An Extensible Object-Oriented Database Testbed *

Magdi M. Morsi & Shamkant B. Navathe
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
sham@cc.gatech.edu

Hyoung-Joo Kim
Computer Engineering Department
Seoul National University
Seoul, Korea
hjk@krsnucc1.bitnet

Abstract

This paper describes the object-oriented design and implementation of an extensible schema manager for object-oriented databases. In order to achieve extensibility, we have adopted an *open class hierarchy* approach using an object-oriented implementation where an object-oriented database maintains its implementation classes as user-defined classes. A Graphical interface for an Object-Oriented database Schema Environment, *GOOSE*, has been developed. *GOOSE* supports several advanced features which include *schema evolution*, *schema versioning*, and *DAG rearrangement view* of a class hierarchy. *Schema evolution* is the ability to make a variety of changes to a database schema without reorganization. *Schema versioning* is the ability to define multiple schema versions and to keep track of schema changes. A new type of view for object-oriented databases, namely the *DAG rearrangement view* of a class hierarchy, is also supported.

1 Introduction

Database management systems have been successful in supporting traditional business applications. This success has created an interest in investigating efficient support for nontraditional applications, such as software engineering environments, engineering applications for CAD/CAM/CIM and VLSI data, knowledge based systems, multimedia information systems, and office information systems [1, 2, 3].

Nontraditional applications require support for more complex modeling primitives and data structuring capabilities. Moreover, they require additional functionality to deal with evolving environments because design applications, in general, tend to be iterative in nature, where the completion of the design process may require investigating several alternatives. The object-oriented data models are more powerful than traditional data models due to their support for inheritance in a class hierarchy, complex objects, object identity, and an easy incorporation of new data types.

Support for dynamically evolving objects includes the evolution of objects as well as the evolution of their schema. The evolution of the schema supports alternative representations of object classes. The schema versioning mechanism enables the user to manipulate versions of a class in a single class hierarchy. Also, the class versioning should be independent of the object versioning mechanism supported. For example, a versioned class does not have to have all objects in its scope versioned. The incorporation of schema versioning enables the user to experiment with alternative objects as well as alternative classes.

The *open class hierarchy* approach has been adopted to achieve the extensibility of the implementation. In this approach, the system meta information is implemented as objects of system classes. One of our ultimate objectives is to extend the notion of consistency to the methods. *Method consistency* is based on the robustness of the method interface and the validity of a method implementation in a dynamically changing schemas. In order to maintain method consistency, the system maintains the method sources and their interfaces. The advanced features, described in this paper, have been implemented as a specialization of the system classes.

1.1 Overview

This paper describes the advanced features and the graphical interface, *GOOSE*, component of an object-oriented database system under development at the Georgia Institute of Technology. A primary objective of this research is investigating support for dynamically evolving objects and their schemas in object-oriented databases. In this paper, we focus more on

*This work is partially supported by NSF grant number IRI-9010120.

0-8186-2545-7/92 $3.00 © 1992 IEEE

the structural aspects of the system and its schema environment.

In this section, we define the terms used and the implementation of an extensible object-oriented data model. Section 2 describes the data model and the extensibility of its implementation. Section 3 briefly describes the architecture of our testbed. Section 4 describes the support for schema evolution and versioning in *GOOSE*. Also, we briefly describe the support for *DAG rearrangement* views of a class hierarchy (DAGR views), and method conversion. Section 5 concludes the paper by describing additional extensions to *GOOSE* and future research issues.

2 Data model and its implementation

In this section, we describe the core concepts of object-oriented data models with which our data model is consistent. The adopted data model maintains its implementation components as well as user-defined classes as a part of the overall schema definitions. We use the term *object* to represent an encapsulation of instance variables containing a state, and methods for manipulating it. *Classes* are used to describe object types, as defined by their instance variables and methods. The *properties* of a class connote its instance variables and methods. Objects communicate via *messages*. The *domain* of an instance variable is the set of values it might have from its class type, which is called the *domain class*.

