SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **29**(5), 479–500 (1999)

Object Versioning in an ODMG-compliant Object Database System

SANG-WON LEE*AND HYOUNG-JOO KIM
Department of Computer Engineering, Seoul National University, Shilim-Dong Gwanak-Gu, Seoul 151-742, Korea
(email: {swlee,hjk}@oopsla.snu.ac.kr)

SUMMARY

In this paper, we describe the design and implementation of the SOP Object Version System which is built upon an object database system, called SOP (*S*NU *O*DBMS *P*latform). SOP is fully compliant with ODMG-93, a standard for object databases proposed by the Object Database Management Group (ODMG). To support object versioning in an ODMG C++ OML (Object Manipulation Language) environment, the SOP Object Version System provides a set of APIs (Application Programming Interfaces) as a class library. One important design goal of the SOP Object Version System was to achieve full compliance with ODMG-93 standard. That is, in designing the class library, we tried to utilize the ODMG-93 C++ OML interface as much as possible, while preserving its semantics as defined in ODMG-93. Our object version model follows the basic functionalities of mainstream object version models. Although this approach burdens programmers with the management of the application-specific version policy, it helps users to apply the SOP Object Version System to diverse application areas. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: object version; object database systems; ODMG-93 data model

INTRODUCTION

An object in a database represents a real world entity which the database is to model, and many database applications require database objects to model the various aspects of real world entities, such as the temporal history of an entity and its alternative designs.

With the advent of the object-oriented data model in the mid 1980s, the object versioning concept† has received much attention from new database application areas including CAD/CAM, CASE and temporal database areas [3–8]. Hence, almost every object database system (ODBMS), for example, Exodus, Itasca, ObjectStore, Objectivity, O2 and Ode, support versioning concepts of their own [9–15]. Recently, the necessity of object versions has been strongly re-motivated in several new object database application areas including Repository [8,16] and WWW [17,18].

*Correspondence to: S.-W. Lee, Department of Computer Engineering, Seoul National University, Shilim-Dong Gwanak-Gu, Seoul 151-742, Korea.
Contract/grant sponsor: Research Institute of Engineering Science.
Contract/grant sponsor: STEPI, ‘A Development on Object-Oriented Component Technology for a Web Transaction Server’; Contract/grant number: 97-058.
†Strictly speaking, the version concept discussed in this paper is about *user-level* versions. *System-level* version concepts, such as for concurrency control [1] and long duration transaction [2] is beyond the scope of this paper.

CCC 0038–0644/99/050479–22$17.50

Received 18 May 1998
Revised 28 September 1998
Accepted 7 January 1999

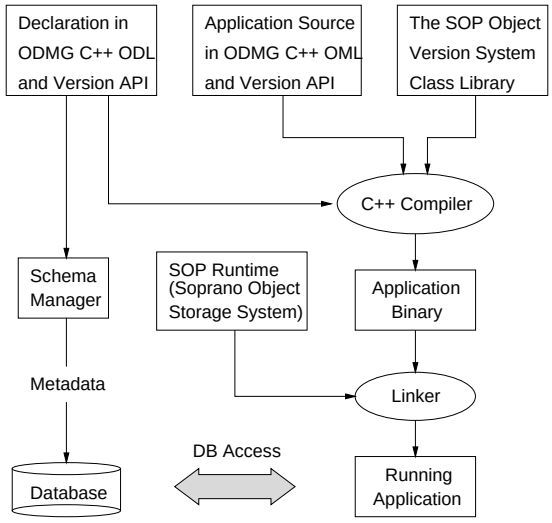

Figure 1. Using the SOP system and its object version system

This paper deals with the design and implementation of version facilities of the SOP Object Version System, which extends the SOP (**S**NU **O**DBMS **P**latform) system [19], which was developed from scratch at Seoul National University and is fully compliant with ODMG-93, the *de facto* object database standard proposed by the Object Data Management Group (ODMG) [20].

In the early 1990s, ODBMS vendors recognized that the lack of a standard for object databases was a major limitation to their products' wide acceptance in the market. They learned that much of the relational database's success was attributable to its standard, SQL. Hence, they organized the ODMG in the summer of 1991 so as to make an object database standard allowing source code portability and interoperability between different ODBMS products. Since the publication of release 1.0, ODMG-93 [21] in the late 1993, the ODMG made several enhancements to the standard, release 1.1, 1.2 and 2.0 (as of 1997). The main components of the standards include (1) Object Model, the common data model to be supported by ODBMSs, (2) Object Definition Language (ODL), the specification languages for ODBMSs, (3) Object Query Language (OQL), a declarative language for querying and updating database objects, and (4) Object Manipulation Languages (OMLs), the bindings of ODBMSs to the popular object oriented languages C++, Smalltalk and Java (new in release 2.0 [22]). We assume in this paper that readers are familiar with the ODMG-93 standard, especially ODMG C++ OML and the C++ programming language [23].

The SOP system supports ODL, OQL and C++ ODL/OML binding of the ODMG-93 standard (currently based on release 1.2 [20]). SOP consists of several main modules, including an object storage system *Soprano* [24,25], a cost-based query processor and optimizer *Sopoqles*, a schema manager module, and a C++ ODL/OML preprocessor *LOD** [26,27].

Application programmers can use the version mechanism of SOP (which we refer to as the SOP Object Version System) in the ODMG C++ ODL/OML bindings [20]. Figure 1 illustrates the use of the SOP ODBMS and its Object Version System. The SOP Object Version System is provided to users as a class library, and an application program using the

version functionality is then compiled together with the class library and then linked with the SOP runtime module, the *Soprano* object storage system, to produce the running application. The software architecture in Figure 1 follows the approach proposed by the ODMG.

The SOP Object Version System, as in the form of a class library, is loosely coupled with the SOP runtime module, and is thus easily portable to other ODMG-compliant ODBMSs. Currently, the ODMG standard does not include any specifications about object versioning. We believe that the approach to object versioning taken by the SOP Object Version System is a natural extension of the ODMG standard, and thus we hope that our approach can be a candidate for object versioning in the ODMG standard.

The rest of this paper describes the versioning functionality of the SOP Object Version System. The following sections deal with its design goals, the SOP object version model, a set of API (Application Program Interface) calls and its usage example, implementation details, comparisons with other version mechanisms, and conclusion. Note that the focus of this paper is on object versioning. Many ODBMS applications also require the functionality of schema versioning which tracks the evolution of the definition of classes and class hierarchy. Although object versioning is somewhat related to schema versioning [28], this topic is beyond of the scope of this paper (see elsewhere [28–30] for the details about schema versioning).

