e., by selecting the entire underlying schema, or part of it, and choosing some instances ("restriction" view) – or, alternatively, restructure the database ("restructure" view) – i.e., by restructuring the schema and then creating the instances.

A view on a multiversion database is itself a (virtual) multiversion database. What would be, in this case, a "restrict" and a "restructure" view? "Restrict" views are those

versioned objects according to a different intension criterion.

Since the intension of a multiversion view is made up of two components – schema and version semantics – there are *four possible ways of constructing the intension*, shown in the following table, where for shorthand notation we will refer to them as types 00, 01, 10 and 11. In general, when we talk of view "of type **(i j)**" where i stands for version semantics and j for schema structuration. In other words, from now on when we say for instance a view is "of type **(10)**" we actually mean that "its intension is constructed by restricting versioning semantics but without altering the source schema".

	Maintains source schema	Restructures source schema
Maintains source version semantics	**Type 00**	**Type 01**
Restructures source version semantics	**Type 10**	**Type 11**

Let us show these differences through a short example, which will be used throughout the text. This example is based on a real life application, which is described at length at the Appendix A. Consider a database for a firm that provides information technology services. These services include configuring and installing hardware and software components, with different types of component configurations. We note that, in this paper, the term *configuration* is used in its usual version context to mean "set of component versions that form a consistent unit". Components have different versions, and they are sold together with the corresponding manuals (which can be in several languages). Part of the firm's database has a class called HardwareComponent, which stores information about hardware components, prices and component compatibility (e.g., parts that can be installed together in a package). This class schema can be defined as

```
Class HardwareComponent type
        tuple (name: string,
               price: real,
               compatible: HardwareComponent)
```

Consider the HardwareComponent objects, at times t_i, under the multiversion database mechanism. Internally all is managed as a sequence of database versions $\{d_1, \ldots d_3\}$. Table 1 shows HardwareComponent instances: the first two columns show internal information managed by the version mechanism (database version identifiers and multiversion object identifiers); the next column indicates the timestamp (in this case, used as the versioning criterion); and the other columns show versionable attributes. Strings Ox are multiversion *oids*. Asterisks (*) indicate value changes from one version to another. For instance, $O1$ is a multiversion object which has three versions (in d_1, d_2, d_3), whereas object $O3$ has only one version in all database versions. At any time, the user will access either (d_1) or (d_2) or (d_3) which consist of consistent units of data. Given this small partial database, we now show examples of the four possible kinds of multiversion view.

d_1	O1	t_1	Disk1	$10	O2
	O2		Driver2	$10	O1
	O3		Server1	$30	O5
d_2	O1	t_2	Disk1	$20(*)	O2
	O2		Driver2	$10	O1
	O3		Server1	$30	O5
d_3	O1	t_3	Disk1	$40(*)	O8(*)
	O2		Driver2	$10	O5(*)
	O3		Server1	$30	O5

Table 1 - Component Class Instances across three Versions

Example 2. An example of view of type **(00)** is Myview00, the standard "select" view found in the literature: the intension reflects the underlying database intension (no change in versioning semantics and no change in schema), and the extension is a query that selects part of the database objects. Suppose, for instance, that it is restricted to HardwareComponents with price equal to $10. This view is shown in Table 2, where (d_j) denotes the source database versions from which objects were selected, *Source Obj* denotes the source objects used to construct the view objects, and vd_j denotes the (virtual) database versions (dbv) within the multiversion view.

Source dbv	Virtual dbv id	Source Obj	Time	Name	Price	CompatWith
(d_1)	vd_1	O1	t_1	Disk1	$10	O2
		O2		Driver2	$10	O1
(d_2)	vd_2	O2	t_2	Driver2	$10	O1
(d_3)	vd_3	O2	t_3	Driver2	$10	O5

Table 2 - No Intension Change - View of type (00)

Informally, this can be expressed by the query that follows, over the database versions in the database (MyDatabase). Variables range over database versions **(d)** or objects in a class **(c)** (e.g., see the VQL language for querying database versions of [Abd97]). The expression **c[d]** restricts object variable **c** to range over its logical versions in database version **d**.

```
create view MyView00 as
       select d
       from d in MyDatabase
            c in HardwareComponent
       where c[d].Price = $10
```

This view has three virtual database versions; vd_1 contains two virtual logical object instances, whereas the other database versions contain only one virtual logical object each.

Example 3. In the second case, MyView01, the intension's schema is restructured, but not the versioning semantics. Suppose this new schema is specified as containing one single

```
Virtual Class CompatibilityPairs type
tuple (hardcmp1:  tuple(name,Price) from HardwareComponent,
       hardcmp2:  tuple(name,Price) from HardwareComponent,
       cost: real)
```

where the indication *fromHardwareComponent* denotes that these schema elements were derived from the HardwareComponent class schema. Suppose also that the extension is now defined by selecting pairs of Components which are compatible, and adding up their prices to a total cost (a new virtual attribute Cost). Again using the informal query syntax adopted, the extension for this virtual class can be defined as

```
create view MyView01 as
       select new CompatibilityPairs ( c1[d].sum(c2[d].cost()) )
       from d in MyDatabase
            c1 in HardwareComponent
            c2 in HardwareComponent
       where c1[d].CompatWith = c2
```

Source dbv	Virt. dbv id	Source Obj	Time	Hardcmp1	Hardcmp2	Cost
(d_1)	vd_1	(O1, O2)	t_1	<Disk1, \$10>	<Driver2, \$10>	sum(\$10, \$10)
(d_2)	vd_2	(O1, O2)	t_2	<Disk1, \$20>	<Driver2, \$10>)	sum(\$20, \$10)

Table 3 - Intension Modification – Modifying the Schema - View type (01)

This view has a *new* class, but the versions are still organized according to the original versioning semantics (i.e., time). Thus, each virtual database version is still mapped to a single source database version. The virtual extension now has new objects (component pairs), demanding management of virtual *oids*.

Example 4. In the third case, MyView10, the intension is built without restructuring the source schema, but modifying the versioning semantics. Suppose, for instance, that the user now wants to group the versions according to a different time frame. If, for example, Time had been stored before in weeks and now the user wants a biweekly view, furthermore providing average prices over the two weeks. Times t_1 and t_2 are generalized into t_{12}, and only the state of t_2 is kept; t_2 and t_3 are generalized into t_{23}, keeping the state of t_3; and the biweekly interval which should start at t_3 is ignored. This is a standard operation in temporal databases [Sno95], sometimes called temporal generalization, which implies some kind of generalization operation on time and consequently on related values. The resulting view takes the assumption, common in temporal databases, that this generalization through time ignores some of the intermediate states.

[1] This repeats the classical example in object-oriented views when one wants to show schema reorganization through views (e.g., [AB91]).

(d_1, d_2)	vd_1	O1	t_{12}	Disk1	avg($10,$20)	O2
		O2		Driver2	avg($10,$10)	O1
		O3		Server1	avg($30,$30)	O5
(d_2, d_3)	vd_2	O1	t_{23}	Disk1	avg($20,$40)	O8
		O2		Driver2	avg($10,$10)	O5
		O3		Server1	avg($30,$30)	O5

Table 4 - Intension Modification – Modifying Version Semantics - View Type (10)

Informally, this may be expressed as follows, where *biweek* is a function that transforms two one-week timestamps into the equivalent biweekly timestamp.

```
create view Myview10 as
       select c[d]
       from d in (d1 U d2)
            d1 in MyDatabase
            d2 in MyDatabase
            c in HardwareComponent
       where d1.Time= d2.Time+7
       with   c[d].Price = avg (c[d1].Price, c[d2].Price)
              and c[d].Time = biweek (c[d1].Time, c[d2].Time)
              and c[d].CompWith = c[d2].CompWith
```

Notice that this type of intension operation may provide other view extensions, depending on the user's semantics, which are reflected in the way the extension is built. Again, this is to be expected, since this is equivalent to changing the temporal granularity with which data are analyzed. We point out that, besides the temporal database connotation, this type of operation is typical of OLAP environments [Sho97], where data organized according to some criteria (dimensions) are restructured either along the same dimension – a OLAP *roll-up* operation – or along another dimension (which would correspond to a *rotate* in OLAP).

