# A UNIFYING VERSION MODEL FOR OBJECTS AND SCHEMA IN OBJECT-ORIENTED DATABASE SYSTEM

DISSERTATION

Submitted to the Graduate Council of the

University of North Texas in Partial

Fulfillment of the Requirements

For the Degree of

Doctor of Phylosophy

By

Dongil Shin, B.S., M.S.

Denton, Texas

August 1997

Shin, Dongil, A Unifying Version Model for Objects and Schema in Object-Oriented Database System. Doctor of Philosophy (Computer Science), August, 1997, 78pp., 1 table, 8 illustrations, references, 55 titles.

A traditional database management system generally supports applications that consist of a large number of instances for relatively few number of data types. In contrast, application environments that are supported by object-oriented databases (OODBMS) are characterized by a small number of objects (i.e., individual records) linked to a large number of classes (i.e., schema). The large number of types or classes in an OODBMS makes it more likely that these data elements will be changed sometime during the life of the database. These changes may occur for several reasons such as flaws in the design, changes to requirements, or the need to reuse classes. As a result, there is a greater need for object-oriented database systems to support the efficient management of object and schema versionings to handle changes to the data.

There have been a number of different versioning models proposed. The research in this area can be divided into two categories: object versioning and schema versioning. Although researchers acknowledge that both object and schema versioning is necessary, they have tended to handle these items separately. That is, most OODBMS models provide users with two different sets of functions; one for handling changes to objects, and another for schema changes. As a result, most OODBM systems contain code that supports both object and schema versioning, and programmers must learn two different versioning techniques to handle the two types of changes.

In this dissertation, both problem domains are considered as a single unit. This dissertation describes a unifying version model (UVM) for maintaining changes to both objects and schema. UVM handles schema versioning operations by using object versioning techniques. The result is that the UVM allows the OODBMS to be much smaller than previous systems. Also, programmers need know only one set of versioning operations; thus, reducing the learning time by half. This dissertation shows that UVM is a simple but semantically sound and powerful version model for both objects and schema.

# A UNIFYING VERSION MODEL FOR OBJECTS AND SCHEMA IN OBJECT-ORIENTED DATABASE SYSTEM

DISSERTATION

Submitted to the Graduate Council of the

University of North Texas in Partial

Fulfillment of the Requirements

For the Degree of

Doctor of Phylosophy

By

Dongil Shin, B.S., M.S.

Denton, Texas

August 1997

## ACKNOWLEDGEMENTS

First of all, I really appreciate Dr. Robert P. Brazile for his excellent advising and encouragement during the course of this dissertation. I would like to thank Dr. Swigger for her consturtive comments on the structure of the dissertation and careful reading of my work. I would also like to thank Dr. Farhad Sharokhi and Dr. Tom Jacob for serving on my Ph.D. committee.

I also owe to Dr. Kathleen Swigger and Dr. Robert P. Brazile for hiring me as a Research Assistant for the course of my Ph.D. study. They helped me financially and allowed me to participate in research activities.

I am especially grateful to my parents. They made it financially possible for me to study and live since I started my studies in United States. I thank my father for putting pressure on me to speed up my writing, and my mother for encouraging me all the time.

I would also like to thank my two daughters, Rachel and Michelle, for giving me inspiration and the joy of life. Most of all I'd like to thank my wife, Sunyoung Chung, for being so supportive despite having two great children to take care of. **I dedicate this dissertation to my wife, Sunyoung**.

TABLE OF CONTENTS

Page

LIST OF TABLES .................................................. vii

LIST OF ILLUSTRATIONS .......................................... viii

CHAPTER

1. INTRODUCTION .................................................. 1

2. PREVIOUS RESEARCH ON VERSIONING .............................. 5

Basic Concepts
Versions of Objects
Batory and Kim
Chou and Kim
Klahold, Schlageter, and Wilkes
Landis
Rumbaugh
Vines, Vines and King
Agawal, Buroff, Gehani, and Shasha
Versions of Classes
Bjornerstedt
Chiueh
Kim and Chou
Penny and Stein
Skarra
Zicari
Summary

3. UNIFIED VERSION MODEL ........................................ 20

Description of the Model
UVM Operations
Creating Different Versions of Objects and Classes
Objects

Classes
Deleting Versions of Objects and Classes
Accessing Versions of Objects and Classes
Accessing Instances When There Are Versions of Classes
A Complete Example

4. IMPLEMENTATION OF UVM .................................... 37

Persistent Objects and the DBO Class Library
Using DBO to store Versions of Objects and Classes
Modifying the UID for Versioning
The Set of Classes for UVM
Class UVM
Class UVMNode
Class UVMAttribute
Special Implementation Issues
The C++ Specification for UVM

5. CONCLUSION ................................................ 48

UVM Features
Implementation Results
Potential Disadvantage
Future Research
Class Evolution
Evaluation
Alternate Implementation Strategies
Conclusion

Appendix A. C++ Class Definitions .................................. 54

Appendix B. Introduction to DBO: A Simple Object Manager ................ 60

BIBLIOGRPHY ................................................ 70

# LIST OF TABLES

Table Page

1. Versioning functions in UVM........26

## LIST OF ILLUSTRATIONS

Figure Page

1. Unifying Version Model. ....22
2. Creating Versions of a Person Object. ....27
3. Creating Versions of a Person Class. ....30
4. Creating Versions of an Instance and a Class. ....34
5. Accessing Version of an Instance for the Different Class ....36
6. A Derivation Hierarchy of the Unifying Version Model. ....38
7. Version Derivation Tree. ....42
8. Dynamic Object Conversion. ....46

# CHAPTER I

# INTRODUCTION

Traditional data processing applications usually have a large number of instances of a relatively small number of different types of data. As a result, traditional database systems exist to support the long-term persistence of this kind of data. For example, a typical accounting database for a large bank contains millions of master records that describe each deposit or withdrawal credited or debited against an account over the course of a billing period. In order to accommodate these types of transactions, conventional database management systems have evolved to provide efficient ways of storing and accessing such regularly structured data.

In contrast, application environments which object-oriented databases (OODBMS) support are characterized by a small number of objects (i.e., individual records) linked with a large number of classes (i.e., schema). Moreover, frequent object and schema changes are the rule rather than the exception (Atkinson et al. 1989). Design objects will often undergo a number of changes to both values and their logical structures. These changes arise due to many reasons: design flaws, changes in requirements, the need for reusability and extensibility of classes, or the need for comparability with other systems (Tresch. and Scholl 1993). For example, an OODBMS that supports design information for a microprocessor chip will have to contain descriptions for each part in terms of several million interconnected transistors. In addition, one can expect these descriptions to change radically as the chip is being developed. It is inevitable that the programmer will have to make changes to objects as well as the schema in response to new requirements, bugs, or performance bottlenecks. Thus, efficient management of object and schema versions to handle object and schema changes is one of the most important functions of an OODBMS.

In order to improve both the efficiency as well as management of changes to objects and schema in Object Oriented Databases, there have been a number of different versioning models proposed. The research in this area can be divided into two categories: object versioning (Zdonik 1986, Kim 1986, Chou and, Katz 1990, Agrawal et al. 1991, Ahmed and. Navathe 1991, and Sciore 1991) and schema versioning (Banerjee et al. 1986, Penney and Stein 1987, Skarra and Zdonik 1986 and 1987, Bjornerstedt and Hulten 1989, Kim and Chou 1988, Zicari 1989, Casais 1990, Lerner and Habermann 1990, Zdonik 1990, Andany, Leonard, and Palisser 1991, Clamen 1991, Roddick 1991, Bertino 1992, Odberg 1992, Bratsberg 1993, Monk 1993, and Scherrer, Geppert, and Dittrich 1993). Object versioning techniques keep track of different versions of the same object whenever the data in the object changes but the schema remains the same. For example, if a specific dimension value for an object is changed from five units to six units, the system retains both versions of the object (the old and the new object) in the database. On the other hand, schema versioning keeps track of changes to the schema and the corresponding objects that relate to the different schemas. For example, if a new data member is added to a class definition, the system must store objects created under both the new and old versions of the schema.

Although researchers acknowledge that both object and schema versioning is necessary, they have tended to handle these items separately. That is, most OODBMS models provide two different sets of functions, one for handling changes to objects and another for schema changes. As a result, most systems contain redundant code to support both object and schema versioning, and programmers must learn two different versioning processes to handle these two items.

In response to the problem presented above, this dissertation describes a single model for versioning that includes both object versioning and class versioning. It was observed that the schema is maintained in a database as instances of other (Meta)classes, or in other words, as objects. It was also observed that the data and operations, at least conceptually, are very similar for maintaining versions of objects and versions of classes. This observation led to the idea that there could be a single model of versioning which would be suitable for both objects and schema. While there are additional data and operations necessary for the maintenance of instances of the classes that change, there is a core set of data and operations for the maintenance of versions of both objects and classes that can be identified and included in a single model.

The benefits of a single model are: (a) it is less complex; (b) there is a reduction of redundant code and data; and (c) it is a more elegant model for versioning.

Therefore, this dissertation describes a unifying version model (UVM) for maintaining changes to both objects and schema. UVM handles schema versioning operations by using object versioning techniques. The result is that the UVM allows the OODBMS to be smaller and simpler than previous systems. Also, programmers need know only one set of versioning operations, so it reduces the learning time by half. This dissertation will show that UVM is a simple but semantically sound and powerful version model for both objects and schema.

The schema is the definition of the database, and in an object-oriented system the schema is represented by a set of class definitions. Therefore in the remainder of this discussion, the schema and schema versions will be referred to as classes and class versions.

The next sections of the dissertation are organized as follows. Section 2 presents a discussion of previous research on object versioning and class (schema) evolution. Section 3 introduces the Unified Version Model and provides a detailed description of the various components of the model. Section 4 presents the DBO Class Library which can be used to implement the model. Section 5 presents the conclusion and lists topics for future research.

# CHAPTER II

# PREVIOUS RESEARCH ON VERSIONING

## Basic Concepts

This section introduces the basic concepts and problems of management systems, with specific reference to object-oriented databases (OODBMS). In OODBMS, an object that is associated with a *state* and a *behavior* represents each real world entity (Chou and Kim 1988). The state is represented by the values of the object's attributes, while the behavior is defined by a method acting on the state of the object whenever it is invoked (Chou. and Kim, 1986).

A set of objects that have exactly the same attributes and methods are called a *class*. The class defines the implementation of a set of objects. The term *Instantiation* means that the same class definition can be used to generate objects with the same structure and behavior. In this sense, a class acts as a template for the creation of objects. A *schema*, in turn, is a set of class definitions, including any relationships to other classes. This relationship is commonly represented as a hierarchy, and classes are said to be associated with each other through an inheritance hierarchy (Chou. and Kim 1986).

A *version* of an object can be thought of as a semantically significant snapshot of the object taken at a given point in time (Chou. and Kim 1986). The term *semantically* *significant* means that a new version of an object is created only when a major, as opposed to any, modification has occurred. *Derivation* and *history* are terms used to describe the conceptual mechanism of version management.