SOP OBJECT VERSION SYSTEM: DESIGN PRINCIPLES

In the design of the SOP Object Version System, we followed the guiding principles listed below. In this section, we briefly discuss these design principles.

1. Compliance with ODMG object database standards.
2. Minimal impacts on *Soprano* object storage system.
3. General version mechanism, instead of a specific version policy.

ODMG compliance

ODMG C++ OML is used for accessing and manipulating persistent objects in the database [20]. Since versioned objects are also persistent objects, all of the ODMG C++ OML interfaces for persistent objects should be valid for the versioned objects. For instance, the references to persistent objects in ODMG C++ OML are based on a smart pointer, `Ref`, which behaves like a normal C++ pointer but provides additional mechanisms for guaranteeing the integrity of references to non-memory resident persistent objects. This smart pointer, we think, could also refer to versioned objects. In addition, we tried to use the ODMG C++ OML interfaces as much as possible when implementing the SOP object version model. For example, to prohibit the modification of versions of a particular type, we extended the interface `mark_modified()` of ODMG C++ OML, which is used to notify the database runtime that the state of an object will change. The main reason why we tried to achieve maximal ODMG-compliance in designing the SOP Object Version System is to alleviate the user from the burden of learning new interfaces for object versioning.

Minimal impacts on *Soprano* object storage system

The next design goal of the SOP Object Version System is to minimize the performance overhead for non-versioned objects. The goal of ODMG-compliance is achieved at the expense of a performance degradation in accessing non-versioned objects, but this

performance degradation is negligible. Also, we tried to minimize changes in the *Soprano* object storage system so that the SOP Object Version System could be independent of any particular storage system, as far as possible. In fact, only two member functions of the *Soprano* object storage system are changed, and one member function is newly added to *Soprano*. Except for these member functions, all other components of the SOP Object Version System are completely independent of the object storage system. Thus, the SOP Object Version System is easily portable to other ODMG-compliant ODBMSs.

General version mechanism

Instead of providing a specific version policy required in some applications, the SOP Object Version System supports general version mechanisms. Although this burdens the user with managing the semantics of the versions for some applications, we believe that it allows the SOP Object Version System to be applicable to wider application areas. Moreover, in future we will relieve users of the burden of managing version semantics by adding the functionality of customizing version semantics, as in Objectivity [31] and Itasca [13].

SOP OBJECT VERSION MODEL

In this section, we describe each component of the SOP object version model, including the creation of versioned objects, the derivation of new versions, deletion of versions, version status, static/dynamic binding to versions, and traversal of versioned objects. To illustrate the SOP object version model, the versioned object example shown in Figure 2 is used throughout this section.

Versioned objects and versionable classes

A versioned object in the SOP Object Version System is also conceptually a persistent object. We use the term *conceptually* because it represents an entity in the real world, even though the versioned object consists of a *generic version* and one or more *specific versions*, all of which are also persistent objects and thus have their own unique object identifiers (OIDs). Both generic versions and specific versions are visible to users. Thus, users can access a versioned object via either its generic version or a specific version. With regard to access via the generic version, one of the specific versions is, at a point of time, designated as the *default version* of the versioned object, and then all the references to the generic version are forwarded to the default version. In addition, a specific version of a versioned object can reference its generic version and even itself, as well as other non-versioned and versioned objects.

To clarify these concepts, let us see the example in Figure 2. The dotted box represents a versioned object which consists of a generic version v_g and six specific versions v_i, where specific versions represent the history of the evolution of the versioned object. Version v_7 is currently designated as the default version, and thus all the requests from object *o1* through v_g are forwarded to v_7. Note that the generic version points to the current default version. Finally, though not depicted in Figure 2, a specific version (e.g. v_5) may reference another version, say v_6.

The SOP object version model, as in the Orion version model [32], distinguishes versionable classes and non-versionable classes. An instance object of a versionable class is called a versioned object, while a non-versionable class is a non-versioned object. In this

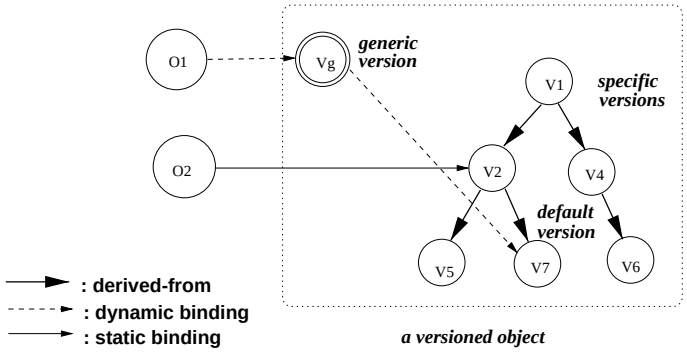

Figure 2. A version example

sense, a persistent class in the ODMG data model, which inherits from root class `PObject`, is a non-versionable class, and its instance objects are non-versioned objects. To make a class versionable in the SOP Object Version System, the class should inherit (directly or indirectly) from class `Version` of the SOP Object Version System, and its instance objects are then versioned objects. Although we distinguish versionable and non-versionable classes in our version model, all of the ODMG interfaces for accessing and manipulating persistent objects can be applied to an object, irrespective of whether it is a versioned object or not, as well as a persistent or transient object.[‡]

In many ODBMS applications such as CAD and CASE, which necessitate long-duration transactions, object versioning, in combination with the check-out and check-in of objects, can be used to avoid the problem of long-duration-waits [32]. Therefore, it might be desirable for ODBMSs to support type-orthogonal version capability. However, versioning in type-orthogonal version systems also incurs both processing and storage overhead to maintain such a generic object, and the overhead is not less than that of other version systems distinguishing versionable and non-versionable classes, such as the SOP Object Version System. Moreover, the problem of long-duration-waits for the checked-out objects can be solved in several ways other than using versioning, such as group transaction techniques and soft locks, and we think that not all of the usage patterns of the objects in databases has to be based on check-out/in. In this respect, it is a reasonable approach to distinguish versionable and non-versionable classes, although the user has the burden of statically deciding which class should be declared as versionable/non-versionable. Moreover, this inflexibility can also be overcome by the functionality of the schema evolution found in Kim [32]; to change a class from non-versionable to versionable classes, and *vice versa*.

Creation of version objects

In the SOP Object Version System, as in the ODMG-93 standard, application programs can create a versioned object only through the operator `new()` of ODMG C++ OML binding. When an instance object of a versionable class is created, the first specific version of the versioned object, in addition to its generic version, is created.