Inheritance is a mechanism used for defining specialization relationships between classes. Given a property V applicable to a number of classes, we define the *origin* of V as the most generalized class where V is locally defined. *Multiple inheritance* allows a class to inherit properties from several superclasses. If a property is inherited from more than one superclass that has a single origin, only one property is included in the class definition.

We use the notion of *full forced inheritance* (*FFI*). Under *FFI*, even if there is a naming conflict among inherited properties, all of these properties must be inherited and none should be rejected. To resolve the naming conflict, the property name is prefixed with the name of the class in which it is locally defined.

The ISA relationship captures the relationship between a class and its superclass and is represented in a class hierarchy. Because of multiple inheritance, the resulting structure is not just a hierarchy, but a single rooted DAG (lattice). The database schema consists of a class hierarchy along with the set of locally defined properties of each class.

Subclasses are allowed to restrict the domain of an inherited instance variable. The domain of an instance variable inherited from multiple superclasses is restricted to the intersection of the instance variable domains.

The *scope* of a class C contains all objects that are visible through C which consist of all objects of type C as well as those of the subclasses of C. The *direct scope* of a class C contains all objects in the scope of C which are not in the scope of any subclass of C.

2.1 The extensibility of our data model

In our implementation, we extended the notion of unique object identifiers to system objects which capture the database schema, namely *classes, instance variables*, and *methods*. Each of these components is an object of a meta class. This approach supports a consistent data model where full forced inheritance is supported. For each user-defined instance variable, its name, origin class, domain class, and domain constraints are maintained. Also, for each method, its name, origin class, and parameters are maintained. The properties of each class include its name, the set of superclasses, the set of methods, and the set of instance variables.

Schema Invariants The database schema invariants are based on the invariants defined by Banerjee et al. in [4]. The first invariant ensures that the class hierarchy is a single rooted DAG. The second invariant ensures unique naming of the database schema components. However, in our model, this invariant ensures the unique naming of classes for *only the locally* defined properties associated with each class. The third invariant ensures unique origin of each property. The fourth invariant ensures the domain compatibility of the properties in the subclasses of their origin; that is, a class C may modify the domain of an inherited property P as long as the new domain is a subset of the domain of P in the superclasses of C. The fifth invariant ensures full inheritance among a class and its subclasses; that is, a class inherits all the properties of its superclasses. The naming conflict that may arise because of inheritance is resolved by prefixing the property name with its class of origin. In our implementation, the external name of a property is a computed value which may change as a side effect of some schema evolution operations.

The objective of this approach includes providing a consistent and extensible environment where the system classes and the user-defined classes are maintained. That is, a *full fledged object-oriented im-*

plementation where the schema components and the sources of the implementation are maintained. The advantages of this approach include:

1. Extending the implementation to support several advanced features and customization of supported features for specific applications.

2. Maintaining the source code of methods implementation enables the system to perform a method conversion as a side effect of dynamic schema changes. For example, renaming of a class or a method would result in the conversion of the affected methods source code.

3. Extending the notion of schema consistency to method consistency. For example, dropping a class C may result in notifying the user of potential inconsistency if an instance variable or a method parameter is of type C. Also a method implementation may be invalidated if it contains a local variable of type C.

2.2 Implementation of data model

In this section, we briefly describe the system classes which are necessary to implement and maintain the database schema and its evolution operations. These classes are shown in Figure 1. A more comprehensive description of these classes is found in [5].

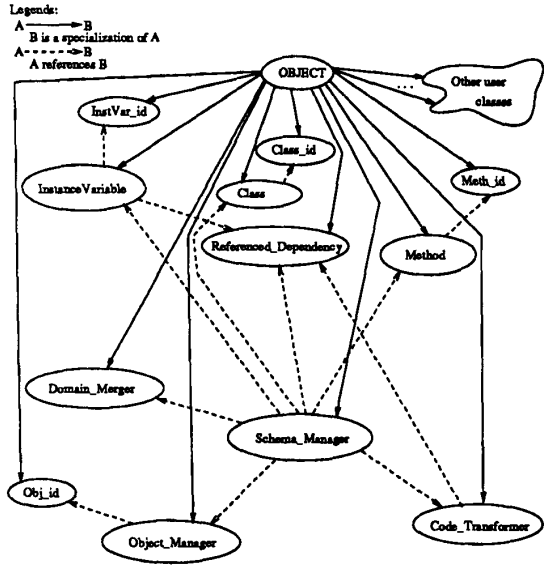

Figure 1: Class hierarchy of system components for supporting schema evolution

We classify the system classes used to maintain the database schema into six categories. These categories present functions and abstractions of the system.