A *version model* is defined as a description of the data structures that store the information necessary to maintain versions of objects and the operations necessary for the manipulation of those data structures. Kats (Katz 1990) surveyed various version models and provided a common terminology and collection of mechanisms that underlies any version modeling approach. For example, his research suggests that *derivation* and *history* are two basic conceptual ideas used in version management. An object's derivation refers to the object's previous version, while an object's history defines how a given version was arrived at over a period of time (Landis 1986). The relationship among versions can be represented as either a tree or graph. Further, the history of an object or class is maintained to show how a given version was derived. History information can be structured and saved in many forms.

In general, a version of a complex object consists of specific versions of its component objects which is sometimes referred to as an object's *configuration* (Vines, Vines, and King 1988). This particular feature enables a version of a complex object to establish a link between itself and versions of each of its component objects. A configuration is said to be either *static* or *dynamic*. If the user explicitly and permanently defines the link, then the configuration is considered static. If the versions are linked together at runtime, then the configuration is dynamic (Vines, Vines, and King 1988).

Because objects can sometimes contain references to other objects (primitive or complex), which can, in turn, be referenced by still other objects or versions, programs that manage versions must have some type of mechanism for *notifying* and for *propagating* changes (Kerr, Chan, and Cooper 1992) to objects. It must also have a way of supporting different versions of instances of classes, objects, and versions of versions. Most versioning systems contain three general operations for version control: creating, accessing, and deleting versions (Chou. and Kim, 1986). Creating a new version requires that the system locates a particular object, and records and stores any changes to that object. Accessing a particular version requires that the system understand how to convert the original version to the latest version, either in a backward or forward manner. Backward and forward compatibility and automatic conversion (building) of versions is a desirable feature of versioning systems (Clamen 1991). Deleting a version requires that the system identify the correct version to delete.

One final aspect of version management is the need to apply all the above functions to supporting class versioning. Because of the nature of OODB applications, it is probably more common to alter or change the structure of classes than instances of objects. Therefore, it is useful to have a versioning mechanism that applies to both classes and instances of classes.

Due to the lack of a standardized model for version control, various version models have been proposed. Although many issues have been addressed by this research, no single model has been proposed that is able to create versions for both classes and instances of classes. Current models concentrate on creating either versions of instances of classes, where the primary issues are maintenance of derivation information, configuration management, and change notification or versions of classes, where the primary issues are backward/forward compatibility and automatic conversion of instances among the various class versions (Monk and Sommerville 1992). The current activities in these two categories of models are summarized in the following sections.

## Versions of Objects

Various version models have been proposed and implemented, although research in this area has focused on creating versions for a single object. In this section, key research describing the current version models will be reviewed. The major areas of research have centered on implementation issues in three areas: (1) maintenance of information about how to derive versions of objects; (2) configuration management; and (3) notification to objects about changes among the various versions. Below is a description of some of the systems that address these particular issues.

### Batory and Kim

Batory and Kim (Batory and Kim 1985) examine the issue of representing versions for a VLSI CAD application and propose using an extension of the E-R model for this purpose. Other researchers who address the version representation issue are Beech, Chou, Dijkstra, and Kafer.

This particular research introduces four basic concepts: molecular objects, type-version generalization, instantiation, and parameterized versions. Molecular objects are introduced as a way of aggregating more primitive objects and their relationships. In the Batory and Kim model, interfaces are used to specify an object type, and versions are represented as instances of that type. Versions are viewed as an alternative way of implementing revisions to previous versions, with no explicit support for derivation history.

The model's instantiation mechanism separates use of design data from its definition. Parameterized versions are basically a way to support dynamic configurations.

The fourth mechanism, parameterized versions, is used to support dynamic configurations. Since a molecular object can be bound to a specific version of its component or can reference a component's type, it can be used to gain access to any component.

Batory and Kim also address the issue of change notification by using timestamp information to limit the range of messages that must be sent.

## Chou and Kim

Chou and Kim (Chou. and Kim 1986 and 1988) describe a model that supports versions, derivations, and operations based on workspaces that provide check-in/check-out operations and dynamic configuration binding via a context mechanism. They are particularly concerned with the problem of notifying the system about changes that have occurred. They distinguish between message-based and flag-based notification mechanisms. Message-based notification can be sent either immediately or later, whereas flag-based notification is sent whenever the user is working on a particular object that has undergone a change. Because of the difficulty of successfully limiting the scope of changes, the authors' approach to change propagation is to limit changes to a single level. That is, only the objects that directly reference the changed object undergo change, and the process does not recurse.

## Klahold, Schlageter, and Wilkes

Klahold et al. (Klahold, Schlageter, and Wilkes 1986) propose a version model based on the concept of a *Version Graph* that is designed to represent the ancestor/descendent interrelationships among versions. Similar to other authors who use graphical representations (Chou. and Kim 1986 and Kim and Chou. 1988), the authors employ partitions to show groups of versions according to their level of consistency. They also provide support for different views of versions.

## Landis

Landis (Landis 1986) describes a version model that was used for the initial design of the Ontologic data model. The research introduces four important ideas: non-linear history, version references, change propagation, and scope limitation. Because versions of objects are organized like branches on a tree, the history of a particular object can be derived by simply following the links up (or down) the tree. Each branch of the tree represents a particular version of the object. As a result, the system can easily identify the current version as well as any default branch.

Version references are proposed as a mechanism to support dynamic configurations. Historical references are always bound; that is, once a version is superseded, any reference to other versions cannot be altered. References from current versions that are not explicitly bound always refer to the current version of the target object.

## Rumbaugh

Rumbaugh suggests using a simple mechanism for controlling change propagation which is based on the idea of assigning a propagation attribute for particular operations (Rumbaugh 1988). These attributes can take on one of four values: 1) none, if the particular operation does not propagate; 2) propagate, if the operation should be applied to both the relationship instance and the related object; 3) shallow, if the operation should be applied to the relationship instance but not the related object; and 4) inhibit, if the propagation should be suppressed for a particular period of time. Although the mechanism was originally proposed for operations such as copy, destroy, print, and save, it proved was for controlling change propagation.

## Vines, Vines and King

Vines et al. describe the version and change control model of GAIA, an object-oriented framework for an ADA-programming environment developed for Honeywell (Vines, Vines, and King 1988). Their approach for change control is based on four related concepts. First, timestamps rather than version numbers are used to keep track of the relationships among various versions. Second, explicit relationships between objects are created to define the impact of a specific change. Third, a special object is created whenever a human or the machine issues a request to change an object. This special object then tracks the change as it evolves and provides the anchor for an audit trail. A change notification object is spawned whenever a change request is propagated along version- or change-sensitive relationships. Finally, configuration objects are introduced as a way of grouping changes.

Agrawal, Buroff, Gehani, and Shasha

Agrawal et al. (Agrawal et al. 1991) present versioning techniques used in the Ode (Agrawal and Gehani 1989) object-oriented database system. The authors built a number of powerful language primitives that implement a variety of functions for performing versioning. The following are the important features introduced in ODE.

- Object versioning is orthogonal to type; that is, versioning is an object property and not a type property. As a result, users can create new versions of an object without having to make changes to the type definition of that object. All persistent objects can be versioned, and both versioned and non-versioned objects of the same type can be created.
- Reference to an object can be bound statically to a specific version of the object or dynamically to the latest version of the object. Static binding is useful for configuration management because it can be used to refer to specific versions of component objects. Dynamic references are useful

because it can be used to track the latest versions of constituent objects.

- Both temporal and derived-from relationships between versions of an object are maintained. Temporal relationships that reflect the history of an object are important for historical databases and for databases that support time. Derived-from relationships reflect the past history of an object and can be useful for software engineering environments. The derived-from relationship can also be used to identify differences between two versions of the same object, a method sometimes used to store the version itself.

## Versions of Classes

Whenever the designer changes a class definition, there is a possibility of creating a new version of the class. This is known in the literature as schema evolution (Clamen 1994). Since schema evolution is a term applied to both non-object-oriented databases as well as OODBs, this paper will refer to the process of changing the database definition as changing the class definition or class versioning. Class changes result in a gradual changing of the structure of the database, either to rectify the lack of some unforeseen data storage requirement or to make the database conform to the changes in the real world.

Thus, the definition of a database cannot be changed without considering what will happen to the data whose structure is described by that definition. Unfortunately, data may be created under the new definition that has a different structure than the data created under the old definition. As data continues to be changed, the data and database definition become inconsistent. Thus, one of the major issues with respect to class versioning is insuring that the data is both backward and forward compatible. A class is said to be backward compatible if a query that is formulated after the class definition has changed can access data created before the class definition was changed in the same way as data created after the change. Similarly, a class change can be said to maintain forward compatibility if any query that is embedded in a program to access instances of a class can retrieve instances that were created after the class changes were made.

There are basically three techniques used to do class versioning: Database conversion which means that the system simply restructures the database; Class versioning with views which means that the system creates a separate view of the database whenever changes are made; and Class versioning without views which means that the system continually manages the classes as well as all the versions of the classes. Only class versioning without views supports both backward and forward compatibility.

The second major issue surrounding versions for classes is the problem of restructuring the objects to conform to any modifications to the database that occur as the result of the creation of a new version. The most straight forward approach to this condition is to simply convert the database whenever changes occur (Banerjee and Kim 1987, Kim and Chou 1988, Narayanaswamy and Bapa Raol 1988, Ariav 1991, Ewald and Orlowska 1993, Tresch. and Scholl 1993, Koerkotte and Zachmann 1993, and Breche 1996). A second approach is to wait until the user requests a version and then make the necessary changes (Penney and Stein 1987, Tan and Katayama 1989, Nguyen and Rieu

1989, and Roddick 1991 and 1992).

Similar to the discussion of object research, descriptions of some of the systems that address some of these issues are given below. The systems reflect both old and new techniques for maintaining versions of classes.

## Bjornerstedt

The AVANCE project developed general object versioning as part of their prototype OODB (Bjornerstedt and Britts 1988,Bjornerstedt and Hulten 1989) and extended this model to the versioning of class definitions. The system adopts a similar approach to ENCORE in that it uses exception handling to resolve mismatches between the expected and actual version found by the query. The exception handlers service the query with values appropriate to the version of the class. Some unusual features of AVANCE are its rigid separation between specification and implementation of classes and its strict encapsulation of objects.

## Chiueh

As a contrast to the other systems in this section, the Trait system avoids the problem of schema evolution by dispensing with the idea of a rigid schema. Chiueh (Chiueh 1994) justify their system by arguing that - "Pure type-based OODB models are too restrictive to be useful in the context of engineering design (pp 168.)" He goes on to propose a data model that supports classes and arbitrary attribute attachment.

In this system, a class definition defines the classes' initial attributes. The attributes of individual instances may then be changed arbitrarily. In a traditional, rigid class definition data model, the need for additional attributes would be handled in one of two ways: 1) By modifying the whole class to include the new attribute, or 2) By creating a new subclass for the instances requiring the new attribute and moving the instances to the subclass. This approach runs contrary to the generally accepted idea that a class definition should hold true for all instances of the class such as happens in GemStone' s representation invariant. Because of this problem, the database can have difficulty managing changes to the attributes. The schema of the database becomes a suggestion rather than the law. To some extent Trait represents a hybrid between inheritance-based systems and delegation systems and may be better suited to applications that do not require a rigid schema such as single user design applications.