[‡]We would like to remind readers that in the ODMG data model, a class `T` inheriting from `PObject` is *persistent-capable*, in that it does not make all its instance objects automatically persistent. The persistency of an object of a persistent-cable class is determined at its creation time by the user. Thus, the lifetime of an object ODMG is either persistent or transient. The same is true for the versionable classes in the SOP Object Version System, since they also inherit from `PObject` class.

Derivation of new versions

The SOP object version model allows the derivation of a new version v_j (child) from an existing version v_i (parent). Note that new versions are manually derived from an existing version by the applications, not automatically by the modifications to the version. The most recently derived version is automatically set to be the default version, but users can explicitly change the default version to other specific versions later.

The derived-from relationships between pairs of specific versions of a versioned object give rise to a Version Derivation Hierarchy (VDH) for the given versioned object. This VDH information is maintained in a generic version. More than one child version can be derived from one version. The versions derived from the same parent version are called *sibling versions*. However, the SOP Object Version System does not support version merging, which derives a new version by merging two or more existing versions. Thus, in the SOP Object Version System, a VDH results in a tree, not a DAG (Directed Acyclic Graph).

The functionality of version merging might be useful for collaborative works, for example, to combine independently bug-fixed versions of a software module into a new version. And with version merging, the semantics that a version evolved from two or more parents can be directly expressed in VDH. In spite of these advantages, automatic version merging is nearly impossible to achieve in practice, and thus the logic for resolving the differences among parent versions must be left to the applications.

References to version objects

The SOP object version model supports both static and dynamic bindings to a versioned object. With dynamic binding, an object refers to the versioned object via its generic version, and at runtime, the referencing object is automatically bound to its default version. In the SOP Object Version System, when the operator `new()` completes the creation of a new versioned object, it returns the OID of its generic version. Using this OID, other objects can refer to the versioned object dynamically. With static binding, an object directly refers to a specific version via its OID. An object could get a reference to a specific version in several ways, as we describe below. Note that, in the SOP object version model, applications see that both generic versions and specific versions of a versionable class have the same type, and thus a referencing variable for class `T` can point to either generic versions or specific versions of class `T`.

To illustrate the concepts of static and dynamic binding, let us see the example in Figure 2. In this figure, object *o1* has dynamic binding to version v_7 via generic version v_g (we assume that version v_7 is the current default version), while object *o2* is statically bound to version v_2. Later, if another version (e.g. v_6) becomes the new default version, *o1*, at the next traversal to the versioned object via v_g, is dynamically bound to v_6. However, the same is not true with the reference from *o2* to v_2.

Version status

In the SOP object version model, each specific version could have either a `FROZEN` or `WORKING` status at a point of time, based on its updatability. A version in the `WORKING` status is updatable, while updates to a `FROZEN` version are prohibited. The effect of the status of a version is confined to the version itself, that is, the effect does not propagate either to its child version or to other objects connected to it.

A new version can be derived from either the `WORKING` or `FROZEN` version, and a new version, when it is created or derived from an existing version, has a `WORKING` status. To change the status of a version, a user should explicitly promote a `FROZEN` version to a `WORKING` version, or *vice versa*. A version may be deleted regardless of its status.

We believe that to fulfill various application requirements, it is necessary for ODBMSs to unbundle policies of managing the versions' status from their version models. Thus, the SOP object version model does not provide any built-in policies governing version status. Currently, the logic for managing version status is left to the applications. In this respect, the SOP version model is similar to the Orion model [32], except that the Orion model supports three types of version status: transient, working and released.

Deletion of versions

In the literature on object versioning, the version deletion semantics are rarely discussed, except for the Orion model [32] and the EXTRA-V/EXCESS-V model [6]. In the Orion model, a specific version can be deleted even if it is not a leaf node in the VDH. Also, if the only specific version of a versioned object is deleted, the versioned object itself is deleted, i.e. both the generic version and the only specific version are deleted together. In the case of generic version deletion, the Orion version model deletes all of its specific versions, along with itself. Similarly, in the EXTRA-V/EXCESS-V model, Scoire [6] proposes two commands for version deletion, *delete* and *delete version*. The *delete* command removes a versioned object, including all of its versions, while the *delete version* command deletes only a specific version.

With respect to version deletion, the SOP object version model is exactly the same as the Orion model, except in the case of the deletion of non-leaf versions from the VDH. The Orion keeps the information of already deleted non-leaf versions in the VDH, so as to maintain the history of derived-from relationships. In contrast, the SOP Object Version System removes derived-from relationships involving the version from the VDH, as well as the version itself. Thus, when a version is removed from the VDH, its parent version becomes the new parent version of each of its child versions (if any). In particular, when a root version is deleted, the VDH results in a forest. For instance, when version v_1 in Figure 2 is deleted, both version v_2 and v_3 become new roots of different trees.

VDH traversal

The SOP object version model provides users with several ways to traverse the VDH, as follows:

(a) Traversal to parent version: allows users to traverse from a specific version to its parent. In the case of a root version, this returns *null*.
(b) Traversal to sibling versions: allows users to traverse from a specific version to its elder/younger sibling versions. In the case of the eldest version, traversal to elder siblings returns *null*. Similarly, in the case of the youngest version, traversal to younger siblings returns *null*.
(c) Traversal to the eldest child version: allows traversal from a specific version to its eldest child version. When applied to a leaf version in the VDH, this returns *null*. This traversal, in combination with traversal to sibling versions, allows users to traverse all the children of a specific version.

(d) Traversal to the default (generic) version: allows traversal to the default (generic) version of a versioned object.
(e) Historical traversal: allows traversal of all the versions in the VDH in order of creation. We provide interfaces for the oldest version, latest version and next/previous of a specific version.
(f) When each of the above traversals is directed to the generic version, it returns the same result when applied to the current default version.

SOP OBJECT VERSION SYSTEM: API AND ITS USAGE

In this section, the API of the SOP Object Version System is explained, and then its usage is illustrated using an example.

SOP object version system: APIs

Figure 3 shows these APIs, each of which is a member function of template class `Version`. The way in which these member functions work internally, that is, implementation details of the SOP Object Version System, is given in next section.

As noted above, to create a versionable class in the SOP Object Version System, users should declare it as a subclass of the class `Version`. As shown in Figure 3, the class `Version` is a direct subclass of the persistent root class `PObject` in the ODMG model (that is, a versioned object *is a* persistent object). Also, note that the class `Version` is declared using a C++ template [23]. The parameter class `T` is used in defining the class `Version`, i.e. in declaring its members and implementing its member functions. The parameter `T` is the versionable class itself being declared as a subclass of class `Version`.