Example 5. Finally, the fourth case, MyView11, is simply achieved by combining cases **(10)** and **(01)**. One example is to group compatible component pairs in a biweekly basis, combining examples 3 and 4.

A more realistic example, partially shown in Table 5, groups components by Price, and ignores the Compatibility attribute. For instance, virtual database version vd_1 groups all logical object versions with price $10. The original database with three database versions is seen as a database with four versions, where objects are furthermore organized in another way. In its (11) intension, the (10) dimension – version semantics – corresponds to versioning by price instead of by time; the (01) dimension – schema modification – ignores Compatibility. The new schema is given by

```
Virtual Class PriceClassif type
        tuple (name:string, time:real)
```

$(d_1,d_2,d_3$	vd_1	(O1,O2)	\$10	$\{<$ Disk1, $t_1 >, <$ Driver2, $t_1 >, <$ Driver2, $t_2 >, <$ Driver2, $t_3 > \}$
(d_2)	vd_2	(O1)	\$20	$\{<$ Disk1, $t_2 >\}$
(d_1, d_2, d_3)	vd_3	(O3)	\$30	$\{<$ Server1, $t_1 >, <$ Server1, $t_2 >, <$ Server1, $t_3 >\}$
(d_3)	vd_4	(O1)	\$40	$\{<$ Disk1, $t_3 >\}$

Table 5. Intension Modification – Change Version Semantics and Schema - View Type (11)

This extension can be expressed as follows, where *Flatten* is one of the basic multiversion view intension creation operators described in section 4.2 and partition is a reserved keyword for the result of a group-by expression. Flatten, as will be seen, makes all logical object versions available in a single database version to allow recombining them according to other criteria. Intuitively, it "flattens" all database versions into a single version, thus making them all visible at once.

```
create view Myview11 as
       group c in HardwareComponent
       from d in Flatten (MyDatabase)
       by (Price: c[d].Price)
       with (Components: select new PriceClassif (c[di])
                         from c[di] in partition
            )
```

The subsequent sections will now formalize multiversion view construction. Sections 4.1 and 4.2 define the operators that specify the view's intension (by respectively restructuring schema and version semantics). Section 5 shows how to create the extension by querying the source database.

4 Defining multiview intension

4.1 Defining the intension: schema structuration operators

This section analyzes operations that restructure the schema (**01**-view intensions). The goal of these operations is the construction of a *virtual* schema from the source schema of the multiversion database. We borrow [LDB97]'s definition of a database schema and define the schema as being formed by the classes, persistency roots, methods and inheritance graph of a database. Different proposals for schema restructuring operations have appeared in the literature, aiming at view construction (e.g., [TYI88, Run93, LDB97, RR97, LDB98]). However, most of these view definition mechanisms mix up the issue of schema definition with that of extension construction. We adopt the set of schema structuration operators defined by [RR97], extending them to encompass roots.

A virtual schema S_v restructures a *source* schema S by applying *update* operations on S [RR97]. These updates (insert, delete, modify) can be applied to all schema components (classes, roots, methods and inheritance hierarchy). The schema restructure operations

extension. Extension creation is the last part of a view definition, and is accomplished by queries – see Section 5.

The operations which we propose are therefore:

- Operations on classes: Class creation by specialization and/or generalization (adding and hiding attributes) of existing classes; Class modification, by adding, eliminating or modifying attributes, and Class elimination. All these operations have repercussions on the inheritance graph.
- Inheritance graph updates: edge elimination and creation
- Root creation or elimination
- Method creation or elimination

Other update operations (e.g., the addition, deletion or domain change in attributes of [RR97]) can be achieved through combinations of these operations. In what follows, we consider that the inheritance graph has a single root, which is the class *Object*, from which all class hierarchies descend (a standard assumption in object-oriented systems).

Persistency root updates. We extend the work of [RR97] by considering virtual persistency roots. Virtual extensions (objects in virtual classes) are attached to virtual roots. Virtual roots concern us only insofar as they play the role of allowing access to virtual objects. Virtual roots can either be imported from the source or defined by updates. The deletion of a root corresponds to hiding it in the view. The creation of a root corresponds to defining a new virtual root in the view, which will refer to virtual objects. The extension of a root is defined by a query (e.g., see example 3). When a root is deleted, then the objects attached to it are no longer accessible in the view, unless they are also reachable via another persistency root which is available in the view.

Class updates Schema updates must consider the effect on the class extension. Thus, the set of valid schema updates is restricted to those that present no ambiguity for handling class exensions. For this reason, just like [RR97, LDB97], when an update creates or modifies a class so that it generalizes several existing classes, we impose that it becomes a direct subclass of *Object*. The same considerations occur for class elimination (and thus elimination of edges from the inheritance graph). Yet another operation is class *hiding*, when the class is in the middle of the hierarchy. The class disappears from the view, but it still exists, and therefore its subclasses still inherit from it.

Method updates. The creation of new virtual classes pressuposes that new methods may be defined. Furthermore, methods may also be hidden from the user, or have their bodies redefined. In all cases, we assume that this is done explicitly when the virtual schema is created – adding, deleting or re-defining methods.

itance graph, two other operations may be defined exclusively on the graph – adding new edges or eliminating edges – without touching the class set. This may introduce multiple inheritance relationships (edge addition) or eliminate multiple or single inheritance (edge deletion).

4.2 Defining the intension: operators to reorganize version semantics

Section 4.1 enumerated the operators that specify the virtual schema (**01**-view intensions). Versioning semantics dictates the way database versions are constructed. This section defines the operators that allow this construction (**10**-view intensions). These operators combine sets of source database versions, constructing new sets of database versions, which can be progressively composed to form arbitrarily complex versions.

There are two types of reorganization operators: (a) operators which are applied to sets of database versions, ignoring their extensions; and (b) operators on extensions, which consider multiversion objects.

In the database version approach, classes, methods and objects are the versionable entities. Interdependencies between different entity versions of a multiversion entity are determined by the following operations:

- *logical(multiversion_entity,d)*, which returns the logical entity version of a multiversion entity contained in a given database version d:

$$logical(multiversion_entity, d) = < multiversion_entity, d >;$$

- *physical(logical_entity_version)*, which maps a logical entity version to one and only one physical entity version:

$$physical(logical_entity_version) = < physical_entity_version >;$$

- *ident(logical_entity_version)*, which returns the identifier of the multiversion entity for a given logical entity version:

$$ident(logical_entity_version) = < multiversion_entity >;$$

- *dV(logical_entity_version)*, which returns the database version identifier for a given logical entity version:

$$dV(logical_entity_version) = < d >;$$

- *value(logical_entity_version)*, which returns the value of a logical entity version contained in a given database version d:

$$value(logical_entity_version) = value(physical(logical_entity_version)) = < value > .$$

The operators that ignore extensions are set operators that combine source multiversion databases in order to build a set of database versions. The idea behind these operators is to create new sets of database versions from an existing set without considering the contents of these versions.