## Kim and Chou

The ORION system uses a deferred approach to class versioning; that is, it updates instances of a changed class only when it is accessed (Kim and Chou 1988). Orion defines four invariants of class modification that cannot be violated if a change to a class occurs. The four invariants deal with the class hierarchy, name, origin, and full-inheritance. The Name Invariant ensures against duplicate name fields for classes, attributes, and methods. The Origin Invariant maintains single inheritance between class and attributes with similar names. The Full Inheritance Invariant means that classes inherit all attributes and methods of their ancestors, except in cases instances where there are name conflicts.

Every OODB that maintains versions for classes must contains some form of these invariants. There is often more than one way to satisfy the conditions for these invariants, so rules are sometimes used to specify how the database should be changed.

## Penny and Stein

GemStone performs class versioning similar to Orion. In addition to the Invariants defined above, GemStone introduces four more (Penney and Stein 1987). These constraints are used largely because of the differences between Orion's and GemStone's data model. That is, GemStone does not support multiple inheritance and uses a form of garbage collection rather than explicit deletion to maintain the classes in the database. This simplifies the maintenance problem considerably because it means that, unlike Orion, GemStone does not need rules to select the appropriate invariant.

## Skarra

The ENCORE system is a prototype OODB that addresses some of the problems of class versioning (Skarra and Zdonik 1987). Classes can be versioned, and the set of versions of one class is defined as the version set of the class. In every version set, there is only one version is termed the current version, and it is always the most recently created version.

One serious limitation of ENCORE is its inability to associate additional storage with existing attributes. This means that only a fixed, read only, default value can be assigned to an attribute in the pre-version form of a class, although it is possible to get around this problem by defining one or two additional attributes for each class.

In ENCORE, attributes with the same name, but referenced in different versions of a class, are assumed to represent the same information. This means that it is impossible to represent a change in the semantics of an attribute among versions, such as that which occurs when you change the units of a height attribute from cm to inches. Such a change requires the system to scale the numeric values during the conversion process.

### Zicari

O2 (Zicari 1989 and 1991) makes modifications to the database either in an incremental fashion using specific primitives or by redefining the structure of single classes as a whole.

No matter how a class is modified, O2 performs only those modifications that keep the database definition consistent. O2 supports both the immediate and deferred transformation of the database, where the deferred transformation is used by default. O2 transforms objects by means of either a series of default transformation rules or through user-defined conversion functions.

## Summary

There have been many approaches to both object versioning and class versioning, and a list of these approaches has been presented in this chapter. However there is no current approach that contains both types of versioning in a single model. The need for a unified version framework has increased over the last several years, but unfortunately such a framework has failed to appear. The subsequent chapters of this dissertation contain a description for just such a model. Although the model itself is unique, there are many elements of the UVM system that are similar to existing systems. For example, many of the ideas for implementing versioning for objects UVM can be found in (Agrawal et al. 1991). The overall concepts for versioning for classes for UVM are, in part, from (Clamen 1994). Specific ideas for implementing backward and forward compatibility are similar to (Clamen 1992). Finally, the model's dynamic conversion techniques resemble those found it (Monk and Sommerville 1993). The remainder of the dissertation now describes the model and its implementation in greater detail.

# CHAPTER III

# UNIFIED VERSION MODEL

One of the basic functions of any database system is to create and maintain different versions of the data. A database system must also provide the ability to access different versions of the data. These basic functions are also part of all object oriented database management systems. However, previous research suggests that separate systems are required to create and maintain different versions of classes and objects (Brazile and Shin 1995a and 1995b). In spite of the large number of papers in the area, researchers have proposed systems for unified versioning system of only objects or only classes. In (Katz 1990) a generalized and comprehensive object version model for engineering databases was presented, and a unified versioning for views and class changes was explored in (Kwang and McLeod 1993). However, neither of these research projects was able to unify both object and class versioning into a single model.

In contrast to the above research, this dissertation argues that a single, unified version model (UVM) for maintaining and using object and class versions is possible. The proposed model permits versions of both classes and objects to be represented in the same way. The actual distinction between the two data types need occur only at the implementation level. The UVM consists of 1) the data objects that are used to describe the model, and 2) the operations that are necessary to maintain the model. The data description includes a definition of the information that identifies the object or class from which the new version is derived, and a list of the changes necessary to create the new version. The list of the operations includes the functions that are necessary to create, delete, and access different versions of objects and classes. In addition, this particular UVM supports 1) versions that are orthogonal to type and 2) backward/forward compatibility between different versions. Details of the basic model and its operations are presented below.

## Description of the Model

The object model for the UVM consists of three basic classes: UVM, UVMNode, and UVMAttribute (see figure 1). The UVM class serves as a base class for the versioning process and is inherited by every object or class that needs to have versioning capability when it is initially defined. The primary function of the UVM class is to maintain the references to the list of different versions of the objects or classes. A separate UVMNode object is created for each new version of the object or class. Thus, the primary function of the UVMNode is to store the instance of a particular version of an object or class. UVMAttribute objects are created whenever a UVMNode is created and are used to store the differences between the old and new versions. A more detailed description of these elements now follows.

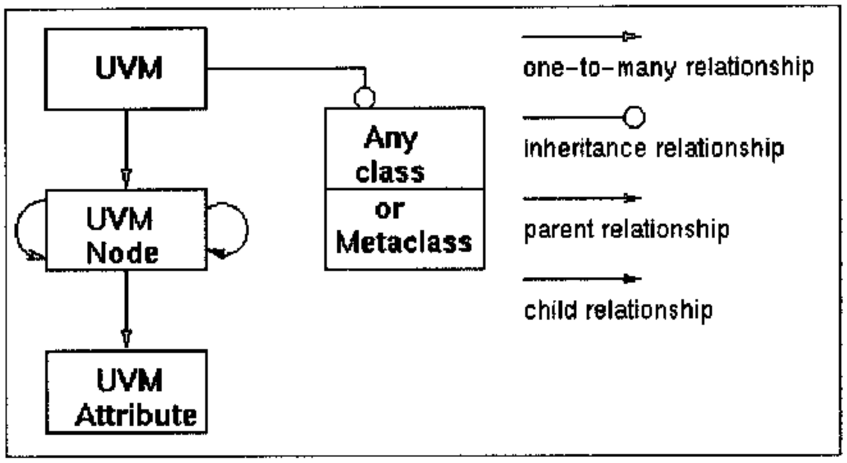

Fig. 1. Unifying Version Model

As previously mentioned, the UVM class is the base class for the versioning process and is inherited by every class or object that one can anticipate will have a version at some future time. Initially, the UVM portion of the object contains no information. But as new versions of the class or object are created, the UVM class maintains the list of references to the various versions of that class or object. The unifying property of the UVM class is apparent in that all classes and objects inherit or use the same basic structure for maintaining their versions. If the classes are ordinary, then the versions represent modifications to data in individual object instances. If the classes are meta-classes, that is, part of the definition of another class, then the versions represent modifications to the class definitions of ordinary classes. While operations for the classes and objects may differ at the implementation level, the basic data structure for versioning is the same for both classes and objects.

Whenever a new version of a class or object is created, the system generates a UVMNode. As new versions are added, a set of UVMNodes is created and stored as a tree that has references to both the parent and children of each UVMNode. Information about whether a specific UVMNode represents a version of either a class or object is stored in the versionID attribute. This is the only mechanism required to distinguish between versions of classes and objects. Because specific modifications are stored in the UVMAttribute object, the UVMNode maintains only a list of the attributes that have changed in each version. This feature also allows the UVMNode to have a one-to-many relationship with the corresponding UVMAttribute objects. The importance of the UVMNode class is that it allows the system to store and retrieve distinct versions of an object and show how these versions are related to one another. The specific attributes contained within the UVMNode are as follows:

- versionID - a number identifying the specific version of the object or class and the type (i.e., whether it is a class, object or a version of a class or object)
- references - a variable that stores the number of times this specific version has been referenced
- parentID - a pointer that designates the parent version of the object
- attribute list - list of attributes that have been changed from the generic object

- children list - list of child version objects in the version derivation tree

As previously mentioned, whenever a new version is created, the system also creates a UVMAttribute object. The UVMAttribute object stores information about the attributes of a class or object that have been added, deleted, or updated. The UVMAttribute contains information about each attribute's name, type size, position and value. Whenever an attribute is deleted, the attribute's size field is set to -1. Whenever a new attribute is created, the attribute's value field contains a system default value for the new attribute or the *update* function defined by the user. The importance of the UVMAttribute class is that this instance is created for ONLY those attributes that change. Those attributes that do not change do not have a corresponding UVMAttribute instance. Each UVMAttribute object includes the following attributes:

- attribute name - name of the changed attribute with generic object
- attribute type - type of the changed attribute
- attribute size - size of the changed attribute
- attribute position - byte position of the changed attribute in the generic object
- attribute value - changed value of the attribute

As different versions of an object/class are created, a list of the changes are kept along with a pointer to the generic object (i.e., the first version). The original version of the object is stored in the database and is not modified until the entire class is destroyed. Whenever the user asks for a specific object, the instance of the object (at the time of the request) is used to derive a specific version of the requested object. However, if the reference counter in the UVMNode exceeds a specified threshold, then the system automatically creates a new object in the database rather than continue to derive the correct version.

## UVM Operations

A previously mentioned, UVM contains operations that create, delete, and access versions of various objects and classes. While the specific details of the operations may vary depending on whether they are applied to classes or objects, the functions are basically the same for both types of data. For example, the system creates a new version of an object whenever a value of a property changes, such as when a number is changed from 5 to 6; whereas it creates a new version of a class whenever the property itself changes, such as when the length of an attribute string is changed from 5 to 6. Similarly, the system deletes versions of an object by simply deleting an instance of the object, whereas it deletes versions of a class by deleting the object itself and all other objects that represent a particular version. Finally, accessing a version of an object may require building the version by starting with the generic object and applying whatever changes are required to construct the target version. However, the system handles this building process automatically.

Table 1. provides a detailed comparison of the basic operations for the UVM's objects and classes. The table clearly illustrates that the functions differ only in the details of their implementation. A more detailed description of these functions now follows.

Table 1. Versioning functions in UVM.

| | Object | Class |
|---|---|---|
| Creating versions | $UID_{ov}$ = newVersion($UID_o$) | $UID_{cv}$ = newVersion($UID_c$) |
| | $UID_{ov}$ = newVersion($UID_{ov}$) | $UID_{cv}$ = newVersion($UID_{cv}$) |
| Deleting versions | DeleteVersion($UID_o$) | deleteVersion($UID_c$) |
| | deleteVersion($UID_{ov}$) | deleteVersion($UID_{cv}$) |
| Accessing versions | $UID_{ov}$ = parentVer($UID_{ov}$) | $UID_{cv}$ = parentVer($UID_{cv}$) |
| | $UID_{ov}$ = childVer($UID_{ov}$) | $UID_{cv}$ = childVer($UID_{cv}$) |
| | $UID_{ov}$ = prevSiblingVer($UID_{ov}$) | $UID_{cv}$ = prevSiblingVer($UID_{cv}$) |
| | $UID_{ov}$ = nextSiblingVer($UID_{ov}$) | $UID_{cv}$ = nextSiblingVer($UID_{cv}$) |
| | $UID_c$ is a UID of a class. $UID_{cv}$ is a UID of a class version. $UID_o$ is a UID of an object. $UID_{ov}$ is a UID of an object version. | |

Creating Different Versions of Objects and Classes

Objects

As previously mentioned, the user is allowed to change an object without necessarily creating a new version of the object. For example, if the user detects an error in the data, then he can change the information in the object without creating a new version. In this particular model, a new version of an object is created only when the user makes an explicit call to the function `newVersion`. It should also be noted that the system does not actually create a new version of an object. Rather it creates a new UVMNode and links it to the object. The UVMNode contains a version number and pointer(s) to the information that has been changed. Thus, whenever the system needs to retrieve a specific version of an object, it actually derives the information rather than accessing it directly.