The constructor of the class `Version` is responsible for creating a versioned object, and the destructor implements the version deletion semantics of the SOP object version model. This is related to our `ODMG-compliance` design principle, in that we exploit ODMG interfaces for the creation and deletion of versioned objects, instead of introducing new interfaces. Like non-versioned objects, a versioned object is created via calling the ODMG C++ OML `new()` interface, which then calls the `Version` class constructor. Similarly, a version is deleted using the ODMG `Ref::delete_object()`, which calls the `Version` class destructor.

The APIs for deriving new versions, setting default version, traversing VDH, obtaining a reference to the generic/default version and checking version status are provided as public member functions of the class `Version`. Many of these member functions return, as the result, a smart pointer `Ref<T>` to a specific or generic version. Recall that object references in the ODMG C++ OML are based on this smart pointer `Ref`. When a persistent object referred to by a `Ref` is dereferenced using the operator `->`, the object, if not memory-resident, is automatically fetched into the memory from the disk, and its virtual memory address is returned as the result of the dereference.

The member function `deriveVer()` is used to derive a new version from the current version, and returns the `Ref<T>` of the newly derived version. A version can be explicitly set as the default version using the member function `setDefault()`. The member functions for traversing the VDH are self-explanatory; each of them returns the target version as its result, as defined in the previous section. Note that the member function `genericObj()` returns `Ref<T>` to a generic version, while other APIs return `Ref<T>` to a specific version. The member function `genericObj()` is thus used to get the reference for dynamic binding.


```
. file version.h .

enum version_flag { WORKING, FROZEN, DUMMY }; // version status

template<class T>               // template class 'Version'
class Version: public PObject {
public:
    // Constructor and destructor.
    Version(void);
    Version(T*);
    ~Version();

    Ref<T>  deriveVer();        // derive new version
    void    setDefault();       // set this version as default version

    // VDH traversals
    Ref<T>  parentVer();        // parent version
    Ref<T>  childVer();         // eldest child version
    Ref<T>  prevSibVer();       // elder sibling version
    Ref<T>  nextSibVer();       // younger sibling version
    Ref<T>  oldestVer();        // oldest version
    Ref<T>  latestVer();        // latest version
    Ref<T>  prevVer();          // temporal previous version
    Ref<T>  nextVer();          // temporal next version
    Ref<T>  defaultVer();       // default version
    Ref<T>  genericObj();       // generic version

    // version status manipulations
    void    freeze();           // WORKING to FROZEN
    void    unfreeze();         // FROZEN to WORKING
    version_flag  ver_status(); // current version status


private:
    int             ver_num;       // version number
    Ref<GenericVersion<T>> gen_ver;// reference to generic version
};
```

Figure 3. `Version` *class and its API*

The member function `freeze()` (`unfreeze()`) sets the version status of the current version to `FROZEN` (`WORKING`). The member function `ver_status()` is used to check the current version status of a specific version.

In this section, we comment on two private members of class `Version`, `ver_num` and `Ref<GenericVersion<T>> gen_ver`, though they are not APIs of the SOP Object Version System. The member `ver_num` represents the version number of the version. In practice, the SOP object version model itself does not support the concept of a version number. However, to support historical traversal of the VDH, the SOP Object Version System assigns a version number to each specific version. The member `gen_ver` is used to refer to its generic version.

SOP object version system: Usage example

Figure 4 shows an example of how to use the SOP Object Version System in an ODMG C++ OML environment. A versionable class `Emp` in Figure 4 is defined as a subclass of the class `Version`. In contrast, the non-versionable class `Dept` is declared as a subclass of the persistent root class `PObject`, and has a member `mgr` of type `Ref<Emp>`, to refer to a versioned object of class `Emp`. It should be noted that a smart pointer `Ref` of ODMG C++ OML is used to refer to versioned objects. We also stress that, when declaring the `Emp` class, the example gives `Emp` itself as the argument of the superclass `Version`, because the class `Version` is defined using the C++ template. For simplicity of presentation, from the example we omit the codes for starting the *Soprano* object storage system and shutting it down.

In the example in Figure 4, a versioned object of the class `Emp` called 'root' is created and then a (non-versioned) object of the class `Dept` referring to the versioned object. The variable `emp1` refers to the employee version dynamically via its generic version. The example shows how to derive three more versions. The first `cout` statement prints the name of a specific version `ver4`, because at this point of time, according to our version model semantics, the current default version is `ver4` and the variable `emp1` is dynamically bound to `ver4`.

The example continues showing how to traverse the VDH using such interfaces as `defaultVer()` and `parentVer()`. For example, at the end of the example, we show the usage of the dynamic binding functionality of the SOP Object Version System; 1) to get `Ref` of the generic version using `genericObj()`, and 2) to traverse the default version via this `Ref`. Note that before this, `ver3` is set as the default version using the member function `setDefault()`. The result of the sample program is as follows.

```
emp1: ver4
default: ver4
parent_emp4: ver2
generic: ver3
```

IMPLEMENTATION DETAILS

In this section, we describe the implementation details of the SOP Object Version System, i.e. the way in which each API works internally. One major implementational aspect of the SOP Object Version System is to implement the classes `Version` and `GenericVersion`. Another is the extension of the *Soprano* object storage system to support version functionality.

For ease of understanding, for each concept of the SOP object version model, we explain how the corresponding API function of the class `Version` implements its semantics, in cooperation with other functions of the class `GenericVersion`. Also, we show how we take advantage of the ODMG C++ OML interfaces to implement the SOP Object Version System, and when necessary, we touch upon the impact of the SOP Object Version System on the *Soprano* object storage system.

The SOP Object Version System is developed on the Sun Sparc Solaris V2.5.1 Unix platform, and is based on ODMG-93 release 1.2 [20]. Before proceeding with implementation details of each API, in the next two subsections, we give the overall architecture of the SOP Object Version System and introduce the class `GenericVersion<T>`.