These operators, defined on Figure 3, rely on the concept of *membership*. A database version d_1 is *member* of a set of database versions $s=\{d_a, d_b, \ldots d_n\}$ if one of the database versions in s has the same identifier as d_1. The identifier of a database version is given by the function *ident*. Let *DVersion* denote a multiversion database.

$$\forall\ s\ \subset\ DVersion,\ \forall\ d \in DVersion,$$
$$member(d,\ s) = true\ if\ \exists\ d_i \in s \mid ident(d) = ident(d_i)\ else\ member(d,\ s) = false$$

Figure 2: Membership specification

Difference. Let s_1 and s_2 be two sets of source database versions. The *difference* operator (s_1 - s_2) returns a set of the database versions which are members of s_1 only.

Union. The *union* operator returns a set of database versions which are members of either or both the sets.

$\forall\ s_1,\ s_2 \subset DVersion,\ difference(s_1,\ s_2) = \{\ v \mid member(v,\ s_1) = true$ and $member(v,\ s_2) = false\ \}$
$\forall\ s_1,\ s_2 \subset DVersion,\ union(s_1,\ s_2) = \{\ v \mid member(v,\ s_1) = true$ or $member(v,\ s_2) = true\ \}$

Figure 3: Operators on Sets of Database Versions

Other set operators (e.g., intersect) can be derived from these operators, using standard set theory. Notice that here we do not need to consider object identifiers, just database version identifiers.

4.2.2 Operators applied on the extension of database versions

The previous operators did not consider individual multiversion objects. The intension construction operators applied on the extension of database versions are those that, instead, are applied on object values:

- a *select* operator, which selects (sets of) database versions based on the values of some of their objects;

in a database version denoted d_T ($intersect_v$, $difference_v$). The v suffix is added to differentiate these operators from the previous ones;

- set operators that allow to combine a set of database versions into a single database version denoted d_T, without ($union_v$) or with (*flatten*) value duplicates.
- and a *factor* operator which takes one single database version and partitions it into a set of new database versions according to a user-defined criterion.

These operators can be orthogonally combined to each other, since they are all applied to database versions to return one or several database versions.

Intuitively, the set operators are similar to standard set operators, extended to consider individual multiversion objects. The operators are based on simultaneously checking the value of a multiversion object in different database versions. We recall from section 3.1 that a multiversion object is shared by different database versions d_1,..,d_n if its logical object versions within d_1, ..., d_n share the same physical object version. The boolean function *share* performs this functionality.

$\forall\ mo \in MObject,\ \forall\ s \subset DVersion,\ \forall\ d_i,\ d_j \in s,$

share(mo, s) = true if physical(logical(mo, d_i)) = physical(logical(mo, d_j))

else share(mo, s) = false

Example 6. Consider again the database of section 3.3 (Table 1). Object $O2$ has the same value in database versions d_1 and d_2, corresponding to a given state of Driver2. If $s = \{\ d_1, d_2\ \}$, then $share(O2, s) = true$, but $share(O1, s) = false$, because object $O1$ (Disk1) had a change in price from d_1 to d_2.

Let $s = d_1,...,d_n$ be a set of source database versions. We now define the operators on s which require querying objects on database versions. These operators are specified in Figure 4.

Select. *Select* is an operator which, applied to a set of database versions s=$\{d_a, d_b, \ldots d_n\}$, returns a set $s_T \subset s$ which satisfies the *predicate* of the selection. A select operator corresponds to a complex query across a set of versions. Select predicates are expressed either on database versions or on their objects, similar to the VQL query language of [Abd97]. Furthermore, they can involve the use of all operators presented here, i,e., they can perform queries on the result of operator application to database versions.

Predicates are expressed by means of formulae of the form:

$$Q_{11}\ x_{11}, \ldots, Q_{1m}\ x_{1m}\ (p_1(x_{11}, \ldots, x_{1m})\theta\ \ldots\ \theta\ Q_{k1}\ x_{k1}, \ldots, Q_{km}\ x_{km}\ (p_k(x_{k1}, \ldots, x_{km})$$

where $Q_{11}, \ldots, Q_{kn}$ are quantifications over database versions or over objects; $p_1\ \ldots\ p_k$ are formulae respectively applied on these variables; θ is $\vee$ or $\wedge$.

$\forall s \subset DVersion\ select(s,F) = \{\ v \mid F(v) = true\} \vee \{\ set(d_1,\ldots,d_p) \mid F(d_1,\ldots,d_p) = true\}$
$intersect_v\ (d_a,\ d_b) = d_T$ with $extension\ (d_T) = \{logicalV(mo,\ d_T) \mid share(mo,\ d_a,\ d_b) = True\}$
$difference_v\ (d_a,\ d_b) = d_T$ with $extension(d_T) = \{logicalV(mo,\ d_T) \mid logicalV(mo,\ d_a) \in extension(d_a)$ and $share(mo,\ d_a,\ d_b) = False\}$
$flatten(s) = d_T$ with $extension(d_T) = \{\ logicalV(<vmo>,\ d_T) \mid \bigvee_{i \in 1,n} vmo = <mo, d_a> \in extension(d_i)\ \}$
$union_v(s) = d_T$ with $extension(d_T) = \{\ logicalV(<vmo>,\ d_T) \mid \bigcup_{i \in 1,n} vmo = <mo, d_a> \in extension(d_i)\ \}$
$forall\ v\ \in\ DVersion,\ factor(v,[vd_1 : expr_1, \ldots, vd_p : expr_p]) = \{d_i \mid i \in [1,\ldots,p]$ and $extension(d_i) = \{<<mo, v>, d_i> \mid physicalV(<<mo,v>,d_i>) \in result(expr_i)\}\ \}$

Figure 4: Operators which consider extensions

Difference$_v$. Let d_a, d_b be two source database versions. The *difference$_v$* operator returns a database version d_T whose extension is composed of all logical object versions on d_a which do not share a physical version with logical object versions on d_b.

Intersect$_v$. The *intersect$_v$* operator returns a database version d_T whose extension is composed of all multiversion objects whose logical versions on d_a and d_b share the same physical version.

Union$_v$. The *union$_v$* operator returns a database version d_T whose extension is defined by choosing all multiversion objects whose logical version appears in at least d_a or d_b. *Union$_v$* can be generalized to apply to sets of database versions $s = \{\ d_1 \ldots d_n\ \}$.

Flatten. The *flatten* operator on s returns a database version d_T whose extension contains all logical object versions which appear in any d_i in s, without eliminating duplicates.

Flatten and *Union$_v$* are operations which can be seen as regrouping all object versions within a single flat database. They are typically used as an intermediate step for reorganizing logical versions according to a new criterion. Maintaining or not duplicates depends on what the user would want to do next. Keeping duplicates means that the entire set of logical versions of each object is available, even when their value does not change through versions; eliminating duplicates means that the user is concerned only with state changes. In the first case, it is always possible to recover the original source database versions, by applying other operators (see *factor*) whereas in the second case this may not be possible. The result of either operation is one single database version where all logical versions are reflected at once, and to which the user can afterwards apply more complex operations.

One common use for *flatten* is motivated by the need to handle object evolution across versions in a single (flat) view. In this case, the user first applies a *flatten* operation and next structures the objects in lists according to their value along time. In a historical database, this is often implemented as a query through time (or, in temporal database terms, query through time slices [Sno95]).

without applying any aggregate function, and just nesting attributes in the other dimensions. More complex uses of the result of *flatten*, however, have no equivalent OLAP operation, since they derive from specific application needs. For instance, the user may want to create consistent configurations of HardwareComponents, obtained by constructing complex objects from the "flattened" view of the database.

Factor. The *factor* operator takes one database version, d and generates a set of new database versions d_T $=\{d_1, d_2, \dots d_p\}$ according to a user-defined criterion. A *user-defined criterion* c is a tuple $[vd_1 : expr_1, vd_2 : expr_2, \dots, vd_p : expr_p]$ where vd_i defines the name of the ith new virtual database version and $expr_i$ is an expression that defines its extension. This operator allows to partition one source database version into different database versions according to a new versioning semantics specified by criterion c – i.e., factoring the data.