1. The identifier classes of the system and user objects are rooted by the **Identifier** class which has a single instance variable **ID**. The **Versioned_id** and **Typed_id** are subclasses of **Identifier**. The **Versioned_id** class has a local instance variable, **Version_No**, whereas the **Typed_id** class has **Type** as a local instance variable. The **Versioned_Typed_id** is a subclass of the **Versioned_id** and **Typed_id** classes. Each system object of class (**Class**, **Method**, and **InstanceVariable**) is uniquely identified by a system identifier (**Class_id**, **Meth_id**, and **InstVar_id**, respectively).

2. The classes for maintaining the description of the database schema include **InstanceVariable**, **Method**, and **Class**. The decomposition of the database schema into instance variables, classes, and methods enables the database implementor (DBI), to define additional properties and constraints on system objects.

3. The **Schema_Manager** class captures the semantics of the data model and the schema evolution operations. It maps each schema evolution operation to a set of operations on classes that maintain the object-oriented database schema and propagates the changes to the underlying system objects in their scope. This includes parsing and validation of the user provided operations.

4. The **Object_Manager** component maintains information about the object instances in the database. This information includes object instances in the direct scope of each class in the database schema.

5. The **Domain_Merger** class is used for the computation of the domain intersection of inherited properties and relationships among domains, such as subset and disjointedness.

6. The support classes include the **Code_Transformer** class and the **Reference_Dependency** class. The **Code_Transformer** class maintains the source code of the methods in a format independent of the user naming space. The **Reference_Dependency** class keeps track of the instance variables and methods reference by a method. Consider the following example.

Example: 1 A database schema contains the two classes: *Auto* and *Employee*. The instance variables of the *Employee* class include *Car* whose domain class is *Auto*. This would result in a dependency between the *Car* instance variable and its domain class *Auto*. The deletion of the *Auto* class triggers a warning to the user of a possible inconsistency in the database schema. □

3 System Architecture

In order to provide extensibility of the implementation, the object identifiers used in *GOOSE* are independent of the physical location of the object as well as the object's home class. This information is maintained in an object catalog along with the object label. The object label is an optional symbolic label that is used to logically manipulate objects in *GOOSE*. For example, in order to delete, modify, or assign an object reference to a value of an instance variable, the object label is used instead of the object identifier.

The architecture of the system consists of four layers. The bottom layer consists of the Wisconsin Storage System, *WiSS* [6]. The object manager is implemented on top of *WiSS*. It contains two classes: *Object* and *Object Catalog*. The *Object* class is used to store and retrieve objects using class descriptions which describe the type of its instance variables. The *Object Catalog* associates with each object identifier its home class, the physical location, and its label. The *Schema Manager* is implemented on top of the object manager and supports schema evolution, versioning, and DAGR views. It also maintains several main memory structures for caching information, such as *Class Scope* and *Object Buffer*. The *Class Scope* associates a set of object identifiers of those objects in its direct scope. The *Class Scope* is created upon the first access to the class scope and it provides operations on all objects in its direct scope, such as dropping or adding an instance variable. *Object Buffer* maintains a main memory image of each accessed object. The top layer consists of the graphical interface (*GOOSE*).

Separating the objects from their physical locations and their home classes enables the system to support *object migration* where objects may change their home classes either implicitly as a side effect of their constraints or explicitly through user commands. The object-oriented design and implementation of *GOOSE* provides extensibility of the system components through subclassing. For example, the support for schema versioning is implemented by specializing the classes used to support schema evolution.

Also the support for viewing system classes enables the user to manipulate the system classes as user classes which are subject to schema evolution and versioning. This allows the user to add constraints on system objects as user objects. For example, it is possible to disallow multiple inheritance by restricting the set of superclasses to a single class identifier.