SOP object version system: Architecture

The overall architecture of the SOP Object Version System is shown in Figure 5. The dotted rectangle represents a conceptual versioned object of class `T`. The dotted circle corresponds


```
OBase obase;                        // Object Base of Soprano
        :
class Emp: public Version<Emp> {  // Versionable class 'Emp'
public:
    char name[20];
    Emp(char* Name): Version<Emp>() { strcpy(name,Name); }
    void setname(char* Name);
        :
};

class Dept: public PObject {      // Non-versionable class 'Dept'
public:
    Ref<Emp> mgr;
    Dept(Ref<Emp> Mgr) { mgr = Mgr; }
        :
};

void main() {
    // Soprano Start-Up!!

    Ref<Emp>  emp1 = new(obase) Emp("root"); // create an 'Emp' object
    Ref<Dept> dept = new(obase) Dept(emp1);  // create a 'Dept' object

    // derivation of new versions 'ver2', 'ver3' and 'ver4'
    Ref<Emp> emp2 = emp1->deriveVer();
    emp2->setname("ver2");
    Ref<Emp> emp3 = emp2->deriveVer();
    emp3->setname("ver3");
    Ref<Emp> emp4 = emp2->deriveVer();
    emp4->setname("ver4");

    cout << "emp1: " << emp1->name << endl;   // print 'ver4'

    Ref<Emp> default_emp = emp2->defaultVer();
    cout << "default: " << default_emp->name << endl;

    Ref<Emp> parent_emp4 = emp4->parentVer();
    cout << "parent_emp4: " << parent_emp4->name << endl;

    emp3->setDefault();   // set 'ver3' as default version

    Ref<Emp> generic_emp = emp4->genericObj();
    cout << "generic: " << generic_emp->name << endl;

    // Soprano Shutdown!
}
```

Figure 4. SOP object version system: ODMG C++ OML usage example

to its generic version, and each specific version is depicted as a small rectangle. The dotted arrow from a `real` generic version to a specific version represents the pointer to the default version from the generic version. Other objects outside the versioned object can refer to either

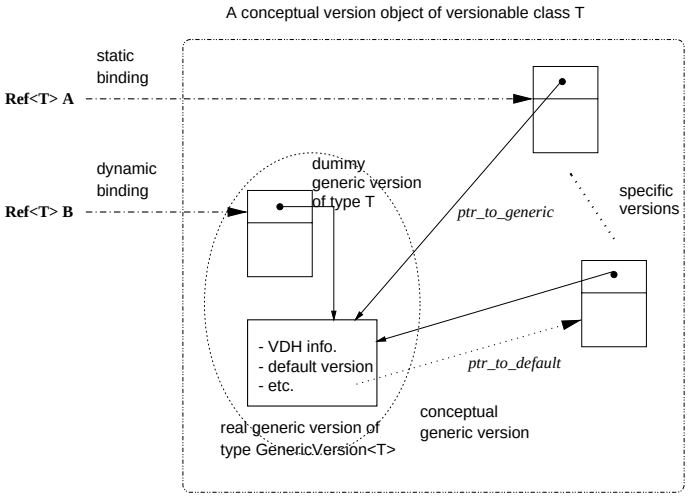

Figure 5. SOP object version system: architecture

a specific version directly (that is, static binding), or the versioned object dynamically via the generic version.

Figure 5 shows that a generic version consists of two objects, that is, a `dummy` generic version and a `real` generic version. The `real` generic version, of type `GenericVersion<T>`, keeps all such information as the default version and the VDH of the versioned object. Meanwhile, the `dummy` generic version, of type `T`, is introduced to make it possible for other objects to dynamically bind the versioned object. Note that, because ODMG C++ OML is strongly-typed, it is impossible for a variable of `Ref<T>` to refer to an object of `GenericVersion<T>`. Thus, dynamic binding in the SOP Object Version System requires two in directions: (1) from the `dummy` generic version to the `real` one; and (2) from the `real` generic version to the default version, detailed below.

Another implementation strategy for generic versions is to move all the information of the `GenericVersion<T>` to the class `Version<T>`, and remove `GenericVersion<T>`, but this strategy has the serious drawback of storage overhead. With this approach, all specific versions, as well as the `dummy` generic version, come with the unnecessary portion of `GenericVersion<T>`. This space overhead thus leads us to decompose the classes `Version` and `GenericVersion<T>`, although the SOP Object Version System suffers from runtime indirections for dynamic binding.

Finally, we would like to mention the role of arrowed lines from specific versions or the `dummy` generic version to the `real` generic version. Within each specific version or `dummy` generic version, the SOP Object Version System keeps a pointer to the `real` generic version, in order to get information such as the VDH from the `real` generic version at runtime. For instance, when a specific version is asked for its parent version, it redirects this request to the `real` generic object.

GenericVersion<T>

Figure 6 shows part of the definition of the template class `GenericVersion<T>`, whose purpose is to define `real` generic objects of type `T`. The SOP Object Version System exploits the template functionality of the C++ language, in the sense that, for each versionable class `T`,


```
. file generic.h .
typedef unsigned VNUM;      // type VNUM

// data structure for VDH(version derivation hierarchy)
struct VDHEdge {
    VNUM parent;            // parent version number
    VNUM child;             // child version number
};

template<class T>           // template class GenericVersion<T>
class GenericVersion: public PObject  {

    Ref<T> _dummy;          // dummy generic object
    Ref<T> _default;        // default version

    Ref<T> versions[20];    // array of current specific versions

    // VDH information
    VDHEdge VDH[20];        // VDH
    int edgenums;           // number of derived-from relationships
    VNUM    VTH[20];        // temporal history of version creation
    int vernums;            // number of versions
    VNUM    nextvernum;     // next-version number

    // other member functions for the implementation of
    // version functionality of SOP Object Version System
            :
}
```

Figure 6. Definition of `GenericVersion<T>`

its corresponding class `GenericVersion<T>` is defined. As shown in Figure 6, members of class `GenericVersion` include a reference to the `dummy` generic object and a reference to the default version. Also, although not listed in Figure 6 for simplicity of presentation, several member functions implementing the version functionality of the SOP Object Version System are included in the class `GenericVersion`. A detailed explanation of each of these member functions is given in the appropriate part of this section.

Implementation of version status

Version status encoding

The SOP Object Version System, as noted above, supports two states for a specific version, i.e. `WORKING` and `FROZEN`. The current status information of each specific version is encoded in the version itself. For the encoding of status information, the SOP Object Version System exploits the data structure `PHeader`[§] for persistent objects in the *Soprano* object storage system, which is invisible to application programmers.

The data structure `PHeader` was introduced in *Soprano* for the implementation of internal functionalities of the object storage system. `PHeader`, as partially depicted in the following,

[§]This part locates the head part of the layout of every persistent objects physically stored in the database.

includes utags, which is reserved for supporting future user requirements. The SOP Object Version System uses this field to encode the version status of the corresponding object:

```
struct PHeader {
        :
    ushort  stags;   // flag for internal use by Soprano
    ushort  utags;   // flag reserved for future user requirements
};
```

As mentioned above, the type of the dummy generic version of a versionable class T is also T, as for its specific versions. We use the status DUMMY for this dummy generic version, to differentiate it from specific versions. This status DUMMY is not visible to application programs, and is used only for the purpose of implementing the SOP Object Version System.