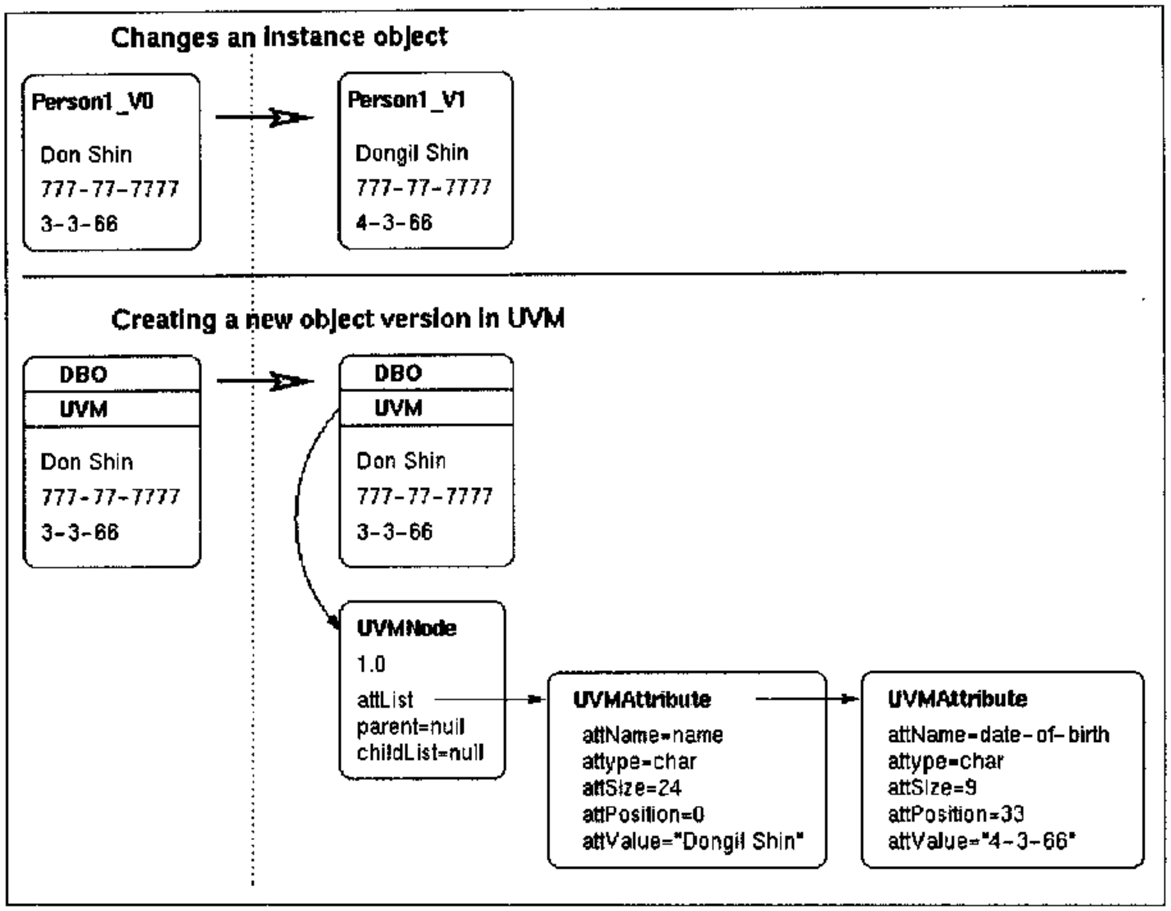

Fig. 2. Creating Versions of a Person Object.

Figure 2 shows an example of how the system creates a new version of an object. The top portion of the figure shows the original data for the Person object and the changes the user wishes to make. As is indicated, the Person object contains attributes for name, number, and date-of-birth. The changes the user wishes to make are to the name (from Don Shin to Dongil Shin) and the date (from 3-3-66 to 4-3-66) fields. The bottom portion of the figure shows how the system creates a new version of this data. The lower-left part of figure shows the original instance for the Person object. The system first notes the changes that need to be made by checking the differences between the two objects; creating a new UVMNode with the appropriate version number (in this case a 1, because it is the first version of this particular object); creating two new UVMAttribute objects to store the changes; and updating the UVMNode's attribute list to include pointers to the new UVMAttribute objects. The result of these operations are shown in the lower-right of figure 2. Each UVMAttribute stores the changes to its corresponding attribute.

## Classes

Similar to the procedure used above, UVM creates a new version of a class only when the user explicitly requests it. A new class version is created in the same manner that a new object version is created; that is, the user invokes a special function `newVersion` that takes as an argument a class id or a class version id number (UID). Although the operation is the same, the creation function for classes is somewhat different. For example, the system maintains all the class definitions in a meta-class called a Persistent Class Type (PCT). Similar to other classes in the database, the PCT inherits its versioning capabilities from UVM. Therefore, whenever a new version of a class is created, the system actually creates a new version (i.e., a UVMNode instance) of the PCT instance that represents the specified class. For example, the Person class used in the previous example would also have an instance of the PCT class called "Person." Associated with the PCT instance of Person would be a set of instances of Attributes, each defining one of the attributes in the Person class. For this particular Person class, there are three Attribute instances: name, number, and date-of-birth. Thus, as changes to the class definition are made, the UVMAttribute instances contain the changes to the Attribute instances associated with the PCT instance. This particular operation is illustrated in figure 3.

Again, the top portion of figure 3. shows the original data for the generic Person class along with the changes the user wishes to make. The original class definitions for Person include name, number and date-of-birth. The changes to the class appear in the upper right hand corner of the figure and show that the user wishes to add a new attribute called address and to change the number from character to long. The system view of the original Person class is represented in the bottom left-hand corner of the figure. The right-hand corner picture illustrates what happens when the system creates a new version of the Person class. Similar to the object example, the system adds a UVMNode with version id 1.0 to designate the newly created node as being the first version of the Person class. Two UVMAttribute objects are then created and linked to the UVMNode. Each UVMAttribute object represents addition/deletion/changes to the corresponding attribute.

Therefore, one UVMAttribute is created for the change to the number and another is created for the addition of the address field. The difference between this versioning procedure and the preceding object example, is that the UVMAttribute instances are recording changes to the class definition; that is, changes to the attribute definitions of the class rather than changes to the actual data associated with the attributes of the instance of the class.

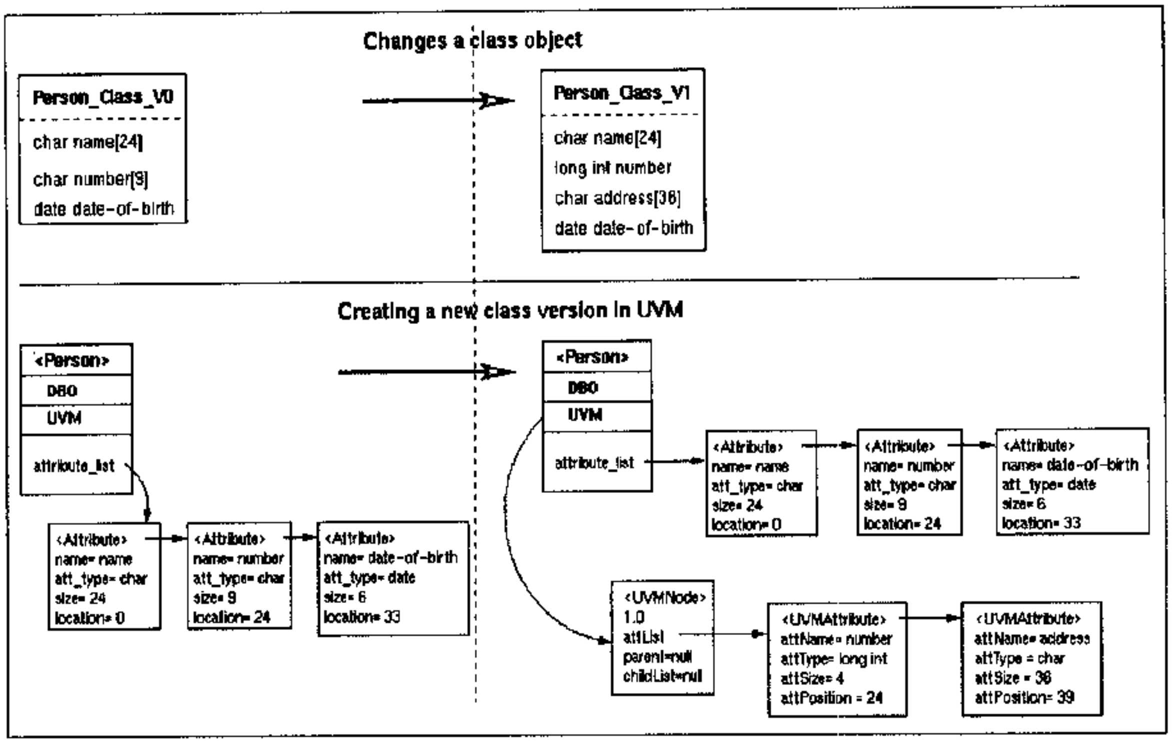

Fig. 3. Creating Versions of a Person Class.

## Deleting Versions of Objects and Classes

The UVM allows the user to delete entire objects and classes or a specific version of an object or class. The delete functions for objects and classes act in a similar way. For example, if the user calls the delete object function (i.e., `deleteVersion`) with an ID value consisting of an object version ID, then the system deletes the specific version of that object. On the other hand, if the user calls the delete function (i.e., `deleteVersion` with an object ID, then the system deletes the object and all versions of that object. In a similar manner, if the user calls the delete class function (i.e., `deleteVersion`) with a class version ID, then the system deletes the version of the class specified. If the user calls the function with just a generic class ID, then the system deletes the class, all instances of that class, and all versions of that class. Whenever the system deletes an object or a class, then that object or class is marked as deleted. If the user deletes a version of a class or object, then the corresponding version is deleted although access to the children of the class or object is still allowed.

When a new version of a class is defined, the user can attach special update/backdate functions or default values to particular fields of the class or object. In the above example, the user added an address field to the new version of the class. At this particular time, the user can specify a default value for older instances that may be converted to the new class definition. Alternatively, the user can specify a function, such as a lookup function or a data input form function, that would supply a value for the address attribute when needed.

## Accessing Versions of Objects and Classes

UVM keeps a copy of the generic objects and classes and stores versions of the different objects and classes as changes occur. UVM does not actually create the object represented by the new class until runtime. The technique used to do this is called Dynamic Object Conversion (Monk 1993). More specifically, UVM stores only the changes and generates the correct version of the object whenever it is requested. Thus, UVM creates the requested version from the generic version (i.e., delta storage) rather than storing every copy of the version (i.e., omega storage). A more complete description of the specific method used to do the dynamic object conversion in UVM is given in Chapter 4.