GOOSE has been implemented using X windows [7] and the C++ programming language [8]. The components of the database schema and user-defined objects are stored using the Wisconsin Storage System, *WiSS* [6]. The purpose of *GOOSE* is to provide a user friendly graphical interface for the database designer or the DBA for designing and modifying object-oriented schemas. It may be looked upon as a quick prototyping interface for building and prototyping to obtain user approval by showing users how various update operations (insert, delete, and modify) affect underlying object instances.

4 Schema evolution, versioning, and DAGR view support

In this section, we briefly describe the design and implementation of *GOOSE* and our approach to support the schema evolution and versioning mechanisms.

4.1 Schema evolution

An important requirement for nontraditional applications is schema evolution: the ability to make a wide variety of changes to the database schema dynamically without resorting to database reorganization. These changes include modifying the structure of a class hierarchy and changing a class properties. The schema evolution operations are provided as methods of the **Schema_Manager** class that manipulate the underlying database schema. Our framework for schema evolution is based on the framework described in [10].

4.1.1 Issues and related work

Two main approaches to propagating schema changes to the affected objects have been suggested. The *Orion* approach, called *screening* [11], defers the propagation of schema changes to persistent objects. These changes are propagated to the affected objects only when they are retrieved. In the *GemStone* approach, called *conversion* [11], schema changes are made instantly which include auxiliary structures of the affected objects. Our approach to propagate schema evolution operations differs from the *ORION screening* approach and the *GemStone conversion* approach

in propagating schema changes to the affected objects. *GOOSE* propagates the changes instantly to main memory objects which are made permanent upon issuing a **save** operation. We consider this a more reasonable compromise. *GOOSE* uses a *screening* approach to resolve dangling references.

4.1.2 Schema evolution operations

GOOSE supports a wide set of schema evolution operations, which include addition, deletion, and modification of schema components, such as classes, instance variables, and methods.

The schema evolution operations manipulate instance variables, methods, classes, and ISA relationships among classes. These operations include the adding and dropping a class, an instance variable to/from a class, a method to/from a class, and an ISA relationship. Operations on instance variables include modifying its name, the class of origin, the domain class, or the domain constraints. Similarly, operations on a method include specialization of the interface, modifying its name and the class of origin, and overriding of its code.

4.2 Schema versioning

Schema versioning is supported by extending the schema evolution framework to allow the users to manipulate several schema versions of an object-oriented database in a single class hierarchy. Our framework is based on the schema versioning mechanism described in [10]. The major benefits of our schema versioning is in keeping track of schema versions and the support for more flexible derivation of object versions; that is, object version derivation may cross multiple schema versions. Each version of a class is subject to schema evolution operations. Also it may have its own direct scope.

4.2.1 Issues and related work

Related work in supporting multiple versions of the schema includes *Encore* [12], and a proposed schema versioning approach for *ORION* database system by Kim and Chou in [13].

The adopted schema versioning mechanism is based on versions of a class. The implementation of the schema versioning mechanism is incorporated with the schema evolution operations. The schema versioning mechanism is based on functionality rather than a mechanism to improve performance, such as the approach proposed in *ORION* [13].

4.2.2 Schema versioning operations

The derivation of a class version from its parent class is either through *specialization, generalization,* or *sibling* derivation. *GOOSE* supports two system classes for each versioned class, namely *generic* and *dictionary*. The purpose of the *generic* class is to provide access to all objects in the direct scope of all versions of a class. A *generic* class of a versioned class is created as a subclass of the root class, *OBJECT*, and has no direct scope associated with it. The properties of a *generic* class include the intersection of the instance variables and methods which are locally defined in all the class versions. The *dictionary* class of a versioned class is defined to contain a union of all locally defined instance variables and methods in every version of its class. The *dictionary* class is created as a subclass of the *generic* class and has neither a scope nor a subclass. The unique naming invariant of a class property is enforced among all versions of a versioned class. For example, if CAR is a versioned class and CAR;1 is a class version of CAR which has *numberOfDoors* as a locally defined instance variable, no other class versions of CAR may have another local property with the same name. However, any class version of CAR which does not have *numberOfDoors* may include it from the class dictionary, which is called CAR.Dictionary.