Example 7. Let us now briefly re-examine the examples of Section 3.3. Example 2 – view of type (00) – is a simple case of view creation by direct application of a query, and is equivalent to the standard non-versioned view creation mechanism. Example 3 – view of type (01) – uses a schema restructuring operation of type class creation using attribute hiding (attribute *compatible* of class HardwareComponent) and attribute addition (new virtual attribute *cost*). Example 4 – view of type (10) – changes versioning semantics by applying a *Union* followed by operations which create versioning aggregates along time. Example 5 – view of type (11) – is more complex. It is the result of applying a *flatten* operation followed by an object reorganization using *factor* on Price (thus, two operations of type **(10)**). This is further combined to a (01) schema restructuring operation of type class creation by specialization, followed by class modification (CompWith attribute hiding).

We now present some additional examples to show further application of the operators, taken from the real life case study described in Appendix A.

Example 8. Let us start from the HardwareComponent class of Table 1. Suppose the user wants to find components which did not change throughout the initial evolution of the database. This is achieved by the *intersect* operator on the extensions:

$intersect_v(d_1,\, d_2) = d_T = \{O2, O3\ \}$.

Since operators can be repeatedly applied to sets of database versions, one can find out the intersection of all three database versions by applying $intersect_v$ twice, i.e.,

$intersect_v(d_3,\, intersect_v(d_1,\, d_2)) = d_T = \{O3\ \}$.

This shows that O3 (Server1) was the only Component which never changed. Similarly, the $difference_v$ operator would allow the user to find Components whose characteristics had changes from one period to another, for instance

$difference_v\ (d_1, d_2) = d_T = \{\ O1\ \}$

indicates that only Disk1 had some change from version d_1 to d_2.

Example 9. Assume that now the user wants to consider the evolution of each Component. This requires first applying the *flatten* operator on the database versions of HardwareCom-

flatten (d_1, d_2, d_3).

Flatten returns a single database version d_T, where all instances of $O1$, $O2$ and $O3$ appear in a (flat) state. Now, the versions of each object can be combined in a new class, to provide its evolution. For instance, all versions of O1 (Disk1) can be structured into a virtual multiversion object vO_1 , organized here as a list of all logical versions of $O1$:

$vO_1 = (< Disk1, t_1, \$10, O2 >, < Disk1, t_2, \$20, O2 >, < Disk1, t_3, \$40, O8 >)$.

To better illustrate *Factor*, we will now enhance the database with a new class – Manual – whose objects are manuals which describe the components' characteristics

```
class Manual type tuple
            (name: string,
             content: Text,
             describes: HardwareComponent,
             language: string)
```

A given HardwareComponent may have several manuals in different languages. Furthermore, a Component's manuals may be versioned whenever the Component is versioned. The next table instantiates class Manual for the three database versions $\{d_1,\ d_2,\ d_3\}$ of the running example of section 3.3.

Version id	Oid	Time	Name	Content	Describes	Language
	M1	t_1	Man1	Text1	O1	French
d_1	M2		Man2	Text2	O1	Port
	M3		Man3	Text3	O2	French
	M1	t_2	Man1	Text1	O1	French
d_2	M2		Man2	Text4	O1	Port
	M3		Man3	Text3	O2	French
	M1	t_3	Man1	Text1	O1	French
d_3	M3		Man3	Text3	O2	French
	M5		Man5	Text5	O3	French

Table 6. MANUAL Class instances

Example 10. Suppose the user wants to class manuals according to the language they are written in. This is similar to an OLAP cube rotation operation, along the Language axis, This can be achieved by the Factor operation as follows

$factor(d_1, [vd_1 : Manual.language = \text{“}French'',\ vd_2 : Manual.language = \text{“}Port''])$

Example 11. Factoring by language is useful when the user wants to find out which components are ready to be exported to which countries. In order to do this, the user must consider both database classes (HardwareComponent and Manual) into a single operation. The following expression will factor database version d_1 grouping components and manuals, according to the language:

$factor(d_1, [vd_a : Manual.language = \text{“}French'' \wedge Component = Manual.describes,$
$vd_b : Manual.language = \text{“}Port'' \wedge Component = Manual.describes])$

operator applied to database version d_1 would give origin to two virtual database versions:

vd_a = { HardwareComp = {O1, O2}, Manual = {M1, M3} }

vd_b = { HardwareComp = {O1}, Manual = {M2} }

e.g, the virtual database version vd_a contains components $O1$ and $O2$ and their respective manuals $M1$ and $M3$, which are written in language "French". Notice this has not changed the database schema, just reorganized the versions according to a distinct criterion, thus corresponding to a (10) operation.

We point out that, in cases like this, where more than one class is involved, there is no equivalent OLAP operation, unless the entire database (all classes together) is seen as a single unit, with all complex objects broken down into their individual components. If one is willing to accept this combination, then this example is similar to rotating this unit along the Language dimension followed by a roll-up along all attributes of the HardwareComponent class.

Example 12. Finally, consider now a more complex operation, which shows that the operators proposed extend beyond OLAP requirements. Suppose the user wants to reorganize versioning according to the type of object involved, separating Components from Manuals. This requires combining two operations: a Flatten, which will place all database objects into a single view, followed by a Factor operator which will version objects according to their nature:

$factor(flatten(d_1, d_2, d_3), [vd_a : d = HarwareComponent, vd_b : d = Manual])$

As a result, virtual database version vd_a will contain all HardwareComponent objects through time, and vd_b all Manual objects through time. This is a partitioning ultimately directed by types and not by values. These two versions (vd_a and vd_b) can themselves originate new sets of database versions by application of another Factor operation:

$factor(vd_a, (vd_{a1} : Time = t_1, vd_{a2} : Time = t_2, vd_{a3}; Time = t_3))$

$factor(vd_b, (vd_{b1} : Time = t_1, vd_{b2} : Time = t_2, vd_{b3}; Time = t_3))$

$\{vd_{a1}, vd_{a2}, vd_{a3}\}$ contains only Components; $\{vd_{b1}, vd_{b2}, vd_{b3}\}$ is a set of database versions containing only Manuals. Different users can work on each set separately, while their links and original versioning semantics are still maintained by the underlying database version mechanism.

5 Defining the virtual extension in a multiversion view

Up to now, we were concerned with multiversion view intensions. Here, we discuss view extensions, providing means for mapping virtual objects and virtual database versions back to the original source objects and versions. This is a classical problem in object-oriented view extension management.

In order to specify identifiers of virtual multiversion objects, we borrow the concept of *referent* of [LDB97]. Referents are the means through which view objects are traced to their origin in the source database. Intuitively, referents are identifiers of virtual objects.

A *referent* is a a pair (object identifier, class name) that allows associating a virtual

more than one class. Referents point to an object *as it is seen and as it behaves* in a given class (italics taken from [LDB97]). We use the same concept to define identifiers of logical objects within multiversion views,

Virtual multiversion object identifiers. We recall that a logical object is denoted by $< mo, d >$, where mo is the multiversion object identifier and d is the identifier of a database version. Logical *virtual* objects are also defined in the same way, i.e, $< vmo, vd >$, where vmo is the identifier of a virtual logical multiversion object and vd is a virtual database version identifier. A virtual logical object may, however, be constructed from several source objects, and thus the identifier $< vmo, vd >$ may become very complex, since both vmo and vd may themselves be complex identifiers.