Forbidding changes of FROZEN *version*

The status of a specific version can be, as described above, changed explicitly by users with two member functions, freeze() and unfreeze(), of the class Version.

The *Soprano* object storage system provides a member function mark_modified() of the class PObject, whose purpose is to communicate to the runtime ODBMS the fact that the state of an object will change. This member function is included in the class PObject because the ODBMS, in the default case, cannot automatically detect when an object is modified. With the help of an optional C++ OML preprocessor, the programmer may omit mark_modified() calls; in this case, the preprocessor automatically detects when the objects are changed. In any case, through this member function, updates to a persistent object in the ODMG C++ OML environment are made visible to other users of the database after the transaction performing the modifications commits.

To prohibit the modification of FROZEN versions, the member function PObject::mark_modified() of the *Soprano* object storage system is extended so as to check the version status, before proceeding with its normal operations, as follows. In the case of the FROZEN version, it notifies an error code to the users:

```
int PObject::mark_modified() {
    if (utags == FROZEN)
        return error;

    // original mark_modified() code;
          :
}
```

Creation of version objects and dynamic bindings

When a versioned object is created from any subclass of the Version class through the ODMG new() interface, the SOP Object Version System creates corresponding dummy and real generic versions by calling each constructor of the classes Version and GenericVersion in sequence, as shown below:

```
template<class T> inline
Version<T>::Version(void) {
    // mark this object as dummy generic version
    markflag(DUMMY);
```



```
    // create real generic version
    gen_ver = new(obase) GenericVersion<T>(Ref<T>((T*)this));
}

template<class T> inline
GenericVersion<T>::GenericVersion(Ref<T>& dummy) {
    // set reference to dummy generic version
    _dummy = dummy;

    // initialize default version reference to null
    _default.clear();

    // initialize other informations
    edgenums = 0;
    nextvernum = 1;
        :
}
```

Readers should note that, in the above code, the constructor of the class `Version` marks the newly created version as a `dummy` generic version, not as a specific root version. For example, the variable `emp1` in Figure 4 refers to a `dummy` generic version of the newly created `Emp` object. Thus, at the time point when the `new()` operation completes, `dummy` and `real` generic versions exist, but the root version of the versioned object has not yet been created. This is because in C++, it is impossible to create two instances of a class by calling the constructor once. The root version is automatically created when the versioned object is accessed for the first time after its creation, via a reference to its `dummy` generic version.

```
template<class T> inline
T* Ref<T>::operator ->(void) const {
    // Soprano object storage system's normal operations
    // for object fixing and pointer swizzling, etc.
                    :
    // In case that this Ref refers to a dummy generic version,
    // return pointer to default version(that is, dynamic binding)
    if ( ((PHeader*)memptr)[-1].utags == DUMMY ) {
        return (T*)(((PObject*)memptr)->defaultver());   // (1)
    }
    // In case of static binding, return pointer to specific version
    else {
        return (T*)memptr;
    }
}
```

Figure 7. Extension of `Ref<T>::operator->` *for dynamic binding*

For the implementation of dynamic bindings, we modified ODMG's dereference operator `Ref<T>::operator ->`, as shown in Figure 7. When a `dummy` generic version is accessed through the `operator ->`, it calls its member function `defaultver()` to get the pointer to its default version. For statement (1) in Figure 7 to be compiled, the

class `PObject` should include the following virtual member function, because the class `Version<T>` is a subclass of class `PObject`.

```
virtual PObject* defaultver(void) { return this; }
```

This member function is inherited by the class `Version`, and is redefined to support dynamic binding, as follows.

```
template<class T>
T* Version<T>::defaultver(void) {
   T* _default;
   if (usertags() == DUMMY) { // In case of dummy generic version,
     markflag(WORKING);       // mark this as WORKING temporarily,
     _default = gen_ver->defaultver(); // forward to real generic version,
     markflag(DUMMY);         // reset to DUMMY
   }
   else {                     // In case of specific versions,
     _default = gen_ver->defaultver(); // forward to real generic version
   }

   return _default;
}
```

This member function forwards the request for the memory pointer of the default version of a versioned object to the `real` generic version of the object. Note that, before calling `defaultver()` of the class `GenericVersion`, in the case of the `dummy` generic version, `Version<T>::defaultver()` marks its status as `WORKING`. This is an implementation trick, and a detailed explanation of this trick is given below.

The member function `defaultver()` of class `GenericVersion` is shown below.

```
template<class T>
T* GenericVersion<T>::defaultver(void) {
    // If root version does not exist yet, create the root version
    if (_default.isnull()) {
         genroot(_dummy.operator->());
    }

    // return pointer to default version
    return _default.operator->();
}
```

As stated above, when a versioned object is accessed for the first time after its creation via dynamic reference to its `dummy` generic version, the root version (at the same time this root version is the default version) does not yet exist. So, the `real` generic version creates its root version via calling its member function `genroot()`. This member function `genroot()` requires a memory pointer of the `dummy` generic version, i.e. `_dummy.operator->()`, because the `dummy` generic version needs to be bit-wisely copied to the default version. At this point, readers should note that the `operator->()` is recursively called and, if the status of the `dummy` generic version is still `DUMMY`, this operator is repeated infinitely. To avoid this infinite loop, the member function `Version<T>::defaultver(void)`, as noted above, marks its status as `WORKING` before calling `gen_ver->defaultver()` in the case of a dummy generic version. Only when the member function `genroot` of the class `GenericVersion<T>` completes, a specific root version of the versioned object is finally created.

Before concluding this subsection, we comment on the performance overhead for non-versioned objects, which is attributable to the SOP Object Version System. In ODMG C++ OML binding, the dereference operator `->` is used to access members of the persistent object addressed by a given object reference `Ref<T>`. To this operator, as shown in Figure 7, the SOP Object Version System added a routine for checking whether the given object is the `dummy` generic version or not. For this comparison routine, the SOP Object Version System does not yield any other overhead for the performance of non-versioned objects. There may be alternatives to this implementation, which do not degrade the performance of non-versioned objects. One possible alternative is to introduce a new type of smart pointer, for example `VRef`, for reference to versioned objects. In this case, all the non-versioned objects are referenced via `Ref`, while references to versioned objects are done via `VRef`. Thus, the SOP Object Version System could yield no performance degradation. However, we discard this alternative because it has a disadvantage, which we think is worse than the performance degradation of the current SOP Object Version System; that is, application programmers have to be aware that there are two different types of references available, and be cautious when choosing the reference type for versioned objects and non-versioned objects.