Derive a class version Vj from another class version Vi: The derivation of a class version Vj from its parent class Vi through *specialization* creates Vj as a subclass of Vi without any additional local properties. Subsequently, the user may add local properties to Vj. The derivation through *generalization* creates Vj as a superclass of Vi. Vj will have only the generic class as its superclass. Also the locally defined properties of the Vi class are promoted to Vj; That is Vj become the origin of all locally defined properties in Vi. Subsequently, the user may change the origin class of some of these properties. Similarly, the derivation of Vj as a *sibling* of Vi creates Vj as a sibling class version with the same superclasses and locally defined properties as those of Vi. However, Vj will not have any subclasses. Subsequently, the user may drop and add local properties to Vj.

The class derivation hierarchy of a versioned class is maintained. This derivation hierarchy provides labeling of the nodes (classes), *released* or *transient*, where each node present a class version. Also the type of derivation of a class version from its parent class is also maintained as an edge label.

The implementation of the schema versioning com-

ponent, as depicted in Figure 2, is based on the specialization of a generic version control and schema evolution classes. This approach enables the user to access class versions in a manner similar to object versions. Its also reduces the redundancy of the system code. In the rest of this section, we briefly describe the specialization of the system classes to support schema versioning. A more comprehensive description of the implementation is found in [5].

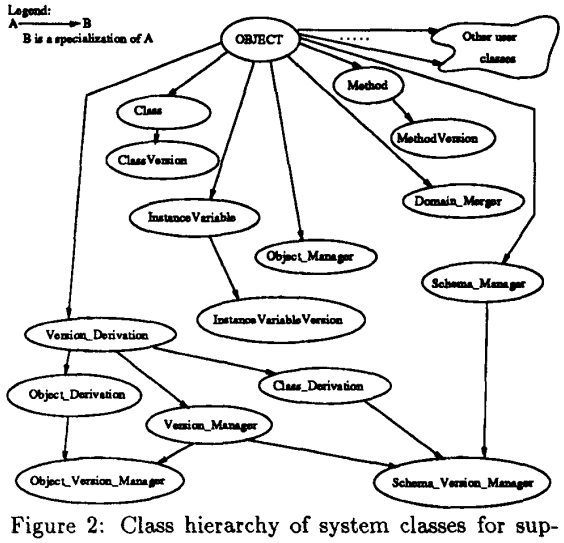

Figure 2: Class hierarchy of system classes for supporting schema versioning

Each instance of a class is uniquely identified by an instance of the `Class_id` class. `Class_id` is a subclass of `Versioned_Typed_id`. Since a class may have several versions, the version number is used to uniquely identify each. The different class types include *non-versioned*, *versioned*, *dictionary*, and *generic*.

The `InstanceVariableVersion` class is a subclass of `InstanceVariable` class. Since an instance variable of a versioned class may be locally defined in several class versions, an additional instance variable is added to maintain auxiliary origins which contains a set of `Class_id`'s. Similarly, the `MethodVersion` class is a subclass of `Methods` class.

The `ClassVersion` class is a subclass of the `Class` class for supporting system classes. Also, the `Add_Class` method has an additional parameter indicating the class type.

4.3 Graphical interface for schema evolution and versioning operations

The schema evolution and versioning operations are incorporated in a single interface. The *GOOSE* environment enables the user to manipulate an entire schema using *Create, Load, Copy,* and *Save* operations. These operations are shown in a separate window, called *GOOSE Graphical Schema Environment.*

The *GOOSE Editor* component supports the schema evolution and versioning operations. The *GOOSE* editor command line is used to display the currently executing command; whereas the *GOOSE* editor message displays warning and system messages. The editor operations are grouped into eight groups, namely *Show, Add, Modify, Drop, Cancel, Save, Print,* and *Layout*, where each group has an associated button at the bottom of the editor window. Clicking on a group button may result in either carrying out the associated operation (*Cancel* and *Save*) or creating a window with a set of options. An option window defines a set of operations associated with a class or a property of a class. For example, the *Drop* option window includes dropping a class, an instance variable or a method from a class, an object from the direct scope of a class, and a constraint associated with an instance variable. The classes of the database schema are graphically represented where each class is presented by a push button. For example, to drop a class, the user has to click on the *Drop* button. The *Drop* option window appears from which the *Class* option is selected. In order to perform this action, the user has to click on one or more classes to be dropped.