In order to specify complex identifiers, we again turn to [LDB97], whose complex referents are represented by ((o_1,c_1), $\wedge \ldots \wedge$ (o_n,c_n)), denoting that a virtual object is built from the set of source objects $\{ (o_i,c_i) \}$. In a similar manner, virtual multiversion objects may have a complex identifier. The expression:

$$< vmo, vd > = < (< mo_1,\ d_1 >, \ldots < mo_n,\ d_j >),\ vd >,$$

denotes a logical virtual object in the virtual database version vd. This logical virtual object is constructed from source logical multiversion objects $\{ < mo_i, d_i > \}$.

Example 13. To illustrate this issue, consider a view containing only one virtual database version, called vd_1, mapped directly from source database version d_3 in the HardwareComponent database of section 3.3 (i.e., $vd_1 = d_3$). The mapping between each object in this view and the source database is given by

$$< vO_i, vd_1 > = << O_i, d_3 >, vd_1 >,$$

where vO_i denotes a virtual multiversion object. Intuitively, this denotes that virtual multiversion object vO_i is constructed from the source object $< O_i, d_3 >$; and that it appears in the virtual database version vd_1.

To extend this to a complex virtual object identifier, let $s = \{d_1 \ldots d_j\}$ be a set of database versions, mo be a multiversion object in s and consider the result of $d_T = \mathit{flatten}\,(d_1, \ldots, d_j)$. Each logical version of mo will appear in d_T, with identifiers constructed by

$$<< mo,\ d_1 >, d_T >, \ldots << mo,\ d_j >,\ d_T >,$$

The user can next combine these versions according to new needs, as we have seen in the previous section. In particular, as shown in Example 9, the user can combine all versions of a given object in a list to construct historical chains, e.g.,

$$vO1 = < (<< o_1,\ d_1 >, d_T >, \ldots < o_1,\ d_j >,\ d_T >)\ , d_{Ta} >$$

all logical versions of o_1 which appear in the "flattened" view d_T, and combining them in a list available through another view d_{Ta}.

This type of stepwise creation of identifiers allows tracking virtual objects back to their source. To complete this task the view mechanism must know which operators were used. In order to do this, we use virtual graphs, described next, which provide a means for completely recreating the construction history of a multiversion view. Figure 4 presents the operators of section 4.2 together with their extension definition in terms of identifiers.

Virtual graphs - tracking version semantics *Virtual graphs* record the sequence of steps performed in order to create the version semantics part of the intension of multiversion views == i.e., for views of type **(10)** or **(11)**. In this sense, these graphs serve the same purpose as, for instance, query graphs in database language representation. From an implementation point of view, these graphs allow keeping track of version and logical object identifiers, and thus ensure the mapping between a virtual object id and its source object identifiers.

A *virtual graph* is a directed acyclic graph whose nodes are logical database versions. A directed edge (d_a,d_b) in a virtual graph denotes that d_a is one of the source database versions used to construct d_b. Node labels specify the (10) intension operation used to construct the corresponding database version, and allow uniquely determining how the corresponding database version identifier is to be computed. Intermediate database version identifiers may be attached as tags to each node.

Figure 5 shows an example of the graph that builds a temporary multiversion database from a set of database versions $S = \{d_1 \ldots d_3\}$. Node vd_1 identifies an intermediate operation: $vd_1 =$ flatten (d_1, d_2, d_3). Database versions d_a and d_b have been created by applying a *factor* operator to dv_1. This figure, in fact, graphically reproduces the operations performed in the beginning of Example 12.

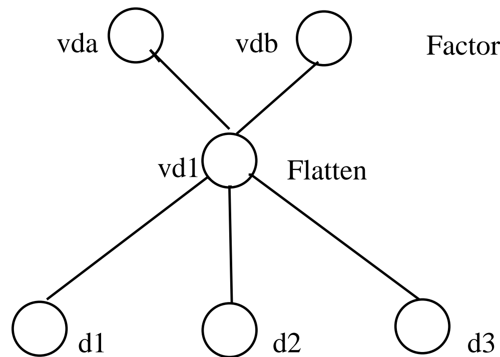

Figure 5: Virtual graph

The function *sourceDv* determines the identifier of the source database version which must be considered. This function is polymorphic; as a result, it is defined for each operator and for a sequence of operators. Let $sequence = x_1\ \theta_1\ x_2\ \ldots\ x_n$ where x_i may be a database version or a set of database versions and θ_i the operators previously defined. The function

dv identifier resulting of operators on set of d_s
$\forall\ \theta \in \{difference, union\}$,sourceDv$(\theta(s_1,\ s_2)) = \{\ sourceDv(d_i)\ \mid\ d_i \in (s_1,\ s_2)\}$
sourceDv(element($\{dv\}$))= sourceDv(dv)

dv identifier resulting of operators on d_s
sourceDv(dv) = dv
$\forall\ \theta \in \{intersect_v, difference_v, union_v\}$, *sourceDv*$(\theta(d_1,\ d_2))$ = *sourceDv*(d_1)
sourceDv(flatten(s))= $\bigvee_{i \in [1 \ldots n]}$ *sourceDv*$(d_i)\ \mid\ d_i \in s$
sourceDv(factor(dv,$[cr_1 : expr_1, \ldots, cr_p : expr_p]$*)= sourceDv(dv)*

Figure 6: Retrieving identifiers of source database versions

Construction of view extensions View extensions are constructed by queries, as is the standard practice for all view mechanisms. In our case, these queries can be posed using VQL [Abd97], extended with the operations defined in this paper.

Managing view extensions require the definition of virtual *oids*– i.e., to allow users to manipulate virtual objects. Virtual identifier construction is achieved by combining the notion of referent (complex multiversion object identifier) to the virtual graph and *sourceDv* operator.

Example 14. Consider the graph of figure 5. The figure stands for Example 12, which reconfigured the original database to provide views where data are versioned along two axes: Components and Manuals. We recall this example refers to the expression

$factor(flatten(d_1, d_2, d_3), [vd_a : d = HarwareComponent, vd_b : d = Manual])$

Let us just consider objects O1 and M1, and keep track of their evolution. These objects first go through a *flatten* operation which will make them visible in a virtual database version vd_1, with identifiers $vO_{i1} = \{ < (\text{O1}, d_i),\ vd_1 > \}$, i.e.

$\{ < (\text{O1}, d_1),\ vd_1 >, < (\text{O1},\ d_2),\ vd_1 >, < (\text{O1},\ d3)\ ,\ vd_1 > \}$

Analogously, the several logical versions of $M1$ appear in virtual objects vM_{i1}, i.e.,

$\{ < (\text{M1}, d_1),\ vd_1 >, < (\text{M1},\ d_2),\ vd_1 >, < (\text{M1},\ d3)\ ,\ vd_1 > \}$

Next, virtual database version vd_1 is factored into two virtual database versions vd_a and vd_b. Objects vO_{i1} will be the sources of virtual multiversion objects in vd_a, and vM_{i1} will be the sources of virtual multiversion objects in vd_b. Logical object versions in vd_a and vd_b are identified by

$$< vO_{i1},\ vd_a > \text{ and } < vM_{i1},\ vd_b >, \text{ e.g.,}$$

the logical version of $O1$ in d_1, applying some operation that allows it to appear in virtual database version vd_1 and finally appear by some other transformation in vd_a. The identification of which operations were applied is recorded in the graph and the source objects are found out by *sourceDv*.

Thus, given the virtual multiversion identifier *MyObject*, in vd_a, its source objects can be traced back to the original database by traversing the graph in reverse order, using *sourceDv*, applied to the complex virtual identifier of MyObject.

6 Implementation Issues

This section presents an architecture for implementing multiversion views in a general context, and describes the present implementation stage of a prototype which specializes this architecture for a specific database system. The general architecture, shown in Figure 7, is based on the *Java Database Binding* tools [BST98] developed by *Ardent Software*. We just give enough details in order to show the implementation context. For a more detailed description of this architecture, see [BC00].