Derivation of new versions

Figure 8 shows the member function `deriveVer()` of the class `Version`, which is used to derive new versions from an existing version. If a `dummy` generic version receives this message, it redirects the message to its default version via its `real` generic version, thus deriving a new version from the default version. When a specific version receives the message `deriveVer()`, it creates a new version by calling the copy constructor of its class `T` (default copy constructor if `T` has no copy constructor), and then updates the VDH information kept in the `real` generic version, and returns the `Ref<T>` of the new version. The copy constructor of the class `Version` does a bit-wise shallow copy. Therefore, immediately after a new version has been derived, its value is exactly the same as that of its parent, except for the version number.

Traversals of VDH

As previously mentioned, the `real` generic version of a versioned object maintains its VDH information, and the SOP Object Version System then uses this information when required to traverse the VDH of the versioned object. A specific version, when required to traverse the VDH, calls the corresponding member function of the `real` generic version with its version number as an argument, and returns the result from the `real` generic version. Meanwhile, in the case of a `dummy` generic version, the request for VDH traversal is forwarded to its default version via the dynamic binding mechanism, and the default version returns the result to the caller.

Version deletion

The SOP Object Version System does not introduce any new interfaces for deleting versions. Instead, in the SOP Object Version System, versions can be removed by calling ODMG C++ OML interface `Ref<T>::delete_object`.[¶] During execution, this

[¶]The *Soprano* object storage system implements this member function with the name of `destroyobj`.


```
template<class T>
Ref<T> Version<T>::deriveVer(void) {

    Ref<T> newver;

    // In case of dummy generic version,
    // forward the request to real generic version
    if (usertags() == DUMMY) {
        newver = gen_ver->deriveVer();
        return newver;
    }

    // Create new version via calling the (default) copy constructor
    newver = new(obase) T(*(T*)this);

    // Set ver_num of new version
    VNUM newvernum = gen_ver->nextvernum;
    newver->ver_num = newvernum;

    // Update VDH information in real generic version
    gen_ver->mark_modified();
    gen_ver->nextvernum++;
    gen_ver->add(newver);
         :

    return newver;      // return new version's Ref
}
```

Figure 8. Member function `deriveVer()`

`delete_object` operator calls the destructor of the versionable class. Based on this fact, the SOP Object Version System implements the semantics of version deletion using the destructor of class `Version`, as follows.

```
template<class T>
Version<T>::~Version(void) {
   if (usertags() == DUMMY) {
      // In case of dummy generic version,
      // delete all specific versions and dummy/real generic version
      gen_ver->deleteAllVers();
      gen_ver.destroyobj();
   }
   else {
      // In case of specific versions,
      // remove the version information from real generic version
      gen_ver->deleteVer(ver_num);

      // If this version is the only one of the versioned object,
      // delete dummy/real generic version also
      if (gen_ver->vernums == 0)
         gen_ver.destroyobj();
   }
}
```


Table I. Comparisons of the version models of various ODBMSs

	VDH	Dynamic binding	Type orthogonality
SOP Object Version System	Tree	yes	no
ITASCA [13]	Tree	yes	no
O2 [33]	DAG	yes	yes
Objectivity/DB [15]	Tree	yes	no
ObjectStore [14]	DAG	yes	yes
Ode [9]	Tree	yes	no
Versant [34]	DAG	yes	yes

RELATED WORKS

In this section, the SOP Object Version System is compared to the version mechanisms of other object database systems with respect to version model and implementation techniques.

Comparisons: version models

Table I compares a variety of ODBMSs on the basis of their support for VDH and dynamic binding. The fourth column in Table I indicates whether a specific ODBMS supports type orthogonality for version functionality, i.e. it can create both versioned and non-versioned objects of the same class, regardless of its type. As far as we know, with every ODBMSs supporting version merging, users have to take responsibility for merging the versions.

Many new database applications require the various version semantics which are specific to each application, but the basic version functionalities of contemporary ODBMSs do not seem to be able to satisfy these requirements [6]. That is, current ODBMSs support their own hard-coded version semantics. In this respect, a more generic approach to version mechanisms is necessary; Sciore [6] proposes a high level version model extending the Exodus/Extra data model, and both Itasca and Objectivity/DB allow users to customize their own version semantics according to the requirements of the applications.

Besides, current ODBMSs, with regard to version modeling power, lack support for configuration management, i.e. how to consistently maintain a collection of component versions of a complex object. The importance of configuration management has been widely recognized in several ODBMS application areas, including CAD [5] and CASE [4]. However, with almost every ODBMS, including the SOP Object Version Systems, application programmers themselves should manage configurations using the basic functionalities of versions and dynamic binding. It is time, we think, to push the functionality of configuration management into ODBMSs, so that they can suit the needs of target applications, and thus be more widely accepted in the marketplace.

Comparisons: implementation techniques

To the best of our knowledge, the SOP Object Version System is the only system which provides version functionality by extending the ODMG standard, specifically ODMG C++ OML. Some commercial ODBMSs, such as O2 and Objectivity, support version mechanisms within the particular database programming language environments of their own. For example, the O2 system supports version functionality only through the O2C language.

To compare the SOP Object Version System and the version mechanisms of these commercial ODBMSs with respect to implementation details, we tried to find related literature, but failed. So in this subsection the SOP Object Version System is compared with only two prototype ODBMSs; Orion [32] and Ode [9].

The Orion system provides version functionality within the extended Lisp environment [32], and thus this interpreter-based language environment seems to be easier to implement version mechanism in than the compile-based language environment, such as ODMG C++ OML. In the Lisp environment in particular a variable can refer to objects of any type. This makes the implementation of dynamic binding mechanism easier, because a pointer can refer to a specific version and a generic version, which have different types. In contrast, to avoid type problems of dynamic binding, the SOP Object Version System introduced two types of generic versions, `dummy` and `real` generic versions, for a versionable class, as described above.

The Ode ODBMS provides version functionality within its own O++ database programming language, which extends the C++ language to integrate the database and programming language [9]. This O++ language is based on the O++ preprocessor, which translates O++ programs into C++ programs. For object versioning, O++ provides a class library, similar to the SOP Object Version System. However, for its implementation, the O++ version class library needed template functionality, which no C++ compiler supported at the time at which the O++ was being developed. So, they simulated a template using the `define` macros, and we think this is one main reason why the Ode system was based on the O++ preprocessor. In contrast, contemporary C++ compilers allow us to implement the SOP Object Version System with the classes `Version<T>` and `GenericVersion<T>`, which fully exploits their template facility. Hence, the SOP Object Version System needs no preprocessors like the O++ preprocessor.

Meanwhile, Agrawal *et al.* [9] report that the Ode system supports dynamic binding, but they do not give either usage example of the dynamic binding or its implementation details, but in this paper we present the dynamic binding mechanism of the SOP Object Version System in depth, including its semantics, usage and implementation details.