4.4 DAG rearrangement views

A new type of view for object-oriented databases, namely DAG rearrangement (DAGR) view of a class hierarchy [10], is being developed by specializing the classes of the schema manager. We refer to them as DAGR views. The main premise behind the DAGR views is that users would like to see certain classes and properties that are relevant to their applications. These views have to be consistent with the underlying database schema.

DAGR view invariants The first invariant requires that the set of properties of a class in a DAGR view is a subset of its underlying properties. This ensures that the state of a viewed object is consistent with its state in the database. The second invariant requires that the scope of a class in a DAGR view is a subset of its underlying scope.

The operations used to define a DAGR view include hiding a class, hiding an ISA relationship between two classes, adding a new class, and adding an ISA relationship between two classes. For example, the semantics of the hiding operation on a class C include dropping all locally defined properties of C from its subclasses. However, all objects in the direct scope of C will remain visible in the scope of its superclasses.

Also the schema of a class may be modified either through hiding a property, demoting a property, or constraining the domain of a property. The notion of a black box is used to view a set of related sibling classes as a single category class. The *GOOSE* support for DAGR views in described in [14].

4.5 Method conversion

Maintaining the consistency of the database schema is an important issue in object-oriented databases. The effect of schema evolution operations may leave the database schema in an inconsistent state. For example, whenever a class C is deleted and there may exist a set of instance variables having C as their domain class, the user has to be notified of the inconsistency as a result of this operation. Similarly, each method and instance variable has associated with it a set of dependent methods that uses it.

Monitoring the effect of schema changes is an important issue in a dynamically evolving environment in order to preserve the consistency of the database schema. In order to monitor the effect of schema changes, we shall maintain the reference dependencies of system components as well as method source code. A code transformer will be used to translate the references to schema components with their respective object identifiers. It will also produce a list of dependencies used to monitor the schema change operations. The code transformer will also support the reverse operation; that is, producing an updated source code of the methods affected by the schema change operations. These dependencies will be used for notifying the users of possible inconsistency which may result as a side effect of schema changes.

Since the behaviors of objects are captured as methods of the object class, the efficient support of method invocation on versioned objects will be addressed. For example, if an object B contains a static binding to a version of an object, the invocation of a method which updates B may be prohibited or at least notified of possible abortion as dictated by the object version type.

The research for investigating method conversions include classification of schema changes and their effect on method conversion. We shall investigate a mechanism for achieving compatibility of methods in light of dynamic schema changes. This includes a framework for the automatic method conversion and a notification mechanism for user interaction whenever automatic method conversion is limited.

4.6 Status

At the time of writing *GOOSE* supports schema evolution, versioning, and DAGR views. For schema versioning, we currently support two types of versioned classes, *transient* and *released*. The modification of a *released* class version is restricted; that is, no properties can be dropped from it. Two modes of operations will be supported, a *restricted* mode and a *free* mode. The *free* mode is a DBA mode in which the restrictions associated with a *released* class version are relaxed.

The support classes *Code_Transformer* and *Reference_Dependencies* are in the design phase along with the notification mechanism. We expect that this features would be incorporated by the mid 1992.

5 Conclusion and future research

In this paper, we described the implementation of a graphical interface for object-oriented database schema environment. The data model supported is based on generalizing the concept of object identifiers to the database schema components. *GOOSE* supports several advanced features which include *schema evolution, schema versioning*, and *DAG rearrangement view of a class hierarchy.*

This research investigates the support for dynamically evolving objects and their schema in object-oriented databases. It is motivated by the complexity of nontraditional application requirements and the complexity of object-oriented database schema. In order to support these environments, several advanced features have been incorporated in the object-oriented database schema environment. The design and implementation of *GOOSE* generalizes the concept of unique object identifiers of user-defined objects to the database schema components and to the DBMS system components as well.