The goal of the *Java Database Binding* tools is to provide Java with persistent capabilities using the *O2* DBMS [BDK92] or any relational DBMS. These tools already allow a limited form of view specification, since they permit schema restructuration by attribute hiding. Our prototype is based on this same architecture, but simplifying some of its details. In order to implement multiversion views using this architecture, the modules represented in shadowed boxes should be implemented. Multiversion views are thus defined in terms of Java classes, on top of any database management system, where instances are mapped to underlying database classes and objects. A multiversion view is therefore manipulated throughout Java interfaces, and all code is translated into database queries via the *Mapping* blocks of the architecture.

The architecture is composed of three main blocks: the *Development Environment*, which allows users to specify multiversion views and translates user requests into DBMS requests and vice-versa; the *Runtime* block, which is DBMS-independent and provides all the services needed to implement ODMG interfaces; and the *Database Connectivity* facilities, which allows access to different database management systems.

Data flow in the architecture is indicated by the arrows. Users' specification of multiversion views are *preprocessed*, *compiled* in Java and next mapped to a specification of underlying database classes and objects by the *Mapping tools*. Once this mapping is performed, the resulting code is handed to the *Bytecode* processing, which adds interfaces and methods required to read and write objects to/from the DBMS in terms of Java bytecode.

At runtime, all is handled by Java applications which are enhanced with postprocessed Java classes. The management of identifiers of virtual multiversion objects is done at this stage. The *Virtual multiversion object cache* is where database objects are initially retrieved, together with their identifiers. This cache binds Java objects to database objects, using the *multiversion referents* mapping mechanism described in section 5.

The central component of the architecture is the set of *Mapping tools*, which map compiled Java classes into the underlying DBMS, according to predefined mapping rules and as-

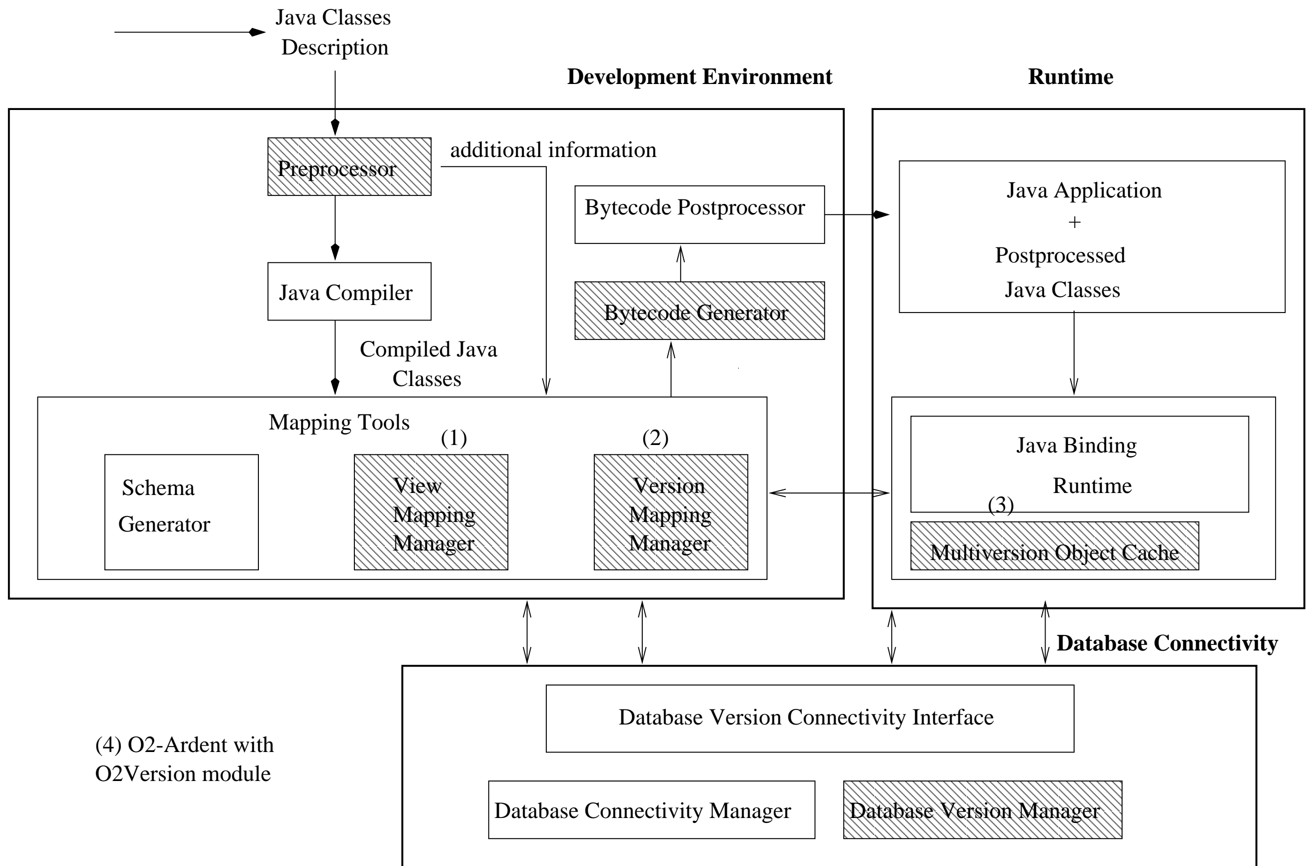

Figure 7: General architecture for implementing multiversion views

inheritance links, method signatures and persistency roots). Their role is to provide the means for transforming Java classes (which are our virtual multiversion view classes) into the physical database. Once this mapping is established, the new classes are transformed into Java bytecode and passed on to applications which will use the view.

The access to the database is performed in three steps. First the *Mapping manager* transforms a user or application request into a DBMS operation; next, the *Mapping manager* creates database queries according to the mapping metadata; finally, the *Database connectivity* block sends these queries to the DBMS and returns the result to the *Runtime cache* block.

The implementation of our prototype, in construction, is centered in the boxes numbered (1) through (3) in the figure. First, it is not geared towards a general database environment; instead, it uses the *O2-Ardent* DBMS, which implements a version mechanism based on the Database Version mechanism of [CJ90], in a module called *O2Version* [O2-98]. Thus, we do not need to implement the Database Version Manager block of the figure, since it is already available in this commercial product. As a consequence, the bottom block (*Database Connectivity*) is replaced in our prototype by *O2-Ardent*, as indicated by number (4) in the figure. In the second place, we do not provide automatic bytecode transformation; rather, the multiversion view operations of Section 4 are transformed by the *Mapping tools* into a set of operations which are part of the interfaces of the Java classes managed by *Runtime*. These interfaces are not generated automatically - i.e., there exists a specific code for each multiversion view specification operation, which is handled by the *View* and *Version mapping* managers.

The *Version mapping manager* translates JAVA operations on multiversion view objects into O2Version code. It also implements the multiversion view intension operations of section 4.2.1. Queries on versions use the VQL language of [Abd97], which has been implemented in a prototype on top of O2Version [Cha99]. The *View mapping manager* handles the remaining intension operations.

View and *Version managers* are kept apart in the architecture to allow handling of multiversion objects by a Java application, regardless of view needs. Thus, for instance, **(00)** views do not activate the *View mapping generator*, since they are just the result of selecting parts of the multiversion database schema, and retrieving the corresponding instances. In this case, user requests are mapped directly via the *Version Mapping Manager* to the underlying O2Version module.