There are four basic functions that are used to access the appropriate version of the object or class. Again, the system recognizes that it should retrieve a version of either an object or class by checking the UID argument passed to the specific function. If the ID refers to a class, then a class accessed. If the ID refers to an object, then a version of an object is retrieved. If the user simply asks for an object or class, then the default version of the object or class is retrieved. If the user wants to access the object or class from which a specific version was derived, then he uses the `parentVer` function. The functions `childVer`, `prevSiblingVer`, and `nextSiblingVer` are used to access the child version of the specific version.

## Accessing Instances When There Are Versions of Classes

As new versions of a class are created, each derivation is added to a tree. As a result the system has a history of all the changes that are made to the database. For example, if a new version is created from the original object, it becomes a sibling of version 1.0. If a new version is, in turn., created from version 1.0, it becomes a child of version 1.0. By following a path through the version history tree, it is possible to convert an instance of any version of the class definition to any other version. This ability to do automatic conversion allows the system to treat any instance as an instance of the class as a whole rather than as a particular version of the class. The actual version to which a particular instance belongs at any specific time is transparent to the user because UVM automatically converts the instance to the version requested by the user. UVM supports not only forward conversion (i.e., from the old version to the new version) but also backward conversion.

As the user creates a new version of a class, he or she can specify default values or update/backdate functions for any attribute or value for the class or object. The user specifies default values for attributes which are used to convert instances between different versions of classes. The update functions are specified for attributes and tell the system how the attribute should be converted in the next version. The backdate function does the reverse of this. These update/backdate functions are stored with the class versions and are invoked by the system whenever the instance(s) are accessed and need to be converted.

## A Complete Example

This section presents a complete example of the UVM and how it handles the creation and retrieval of different versions of objects and classes. It is intended to demonstrate the overall system and how it responds to the creation of different versions of classes and objects.

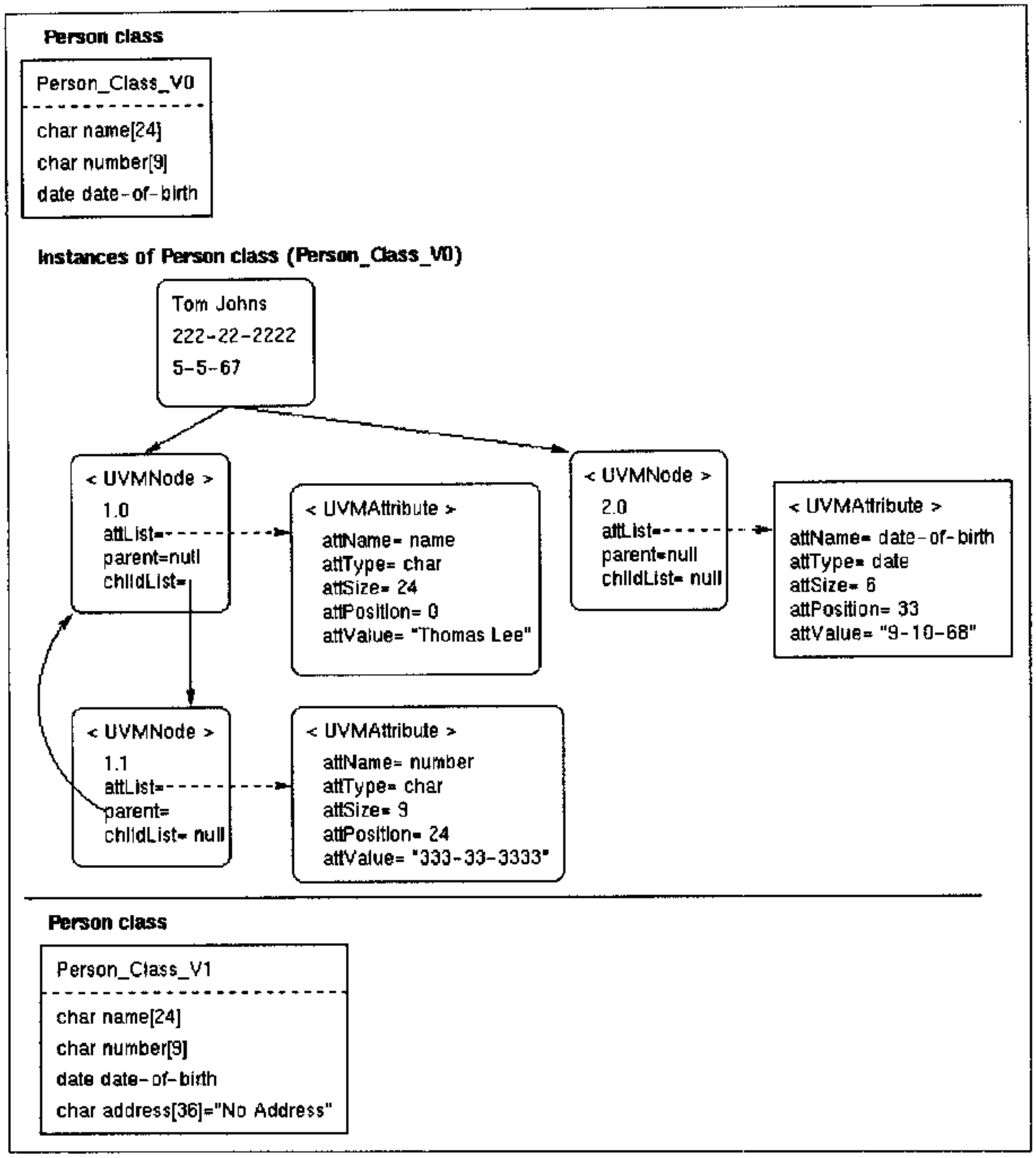

Fig. 4. Creating Versions of an Instance and a Class.

As the upper portion of figure 4. shows, the user begins by creating the Person class and one instance of the Person object with a name attribute of "Tom Johns," a number attribute of 222-22-2222, and a date-of-birth attribute of 5-5-67. The user then creates version 1.0 for "Tom Johns" and changes the name field from "Tom Johns" to "Thomas Lee." In version 1.1 the user changes {\em number) from "222-22-2222" to "333-33-3333".

The user then creates a version 2.0 and changes the date-of-birth field from 5-5-67 to 9-10-68. Since version 2.0 is the last version to be created, it becomes the default version of the "Tom Johns" instance.

The user then adds a new version of the Person Class and a new attribute of Address (see the lower portion of figure 4.). The user specifies "No Address" as the default value for this new attribute. UVM builds a new version of the class similar to figure 3.

Figure 5. shows what happens when the user accesses the 1.1 version of "Tom Johns," which of course is now "Thomas Lee." The system first recognizes the 1.1 version which contains the new number of 333-33-333. Finally, it recognizes that a new version of the Person Class has been created. Because the user has not specified an address for this object, the system simply uses the default value of "No Address." The system then ends the process and displays the information to the user.

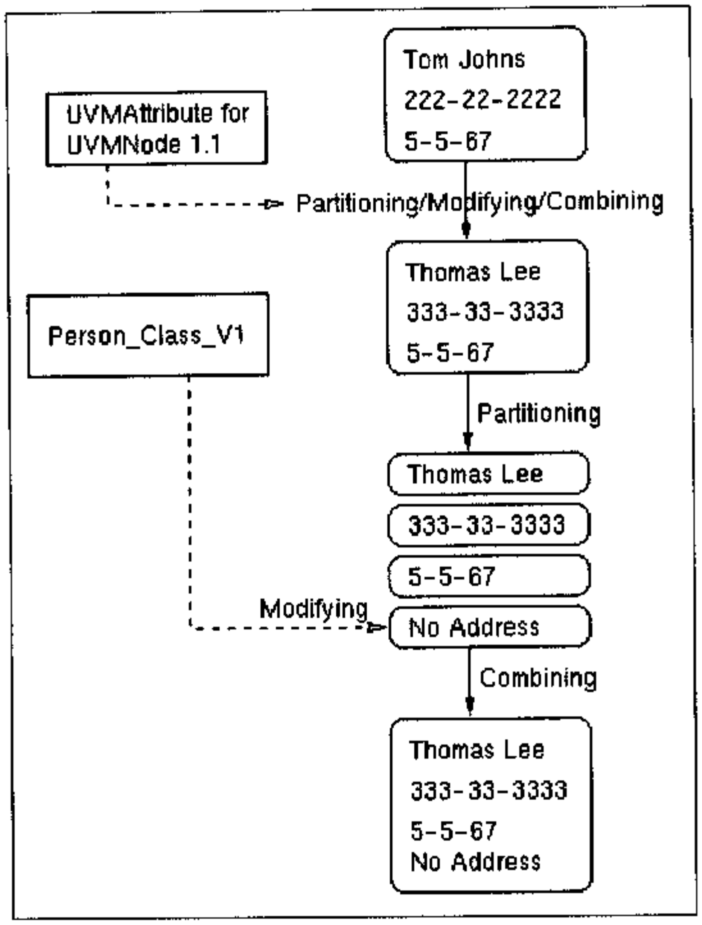

Fig. 5. Accessing Version of an Instance for the Different Class

# CHAPTER IV

# IMPLEMENTATION OF UVM

This chapter describes several issues related to the implementation of UVM. The first section discusses the Database Object System (DBO), a system that was used to manage the persistent objects for UVM. Section 4.2 discusses the unique object identifier (UID). Section 4.3 presents detailed information about the classes that were used to implement versioning in UVM. Finally, section 4.4 discusses special implementation issues such as dynamic object conversion and class conversion.

## Persistent Objects and the DBO Class Library

The system has been implemented using a persistent object manager called the Database Object System (refer to Appendix B) that runs on both DOS and UNIX platforms. DBO is not a fully functional object-oriented database management system, but rather an object manager that allows users to store and retrieve objects. The DBO Class Library consists of a set of C++ classes that can be used to create and maintain **persistent** objects (i.e., objects that continue after the life of the program). Because DBO writes the persistent objects to stable storage, subsequent programs can retrieve them.

The idea behind DBO is that all the persistent classes are derived from a root persistent class called DBO. This root class contains methods that know how to store and retrieve objects from the database. Further, all persistent classes are derived from the DBO class using a *public* derivation method. A complete description of the DBO system is presented in Appendix B. In order to accommodate versions of classes and objects, the embedded UID variable within the DBO model was modified to include a new type of identifier of the different versions classes or objects, and a pointer variable was added to record the version's original object. Figure 6. illustrates how DBO was used to accommodate the idea of versions. The dotted lines illustrate how a new version is stored in DBO. As new versions are created, they are represented in the system as a tree. Each node of the tree is an instance of the class UVMNode (see below for details). A version's place in the tree is determined when the new version is created.