This research is part of investigating an object-oriented design and implementation of an extensible object-oriented database system. The approach adopted to achieve extensibility is an open architecture approach. The goal of this approach is to provide a set of mechanisms for maintaining the database system components that is also used to maintain the

database applications in an integrated environment. The following is a list of additional research problems being addressed.

GOOSE extensions One of these extensions is the support of a graphical query language for object-oriented databases and the incorporation of a tutorial using computer animation. Extensible version management of objects and its efficient support will also be addressed.

Schema integration We shall investigate the use of an object-oriented data model as a meta model for schema integration. A set of primitive operations for mapping from relational, network, hierarchical, and object-oriented database schemas to an object-oriented data model will be provided. Also, the automatic integration of these local schema into an object-oriented global schema will be addressed which includes automatic conflict detection and resolution techniques.

Storage Models This research includes the study of performance of several physical models for storing objects. This study only addresses the effect of schema change operations. Also efficient support for versioned objects in object-oriented databases will be investigated.

References

[1] D. Woelk and W. Kim, "Multimedia information management in an object-oriented database system," in *Proc. of the 13th Conf. on Very Large Databases*, UK, pp. 319–329, Sept. 1987.

[2] H. Assarmanesh, D. McLeod, D. Knapp, and A. Parker, "An extensible object-oriented approach to databases for VLSI/CAD," in *Proc. of the 11th Conf. on Very Large Databases*, Sweden, pp. 13–24, Aug. 1985.

[3] R. Ahmed and S. Navathe, "Version Control of Complex Objects in CAD Databases," in *Proc. of ACM-SIGMOD Conf.*, Denver, CO, pp. 218–227, May 1991.

[4] J. Banerjee, H.-T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H.-J. Kim, "Data model issues for object-oriented applications," *ACM Trans. on Office Info. Sys.*, vol. 5, pp. 3–26, Jan. 1987.

[5] M. Morsi, H.-J. Kim, and S. Navathe, "Object-Oriented Design and Implementation of an extensible Schema Manager for Object-Oriented Databases." In preparation.

[6] H.-T. Chou, D. DeWitt, R. Katz, and A. Klug, "Design and implementation of the wisconsin storage system," *Software-Practice and Experience*, vol. 15, pp. 943–962, Oct. 1985.

[7] R. Scheifler and J. Gettys, "The X Window System," *ACM Transactions On Graphics*, vol. 5, pp. 79–109, Apr. 1986.

[8] B. Stroustrup, *The C++ programming Language.* Reading, MS: Addison-Wesley Publishing Co., 1986.

[9] M. Morsi and H.-J. Kim, "Extensible database management systems: Existing approaches and a new approach," in *To Appear in Progress in Object Oriented Databases* (J. Prater, ed.), Albex Pub. Cor., Submitted for copy-editing.

[10] H.-J. Kim, *Issues in Object-Oriented Database Schemas.* PhD thesis, Department of Computer Science, The University of Texas at Austin, Austin, TX, May 1988.

[11] D. Penney and J. Stein, "Class modification in the GemStone Object-Oriented DBMS," in *Proc. of the 2nd. OOPSLA Conf.,*, Kissimmee, FL, pp. 111–117.

[12] A. Skarra and S. Zdonik, "The management of changing types in an object-oriented database," in *Proc. of the 1st OOPSLA Conf.,*, Portland, OR, pp. 483–495.

[13] W. Kim and H.-T. Chou, "Versions of schema for object-oriented databases," in *Proc. of the 14th Conf. on Very Large Databases,* Los Angles, CA, pp. 148–159, Sept. 1988.

[14] M. Morsi, S. Navathe, and H.-J. Kim, "A Schema Management and Prototyping Interface for an Object-Oriented Database Environment," in *Proc. of the IFIP conf. on the Object Oriented Approach in Information Sys.*, Canada, pp. 157–181, North Holland, Oct. 1991.