7 Conclusions and Directions for Future Work

This paper presented a view mechanism – the multiversion view mechanism – that allows handling multiple versions of objects through views, thereby combining properties of version and view mechanisms. This combination is in itself a contribution, since versions and views have so far been treated in isolated contexts by the database community. In several stages of the paper we contrasted operations on multiversion views with OLAP or temporal databases, to show the examples from a different perspective. The goal was also to further motivate

these domains.

The main advantages of the framework proposed here are:

- Clear construction rules. The construction of multiversion views separates the issues of intension and extension definition, thereby avoiding problems that frequently occur in view management mechanisms.

- Closeness to the user's perspective. Whereas other view mechanisms force upon the users a non-versioned perspective of the world our proposal eliminates this constraint, allowing users to construct views containing as many versions of the world as desired.

This type of approach is of immediate application in several contexts: temporal databases (if versioning is restricted to time), OLAP (for relational databases, or situations where only one class is considered), and in cooperative design environments in general.

First, since Time is a frequent versioning criterion, our framework can be seen as an alternative means of handling temporal databases through views, as shown for instance in Examples 4 and 9. Nevertheless, we cover other situations not considered in temporal databases, since versioning can encompass attributes other than time. Moreover, several versions of a given object can exist in a single time period (e.g., when alternatives are created in design applications). Handling this type of situation is not possible from a temporal database perspective.

The concept of OLAP originates, among others, from statistical databases and statistical table handling. OLAP data are organized in tables, where each instance is considered to be a multi-dimensional description of a real world entity. Tables can be reorganized according to several criteria, and dimensions may be aggregated or broken down (i.e., providing distinct views of the database). Transplanted to our framework, this can be performed by creating views through *modifying versioning semantics*. For instance, Example 10 is a canonical instance of an OLAP operation in an object oriented context. In other words, if versioning is performed along single attributes, and the database schema is relatively simple, each attribute can be seen, up to a certain level, as an OLAP dimension. Therefore, we believe that, if we restrain ourselves to relational databases and non-complex dimensions, our operators can replicate the basic needs (e.g., [Sho97] – roll-up, drill-down, rotate and select).

On the other hand, as we also point out, our operations extend beyond OLAP, since they respond to a different set of users' needs (e.g., see Examples 11 and 12). Furthermore, contrary to OLAP underlying assumptions, we allow multiple attribute hierarchies per dimension. Handling of alternatives is yet another case which has no corresponding OLAP counterpart, since this requires a means of linking different alternatives of a single object (and can only be achieved through the versioning mechanism).

The next steps in this research consist in finishing the implementation of the prototype. Another issue to which we will dedicate attention is that of updatable multiversion views.

This work was developed within the binational cooperation program CNPq (Brazil) - CNRS (France). It was furthermore partially supported by Brazilian grants from CNPq and FAPESP, by PRONEX project SAI (Advanced Information Systems) of MCT-Brazil. We thank Marie-José Blin for providing us with a real life example and careful reading of this text.

References

[AB91] S. Abiteboul and A. Bonner. Objects and Views. In *Proc. SIGMOD Conference*, pages 238–247, 1991.

[Abd97] T. Abdessalem. *"Approche des versions de bases de données: représentation et interrogation des versions"*. PhD thesis, Université Paris IX Dauphine, 1997. Supervisor, Genevieve Jomier.

[AGM+97] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, and Y. Zhuge. Views for Semistructured Data. In *Proc International Workshop on Management of Semistructured Data*, 1997.

[BC00] M. Bellosta and W. Cellary. Consistent Versioning of Java Schemata and their Extensions. Submitted for publication, initial version, 2000.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-oriented Database System*. Data Management Systems. Morgan Kaufmann Publishers, 1992.

[Ber92] E. Bertino. Data Hiding and Security in Object-Oriented Databases . In *Proc IEEE Data Engineering Conference*, pages 338–347, 1992.

[BFK95] P. Brèche, F. Ferrandina, and M. Kuklok. Simulation of Schema Change Using Views. In *Proc DEXA Conference*, Springer Verlag Lecture Notes in Computer Science 978, 1995.

[Bla91] H. Blanken. Implementing Version Support for Complex Objects. *Data and Knowledge Engineering*, pages 1–25, 1991.

[BLP95] M-J Blin, J. Lisicki, and I. G. Puddy. Improving Configuration Management for Complex Open Systems. *ICL Systems Journal*, 10(1), 1995. Also at http://www.icl.com/sjournal/v10i1.

[BLT86] J. Blakeley, P. Larson, and F. Tompa. Efficiently Updating Materialized Views. In *Proc. ACM SIGMOD Conference*, pages 66–71, 1986.

for Object Databases. In *Proceedings of International Symposium on Object Techniques for Advanced Software, Kanazawa, Japan, November 1993.*, pages 220–235, Kanazawa,Japan, November 1993.

[BST98] M. Bellosta, C. Souza, and E. Theroude. Mapping Relations to Java Objects. Technical report, O2-Ardent, 1998. OCBA Project D1.2.2 3GL Bindings.

[BW99] D. Barbara and X. Wu. The Role of Approximations in Maintaining and Using Aggregate Views. *IEEE Data Engineering Bulletin*, 22(4):15–21, 1999.

[Cha99] W. Chaoui-Kerkeni. Langages d'Interrogation de BD à Versions - Etat de l'art et Mise en Oeuvre avec l'approche VBD. Master's thesis, Université Paris IX Dauphine, 1999.

[CJ90] W. Cellary and G. Jomier. Consistency of Versions in Object-Oriented Databases. In *Proc. 16th VLDB*, pages 432–441, 1990.

[CK86] Hong-Tai Chou and Won Kim. A unifying framework for version control in a CAD environment. In *Proc. 12th VLDB Conference*, pages 336–344, Kyoto, August 1986.

[DL96] A. Dattolo and V. Loia. Collaborative Version Control in a Agent-based Hypertext Environment . *Information Systems*, 21(2):127–145, 1996.

[FD96] E. Fontana and Y. Dennebouy. Lazy Propagation of Multiple Schema Changes Using Timestamped Versions . *Ingénierie des Systèmes d'Information*, 4(1):9–33, 1996.

[FR97] G. Fahl and T. Risch. Query Processing over Object Views of Relational Data. *The VLDB Journal*, 6:261–281, 1997.

[FSS79] A. Furtado, K. Sevcik, and C. Santos. Permitting updates through views of databases. *Information Systems*, 4:269–283, 1979.

[GJ94] S. Gancarski and G. Jomier. Managing Entity Versions within their Contexts: a Formal Approach. In *5th International Conference,Database and Expert Systems Applications DEXA94*, pages 400–409, 1994.

[Huy98] N. Huyn. Multiple View Self-maintenance in Data Warehousing Environments. In *Proceedings VLDB 1998*, pages 26–35, 1998.

[ICL00] ICL. ICL site. http://www.icl.com, as of January 2000, 2000.

[Kat90] R. H. Katz. Toward a Unified Framework for Version Modelling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, 1990.

[KC88] W. Kim and H-T Chou. Versions of Schema for Object-oriented Databases. In *Proc. 14th VLDB Conference*, pages 148–159, 1988.

[KS92] W. Kafer and H. Schoning. Mapping a Version Model to a Complex-Object Data Model . In *Proc IEEE Data Engineering Conference*, pages 348–357, 1992.

[KSW86] P. Klahold, G. Schlageter, and W. Wilkes. A General Model for Version Management in Databases. In *Proc XII VLDB*, pages 319–327, 1986.

[LDB97] Z. Lacroix, C. Delobel, and P. Brèche. Object Views Constructed with an Object Algebra. In *Proc.13e Journées Bases de Données Avancées*, pages 219–239, 1997.

[LDB98] Z. Lacroix, C. Delobel, and P. Brèche. Object Views. *Networking and Information Systems Journal*, pages 231–250, 1998.