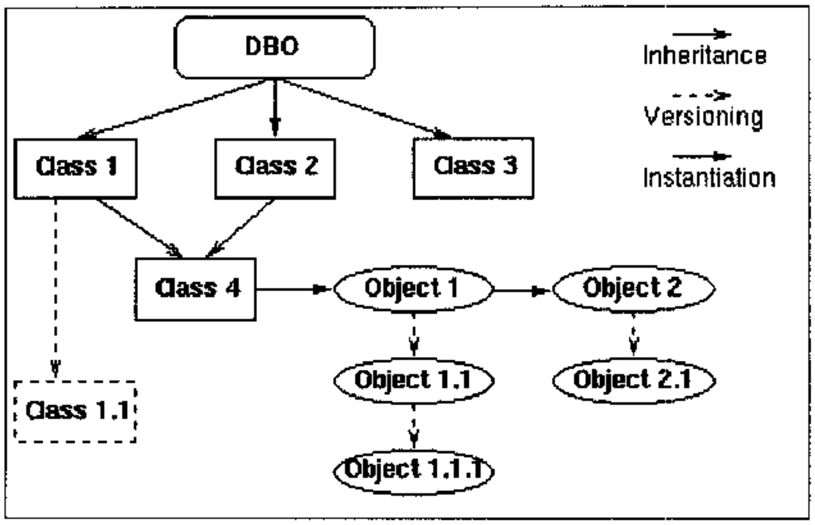

Fig. 6. A Derivation Hierarchy of the Unifying Version Model.

## Using DBO to store Versions of Objects and Classes

As previously mentioned, UVM uses DBO to store versions of objects and classes. In attempting to use DBO for this purpose, several versioning techniques were considered. One method is to store the different versions as a collection of pointers to values (Ecklund et al. 1987). Another method is to store the original object and only changes to the object (Ecklund et al. 1987). A third method is to store the original object and changes to the object, but also a counter that records the number of times a specific version of an object is accessed along with a pointer to the original or generic object.

Since the first two methods require a great deal of execution time in order to recreate the requested version, they were not implemented in UVM. Although the third method requires some additional storage, it uses considerably less time to recreate the requested version. Thus, the third method was used as the method for storing versions of classes and objects in UVM. Each version of a class or object maintains counter and pointer variables. Whenever the user accesses a specific version of a class or object, the counter value is increased by one. If the counter exceeds a specified threshold, the system makes a new copy of the object with the changed values and saves a pointer to the derived version. This implementation needs only a small amount of additional storage space but provides fast access to frequently referenced objects. However, access to infrequently referenced versions requires substantially more time. Therefore, it is very important to find an optimal threshold value. As a result, this feature was created so that the user could modify the variable whenever necessary.

## Modifying the UID for Versioning

The system is able to unify versions because it creates a single type for both objects and classes. By specifying a single type for both classes and versions, the system can call the same functions for either classes or objects. This is accomplished by creating a special UID number and storing it in the UID field. The system stores information in the UID field that indicates where the version is stored and whether the location refers to a class, a version of a class, an object, or a version of an object. Data items that refer to classes are assigned a 0; versions of classes are assigned the number 1; objects are assigned the number 2; and versions of objects are assigned the number 3. The following class definition indicates how the system specifies this field.

```
enum ID_KIND {Class, Class_Version, Object,
     Object_Version};

class UID {
     private:
          long uid;
          ID_KIND kind;
     public:
         ... uid manipulation functions ...
};
```

A version number for an object with a unique UID can be represented as follows:

`UID`[.*version number*]$^*$.

The very first version of a class is given the version number of CUID.1 where CUID is a class UID. The second, is assigned the version number of CUID.2, etc. Similarly, the first version of an object is assigned the version number OUID.1 where OUID is an object version UID, etc. The first version derived from version OUID.1 would be given an identifier of OUID.1.1. Through this method, the system can derive the location, type and version of a specific object, class or version of a class or object.

Instances of versions are objects in their own right and, therefore, are uniquely identified to the system using the above version number scheme. They are related to a common generic instance from which attributes and default values can be inherited. The system does not need to maintain the ancestor/descendent relationships between the instances of versions because the number scheme itself encapsulates this relationship.

## The Set of Classes for UVM

A specific class is given the ability to have versions by designating it as a subclass of UVM. Thus, any class given this designation inherits the generic class *UVM* as one of its base classes. A tree, in turn, is used to represent the set of versions of an object of versioned objects and each node of the tree is in the class *UMNode*. The exact location of a version in the tree is determined by when the new version is created and which version it changes.

A new version (except the initial generic version) is always based on an existing version; the new version has a derived-from relationship to the version it is based on. The derived-from relationship is established when the version is created and does not change even when updates occur. As previously stated, the derived-from relationship between versions of an instance are represented as a tree. Although other researchers have used an acyclic graph to represent the derived-from relationship (e.g., (Banerjee et al. 1986 and Banerjee and Kim 1987)), others have argued that a tree is sufficient for this particular application (Chou and Kim 1986).

Figure 7. illustrates an example of how versions are stored in UVM. The tree's nodes identify the versions, and the arcs represent the derivation relationships. Several parallel versions can be derived from a single version. In this particular example, the default value for the object is the latest version created (i.e., O[3]).

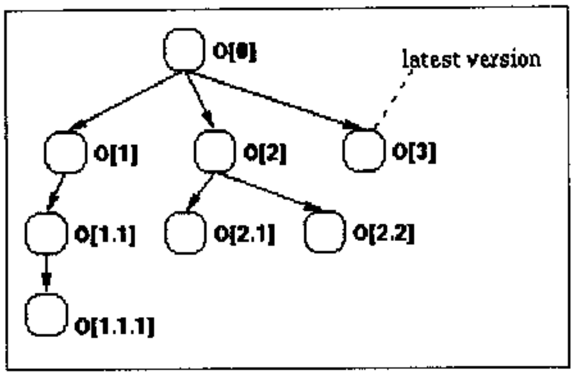

Fig. 7. Version Derivation Tree.

The rest of this section describes the important classes within UVM.

## Class *UVM*

The UVM class is considered a base class and must be inherited by any class or object that has a version of itself. Because of its "meta-class" status, it is assigned a set of protected constructors that guards against any objects of the class *UVM* being created.

A *UVM* object contains a pointer member that links the newly created object into the version tree. It also contains member functions that retrieve information from the tree, create new versions, and delete old versions. The system first copies the values and attributes of the old version into the new version and then inserts the new version as the rightmost child in the version tree.

## Class *UVMNode*

The *UVMNode* class is used to store a version of a particular object. The UVMNode acts as a node in a version derivation tree. It contains attributes as explained in the previous chapter. More specifically, it contains attributes for a version id, pointer to the parent version, a list of child versions, and a list of changed attributes. It also includes functions for manipulating attribute lists and child version lists.

## Class *UVMAttribute*

The *UVMAttribute* object stores changes for the corresponding attributes among the different versions. As stated in the previous chapter, UVMAttribute contains information about any changes to the generic object's name, type, and size. It also contains the new value of the changed attribute and the position of that value in the generic object. Thus, the system uses this information to display to users whenever they request a specific version of an object or class.

## Special Implementation Issues

One of the major concerns for UVM as well as other versioning models is the problem of providing support for changes to the database schema or meta-classes. More specifically, a versioning system must be able to resolve any inconsistencies that arise as a result of a change to any part of the schema. These inconsistencies often occur whenever a user changes the definition of a class. The result of such changes is that instances generated from any previous version of the class become incompatible with the new version.

Several solutions to this problem have been suggested by a number of different researchers. One technique, called class conversion, simply restructures the instances of the modified classes until they agree with the current representation of the classes. Class conversion is supported in the ORION (Banerjee et al. 1986 and Banerjee and Kim 1987) and GemStone systems (Penney and Stein 1987). The primary weakness of this approach is that it does not support backward/forward compatibility. This term is to describe the ability to access previous versions of an object at any time without losing data. Because systems that use class conversion discard all former versions of the object whenever they make the changes, the application programs that must use the former versions of the objects soon become obsolete.

One alternative to the above approach is to maintain all the versions of classes. This technique is called class evolution (Skarra and Zdonik 1986 and 1987). Each "evolution" of the class defines a new version of the class. Thus, old class definitions persist indefinitely or until the user deletes the entire class. Instances and applications are associated with a particular version of a class, and the system is responsible for simulating the semantics of the new class on top of instances of the old, or vice versa.

In the Unifying Version Model (UVM), the schema is treated as a set of current versions of classes, and the schema is maintained as instances of classes designed to contain the necessary information for representing a particular schema. Moreover, a class is constructed with **slots** that contain information about each attribute and method available to the current version of the class. To avoid any loss of data during class evolution, UVM stores the first version of an instance or a class object and keeps track of only the changes made to the object. Any version of the generic object can be accessed dynamically at runtime.

Unlike evolutionary systems such as ENCORE (Skarra and Zdonik 1986) and AVANCE (Ahlsen et al. 1983, Bjornerstedt and Britts 1988, and Bjornerstedt and Hulten 1989), UVM *dynamically* converts objects between different versions rather than emulating other versions. This conversion is *reversible*, and any object in the system may be freely converted between different versions of its classes. In addition, objects are partitioned into pieces while being converted back and forth between versions of classes. Therefore, the term Dynamic Object Conversion is used to describe UVM.

Since instances of objects can be converted from one format of a class version to any other, it is best to consider instances as having an indeterminate type, and that they are given a certain type only when they are accessed. The instances may only be accessed in one of the representations provided by the class versions. The class version is determined by functions used to extract instances from the system.

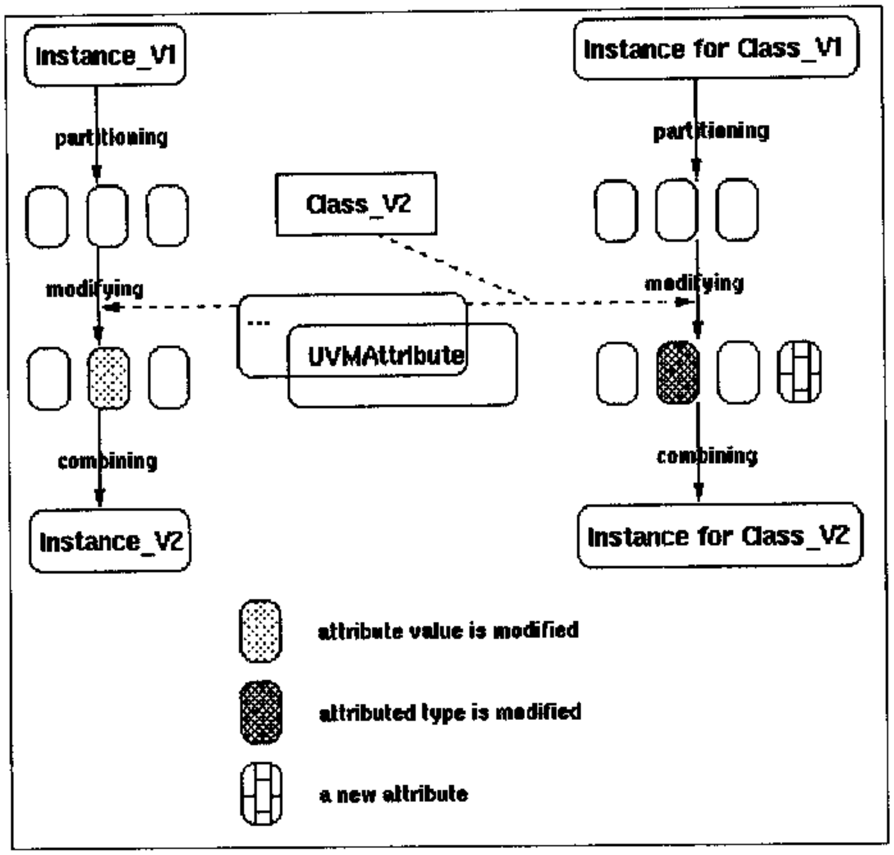

Fig. 8. Dynamic Object Conversion.