CONCLUSION

In this paper, we have described the design and implementation of the SOP Object Version System, which extends ODMG-93 C++ OML to provide version functionality. We gave several design principles when designing the SOP Object Version System, such as compliance with the ODMG-93 standard and minimization of performance overhead on the *Soprano* object storage system. After explaining the version model and application program interfaces of the SOP Object Version System, we presented its use for creating and managing versioned objects, with an illustrative example. Finally, we discussed implementation details of the SOP Object Version System.

To the best of our knowledge, the SOP Object Version System is the only system providing object version facility in the ODMG standard. Another major contribution of the SOP Object Version System is that it gives a solution for dynamic binding in strongly typed languages like C++. We plan four future directions. First, we are going to upgrade the SOP Object Version System to ODMG release 2.0, although it seems to be somewhat trivial. Next, we will extend the current SOP Object Version System to support both the customization of version semantics and configuration management. Thirdly, we will extend both our OQL processor *Sopoqles* and schema manager module to understand the version concept of the SOP Object

Version System. Finally, we would like to integrate the SOP Object Version System with our schema version model [29], and the integrated object/schema version model will provide a powerful environment for the development of many complex database applications.

ACKNOWLEDGEMENTS

We are indebted to the anonymous referees for their helpful comments.

REFERENCES

1. R. Rastogi, S. Seshadri, P. Bohannon, D. W. Leinbaugh, A. Silberschatz and S. Sudarshan, 'Logical and physical versioning in main memory databases', *Proceedings of the International Conference on Very Large Data Bases*, Athens, Greece, 1997, pp. 86–95.
2. W. Kim, *Modern Database Systems – The Object Model, Interoperability, and Beyond*, ACM Press, 1995.
3. M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier and S. Zdonik, 'The object-oriented database system manifesto', *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, 1989, pp. 223–240.
4. R. Conradi and B. Westfechtel, 'Version models for software configuration', *ACM Computing Surveys*, **30**(2), 232–282 (1998).
5. R. H. Katz, 'Towards a unified framework for version modeling in engineering databases', *ACM Computing Surveys*, **22**(4), 375–408 (1990).
6. E. Sciore, 'Versioning and configuration management in an object-oriented data model', *VLDB Journal*, **3**(1), 77–106 (1994).
7. A. Silberschartz, M. Stonebraker and J. Ullman, 'Database systems: Achievements and opportunities', *Communications of the ACM*, **34**(10), 110–120 (1991).
8. A. Silberschartz, M. Stonebraker and J. Ullman, 'Database research: Achievements and opportunities into the 21st century', *ACM SIGMOID Record*, **25**(1), 52–63 (1996).
9. R. Agrawal, S. J. Buroff, N. Gehani and D. Shasha, 'Object versioning in Ode', *Proceedings of the International Conference on Data Engineering*, Kobe, Japan, 1991, pp. 446–455.
10. J.-H. Ahn, S.-W. Lee, H.-J. Song and H.-J. Kim, 'A survey of performance-related features of contemporary object storage systems', *Journal of System Architecture*, **45**(5), 363–386 (1998).
11. Ardent Software, Inc., 'O_2 Version: The O_2 Version Manager'. Available: http://www.ardentsoftware.com/object/papers/O2Version.html, 1998. (White paper.)
12. EXODUS Project Group, 'EXODUS Storage Manager Architectural Overview', November 1991.
13. IBEX Object Systems, Inc., 'ITASCA Technical Summary Release 2.3', 1995.
14. Object Design, Inc., 'ObjectStore Release 4.0 Online Documents', 1995.
15. Objectivity, Inc., 'Objectivity/DB Version 3: Getting Started with C++', 1994.
16. P. A. Bernstein, 'Repositories and object oriented databases', *ACM SIGMOD Record*, **27**(1), 88–96 (1998).
17. A. Bapat, J. Waesch, K. Aberer and J. M. Haake, 'HyperStorM: An extensible object-oriented hypremedia engine', *The Seventh ACM Conference on Hypertext*, Washington, DC, 1996, pp. 203–214.
18. E. J. Whitehead, Jr., 'World Wide Web: Authoring, versioning, and other topics', *Standard View: ACM Perspectives on Standardization*, **5**(1), 3–8 (1997).
19. SOP Team, 'SNU ODBMS PLATFORM: SOP Implementation Document (in Korean)', OOPSLA Laboratory, Department of Computer Engineering, Seoul National University, February 1998.
20. R. G. G. Cattel, editor, *The Object Database Standard: ODMG-93 Release 1.2*, Morgan Kaufmann, 1996.
21. R. G. G. Cattel, editor, *The Object Database Standard: ODMG-93 Release 1.0*, Morgan Kaufmann, 1993.
22. R. G. G. Cattel and D. K. Barry, editors, *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, 1997.
23. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1991.
24. J.-H. Ahn and H.-J. Kim, 'SEOF: an adaptable object prefetch policy for object-oriented database systems', *Proceedings of the Interbnational Conference on Data Engineering*, Birmingham, UK, April 1997, pp. 4–13.
25. J.-H. Ahn, K.-W. Lee, H.-J. Song and H.-J. Kim, 'SOPRANO: Design and implementation of object storage system (in Korean)', *Journal of KISS (C)*, **2**(3), 243–255 (1996).

26. E.-S. Cho, S.-Y. Han and H.-J. Kim, 'A new data abstraction layer required for OODBMS', *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, Montreal, Canada, 1997, pp. 144–150.
27. E.-S. Cho, S.-Y. Han and H.-J. Kim, '*LOD**: An ODMG based C++ database programming language with class-separation support', submitted 1998.
28. W. Kim and H. T. Chou, 'Versions of schema for object-oriented databases', *Proceedings of the International Conference on Very Large Data Bases*, Los Angeles, California, 1988, pp. 148–159.
29. S.-W. Lee and H.-J. Kim, 'A model of schema versions for object-oriented databases based on the concept of rich base schema', *Information and Software Technology*, **40**(3), 157–173 (1998).
30. S. Monk and I. Sommerville, 'Schema evolution in OODB using class versioning', *ACM SIGMOD Record*, **22**(3), 16–22 (1993).
31. Objectivity, Inc., 'Objectivity/DB Technical Overview, Version 3', 1995.
32. W. Kim, *Introduction to Object-Oriented Databases*, MIT press, 1990.
33. O_2 Technology, *A Technical Overview of the O_2 System*, 1994.
34. Versant Object Technology Corp., 'Versant ODBMS Release 4', 1996.