[LST98] F. Llirbat, E. Simon, and D. Tombroff. Using Versions in Update Transactions: applications in Integrity Checking. In *Proceedings VLDB 1998*, pages 96–105, 1998.

[Man00] M. Manouvrier. *Objets Similaires de Grande Taille dans les Bases de Données*. PhD thesis, Université Paris IX Dauphine (France), january 2000.

[MBJ96] C. B. Medeiros, M. Bellosta, and G. Jomier. Managing Multiple Representations of Georeferenced Elements. In IEEE, editor, *Proc. 7th DEXA96 Workshop*, pages 364–371, 1996.

[MM91] J-C. Mamou and C. B. Medeiros. Interactive Manipulation of Object-Oriented Views. In *Proc International IEEE Conference Data Engineering*, pages 60–69, 1991.

[MP96] R. Motschnig-Pitrik. Requirements and Comparison of View Mechanisms for Object-oriented Databases. *Information Systems*, 21(3):229–252, 1996.

[Nov95] G. Novak. Creation of Views for Reuse of Software with Different Data Representations. *IEEE Transactions on Software Engineering*, 21(12):993–1005, 1995.

[O2-98] O2-Ardent, Boulder, CO. *O2Version Reference Manual Rel. 5.0*, February 1998.

[RR97] Y. Ra and E. Rundensteiner. A Transparent Schema-evolution System Based on Object-oriented View Technology. *TKDE*, 9(4):600–623, 1997.

[Run93] E. Rundensteiner. Design Tool Integration Using Object-Oriented Database Views. In *Proc IEEE Intl. Conference on Computer-aided Design*, pages 104–107, 1993.

[SAD94] C. Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In *Proceedings EDBT 1994*, pages 81–94, 1994.

[Sci91] E. Sciore. Multidimensional Version for Object-Oriented Databases. In *Proc. 16th VLDB Conference*, pages 355–370, 1991.

Proc ACM PODS, pages 185–196, 1997.

[Sno95] R. Snodgrass. *Modern Database Systems: The Object Model, Interoperability, and Beyond*, chapter Temporal Object-oriented Databases: a Critical Comparison, pages 386–408. ACM Press, 1995.

[TO96] G. Talens and C. Oussalah. Version d'objets pour l'ingénierie. *Technique et Science Informatiques*, 15(2):145–178, 1996.

[TYI88] K. Tanaka, M. Yoshikawa, and K. Ishihara. Schema virtualization in object-oriented databases. In *Proc IEEE 4th Conference on Data Engineering*, pages 23–30, 1988.

[WR94] W. Wieczerzycki and J. Rykowski. Version Support for CAD/CASE Databases. In *Proceedings East/West Database Workshop*, Workshops in Computing, pages 249–260. Springer Verlag, 1994.

[YKL98] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environments. In *Proceedings VLDB 1998*, pages 136–145, 1998.

[YPS95] X. Ye, C. Parent, and S. Spaccapietra. On the Specification of Views in DOOD Systems. In *Proceedings Fourth International DOOD Conference*, pages 539–550, Singapore, 1995.

The paper uses a small example (starting in Section 3.3) to illustrate multiversion view creation and management. This example was extracted from a real case study of management of configuration of complex information systems. This Appendix gives an overview of this case study, a typical context in which multiversion views can make a difference. For more details on this system and its design see [BLP95].

The application concerns the management of a system called ISS400 of the British information technology firm ICL [ICL00]. ISS400 is a system that handles the configuration of software and hardware for medium to large scale retail points of sale (e.g., super or hypermarkets, department stores). It is highly configurable, and can be adapted to all kinds of client that deal with retail outlets and business operations. ICL development and support teams install and maintain at each client's site the most appropriate hardware and software configuration. This is achieved by selecting, from a large component version library, the appropriate version of each software and hardware component to customize and install at the client's sites.

The example database of this paper is extracted from the ISS400 component library. View creation examples borrow from typical needs of ICL's development and support teams. The component library consists of both hardware and software descriptions, and associated documentation. As well, it contains description of services that ICL teams can provide.

Examples of *hardware components* include servers, peripherals, workstations, tills, networks, power supply units, network cards, bar code readers, and so on. A component may have several subcomponents, e.g., a given till may be composed of main board, secondary boards, keyboard, displays, printers, scanners, cables and interface black boxes. *Software components* include source modules, object modules, data specifications and interface look and feel definitions. *Documentation components* include external customer manuals as well as internal specification files describing support and development tasks, such as requirements, design blueprints and test plans. The *service components* include installation, support, consultancy and training. Each service module requires specific skills and support documentation.

In particular, over 15,000 distinct modules are available in the library, each with approximately 10 versions. Software, hardware, documentation and service components are furthermore subject to complex dependencies which link program versions to hardware, documentation and services, defining valid configurations thereof.

A complex configuration may include, for instance, hardware, the operating system environment, applications, peripherals, cabling, interfaces etc. all of which interact. Any component may be versioned according to its temporal technological evolution, to customer variants and to the business processes it is designed to support and work with. The definition of a configuration for a given client is equivalent to the definition of either a (00) view or a (01) view containing only one version of each object (i.e., one particular version of each relevant component is picked out). However, configuration creation may require several operations on all kinds of multiversion views.

Since ICL has clients in over 40 countries, and a client may operate in many countries, this has an impact on configuration management. For instance, a given supermarked chain may require several types of tills (e.g., for distinct shops or countries), and therefore distinct

See, in particular, examples numbered 9 onwards in the text for typical configuration management requests. Development and installation teams must thus respond to two challenges - environment evolution due to changes or hardware/software error corrections.

Components have several versions related to their history, and several variants related to the language, country market or customer specific requirements. Minor changes in a version generate so-called releases. Each component is identified by internal ICL codes, which also determine its version (similar to multiversion object identifiers) - e.g., TA_IT_10.0 is the initial version of a till application, further releases are identified by TA_IT_10.1, TA_IT_10.2 and so on. Consider for instance till application TA_IT_10.0. It runs on point of sale 9520/150R3, which contains among others the components "CPU board R3" and "Pin pad Dassault LCM 103". It may also run on point of sale 9520/150R4, which differs from the previous point of sale by using "CPU board R4". The till application uses version 3.1 of software "FORTE", which runs on either R3 or R4 CPU boards.

In order to minimize the number of parameters to be handled during configuration specification, ISS400 is structured in four layers – hardware, operating system, base software and application programs. Each layer is handled by a distinct ICL team. To create a new system version for a customer, each team creates a version of a related specific part of the system either by reuse, selection and customization of existing component versions or by development of new component versions. Each of these parts is verified in unit tests, followed by integration and validation before installation. Test documentation is integrated to this configuration. The set of steps previous to and during the installation procedure are also subject to constraints of team member expertise and availability (who can do what, where and when). Again, assigning a team to a task can be simplified by creating specific multiversion views that involve people, constraints and configurations.

The modeling of ISS400 according to the database version model can be seen in [BLP95]. To finish this abridged description of the system, we include a few hardware, software and configuration constraints, which show further need for multiversion views:

- if a configuration contains a check reader of a pin pad, then it must also contain a network card X25;
- a hardware integration configuration must include two servers and at least one till and a hub;
- version 3.1 of FORTE software has been conceived to be installed together with either R3 or R4 CPU boards;
- program TCF must be installed in version AU33_NDBoard3 if the installation has an NCR scanner, a Dassault pin board, a 1200 baud Dassault card reader and no F1 keyboard;
- configuration AU_04_V1L5.7A is compatible with the following: (a) FX486/50 processors, french VGA screen, 2 425MB disks, 32MB RAM; (b)9520/150 family point of sale tills work with an AU keyboard, cash drawer, RS232 printer interfaces; (c) etc ...