Figure 8. illustrates an example of how dynamic object conversion is implemented in UVM. In this particular example, a version of the V1 instance object is partitioned, modified, and combined into a new version (i.e., V2) of the instance. The partitioning is accomplished with information taken from the PCT for the class.All changes to the attributes are extracted from UVMAttribute. When an instance for a class of version V1 is accessed as an instance of the class of version V2, it is first partitioned into attributes. Again this partitioning uses information from the PCT. Next, a special modifying procedure extracts attribute information from class V1 and V2 and uses this information to build a new set of attributes for class V2. Finally, all modified attributes are combined into the instance for class V2.

## The C++ Specification for UVM

Portions of the specifications of the classes used to support UVM are described in Appendix A. The listings include a description of the classes and methods used to implement the major features of UVM. Readers need to be familiar with C++ to understand some of the specific implementation issues specified in these listings.

# CHAPTER V

# CONCLUSION

In this thesis, various research efforts on object and class versioning systems were presented, and the need for a unified version model (UVM) for both classes and objects (instances of classes) was suggested. The major reason for designing the UVM was to introduce a simple but semantically sound version model that uses a single set of operations to support versioning for both objects and classes. Other OODBMS systems consider versioning as two problems: one that looks at problems related to objects, and a second that examines issues concerning class versioning. UVM combines these functions into a single model, thus providing a system that is both easier to learn and conceptually simpler. UVM is able to unify both objects and classes into a single model because it forces classes and objects to inherit their versioning functions from a single meta-class (i.e., UVM) . Such a concept greatly reduces the number of methods that have to be supplied for both classes and objects. In addition, UVM uses the UID attribute to store information about whether the particular instance represents a class or object. As a result, UVM can apply similar functions to both classes and objects, making the need to differentiate between the two transparent to the user.

## UVM Features

Because UVM can be used to create versions for both classes and objects, it contains the following features:

- *One versioning model for both objects and classes*: Operations for creating, accessing and deleting versions of objects or classes are the same.
- *Version orthogonality*: Versions of an instance or a class can be created without requiring any change to the object or the class. Thus, the decision to create a version can be made as needed instead of when the database is initially designed.
- *Backward/Forward compatibility with data retention*: Backward/forward compatibility is provided by using dynamic object conversion. Class versioning using a delta storage scheme supports program compatibility and a variety of class evolutions without data loss.Class versioning also permits multiple views of the database definition to coexist.
- *Dynamic object conversion*: Objects are converted only when needed at runtime. To access a specific version of an object (an instance or a class), the system divides the object into attributes, modifies it, and then merges the changed information into the new version of the object.

## Implementation Results

In order to test the UVM model, an actual OODBMS system was been built as an extension to DBO. Six basic functions were developed to create versions. These functions are: `newVersion`, `deleteVersion`, `parentVersion`, `childVersion`, `pervSiblingVer`, and `nextSiblingVer`. The system distinguishes between classes and objects by checking the UID argument passed to the specific function. A new version of an object or a class is created only when the user calls the function `newVersion` explicitly. The user can delete entire objects and classes or a specific version of an object or class by calling `deleteVersion`. The functions of `parentVersion`, `childVersion`, `pervSiblingVer`, and `nextSiblingVer` are used to access the appropriate version of the object or class. When the user accesses objects for different versions of classes, the system creates the object represented by the specified version at runtime.

The system was tested using data for a simple student information system. The system allows the user to store personal information, change the class structure by adding or deleting attributes, and access different versions of classes and objects. The database for the student information system included one class (Person), containing three initial attributes. Five objects were initially created for the *Person* class, and four different versions of *Person* class were created and named CUID.1, CUID.2, CUID.2.1, and CUID.2.1.1. For each version of the class, five objects were created. The versioning capabilities were then tested by accessing fifteen objects for each version of the class. The system successfully retrieved the correct version of the class for all cases.

## Potential Disadvantage

The system was implemented using C++ and the UNIX operating system to prove the feasibility of the design of the model. However, a full performance test was not completed. One of the potential problems with the design is the amount of time that may be needed to convert the instances of the different versions of objects. For certain types of applications, the time required to access and convert the different versions of objects and classes may be unacceptably slow. However, in those database applications where the number of versions is high but the number of instances low, UVM should perform well. A more complete series of performance tests should be done to determine a range of performance values for different types of problems.

## Future Research

There are a large number of areas that need further investigation as a result of the work suggested in this dissertation. A few of these areas are listed below:

- The development of a new mechanism to support class evolution.
- A validation of the model using real applications.
- An evaluation of the three storage mechanisms proposed in Chapter 4.

### Class Evolution

The current system does not address some of the changes that involve modifications to the class inheritance tree. A potentially fruitful area of versioning research would be to devise a *class set versioning* system that would allow a system to create a version of a whole set of related classes (that could comprise the entire database definition). Class set versioning appears to be a promising area of future research that would extend the range of forward/backward compatibility changes that can be made to a database definition.

## Evaluation

Industrial strength object-oriented database management systems have been limited to class evolution strategies that use database definition modification rather than class versioning techniques. This means that the utility of class versioning has never been tested in the real world. It would be interesting to incorporate a class versioning system such as the one used to implement UVM into a commercial object-oriented database management system and evaluate its usefulness in this area.

## Alternate Implementation Strategies

As suggested in Chapter 4, there are three different ways to store and maintain information about versions. These three ways are: (a) store the different versions as a collection of pointers to values; (b) store the original object and only changes to the object; and (c) store the original object and changes to the object, but also a counter that records the number of times a specific version of an object is accessed along with a pointer to the original or generic object. One significant line of research would be to compare the performance of these three methods and determine which method was most efficient and effective. The following criteria could be used to evaluate their performance.

- Number of pages accessed for processing the same set of user queries
- Number of conversions performed to retrieve the correct version
- Total amount of storage required for processing the same set of user queries

A comparison of these three Strategies would provide valuable insight into how version information should be maintained.

## Conclusion

The last few years has seen a rapid growth in both object-oriented languages and object-oriented databases. As the need for such systems arise, the current object oriented database systems must become more sophisticated to keep pace with the demands of applications in new domains such as CAD/CAM, graphics information systems, and multimedia. As the objects of such applications become more and more complex, the possibility that there will be changes to these objects increases, and the need for better support for object and class versioning rises. The model presented in this thesis attempts to meet this challenge. It provides a unified model for both class and object versioning. Such a model greatly simplifies the conceptual framework that is needed to develop the next generation of object oriented database management systems.

## APPENDIX A.

## C++ Class Definitions

```
class UVM {

// Constructors and destructors are protected, so the users
// cannot create Version objects.
protected:
     UVM(void);
     UVM(void);

public:
// functions for creating and deleting versions
     UVMNode* newVersion(UVMNode* parent);
     UVMNode* newVersion(VID& parent);
     UVMNode* newVersion(void);
     void delVersion(VID v); // for deleting version

// functions for navigating around the version set
     UVMNode* parentVer(void);
     UVMNode* childVer(void);
     UVMNode* prevSiblingVer(void);
     UVMNode* nextSiblingVer(void);
     void resetTraverse();
     void activateTravVersion();
     void activateCurrVersion();
     void displayUvm();

// public data
public:
     Plist *root;
     UVMNode* currentVersion; // the latest version
     UVMNode* travVersion;
          // for traversing version tree
     int counter;
          // the number of existing version instances
}

class UVMNode {
private:
     UVMNode();
     UVMNode();

public:
// attribute list manipulation functions
     int addAttribute(UVMAttribute* att);
     int deleteAttribute(char *attname);
     UVMAttribute* firstAttribute();

     UVMAttribute* nextAttribute();

     void activateAttList();


// child list manipulation functions

     int appendChild(UVMNode* node);

     int deleteChild();

     UVMNode* firstChild();

     UVMNode* nextChild();

     UVMNode* lastChild();

     void activateChildList();


// attribute accessing functions

     int getReferences();

     void incReferences();

     VID* getVID();

     UVM* getRoot();

     void setRoot(UVM* r);

     UVMNode* getParent();

     void setParent(UVMNode* p);

     UVMNode* getCurrentChild();

     void setCurrentChild(UVMNode* c);
```

```
protected:
     Plist *attList;
          // a linked list of changed attributes
          // along with values
     Plist *childList;
          // a linked list of child versions
     UVM* root;
          // root of the version tree (pointer to UVM)
     UVMNode* parent;       // parent of this node
     static UVMNode* currentChild;
     VID versionID;
     int references;        // reference counter
};

class UVMAttribute {
private:
     UVMAttribute(char *dboName, char *attName, char *type,
int size,
     int offset, void *value=NULL);
     char* getAttName();
     char* getAttType();
```

```
    int getAttSize();

    int getAttPosition();

    void* getAttValue();


public:

    char attName[MAX_ATT_NAME];

        // name of the changed attribute

    char attType[MAX_TYPE_NAME];

        // type of the changed attribute

    int attSize;        // size of the attribute in byte

    int attPosition;

        // starting byte position of this

        // attribute in the original object

    char* attValue; // a storage for the new value will be

                    // allocated dynamically by the size

};
```

## APPENDIX B.

## Introduction to DBO: A Simple Object Manager

## B.1 Introduction

The DBO Class Library is a set of C++ classes to be used for creating and maintaining persistent objects. The DBO has been developed on DOS, Windows, Solaris, FreeBSD, and Linux. The main idea of DBO is that any object which is to be persistent must be an instance of a class which has been derived from the root persistent object DBO. DBO has methods which know how to store and retrieve objects from the database. Thus all persistent classes are derived from DBO using a *public* derivation. For example:

```
// ***********  person.h  *************
#include <string.h>
#include "dbo.h"

class Person; //forward reference

class Job: public DBO {
public:
    Job(char *s, int i):DBO("Job",s,sizeof(Job))
    { salary=i; boss=NULL; }
    void display();
protected:
    Person *boss;
    int salary;
};

class Person: public DBO {
public:
    Person(char *s, char *n):DBO("Person",s,sizeof(Person))
    { strcpy(ssn,n); father=mother=NULL; job=NULL; }
    void display()
    {    cout << name << " ssn: " << ssn << "\n";
         if ((job != NULL) && ((UID)job != INACTIVE))
             job->display();
    }
    void setjob(Job *j) { job = j; }
    void setf(Person *f) { father = f; }

protected:
    Job *job;
    Person *father;
    Person *mother;
    char ssn[12];
};
```

```
void Job::display()
{
    cout << "salary: " << salary << "\n";
    if ((boss != NULL) && ((UID)boss != INACTIVE))
        boss->display();
}
```

The constructor for the class must call the constructor for class DBO and pass the arguments classname, objectname and class size. Every object has a name which is kept in an attribute which is inherited from class DBO. This name should be unique because it can then be used to *lookup* the object using the name as a search key. The name of the object must only be unique within the class for this to work properly. If an object is not given a unique name, it may still be found if it is a component of another object which does have a unique name. (see the Complex Objects section)

When an object is created for the persistent class, it is put into a persistent class buffer which will be saved to the database if the transaction which created it commits. If the transaction aborts, the object will not be saved to the database.

When an object is read in from the database, it will not be written back to the database unless the programmer issues a *putObject* command. This causes the system to write the object back to the database if the transaction commits.

## B.2 Complex Objects

Complex objects are objects which *contain* other objects. The idea of *contain* may be that object *a* is physically contained within object *b*. In this case, object *b* cannot be referenced outside the scope of object *a*. If object *a* is a persistent object then object *b* will be stored on the database, too, because it is physically inside of object *a*.

Alternatively, object *b* may be only referenced inside of object *a*. In the above example of Person, the attributes job, mother and father are references to other persistent objects. If object *a* is a persistent object and it references object *b*, then object *b* should be a persistent object also. To use the DBO class library, the reference must be a pointer to object *b* and the fact that it is referenced must be in the Persistent Class Table (PCT). The PCT must be built by the user and contains the name of each persistent class and the number of other persistent objects that are referenced. For example:

```
DBO       0
Pnode     3
Plist     2
Student   0
Keyword   2
Query     1
Element   1
quit      0
```

The format of the file is *class name, number of references to other persistent objects*. This set of data is repeated for each persistent class. The last line of the file has the keyword *quit*. Look at the entry for the Person class for an example of the PCT data for the Person class defined above.

## B.3 Using the DBO Class Library

To access the database, the application opens the database and starts a transaction. It may then access objects in the database, make changes to them and write them back. When the application has completed a consistent piece of work, it commits the transaction, and actually causes the database to be updated with the changes it has made. Alternate

# BIBLIOGRAPHY

Agrawal, R. and Gehani, N.H. 1989. Ode (Object Database and Environment): The Language and the Data Model. In *Proceedings of the ACM-SIGMOD 1989 International Conference on Management of Data*, 36-45, Portland, Oregon: ACM Press.

Agrawal, R., Buroff, S., Gehani, N. and Shasha, D. 1991. Object versioning in ode. In *IEEE 7th International Conference on Data Engineering*, 446-455: IEEE Computer Society Press.

Ahlsen, M., Bjornerstedt, A., Britts, S., Hulten, C. and Soderlund, L. 1983. Making Type Changes Transparent. In *Proceedings of IEEE Workshop on Languages for Automation*, 110—117, Chicago: IEEE Computer Society Press.

Ahmed R. and. Navathe, S.B. 1991. Version management of composite objects in cad databases. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, 218-227, Denver, Colorado: ACM Press.

Andany, J., Leonard, M. and Palisser, C. 1991. Management of schema evolution in databases. In *Proceedings of the 17th International Conference on Very Large Data Bases*, 161-170, Barcelona, Spain: Morgan Kaufmann.

Ariav, G. 1991. Temporally oriented data definitions: Managing schema evolution in temporally oriented databases. *Data and Knowledge Engineering*, 6:451-467.

Atkinson, M., Banchilhon, F., Dewitt, D. and Dittrich, K. 1989. The object-oriented database system manifesto. In *Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases*, 223-240, Kyoto, Japan: Springer-Verlog.

Banerjee J. and Kim, W. 1987. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Record*, 3:311-322.

Banerjee, J., Kim, W., Kim, H. and Korth, H.F. 1986. Schema Evolution in Object-Oriented Persistent Databases. In *Proceedings of the 6th Advanced Database Symposium*, 23-31, Tokyo, Japan.

Batory, D.S. and Kim, W. 1985. Modeling concepts for vlsi cad objects. *ACM Transactions on Database Systems*, 3:322-346.

Bertino, E. 1992. A view mechanism for object-oriented databases. In *Proceedings of the 3rd International Conference on Extending Database Technology*, 136-151, Vienna, Austria: Springer-Verlag.

Bjornerstedt, A. and Britts, S. 1988. Avance - an object management system. In *Proceedings of the Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, 206-221, San Diego, CA: ACM Press.

Bjornerstedt, A. and Hulten, C. 1989. *Version control in an object-oriented architecture*. Edited by W. Kim and F. Lochovsky, Object-Oriented Concepts, Databases and Applications. Addison-Wesley.

Bratsberg, S.E. *Evolution and Integration of Classes in Object-Oriented Databases,* Ph.D.Dissertation, Norwegian Institute of Technology.

Brazile, R.P. and Shin, D. 1995a. A unifying version model for objects and schema. In *Proceedings of International Conference on Intelligent Information Systems*, 245-248,Washington D.C.

Brazile, R.P. and Shin, D. 1995b. A unifying version model for object-oriented engineering database. In *Proceedings of the 9th Annual ASME Engineering Database Symposium*, 221-224, Boston, MA.

Breche, P. 1996. Advanced primitives for changing schemas of object databases. In *Proceedings of CAiSE '96 Conference*: Springer Verlag.

Casais, E. 1990. Managing class evolution in object-oriented systems. In Tsichritzis, D. (Ed.), *Object Management*, 133-195, Geneva: Centre Universitaire d'Informatique, University of Geneva.

Chiueh, T. 1994. Papyrus: A history-based vlsi design process management system. In *Proceedings of the 10th International Conference on Data Engineering*: IEEE Computer Society Press.

Chou, H. and Kim, W. 1986. A unifying framework for version control in a cad environment. In *Proceedings of the 12th International Conference on Very Large Data Bases*, 336-344: Morgan Kaufmann.

Chou, H. and Kim, W. 1988. Versions and change notification in an object-oriented database system. In *Proceedings of the 25th ACM/IEEE Design Automation*

*Conference*, 275-281: ACM Press.

Clamen, S.M. 1991. Managing type evolution in the presence of persistent instances, Ph.D. Dissertation Proposal, Carnegie Mellon University.

Clamen, S.M. 1992. Type evolution and instance adaptation: Carnegie Mellon University, CMU-CS-92-133R.

Clamen, S.M. 1994. Schema evolution and integration. *Distributed and Parallel Databases*, 1:101-126.

Ecklund, D.J., Ecklund, E.F., Eifrig, R.O., and Tonge, F.M. 1987. Dvss: A distributed version storage server in cad applications. In *Proceedings of the 13th VLDB Conference*, 443-454, Brigton, England.

Ewald, C.A. and. Orlowska, M.E. 1993. A procedural approach to schema evolution. In *Proceedings of the 5th International Conference on Advanced Information Systems Engineering,* 22-38, Paris, France: Springer-Verlag.

Katz, R.H. 1990. Toward a unified framework for version modeling in engineering databases. *ACM Computing Survey,* 4:375-408.

Kerr, D., Chan, D. and Cooper, R. The variety of approaches to change propagation in multiversion databases: University of Glasgow, Glasgow, England, 1992.

Kim, W. and Chou, H. 1988. Versions of schema for object-oriented databases. In *Proceedings of the 14th International Conference on Very Large Data Bases,* 148-159: Morgan Kaufmann.

Klahold, P., Schlageter, G., and Wilkes, W. 1986, A general model for version management in databases. In *Proceedings of the 12th International Conference on Very Large Data Bases,* 319-327, Kyoto, Japan: Morgan Kaufmann.

Kwang, J.B. and McLeod, D. 1993. Toward the unification of views and versions for object databases. In *Proceedings of the Object Technologies for Advanced Software; First JSSST International Symposium,* 222-236. Berlin: Springer-Verlag.

Landis, G.S. 1986. Design evolution and history in an object-oriented cad/cam database. In *Proceedings of the 31th COMPCON Conference*, San Francisco, CA.

Lerner, B.S. and Habermann, A.N. 1990. Beyond schema evolution to database reorganization. In *Proceedings of the Conference on OOPSLA (ECOOP'90),* 67-

76, Ottawa, Canada: ACM Press.

Moerkotte, G. and Zachmann. A. 1993. Towards more flexible schema management in object bases. In *Proceedings of the IEEE 9th International Conference on Data Engineering*, 174-181, Wien, Austria: IEEE Computer Society Press.

Monk, S.R. and Sommerville, I. 1992. A model for versioning of classes in object-oriented databases. In *Proceedings of the 10th British National Conference on Databases*, 42-58, Aberdeen, Scotland: Springer-Verlog.

Monk, S.R. and Sommerville, I. 1993. Schema evolution in oodbs using class versioning. *SIGMOD RECORD*, 3:16-22.

Monk, S.R. 1993. *A Model for Schema Evolution in Object-Oriented Database Systems.* Ph.D. Dissertation, Computing Department, Lancaster University.

Narayanaswamy, K. and Bapa Rao, K.V. 1988. An incremental mechanism for schema evolution in engineering domains. In *Proceedings of the fourth International Conference on Data Engineering*, 294-301, Los Angeles, CA: IEEE Computer Society Press.

Nguyen, G.T. and Rieu, D. 1989. Schema evolution in object-oriented database systems. *Data and Knowledge Engineering*, 1:43-67.

Odberg, E. 1992. A framework for managing schema versioning in object-oriented database. In *DEXA 92. Database and Expert Systems Applications: Proceedings of the International Conference*, 115-120: Springer-Verlog.

Penney, D.J. and Stein, J. 1987. Class modification in the gemstone object-oriented dbms. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 111-117, Orlando, Florida: ACM Press.

Roddick, J.F. 1991. Dynamically changing schemas within database models. *Australian Computer Journal*, 3:105-109.

Roddick, J.F. 1992. Sql/se - a query language extension for database supporting schema evolution. *SIGMOD RECORD*, 3:10-16.

Rumbaugh, J. 1988. Controlling propagation of operations using attributes on relations. In *Proceedings of the OOPSLA '88 Conference*, 285-296, New York: ACM Press.

Scherrer, S., Geppert, A., and Dittrich, K.R. 1993. Schema evolution in no$^2$: University of Zurich, No. 93.12.

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

Skarra, A.H. and Zdonik, S.B. 1986. The management of changing types in an object-oriented database. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications,* 483-495, Portland, Oregon: ACM Press.

Skarra, A.H. and Zdonik, S.B. 1987. Type evolution in an object-oriented database. In Bruce Shriver, B. and Wegner, P.(Ed.), *Research Directions in Object-Oriented Programming,* MIT Press.

Tan, L. and Katayama, T. 1989. Meta operations for type management in object-oriented databases - a lazy mechanisms for schema evolutions. In *Proceedings of the first International Conference on Deductive and Object-Oriented Databases,* 241-258, Kyoto, Japan: North-Holland.

Tresch, M. and Scholl, M.H. 1993. Schema transformation without database reorganization. *SIGMOD RECORD,* 1:21-27.

Vines, P., Vines, D., and King, T. 1988. Configuration and change control in gaia. *Communications of ACM.*

Zdonik, S.B. 1986. Version Management in an Object-Oriented Database. In *Proceedings of the 4th International Workshop on Advanced Programming Environments,* Trondheim, Norway.

Zdonik, S.B. 1990. Object-Oriented Type Evolution. In Bancilhon, F. and Buneman, P. (Eds.), *Advances in Database Programming Languages,* 277-288: Addison-Wesley.

Zicari, R. 1989. Schema Updates in the O2 Object-Oriented Database System: Dipartimento di Elettronica - Politecnico di, Milano, 89-057.

Zicari, R. 1991. A framework for schema updates in an object-oriented database system. In *Proceedings of the 7th International Conference on Data Engineering,* Kobe, Japan.